S(o)OS
Service-oriented Operating Systems

State of the Art

Project Deliverable D5.1

Tommaso Cucinotta, Giuseppe Lipari, Dario Faggioli,
Fabio Checconi and Sunil Kumar (SSSA)
Rui Aguiar, João Paulo Barraca, Bruno Santos and Javad Zarrin (IT)
Jan Kuper and Christiaan Baaij (UT)
Lutz Schubert and Hans-Martin Kreuz (USTUTT-HLRS)
Vincent Gramoli (EPFL)

Due date: 31/07/2010
Delivery date: 31/07/2010

SEVENTH FRAMEWORK PROGRAMME

This work is partially funded by the
European Commission under
FP7-ICT-2009.8.1, GA no. 248465

(c) 2010-2012 by the S(o)OS consortium
This work is licensed under the Creative Commons Attribution 3.0 License.
To view a copy of this license, visit http://creativecommons.org/licenses/by/3.0/ or send a letter to Creative Commons, 171 Second Street, Suite 300, San Francisco, California, 94105, USA.

Version History

Version	Date	Change	Author
0.1	10/03/10	First TOC	Tommaso Cucinotta Lutz Schubert
0.2	04/05/10	Merged UT, HLRS and SSSA contributions	Tommaso Cucinotta
0.3	31/05/10	Integrated SSSA, UT, IT, HLRS contributions	Tommaso Cucinotta
0.4	03/06/10	Added preliminary introduction and section on scheduling and synchronization	Tommaso Cucinotta
0.5	08/06/10	Merged further contributions	Tommaso Cucinotta
0.5.1	12/06/10	Merged contributions from UT, IT, HLRS and SSSA	Tommaso Cucinotta
0.6	15/06/10	Merged HLRS comments and EPFL contribution on Transactional Memory	Tommaso Cucinotta
0.7	19/06/10	Refined SSSA contribution on hardware interconnects, merged SSSA section on protection levels and HLRS contribution on compilers	Tommaso Cucinotta
0.8	20/06/10	Merged refined HLRS contribution on compilers, recovered SSSA contribution on locking	Tommaso Cucinotta
0.9	17/07/10	Merged reviewers comments	Tommaso Cucinotta
0.A	22/07/10	Merged references fixes and further minor contributions	Tommaso Cucinotta
1.0	28/07/10	Merged HLRS fixes of formatting issues.	Tommaso Cucinotta

EXECUTIVE SUMMARY

This document provides an overview of the current state of the art in both industrial practice and academic research in the areas of parallel and massively parallel systems, as well as the support for such platforms at the Operating System (OS) and programming levels. From the survey presented in this document, it is evident that most of the existing work undertakes an incremental approach, in which new OS features, kernel architecture and programming models build mainly upon existing and widely used technologies. Only a few of the presented works try to rethink and redesign from scratch the software stack that is needed in order to exploit parallelism to the degree that will be available on future computing many-core platforms. From the perspective of the S(o)OS project, the latter ones are the approaches that need to be more deeply investigated. In fact, the research that will be carried out in the project will move along such innovative directions of action.

The problem of such long-term approaches thereby consist mainly in the dynamic development, making long-term assessments uncertain and hence risky. Such short-term and long-term developments are discussed in more detail in D5.2 “Definition of Future Requirements” [86]. Accordingly, D5.1 and D5.2 together form the basis over which the research undertaken in the context of the S(o)OS project will further investigate.

This document constitutes the output of the S(o)OS Project Tasks T2.1, T3.1, T4.1 and T5.1 on “State of the Art Assessment”.

BY-ND

CC BY-ND

Table of Contents

CC BY-ND

BY-ND

List of Figures

CC BY-ND

List of Tables

CC BY-ND

ABBREVIATIONS

Abbreviation	Description
AD	Analog Devices
AES	Advanced Encryption Standard
AHT	Augmented Hypercube Torus
ALU	Arithmetic Logic Units
API	Application Programming Interface
API	Application Programming Interface
ASIC	Application Specific Integrated Circuit
ASIP	Application-Specific Instruction Processor
CAF	Co-Array Fortran
ccNUMA	Cache Coherent NUMA
CFS	Completely Fair Scheduler
CISC	Complex Instruction Set Computer
CPLD	Complex Programmable Logic Devices
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
DFF	Delay Flip-Flop
DMA	Direct Memory Access
DOR	Dimension-Ordered-Routing
DPCP	Dynamic Priority Ceiling Protocol
DSP	digital signal processors
DTM	decoupled transactional memory
DWDM	Dense Wavelength Division Multiplexing
EDF	earliest deadline first
FIF	First-In First-Out
FP	Fixed Priority
FPGA	Field Programmable Gate Array
FUTEX	Fast Userspace muTEX
GP-GPU	General-Purpose Graphics Processing Unit
GPP	General-Purpose Processor
GPU	Graphics Processing Unit
HIP	Host Identity Protocols
HPC	High Performance Computing
IPC	interprocess communication
ISA	Instruction-Set Architecture
IXS	Internode Crossbar Switches
JiT	Just-in-Time (Compiler)
LUT	Lookup Table
MINLP	Mixed-Integer Non-Linear Programming
MMX	Intel single-instruction multiple-data extensions
MPCP	Multiprocessor Priority Ceiling Protocol
MPI	Message-Passing Interface
MSRP	Multi-processor Stack Resource Policy
Mux	Multiplexer
NoRMA	No-Remote-Memory-Access
NRE	Non-Recurring Engineering
NUCA	Non-Uniform Cache Architecture
NUMA	Non-Uniform Memory Access
OAM	Operations Administration and Maintenance
OS	Operating System
OSI	Open Systems Interconnection (Reference Model)
PCP	Priority Ceiling Protocol

BY-ND

PGAS	Partitioned Global Address Space
PI	Priority Inheritance
POSIX	Portable Operating System Interface
PPE	Power Processing Element
QoS	Quality of Service
QPI	Quick Path Interconnect
RAM	Random Access Memory
RISC	Reduced Instruction Set Computer
RPC	Remote Procedure Call
RR	Round-Robin
RTOS	Real-Time Operating System
S(o)OS	Service-oriented Operating System(s)
SCC	Single-Chip Cloud Computing
SDK	Software Development Kit
SIMD	Single Instruction Multiple Data
SMP	Symmetric Multiprocessors
SMT	Simultaneous Multi-Threading
SoC	System-on-Chip
SPE	Synergistic Processing Element
SRAM	Static Random Access Memories
SRP	Stack resource Policy
SSE	Intel Streaming SIMD Extensions
SSI	Single-System Image
STM	Software Transactional Memory
TDMA	Time-Division Multiple Access
TI	Texas Instruments
TLB	Translation Lookaside Buffer
TM	Transactional Memory
UMA	Uniform Memory Access
UPC	Unified Parallel C
VIA	Vertical Interconnect Access
VLIW	Very Long Instruction Word
WS-BPEL	Web Services Business Process Execution Language

BY-ND

I. INTRODUCTION

Computing systems are experiencing nowadays a complete paradigm shift. On the old-fashioned single-processor platforms, sequential programming used to constitute an easy and effective way of coding applications, and parallelism was used merely to easily realise independent or loosely coupled components. True parallel and distributed programming used to constitute a domain reserved to only a relatively small number of programmers dealing with high-performance computing (HPC) systems. However, recently multi-core systems, such as the Core i7 by Intel or the Phenom by AMD, have become the de-facto standard for personal computing, from laptops to desktop computers. Multi-core processors are being increasingly used in the embedded domain as well. Furthermore, in the context of servers, it is more and more common to see multi-processor and multi-core systems realising up to 8 - 12 cores. The according process just continues: already experimental multi-core processors, such as the Polaris by Intel, reach up to 80 cores and specialised processors, such as the Azul Vega, go up to 54 cores, yet the mass market commercial exploitation of these processors will still take a few years. Along that line, high-performance and massively parallel systems are undergoing a tremendous architectural shift that promises to move towards an unimaginable number of interconnected cores and other hardware elements such as local memory/cache elements, within the same chip.

As a consequence, in the short future parallel and distributed programming paradigms need to become more and more widespread and known across the whole base of (software and hardware) developers, comprising not only the high-performance computing domain, but also the general-purpose one. Furthermore, even in the HPC domain itself, hardware makers foresee a steep increase in the level of parallelism.

However, the entire ensemble constituting the software stack of nowadays computing systems, comprising Operating Systems, programming languages, libraries and middleware, are still too much influenced by the former non-concurrent era, and are slow at adapting to the new scenario. Commercial development is thereby constrained by the demand to maintain downward compatibility which typically implies that only incremental, rather than disruptive, evolutionary steps can be taken. Vendors are scared by the risk that too large steps along the development of software could imply too much (resource) effort to maintain exactly this compatibility and that implicitly they could loose customers.

The S(o)OS project aims at exactly such a disruptive step to investigate on how the elements of the software stack need to be re-engineered in order to cope with the increasingly available parallelism in the hardware platforms of tomorrow. The S(o)OS project investigates on how hardware and software can come closer together to increase efficiency whilst maintaining a maximum degree of flexibility and adaptability. The attention is strongly focused on the Operating System level, as this level builds the major interface between hardware and software and thus is crucial for these goals [4].

In order to undertake the necessary research on such challenging themes, it is of course necessary to have a comprehensive overview of the current status of the industrial practices and research efforts, as well as of the academic research activities, recently developed in the context of parallel, distributed and massively parallel systems.

BY-ND

The purpose of this document is to provide a comprehensive state of the art on these topics, starting from the hardware domain, and moving across the overall software stack built on top of it, comprising Operating Systems, programming languages and paradigms, and middleware components. This document thereby primarily focuses on developments and research results that have shown wider acceptance and uptake, rather than being purely experimental in nature. This aims not only to reduce the size of the document, but mostly to reflect the *solid* existing basis for future development.
More experimental research results and – coupled to this – the long and short term trends that are represented by this research are assessed through project deliverable D5.2 “Definition of Future Requirements” [86], which represents the developments to be expected in the future, with an impact on future software stacks and the ways of dealing with applications and systems.

1. Organization of this document

This document is organized as follows. In Chapter II. we present a survey of hardware architectures, comprising details about processor and instruction-set architecture models, specifics of multi-core and many-core systems, memory architectures and support at the compiler level. In Chapter III. we provide a survey of protocols in distributed computing systems, with a focus on communications and control models for distributed, massively parallel and highly scalable systems. In Chapter IV. we examine various models of Operating Systems for parallel computing machines, along with a few important recent works in the area of OS support for massively parallel systems. In Chapter V. we address the specifics of the high-performance computing domain, from the hardware architecture to the programming paradigm perspectives. In Chapter VI. we present related work in the area of use cases, benchmarks and applications for anticipated future Operating System architectures. Finally, we conclude this document with a set of general considerations in Chapter VII. .

CC BY-ND

II. Hardware and Processor Architectures

This chapter investigates the current status in the layer underlying the software stack, i.e. on the hardware level of current computing platforms. This comprises in particular processors, interconnects and memory architectures in the different range of usage settings and application domains. These have an impact on what is required from a system and where and how much resources can be vested into their production – the most typical example for this difference consists in the High Performance Computing vs. the Mobile Computing areas:

The HPC area focuses in particular on efficiency and high-performance interconnects at an accordingly high cost. It is not unusual in this domain to develop an individual strand of processors, which often serves as a prototype for new processor architectures in the more mainstream models. See for example the Knights Ferry[1] by Intel. On the other side of the spectrum, the mobile devices domain focuses particularly on minimal size and energy consumption with price being an important factor due to the mass-market demand for mobile devices. Accordingly, processors in this domain typically show comparatively little performance and weak interconnects, but a good energy to performance ratio.

The current era of computing has reached petascale, i.e., 10^{15} operations per second, and the first approaches towards exascale (10^{18} operations per second) are already planned. To make this happen, we have to think about a new paradigm of computing that is not only in the direction of software parallelism but also in the one of hardware infrastructure parallelism.

In order to meet the computing requirements of nowadays' applications, one has to scale the computing infrastructure so as to reduce the processing times, thus keep feasibility of the computations. Various techniques are proposed to improve in particular the performance, efficiency and throughput of processing and memory units. Major ones are:

- silicone designers are putting more and more transistors in a single chip;
- chip designers are putting more and more cores on a single chip;
- various inter-connecting networks are being created for multiple cores in the chip;
- interconnections among computing units are being improved as well, both on the side of memory hierarchies and of the Input/Output sub-system;
- software developers are building applications which are split into multiple logical processes that can be mapped onto the physical available topology (cores, processors).

There are various challenges involved in increasing the scalability of the hardware infrastructure – these includes, among others:

- number of cores / microarchitecture: how many cores in which configuration provide the best scaling capabilities for the various types of applications;
- interconnection of multicores: which kind of connectivity between cores and processors is required;
- performance versus reliability: to which degree can the one be sacrificed for the benefit of the other?
- Quality-of-Service (QoS): how to preserve timing and QoS requirements of increasingly complex and parallel applications;

1 More information is available at: http://www.techspot.com/news/39138-intel-announces-50core-knights-corner-hpc-processor.html.

CC BY-ND

- energy consumption vs. performance: how to increase performance without hitting the power wall and even achieving a low power consumption.

In the following, a non-exhaustive description is made of the main mechanisms adopted so far by hardware vendors for tackling the just mentioned challenges. First, the discussion focuses on increasing performance of traditional single-processor systems, then it switches to the multi-core and multi-processor domain, ending in the area of nework-on-a-chip (NoC) and HPC systems.

1. Processor and ISA Models

This section discusses the state of the art of processor architectures and Instruction-Set Architecture (ISA) models, with a particular focus on the consumer market, i.e., general purposes processors that can be found in high-end desktops, laptops and embedded/mobile devices. This includes specialised processor types which can often be found in the embedded computing domain, such as reconfigurable and digital signal processors.

1.a) General-Purpose Processors Principles

State of the art general purpose processors (GPPs), particularly found in desktop machines, but also in large scale clusters, such as the Intel Nehalem architecture, are Reduced Instruction Set Computer (RISC) machines that have decoding mechanisms for their Complex Instruction Set Computer (CISC) x86 Instruction Set Architecture (ISA). To achieve a high average-case performance, GPPs include (multiple levels of) cache(s) to hide memory latency (see also section II. 5.). To increase the instruction level parallelism (ILP) and increase execution of sequential computation in general, these architectures employ multiple optimisation techniques, including out-of-order execution, (super) pipelining, superscalar data paths and (speculative) branch prediction [199]. In order to exploit the potential data-level parallelism in especially audio/video applications, most GPPs include special Single-Instruction Multiple-Data (SIMD) execution units and registers [199]. These GPPs also increase task-level parallelism by issuing and executing concurrent threads on their (superscalar) data paths, a feature called Simultaneous Multi-Threading (SMT) [200]. Besides SIMD execution units, the latest intel GPPs also include accelerators for cryptography routines [201] such as the Advanced Encryption Standard (AES).

Low-power GPPs, such as the ARM Cortex [202], are also pipelined, superscalar machines, and also have branch prediction to increase execution of sequential computations. Instructions are however scheduled in order. Like the high-performance GPPs, the low-power GPPs also employ caches to hide memory latencies. SIMD execution units for the the ARM Cortex line are optional.

1.b) Digital Signal Processors and Configurable Processors

Digital signal processors (DSPs) and Configurable processors are typically employed in the embedded and signal processing domain, including radar, video and audio applications. Traditional DSPs had specialised data-paths so that computations relevant to signal processing (such as Multiply-Accumulate and bit-shifting for division) could be executed in a single cycle. This meant that they were generally running on a lower clock speed as compared to GPPs, achieving a lower average-case performance for generic applications. However, for signal processing applications, the total execution time is several times lower. From the software stack point of view, the specialised data-path also means that the

CC BY-ND

compiler needs to group computations in a certain way to make effective use of this data-path.
The latest conventional digital signal processors (DSPs) by companies like Texas Instruments (TI) and Analog Devices (AD) are mostly Very Long Instruction Word (VLIW) machines, employing 2 or more parallel data-paths, with (modified) Harvard memory architectures [203][204][205]. Same as for the traditional DSPs, compilers for these new VLIW machines needs to group computations to make effective use of the parallel data-paths. While AD seems to be moving away from pure VLIW machines with the introduction of their "general-purpose" DSP called the BlackFin [206], TI usually combines their latest C6x series with an ARM GPP on a System-on-Chip (SoC), as is the case for their successful OMAP SoC [207].
Like the DSPs, configurable processors are typically employed in the (streaming) signal processing domain. Like the DSPs, they run at lower clock speeds than GPPs, meaning they have a lower average-case performance for generic applications, but perform considerately better for the application they are configured for. Configurable processors can be categorized over two dimensions: design-time vs. run-time and coarse-grain vs. fine-grain. In design-time configurable processors architectures, parts of the die can be specified/designed for a specific application (domain); after fabrication this configuration can no longer be changed.

DESIGN-TIME COARSE-GRAIN CONFIGURABLE PROCESSORS

Architectures recently emerged on the DSP market are the Xtensa [208] architecture of Tensilica, and the HiveX [209] platform from Silicon Hive. Both architectures fall within the class of design-time reconfigurable machines. The default Xtensa ISA is a RISC architecture and employs VLIW encoding. The instruction set can be increased or decreased depending on the application requirements, thus making a deployed Xtensa architecture actually an Application-Specific Instruction Processor (ASIP). Reducing the instruction set has the potential to save on the memory needed to store the reduced-size code, and maybe even lower power dissipation. Silicon Hive processors consist of a VLIW data-path, where the individual function units are Vector/SIMD execution units. The function units, register files, I/O interfaces are adapted to the needs of the application code, allowing a Silicon Hive processor to be very specifically tailored to the needs of a specific application.

DESIGN-TIME FINE-GRAIN CONFIGURABLE PROCESSORS

To bridge the gap between Field Programmable Gate Arrays (FPGAs) and (cell-based) Application Specific Integrated Circuits (ASICs), many ASIC manufacturers and start-ups have developed some form of structured/platform ASICs. These structured ASICs have a predefined, and pre-produced silicon layout for many types of logic. These silicon cells range from complete IPs to the silicon parts of their fully routed ASIC cell siblings. The idea is that only the metal layer is customized/created for the custom routing of the cells, thereby reducing the Non-Recurring Engineering (NRE) costs immensely. The different companies use different techniques, ranging from adding Vertical Interconnect Access (VIA) to existing metal layers to adding completely new metal layers on top of existing (metal) layers. Another technique used by FPGA developers is using anti-fuses for the configuration of the devices: a high current is applied to a dielectric turning it into a short circuit, a process that is irreversible. While these N-phase methods of fabricating a die are much cheaper than fabricating a low-volume

CC BY-ND

ASIC directly from silicon, they are not cost-effective for the high-volume production of GPPs.

RUN-TIME COARSE-GRAIN RECONFIGURABLE PROCESSORS

Run-time coarse-grain reconfigurable processors usually employ an array of word-level processing elements, such as multipliers and Arithmetic Logic Units (ALUs). The programmable interconnection between the elements can be configured *at run-time* to create the desired signal flow; the elements themselves can be configured to execute specific operations. Coarse-grained architectures require less configuration than fine-grained architectures and therefore their configuration time is many orders of magnitude shorter (a few mili-seconds vs. several seconds). In some cases the configuration will result in a fixed functionality for the entire processor, no longer requiring it to fetch instructions every cycle, and basically turning the reconfigurable processor into an ASIC. Word-level algorithms within the intended algorithm domain, such as a Fast-Fourier Transform (FFT), can be executed very efficiently on coarse-grained reconfigurable architectures. However, fine-grained algorithms (manipulating 4 bits or less) execute inefficiently on these coarse-grained architectures. Examples of reconfigurable architectures include the Montium[2] [210] and the PACT XPP[3] [211].

RUN-TIME RECONFIGURABLE PROCESSORS

In fine-grained reconfigurable architectures the functionality of the hardware is specified at the bit-level and the programmable interconnect is manipulated as individual wires. The fine-grained flexibility comes at the cost of additional silicon and this overhead hampers the performance of word-level algorithms. Fine-grained architectures allow for manipulations at the bit-level. They are very efficient for complex bit-oriented computations or bit-level masking, shuffling and filtering (such as those used in encryption operations). Fine-grained architectures include FPGAs and Complex Programmable Logic Devices (CPLDs), where the traditional difference between the two architectures is that FPGAs use blocks comprising of a Lookup Table (LUT), Multiplexer (Mux) and Delay Flip-Flop (DFF) [212], whereas a CPLD uses the Sea of Gates (e.g., sum of products) to define its logic. There are many configuration mediums for these devices, ranging from Static Random Access Memories (SRAMs) where the devices have to be programmed every time they are powered on, over Flash memories where the configuration data is retained even when the device is switched off, to anti-fuses where the configuration becomes permanent [213].

1.c) Operation Modes

Modern computer architectures provide protected modes of operation that, together with other hardware mechanisms like memory protection, interrupt handling, etc. are commonly exploited by Operating Systems to efficiently implement multiprogramming (see also section IV. 1. on Operating System Models). The basic need covered by the protected mode of operation is isolating the Operating System from user applications, thus *protecting it* from unwanted voluntary or involuntary user influences and interferences.

The implementation of these modes varies a lot among the various CPU architectures available on the market. The common trait the various types of

2 More information is available at: http://www.recoresystems.com/index.php?option=com_content&task=view&id=73&Itemid=168.

3 More information is available at: http://www.pactxpp.com/.

BY-ND

implementation share is the support for multiple operating modes, with the system behaving differently depending on the active operating mode. At least two operating modes are usually supported:

- *User* mode: the CPU executes program code under various restrictions, e.g., on the instructions of the ISA that can be executed, the operands they may use, the registers they can access, etc.
- *System* mode: most of the restrictions to which user mode is subject are no longer in place, and the program code being executed has full control over the CPU. Some of the restrictions imposed to user mode operation can be configured from system mode.

The protection mechanisms of the CPUs provide also control over the transitions between the operating modes, ensuring their security. Some architectures also support the distinction between two additional modes used to enhance the support of virtualization:

- *Host* or *Hypervisor* mode: the CPU executes regularly and can control what operations are exactly allowed while in guest mode.
- *Guest* mode: the CPU executes a virtualized copy of itself, emulating efficiently the execution of its own instruction set in hardware and resorting back to hypervisor mode when the emulation cannot be done in hardware by the CPU itself (e.g., in case it requires access to I/O devices).

Figure 1 shows the transitions between the CPU modes in the case of x86 CPUs:

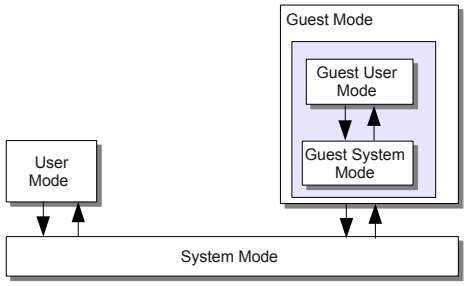

Figure 1: CPU protection modes and possible transitions.

These transitions need to be considered in modern OS kernel designs as they introduce a new kind of context-switch. The typical CPU allows for the following transitions between operating modes (cf. Figure 1):

- From user to system mode:
  - Software interrupts (e.g., system calls), where the user code explicitly asks for a service from the operating system with the hardware taking care of switching mode safely. For example, in the case of x86 CPUs the hardware switches to a system stack to avoid the user forcing the system to access unwanted memory regions.
  - Hardware interrupts or exceptions, generally handled in system mode, using a protection mode switch similar to the one used for system calls.
- From system to user mode:
  - End of interrupt: the hardware usually provides a mechanism to the operating system to resume user mode operation once the interrupt handling is over.
  - Explicit request by system code: for example in the x86 architecture this is done simulating in software the end of an interrupt.
- From hypervisor to guest mode:

CC BY-ND

    - Explicit request by system code, usually with a special instruction.
- From guest to hypervisor mode:
    - Hardware interrupts.
    - Hypercalls: the equivalent of system calls, but with the guest Operating System, with the awareness of being a guest OS, requesting directly and explicitly a service provided by the virtualization hypervisor (or host Operating System), implemented using special instructions.
    - Hypervisor exits: whenever the CPU is not able to continue emulation in guest mode, it asks for assistance from the hypervisor Operating System. For example, consider a write to an I/O port: the CPU in guest mode can be configured to immediately return to hypervisor mode, to let the hypervisor code handle the write emulating the device associated to the port, or to handle a protection error if the guest was not supposed to access the port.
- From guest user to guest system mode:
    - System calls: since the CPU emulates its own behavior on a parallel set of registers while in guest mode, the same sequence of actions used to implement system calls in user mode is emulated in guest mode, and provokes a switch to guest system mode.
- From guest system to guest user mode:
    - Returning from system calls.

Interrupts delivered to the guest Operating System need a special mention. They are generated by the hypervisor kernel (unless direct hardware access is allowed to guest mode) and there is hardware support to propagate them to guest mode.

Memory protection is tightly coupled with protected modes of operation. The usual memory protection mechanisms, like segmentation and pagination, and its simplified forms used in embedded processors, which usually implement a software-managed Translation Lookaside Buffer (TLB), allow for distinct access rights to code being executed in system and in user mode. To improve the performance in guest mode, recent processors provide nested page tables, creating one more level of indirection in the address translation path inside the CPU: the addresses generated while in guest mode are translated using the page tables configured in guest mode, and the result of the translation is a guest linear address, which is then resolved using the host translation mechanisms. In this way the guest Operating System is free to modify its own page tables without assistance from the host Operating System. Instead, without the support for nested page tables, each time the guest OS needs to modify its page mappings, the host OS needs to be informed. Without this kind of hardware support, the hypervisor needs to use more expensive emulation techniques, where the host OS needs to track all the write-accesses of the guest OS to its own page tables.

2. Multi Cores and Multi Processors

Processor manufacturers have gone very far away from the traditional architecture of the processor, in which the chip used to contain a single Central Processing Unit (CPU). From the time when they started to integrate the cache memory elements into the chip for boosting performance, nowadays processors mimic more and more the System-on-a-Chip (SoC) paradigm typical of the

CC BY-ND

embedded domain, in which a multitude of heterogeneous hardware components are all integrated in the same chip.

As explained above, modern CPUs already possess a certain level of internal parallelism, with multiple ALUs, several pipeline stages of execution, vectorial instructions (e.g., with such SIMD extensions as MMX and SSE). Some Intel processors also parallelise the phase of instruction fetch, with the Hyper-Threading[4] technology, with which two fetching stages can independently feed a CPU pipeline in parallel emulating the presence of two CPUs. However, the performance of each hyper-threaded CPU depends on the workload imposed on the internal pipeline elements by both concurrent flows of execution.

Many modern processors embed multiple "cores" within the same chip - full-featured CPUs which are capable of executing independently from one another, hierarchies of cache memories, interrupt controllers, timers and other peripherals. Also, the complexity of the new chips and the growing requirements in terms of computational power (cf. [86]) led to novel paradigms of interconnection among the various hardware components within the chip (and within the system), mainly driven by the need for increasing the level of parallelism in communications. This led to an increasing trend in abandoning the traditional "interconnection bus" concept in favour of Network-on-a-Chip (NoC) interconnection solutions, in which hardware elements communicate with each other by means of packets (similar to how networking among systems used to do), whilst proper routing elements are responsible for delivering these packets to their destination.

Modern GPPs are implemented as homogeneous multi-core processors, including the Intel Core-line, the AMD Phenom-line and the IBM Power series. Not only are these GPPs homogeneous, the cores within these CPUs are also symmetric, in the sense that they have the same single-threaded performance. The above architectures fall under the multi-core paradigm, i.e., parallel "fat" serial cores, that are effectively independent computing units with communication capabilities. There are also architectures that follow the *many-core* paradigm. In a many-core architecture the processing units are basically small, highly coupled cores where a workload is typically distributed over many of these cores. Such architectures can be found for example in graphics processing units (GPUs).

The same architectures are also used for General-Purpose Graphics Processing Units (GP-GPUs), in which all the vector cores are used for general computations, instead of being merely capable of performing 3D rendering operations. Some GPU architectures, such as the Nvidia GT200, have even double-precision floating-point operations within the ALUs of their vector cores [190], especially for scientific computations (3D rendering requires only single-precision floating-point operations) [191][192][214]. Some GPUs are available as special boards especially made for GP-GPU processing, as these boards do not even have a video output [215].

Switching to heterogeneous systems, the IBM CellBE is an example of an heterogeneous multi-core GPP deployed for HPC purposes. The Cell includes a single PowerPC core (GPP) and 8 Vector cores, called Synergistic Processing Elements (SPEs). The lacking success of the Cell could be taken as an indicator against the validity of heterogeneous multi-core architectures. However, the major issues relate to the fact that the processor has high manufacturing costs and it is difficult to code when using current programming models. This currently

4 More information is available at: http://www.intel.com/technology/platform-technology/hyper-threading/index.htm.

CC BY-ND

makes the Cell mostly of interest for specific application areas (note that the Roadrunner by IBM hosts a combination of Opteron and Cell processors [160]).
Intel Core-line of processors use Point-to-Point links between the cores, where for example IBM latest Power processor uses a hybrid ring and crossbar. The IBM CellBE uses multiple rings to communicate between the SPEs and the GPP. As on-chip interconnects is such a broad topic it is covered in more detail in the next section (Section 3.).

3. On-Chip Interconnects

3.a) Scalability and Future Computing Technology

The current era of computing has reached petascale, i.e., 10^{15} operations per second, and the first approaches towards exascale (10^{18} operations per second) are already planned. To make this happen, we have to think about a new paradigm of computing that is not only in the direction of software parallelism but also in the one of hardware infrastructure parallelism.
Peta- and Exa- scale computing architectures are moving towards multiple cores per chip. This kind of chip architecture has to address two main aspects: computation and communication. As the core count increases, the interconnection of cores becomes a more challenging task. We can address the problem at three different levels:

- physical infrastructure;
- communication paradigm;
- software/OS level.

PHYSICAL INFRASTRUCTURE

A good interconnection mechanism determines the efficient execution of an application or set of applications, based on the required computation and communication needs. In single core processors, most components are connected by a hybrid bus topology. But, as the core count increases, one has to think about not only the interconnection of the components inside a single core, but also the interconnection of multiple cores. So an efficient physical topology for multi-core architectures is needed. Up to now, designs are restricted to two-dimensional (2D) topologies, which are best for small-scale systems, but recent advancements in three-dimensional (3D) integrated technology [222][223] will open a new paradigm for design (see also [86]). There are various interconnection mechanisms studies and/or proposals for multiple-cores architectures. Every mechanism has its own advantages and disadvantages. The major performance metrics are: topology regularity, network diameter[5], bisection bandwidth[6], and power consumption.
In Table 1 we report a summary of commonly used topologies, along with their main characteristics, in terms of the just mentioned metrics, as a function of the number of interconnected elements [235].

5 The diameter of network is defined as the maximum (over all node pairs) length of the shortest path between any pair of nodes.
6 Divide a network of N nodes into two sub-networks of N/2 nodes. Then, the bisection bandwidth is the minimum possible bandwidth between these two groups.

BY-ND

Type of Topology	Number of nodes	Degree	Diameter	Bisection bandwidth
Bus	k	k	2	1
nD Mesh	k^n	2n	n(k-1)	k^{n-1}
Ring	k	2	k/2	2
nD Torus	k^n	2n	nk/2	$2k^{n-1}$
nD Hypercube	2^n	2n	n	2^{n-1}
Crossbar	$k^2 + 2k$	4	2 (k+1)	k
Binary tree	$2^k - 1$	3	2 (k-1)	1
Fat tree	Depends on actual structure			
Butterfly	$(k + 1)\ 2^k$	4	2k	2^k

Table 1: Main characteristics of commonly used topologies.

There are various other hybrid interconnection mechanisms proposed in the literature, like Augmented Hypercube Torus (AHT) [216], Torus Fusion (Tofu) [217], Flattened Butterfly Topology [218], Express Cube [219], etc.

COMMUNICATION PARADIGM

The communication paradigm specifies how cores communicate with each other. Here, major challenges are flow control and routing:

Flow-control

There are two possible levels at which flow control is realised: switch-to-switch and end-to-end. Flow-control at the switch-to-switch level mainly depends on switching and buffer management techniques.

Switching techniques are important because they specify when a routing decision has to be made. In a network-based system, there are two types of switching techniques: packet switching and circuit switching. Given that major constraints in silicon are area and power, it is advised to use as few buffers as possible at the network switch and router levels.

Circuit switching is suitable for communications with QoS guarantees because the required link resources are reserved for the entire lifetime of a particular communication and no arbitration is needed [220]. So, it provides very fast communication when needed by the application, but it limits the possibilities of time-sharing of the links across multiple communications. Circuit switching can be combined with Time-Division Multiple Access (TDMA) to provide more sharing connections. Nostrum [225][226] and Æthereal [224] are examples of this scheme.

Packet switching techniques fall into four main categories:

- *Store & Forward* – incoming packets are stored first, then they are forwarded to next hop.
- *Wormhole switching* – each packet is divided into *flits* (small chunks), which are sent one by one to the intermediate router; the router stores and processes the header flit first, then it forwards it to the next hop and the other flits just follow the path taken by the header flit.
- *Virtual-Cut-Through* – *each* packet is divided into flits and sent one by one to the intermediate router; the router stores all the flits in a buffer, then, after the routing decision, forwards them to the next hop.

CC BY-ND

- *Virtual circuits/channels* – it is similar to circuit switching, in the sense that the connection is established before any packets are transferred, and that packets are delivered in order.

Each one of the above schemes has advantages and disadvantages. For example, Store & Forward and Virtual-Cut-Through need the routers to have a buffer-size sufficient for storing an entire packet, while Wormhole switching does not. Finally, in virtual circuits/channels the buffer size requirements are also high.

Among all the packet switching techniques, Wormhole switching seems to be the most suitable for fast computing, because of its lower latency and limited buffer requirements. For QoS-sensitive systems, we can combine virtual circuit packet switching with TDMA.

The performance of a switching technique also depends on channel buffer management between routers to avoid channel buffer overflow. There are two major techniques: Go-and-Stop (or on/off) control and Credit-based control.

In *Go-and-Stop control*, the receiver router sends a *stop* signal to the sender router if the buffer is full. When the available buffer size rises again above a preset threshold and/or the size of a packet, then the receiver router sends a *go* signal to the sender router.

In C*redit-based control, the* receiver router sends *credits* (indicating the available receive buffer size) to the sender router when some buffer is available. Then, the sender router sends packets up to the available *credits* (buffer size).

In addition to the switch-to-switch level, we can control the flow between the source router and the destination router by adjusting the injection traffic rate on and end-to-end basis. There are two popular techniques: Injection Limitation and ACK/NACK flow control.

Injection Limitation – in this case, the sending router suppresses the traffic rate to saturation load, which can be statically or dynamically defined.

ACK/NACK flow control – as soon as a packet is received by the destination router, it is explicitly acknowledged by a ACK packet (or NACK, in case of problems).

Routing algorithm

The routing problem becomes more and more relevant as the number of cores increases. The goal of a routing algorithm is to distribute traffic evenly among the paths supplied by the network topology, so as to avoid deadlock and minimize contention, thus improving network latency and throughput.

Based on the application requirement, routing paths may be adaptive or deterministic, and multi-path or single-path. There are various algorithms which have been proposed, but the most commonly used ones are Dimension-Ordered-Routing (DOR) and Turn-Model routing.

In *DOR*, by looking at the distance vector between the sender and the destination, a message is forwarded first in the lower dimension, then in the other one. DOR uniformly distributes minimal paths between all pair of nodes [229].

In *Turn-Model routing*, the packet has two choices in each router: proceed on the same direction or turn in the other direction, based on the availability of the path or link [230][231].

Quality-of-Service

Quality-of-Service guarantees in communication networks, i.e., predictable and ensured delivery latency and throughput, constitutes also an important issue.

CC BY-ND

Interconnect networks are constrained by on-chip bandwidth, or the number of available communication channels over the link. These are allocated based on traffic requirements (latency critical, data stream, best effort). Therefore, there is a need for proper network resources allocation and scheduling algorithms in order to support QoS.
As discussed for the switching techniques, circuit switching provides the best solution for resources allocation or QoS [220] and virtual circuit/channel packet switching with TDMA also provides a partial solution for this [221]. But, in both cases, resources are allocated to either short time slots (TDMA based) or during the entire lifetime of a connection (circuit switching). But these are not optimized for worst-case situations. Therefore, several dynamic resources allocation schemes have been proposed in the literature [234].

SOFTWARE/OS LEVEL
The goal of this level is to define the access method to the hardware platform by the applications. This can be done based on computation and communication requirements, which may be automatically inferred by the OS by means of proper heuristics, like the ones commonly found in GPOSes for discriminating between batch and interactive processes, or more advanced ones [274][275]. Also, applications may explicitly communicate their resource requirements by using proper APIs [81]. Anyway, the OS and/or application has to deal with application mapping, communication and task scheduling.
The goal of application mapping is to map a set of processor-cores to a specified application in an optimised manner (software-hardware mapping). In multi-core systems, the hardware topology is pre-configured, so we will have to map and schedule applications and their communications on the available hardware resources. Given some knowledge on the application components and their computing and communication requirements, we end up with a graph mapping problem, where the application-level graph needs to be properly mapped to the available physical level topology. These approaches may be based on an a-priori characterisation of the application requirements and an off-line optimisation process [227][228]. However, this is not appropriate for dynamic real-time applications with strongly varying workloads. For this class of applications, online dynamic mapping is more appropriate, where the cores to applications mapping may be dynamically varied depending on the actually imposed workload on the various subsystems [232][233].
For communication and task scheduling, based on available communication infrastructure and communication requirement it is possible to use various scheduling schemes (see Section IV. 2. for details).

4. High-Performance Computing

High Performance Computing (HPC) is always considered a “role model” for many-core systems due to the essential similarity of the architecture and hence the means to develop programs for it. At the same time, the many-core movement leads to a sudden boost of scale in high performance computing without essentially increasing the requirements or costs. Whilst the degree of mutual influence between many-core PCs and HPC will be discussed elsewhere [86], this chapter focuses in particular on providing an overview over the past and present approaches in high performance computing, including programming models to realise distributed and parallel applications.

CC BY-ND

Since the topic is vast in itself, we restrict ourselves to the main issues of relevance for the S(o)OS project. We thereby assume the reader already has basic knowledge in the area, and specifically that he/she is familiar with the concepts found in [157][158][159].

High Performance Computing is often confused with Grid and Cloud computing due to the fact that all these areas investigate into *distributed* computing. However, the degree of connectivity and the type of scale differs enormously between these three systems:

Cloud Computing has the weakest relationship with high performance computing, as its main focus is on ensuring accessibility of data and/or services hosted within the cloud infrastructure by scaling these instances. Implicitly, scale in Cloud systems must be regarded as "horizontal" where instances of data and code are replicated to increase availability [95]. Whilst this is particularly useful for data and service hosts, such as eBay, Amazon, etc., it does have little or no impact on scalable systems acting on "vertical" scale (i.e., where the actual application is split into multiple processes or threads that *jointly* contribute to the application functionality). Recently there have been more and more references to the so-called "HPC Clouds" [96][97][98], yet they either focus on making small parallel machines (in the order of 8 cores in the case of the Amazon EC2[7]) available, or provide access to thousands of cores in an almost unconnected fashion, similar to Compute Grids (see below), as for example in the case of Plura Processing[8].

Grids or *"the Grid"* [99] generally refer to a set of machines that can be used either as compute units (Compute Grids [100][101]), or to access given services (Virtual Organisations [102][103] or eBusiness Grids [104]). Grids primarily provide unlinked or weakly linked capabilities that can be used for embarrassingly parallel tasks, but not for typical HPC, strongly linked parallel execution. Usage of the Grid for such tasks is closely related to peer-to-peer computing, such as Folding@home[9]. HPC Grids on the other hand do expose access to supercomputers and manage job-based scheduling across the HPC units in the grid – the grid itself does *not* realise the computing capabilities. Notably, Grids can theoretically scale vertically and horizontally, depending on usage, but all communication is Internet based and accordingly slow (weak linkage).

Supercomputers are "computers that are at the frontline of current processing capacity, particularly speed of calculation"[10]. However, a supercomputer is not necessarily a *parallel* machine, but any machine that offers enough computational power. Typically a so high computation power is realised nowadays by linking multiple compute units with high-bandwidth, low-latency connections – we will use the term in this sense in the context of this document.

4.A) Supercomputer Architectures

In supercomputers, ***communication*** is key to achieve high performance [105][106][107]. This does not only relate to the network connectivity between nodes, but also to communication on all levels and between any kind of resource in the system (between cores, between processors, towards the cache, any I/O, etc.). Current petaflop machines have more than 100.000 compute cores in a full cluster of over 10-20 units that consist each of approximately 15 racks or towers.

7 Amazon Elastic Cloud Computing - http://aws.amazon.com/ec2/
8 Plura Processing, LP; A Creeris Company - http://www.pluraprocessing.com/
9 More information is available at: http://folding.stanford.edu/.
10 More information is available at: http://en.wikipedia.org/wiki/Supercomputer.

BY-ND

Each rack or tower hosts roughly 12 nodes, which again may have any number of CPUs between 6 and 16, which in turn currently may have 2-4 cores (general purpose) or 8 vector pipelines. It is important to highlight that there is a strong hierarchical structure of the architecture which implies different performance of the interconnects. The fastest connection is realised at the level of core-to-cache and core-to-core, with bandwidths in the order of 19,2 GB/s and latencies of 5.6 ns. The slowest one is between units, e.g., by the use of Infiniband with a bandwidth of 12GB/s and a latency of nearly 1 ms[11].

As a consequence, one of the major challenges for software development consists in distributing and segmenting the code and data in a fashion that any access beyond the immediate cache is reduced to a minimum, and that tightly interacting components are not deployed too far away from each other, in the underlying physical topology. This implies that the code structure needs to match as much as possible with the system architecture. However, hardly any programming language, nor any hardware mechanism does support this specific feature, thus typically all communication is treated equally on the programming level and the slowest connectivity will decide over the overall latencies.

Communication and synchronisation is currently still supported from the hardware side by means of a (virtually) shared memory, i.e., ***hardware coherency support*** (e.g., by using the Quick Path Interconnect protocol by Intel, see below). Coherency thereby means that all processes (or threads) have access to the same memory content – as we shall see, this does not imply that all memories have the same state. The simplest approach to shared data consists in actually physically sharing the memory or cache across multiple (all) compute units in a parallel machine. While such a uniform memory access (UMA) between the nodes and clusters would be ideal for programming purposes (leaving the latency issue aside), it is far from realistic. In fact, in order to reduce costs with the increasing need for scalability, it is more feasible to distribute memory so that each compute unit hosts its own cache and/or local memory. Therefore, the reality is that there is a non-uniform memory access (NUMA) between processors, nodes and clusters. This means that data (and processing) will have to be distributed according to the locality of the computation, otherwise the processor would mostly be waiting for its data.

Coherency in these cases can be achieved in two ways: either by allowing access to remote cache and memory blocks (“NUMA: Non-Uniform Memory Architecture”), or by actually maintaining the same state in all memory instances (“ccNUMA: cache-coherent NUMA”). Both approaches come at the cost of increased data access latency where the slowest memory connectivity restricts the overall speed [108][109][111].

In large scale clusters it is therefore more typical to have different types of shared memories on different hierarchical levels of the system, e.g., between processors on a single node, but not across nodes etc. [109][110].

Note that, with the advent of multi-core processors, NUMA has also become the reality within processors, where the access to a core local cache is 2-4 times faster than accessing the cache of a neighbouring core. In systems with No-Remote-Memory-Access (NoRMA), memories belonging to other nodes or clusters can not be accessed directly at all. In this case, the software (application or Operating System) has to ensure consistency of data across processes or threads. This obviously increases complexity for the developer, because such

11 All figures depend on usage and configuration.

CC BY-ND

devices need to be programmed using some type of message-passing protocol, such as MPI[12].

Cache organisation determines the data throughput of the processor and in particular modern vector machines, such as the NEC SX-8, used a proprietary memory bank structure to reduce access latency (odd-strides and stride 2 being best for access) which is not identical to the one in the NEX SX-9 (where stride-1 is best) this obviously requires more adaptations on the programmer's side [111]. So far, most cluster machines are homogeneous in nature, i.e., the processors incorporate identical cores and all processors across the system are identical. However, modern day systems, such as the IBM Road Runner, show a clear trend towards ***heterogeneity***, incorporating graphic cards on a node level, as well as heterogeneous cores on a processor level. In the HPC domain [113], the organisation clearly reflects the typical usage area, so that the processor and cache layout is close to the algorithmic maps [112][114] – a circumstance that can not be employed in desktop PCs as they try to address a large usage scope.

5. Memory Architectures

High-Performance GPPs use multiple levels of cache(s) to hide the memory latency of the main memory. As cache misses cause a core to stall (5-6 cycles for L1 cache miss, 50+ for L3 cache miss) for a significant part of its operation, cache misses are the biggest performance problem for CPUs. So the main problem for distributed applications is to find the right segmentation of code and data to minimise cache misses.

The first level of cache (L1 cache) is always private per core, L2 caches and higher may however be shared among cores in a processor. The highest-level cache (e.g. L3 in Intel Nehalem) is the cache that sits between the processor and the RAM memory – sometimes the RAM is therefore referred to as the highest-level cache. Typically the higher-level cache is also slower in being accessed due to manufacturing reasons, with the RAM always being the slowest memory (note that modern hierarchical views also consider hard drives and other storage devices as higher order memories [109]). In addition to being slower, access to higher-level cache is also delayed due to the hierarchical organisation of the cache: all access to higher levels are routed via all their lower levels (hence the “hierarchy” of the structure). For example, a L2 cache access has to first go through the L1 cache before being handed over to the L2. This means that the access is delayed further by a factor proportional to the level of the cache being accessed [116].

Some architectures (mostly DSPs) employ some type of Harvard memory architecture, where there is no cache at all, but local memories for both code and data.

The simplest approach to programming parallel applications consists in a shared-memory approach, where each process can easily access data and state across all other processes (see also section V. on programming models). Due to the organisation of multi-core processors, the only effectively shared memory is the RAM, which is at the same time the slowest. In most common desktop applications, shared memory is not even required (as the code is strictly sequential). In order to achieve a shared-memory like behaviour, the caches of different cores need to be kept coherent. Cache-coherency for most multi-core GPPs are implemented in HW in the form of a protocol (as opposed to physically sharing the memory). Cache *coherency* thereby needs to be distinguished from

12 More information is available at: http://www.mpi-forum.org/.

CC BY-ND

shared memory insofar as it does not exactly allow for accessing a remote cache, but ensures that basically all (coherent) caches have the same state. Remote access could lead to deadlocks if no specific care is taken to avoid that, which again may lead to major delays. Intel for example employs the MESIF protocol on top of the "Quick Path Interconnect" (QPI) bus since its Intel Core i7-9xx processor (2008). This protocol does not so much aim at constant consistency across all caches, but it verifies the consistency of a specific memory location at access time. The basic essence of the protocol consists in asking all other cores whether the information is still up-to-date or has been invalidated by any other core, in which case the datum will be overwritten prior to being made available to the querying process. Not only does this require additional steps (3-4 hops, depending on the situation) which delay the memory access even further, it also diminishes the potentially available cache *if* it were shared between the consistent cores. Nonetheless, the QPI approach is still slightly faster than other coherency protocols [117]. In order to reduce the delays, such protocols base on the assumption that the cache is fully interconnected (so as to avoid further delays by routing steps). With the growing scale of multi-core processors, this assumption is however not feasible in the long run (see D5.2 [86] for details).

RAM is typically only accessible via a dedicated core (there is only one memory controller), however, newer architectures have a memory controller per core. To access other devices, Direct Memory Access (DMA) is used, where the DMA controller transfers data between the RAM and the device. The CellBE also uses DMA to transfer data between the main memory and the local memories of each SPE.

DSPs usually employ some type of Harvard memory architecture, where the local memory falls within a different address range than the larger main memory. The SPEs on the Cell on the other hand only have direct access to their local memory, and data from the main memory can only be acquired through DMA transfers.

6. Summary

We can categorize the (current) computing hardware architectures as follows: multicore on a single chip or on computing card with multicore chip, multi socket or multiple chip on single board and network of systems or super computer. Table 2 summarises general-purpose in-chip multi-core and many-core systems, a.k.a., Chip Multiprocessor (CMP). This table shows the comparison between some of the recent CMP systems with processor properties like architectural, memory and power consumption. We can conclude that industry is going towards the direction of multi-core tile based systems, currently following 2D topologies but in future we may find 3D based chip multi-core system also.

CC BY-ND

	Intel TeraFlop	Intel SCC	Sony, Toshiba, IBM cell	Sun UltraSPARC T2	Sun UltraSPARC T3	UT Austin TRIPS	Intel Larrabee	Cavium octeon cn6880	Tilera TILE64
Type	CPU	CPU	CPU	CPU	CPU	CPU	CPU/GPU hybrid	NPU	NPU
ISA	X86	x86	Power	SPARC	SPARC	EDGE	x86	MIPS	
Micro architecture	in-order VLIW		2 way In-order CPU, RISC, 2 way In-order 128-B SIMD SPU	RICS, SIMD	RICS, SIMD	SIMD	2 way In-order, 4 way SMT, 512-B SIMD	RISC, SIMD	RICS, 3-way VLIW
Technology	65nm	45nm	90nm	65nm	45nm	130nm IBM ASIC	32nm	90nm	90nm
Number of core	80	48	1 CPU(PPE) 8 SPU(SPE) SIMD	8	16	2	Upto 48	32 cnMIPS II core	64
Operating frequency	3.2GHz - 5.7GHz	-	3.2GHz - 4GHz	1.6GHz	1.67GHz	533MHz	2 GHz	1.5 GHz on each core	866MHz
Data width	32bit	32bit	128 bit	128 bit	64 bit	32 bit	64bit	64 bit	32 bit
Interconnect	8x10 2D mesh	2D Torus	Ring	Crossbar	Crossbar	4x10 2d mesh	Ring	crossbar	8x8 grid 2D mesh
Bi-Section Bw	320GB/s	256GB/s	195GB/s	-	-	64GB/s	-	-	240GB/s
Switching	Wormhole	Wormhole	Pipe-lined Circuit	Circuit switching	Circuit switching	Wormhole	-	-	Wormhole
Flow-control	On/Off	Stall/Go	Credit-based	Handshaking	Credit-based	Credit-based	-	-	Credit-based
Routing	Sourcing Routing (DOR)	DOR , odd-even Turn Model	Shortest-path	Board level routing	Board level routing	DOR	Ring based routing	-	DOR
Number of Threads	-	-	2 on PPE 8 on SPE	64	8 per core (8 x 16)	4	4-way interleaved multi-threading /core	-	-
L1 cache	3KB I-cache, 2KB D-cache	16KB I-cache, 16KB D-cache	32KB I-cache, 32KB D-cache	16KB I-cache, 8 KB D-cache	32KB I-cache, 16 KB D-cache	64KB I-cache, 32KB D-cache	32KB I-cache, 32KB D-cache	37KB I-cache, 32KB D-cache	8KB I-cache, 8KB D-cache
L2 cache		256kb	512kb	4MB	6MB	1MB	4MB	4MB shared	64KB
L3 cache/ other cache		16kb massage passing buffer (shared), total 384kb	256kb scratchpad memory (Local Store)	-	-	-	-	-	Abstractly all on chip cache
Memory model	Message-passing	Message-passing	Shared-memory	Shared-memory	Shared-memory	Shared-memory	Shared-memory	-	shared memory and user-level message passing
Cache coherence UMA/NUMA	Directory based cache coherent shared-memory	Software based cache coherent shared-memory	NONE	MESI SNUCA	2 coherence plane	No hardware based, Non-Uniform (NUCA) L2 Cache	Broadcast	-	-
Development environment	-	-	IBM Cell Broadband Engine SDK C/C++ compiler	-	-	TRIPS compiler based on C and FORTRAN	C/C++ Compiler	OCTEON (CDK) C compiler	Tilera MDE Eclipse based, c compiler
TLB	-	-	-	-	-	16 I / 16 D	-	64 I / 64 D	8 I / 16 D
Power	-	25W - 125W	92W	72 – 123 W	-	36W	-	95W	15-20W

Table 2: Summary of computing hardware architectures.

Table 3 summarises multiprocessor systems based on Card Computing Units, where the multiple computing units are seen by the OS similar to an expansion card (or peripheral).

	NVIDIA Tesla C2050/C2070	ATI Radeon HD 5970	Cisco QuantumFlow	ClearSpeed CSX700
Type	GP-GPU card computing unit	GP-GPU card computing unit	NPU	Numerical Computation Unit
Process Technology	40nm	40 nm	90nm	IBM 90nm
Number of core	512 CUDA core or Stream processors (16 Stream multi-processors and 32 core per Stream multi-processors)	3200 Stream Processing Units (160 SIMD unit & 20 Stream Processing Units per SMID unit)	40 4-way-threaded (160 total) CPU cores called Packet Processing Engines (PPE) 32-bit RISC core	1 core (96 Processing Element), RISC, Array SIMD per chip
Micro architecture	CUDA	TeraScale 2 Unified Processing Architecture	-	
Operating frequency	1.25GHz to 1.4GHz	725 MHz	900MHz	250MHz
Double-Precision Operations per seconds (Gflops)	515	515	-	96
Single-Precision Operations per seconds (Gflops)	1030	4620	-	96
Interconnect	Infiband	-	-	ClearConnect (Hybrid)
L1 cache	64 KB shared per SM	-	-	8KB I-cache, 4KB D-cache
L2 cache	768KB shared	-	-	2x 128kb shared
Memory	3GB/6GB	2GB	Shared memory - 1 Gb of memory to support 128,000 queues and 40 Mb of content-addressable memory (TCAM)	
Memory clock speed	1.5GHz	4.0 GHz		
Memory bandwidth	144Gbps	256.0 Gbps	100Gbps	
Memory bus width	384 bit	512 bit		
Memory Interface	GDDR5	GDDR5		DDR2
Memory model	Shared memory	-		Shared-memory
Cache coherence				
Development environment	NVIDIA nexes IDE	ATI Stream SDK		ClearSpeed SDK Eclipse based
Compiler support	C, C++, FORTRAN, OpenCL, JAVA, Pyton	OpenCL		C
Address Space	64 Bit			64-bit Virtual 48-bit physical
QoS			Priority based	
Power	190W to 225W	51W to 294W		9W
Comment	Concurrent Kernel Execution			

Table 3: Summary of multiprocessor systems based on Card Computing Units.

III. Protocols: Communication Control Models

This chapter overviews research literature in the field of communications and control models for distributed, massively parallel and highly scalable systems. In the context of the S(o)OS project, we plan to leverage general concepts from the world of distributed computing, like scalable mechanisms for load distribution and balancing, services localisation by means of publishing/subscribing mechanisms, etc., in the context of massively parallel systems, even within the chip. However, the necessary differences need to be properly accounted for, because the mechanisms used for communications within the chip, and among different chips on the same system, and the implied latency and throughput figures, are tremendously different as compared to standard networking stacks.

1. Interconnection Networks

In standard networks two of the core aspects constantly under improvement are bandwidth and latency. Current research is focused on the improvement of these aspects by enhancing the underlying technology or by making protocols more efficient, thus decreasing overhead and maximising utility. Even so, bandwidth improves much faster than latency [32], frequently faster than a system can cope with, especially in IP, fibre based, transport networks. In these networks, traffic demand increases very rapidly, doubling every two years (see Figure 2).

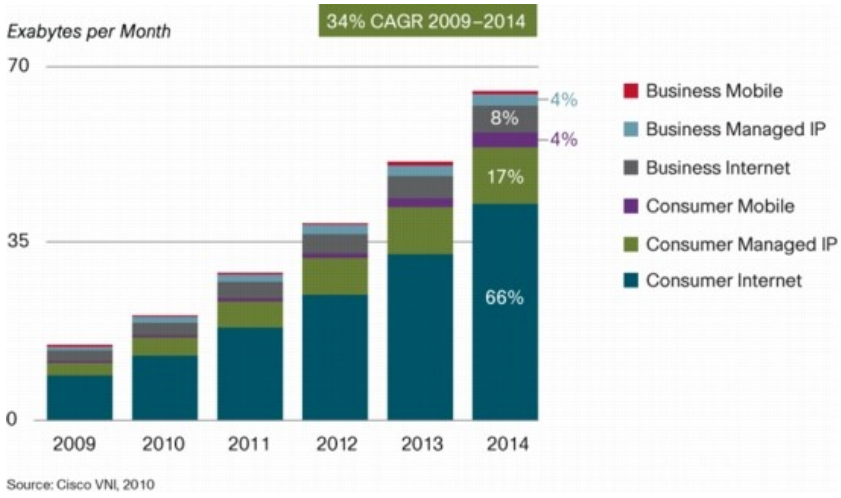

Figure 2: Cisco VNI Forecasts 64 Exabytes per Month of IP Traffic in 2014

Current state of the art optical communication devices transmit at a rate of 100 Pb/s/km using a DWDM (Dense Wavelength Division Multiplexing) deployment of 155 lambdas at 100 Gbs each [29] and over 7000 km. Still, this number only shows up to where technology can be pushed, and it does not refer to real deployments (at least not for transport or HPC networks), because bandwidth between hosts is limited by protocols and controlling electronic equipment.

Most high speed interconnects are based either on copper or optical technology, using point to point connections. The fastest deployments reserved for research purposes as the Internet2, GEANT or CERN. Optical technology was elected for long distance interconnects due to factors such as its low attenuation and distortion, high capacity, low price and raw material availability (now fibre deployment is more cost effective

than copper [30]), and the immunity to electromagnetic interference (or TEMPEST style eavesdropping). Future prospects plan to bring optical technologies into chips, highly increasing integration as well as power efficiency and performance. Considering that the limits of optical technologies are still not reached, we expect bandwidth to increase continuously, while trying to follow demand. Still, for tightly coupled deployments such as HPC environments, copper technologies such as the ones used by Infiniband or Ethernet are pretty much standard.

A more important factor for tightly coupled distributed systems, such as parallel applications, is latency [31]. The higher the latency, the more time is wasted sitting idle, thus hurting overall performance and efficiency in resource usage. With the increase in bandwidth, latency has decreased over time, however network latency is still very far from the latency found between system components. Moreover, the increase in bandwidth requirements is flanked by the need for lower latencies, as systems have synchronization points more frequently. Increasing the bandwidth may not be enough for some classes of applications, but unfortunately some systems with higher bandwidth may present also higher latency figures [31]. Still, considering the overall relation stating that a bandwidth improvement in a technology, generally implies a latency reduction (e.g. Ethernet), after systems start operating at the same speed magnitude as the bandwidth provided by their interconnection network, the latency issue returns, and over time, latency is lagging [32]. This is particularly relevant in the HPC environment where even the low latency of less than 1.5 us provided by Infiniband [37] is a point where improvements are much desired. The only way to actually minimize latency is by optimising algorithms and overall systems. Infiniband allows up to 1.5 us due to high coupling of the communication stack, and strong hardware support. More importantly, Infiniband designers opted for solutions which effectively minimise latency independently of the provided bandwidth, such as cut-through switching, switch fabrics, or serial links.

In more conventional networks, such as Ethernet or DSL, the major aspects are bandwidth, and simplicity, or Operations Administration and Maintenance (OAM) functionalities. In particular it is commonly agreed between authors that the lack of OAM functionality is a major limitation for Ethernet [38]. Latency, while being an existing issue, is not a critical problem as applications are designed to take into consideration high delay between end points.

In order to reduce latency and improve useful bandwidth (at a reasonable cost, depending on requirements and purpose), several different protocols are used, following a layered approach. Between cores, state of the art systems use HyperPath or QuickPath Interconnect [34], inter-processor communications use IXS (Internode Crossbar Switches), in HPC environments Infiniband [37] provides connectivity between nodes, and Ethernet typically connects hosts. Over the Internet, IP is the dominant protocol, and it is used over a multitude of technologies.

Beside the potential capabilities of each technology, it is also relevant *how* systems are interconnected (i.e., the communication topology): traditional home networks make use of either the Bus (such as in 802.11 or 802.3 10Base2) or Star (such as in Ethernet XBaseT) configurations, however these are not optimal for high performance or low latency systems. Instead they favour cost and ease of deployment. One particularly bad example of system configuration, in relation to latency, is the ring model where each node connects to two of their neighbours, forming a circle. Message flow between two nodes must traverse all intermediate nodes, making latency critical communication only possible between direct neighbours. One benefit of this topology is that, if the detailed topology is known, latency (although high) between two hosts can be

predicted. Data-centers for Grids, Clouds and HPC are currently driven by inter-networking topologies much different from their counterparts used in Local Area Networks. In this world, designs maximising bandwidth while keeping latency at its lowest value are desired, and we find Hypercube, N dimensional Mesh, Torus, Dragonfly [33]. Each topology has its advantages, in terms of latency, bandwidth, cost and predictability. The Flattened Butterfly [39] improves the existing topologies, by proposing the use of high-radix routers to reduce network diameter, while improving throughput in 50% for multi-core designs. Moreover, the authors show that routing can have negligible impact in latency, and the design can actually reduce it by 28%.

Communication between software or hardware agents must take into consideration the actual topology, so that parallelism provides the highest possible performance. Several authors already identified the relevancy of this aspect, showing the impact of cross layer adaptation in order to maximise performance [35]. In the context of S(o)OS, we must follow the same approach and make code distribution and parallel execution aware of the underlying topology. In the case that topology information is not directly available, the same can be inferred by quantifying the latency of the multiple links, as it is shown in [36].

2. Communication Protocols (SW Layer)

At the software layer, protocols are much more complex than the ones addressed in the previous section such as Infiniband or Ethernet. This arises from the fact that they are closer to application and business model concepts, providing a much richer set of functionality. They are mostly driven by the TCP/IP or OSI protocol stacks. These approaches consider that protocols are in layers, where each layer makes use of the functionality provided at the lower layers. Typically no two layers provide the same functionality, but because of the multiple possible combination of layers, redundancy and replication of functionality may occur.

Applications reside at the upper layers and exchange data units which differ from application to application. Initially most applications were designed on top of the UDP or TCP layers (or even IP!), but nowadays several common layers, such as HTTP and SIP [41] are shared by many applications. IPv4 is the common protocol driving today and future networks as all services are provided on top of it, with IPv6 following it and allowing for the major deployments required for the Internet of Things. However, as some authors recently suggest [47], transition to IPv6 may not be so soft due to the way operating systems are implemented, and the actual implementation of the protocol may result in lower throughput and higher resource consumption for IPv6.

Many applications, in particular time-sensitive ones, need to have a notion of the available latency and bandwidth in order to deliver services to users and are designed in order to cope with this. Conversely, the further down the protocol layer, the more the protocols are specialised to the characteristics of the local communication environment. Moreover, many applications require to have a detailed knowledge of the underlying transport path and adapt content accordingly [40]. Vertical interoperability is however not easy as the protocol stack relies on a closed layered model, not allowing much access besides to the neighbour layers, and cross-layer optimizations requires particular care. Mostly because some short-cuts can change the expected behaviour of a protocol and break compliance. This makes high level integration of devices for parallel execution more unrealistic, in particular under synchronization critical conditions.

Another protocol highly relevant for a distributed operating systems is 9P. As the name suggests, 9P was created for Plan9, the prototype operating system developed by Bell

labs. Started in 1985, it suffered several refinements with 9P2000 [46] being the latest version. The innovation of 9P consists in a protocol oriented towards a distributed operating system, which allows to distribute filesystems, devices and applications in a grid, however in a transparent manner. That way, resources are used in a way independent of their actual location. It supports locking, client-side caching, secure authentication and even internationalisation. The Inferno Operating System [49] (developed by Via Nuova Holdings) continued research in the 9P protocol, but extending it to more useful scenarios. One important aspect was the support for heterogeneous hardware platforms, through the use of a virtualisation technology (DIS) and a type-safe, C alike language named Limbo. Recent work around this protocol is present in XCPU2, proposed by Lonkov et al. [50]. The authors propose a system allowing users to compose the environment of remote cluster nodes so as to match local resources, creating the illusion that the cluster is a seamless extension of the local machine, much facilitating development and debugging.

SERVICE DISCOVERY PROTOCOLS

Service discovery protocols (SDPs) consist of two groups: protocols that focus on services (such as AppleTalk), and the ones which concentrate on devices (such as DHCP). The activities in SDPs include three main tasks: how to find services; how to publish services; how to use services. Generally UDDI, WSDL, SOAP [51] have been used to implement the mentioned activities in various SDPs. The mechanism of service discovery and advertisement constitutes distributed services or centralized services, AppleTalk, NetBIOS/SMB [52], IPX that were designed following local network (not IP based) features and decentralized discovery (Distributed Service Discovery). UPNP [53] [54] and SLP [55] present distributed solutions to operate in a local scope as well as searching for services in a global scope. This is achieved by exploiting extensions of existing standards (DNS) that enable remote service discovery. The traditional SDPs have weakness in supporting a simple and reliable way to configure and browse services over IP networks. However, to overcome these challenges the next generations of distributed service discovery solutions (zeroconfig SDPs) [56] have been emerged recently. The main advantage of zeroconfig SDPs is their adaptation to various heterogeneous network structures. IPv4 Link-Local Addressing, Multicast DNS, Bonjour [57] and DNS Service Discovery [58] are examples of zeroconfig SDPs. Bonjour is an open source distributed service discovery which is submitted by Apple to IETF as a standard. It enables automatic IP configuration, translating name to IP and service discovery without using DHCP server, DNS server and Directory.

3. Communication Interfaces (API)

There is a close relationship between programming models, tools and the protocol stack. Applications communicate using Berkeley Sockets alike interfaces which vary according to the tools, programming model and environment. These interfaces are exposed to applications by means of libraries which are agnostic of the underlying infrastructure and connect end-points using either UDP or TCP connections (here we focus on distributed systems – see Chapter V. later for programming models for parallel multi-processor and multi-core systems). One underlying limitation of this API is that connections are maintained between IP addresses. In reality what matters to communication is not the actual location but the identity of the hosts. Identifier to locator decoupling is an aspect pursued since a long time such as using HIP [42], but its support is still lacking in current operating systems. Interestingly in the work developed by Khurri et al [48], it is shown that HIP (which makes use of cryptographic functions)

can be successfully implemented in highly constrained devices such as Symbian OS cell phones. Decoupling allows communication to be performed between entities independently of where they are located, as well as IP mobility and session transference, while maintaining active sessions. One way which could be argued to achieve this is by using DNS, however DNS latency is in the order of hours to days, making it unsuitable for today's scenarios.

On top of the socket API, several other APIs provide additional functionality, as well as richer interfaces – for example, related to multi-core and parallel designs, the Message Passing Interface (MPI. The first version (1.0) was released in 1994 rapidly becoming a standard, with version 2.2 released in September 2009 [43]. Research for parallel systems also adopted MPI and is now focused on improving it [44][45]. This specification provides miscellaneous methods such as point-to-point message passing, collection or group communication, process topology and many others. Its specification was created as being language agnostic and has bindings to several languages such as C, C++ or Fortran-95. Many of the basic design principles of MPI can be used in a distributed operating system such as message passing, avoiding memory-to-memory copies, reliable communication between hosts, offload processing to dedicated co-processors (in particular communication), and more importantly platform and hardware agnosticism.

The communication API provided by modern operating systems support both blocking and non-blocking mechanisms. In the latter we have non-blocking synchronous or asynchronous I/O. A blocking interface means that it does not return until the operation is completed, thus preventing a caller to perform other activities. Alternatively, additional threads could be created, but its a limited and expensive resource. By contrast, non-blocking I/O interfaces return without blocking even if the operation is not completed, thus allowing for a more efficient use of threads. Currently, there are two design approaches to develop high-performance concurrent applications that are scalable[13], in particular server applications, using non-blocking I/O: the reactor pattern [59] and the proactor pattern [60]. This mechanisms rely on an event demultiplexer, an object that dispatches events from various sources to the appropriate handlers (or callbacks).

The reactor pattern involves non-blocking synchronous I/O. The event multiplexer waits for events that indicate when an object, such as a file or socket, is ready for a read or write operation. The demultiplexer dispatches the event to the appropriate handler that is responsible for performing the actual read or write. Then the handler can handle the completion of the operation and may renew interest in I/O events.

The proactor pattern, on the other hand, involves asynchronous I/O. A caller issues a read or write on an object and supplies a completion handler. The operating system handles the operation asynchronously and queues a completion event, which is then dispatched by the demultiplexer to the appropriate handler. The handler manages the completion of the operation and may start new asynchronous operations.

By comparison, the proactor pattern has some advantages over the reactor pattern:

- it is easier to write an application, since it relies on the operating system to fully handle the I/O;
- the operating system completion queue handles multiple threads and their concurrency, thus making more efficient use of threads;
- provides better portability, since it can be emulated efficiently on top of the reactor pattern.

13 More information is available at http://msdn.microsoft.com/en-us/magazine/cc302334.aspx and http://www.kegel.com/c10k.html.

Most UNIX like operating systems support[14] the reactor pattern, while support for the proactor pattern can be found on Windows NT[15] based operating systems and Solaris[16]. Libraries, such as ACE[17] and Boost.ASIO[18], provide a unified proactor API for a wide range of operating systems. The former library also provides a reactor API. On the other hand, Linux supports asynchronous I/O by means of an implementation[19] of the POSIX asynchronous I/O API[20].

14 POSIX *select()*, *poll()* and variants like *epoll()* and *kqueue()*.
15 More information is available at: http://msdn.microsoft.com/en-us/magazine/aa365198.aspx
16 More information is available at: http://developers.sun.com/solaris/articles/event_completion.html
17 More information is available at: http://www.cs.wustl.edu/~schmidt/ACE.html
18 More information is available at: http://think-async.com/Asio/
19 More information is available at: http://lwn.net/Articles/94566/.
20 More information at: http://www.gnu.org/software/libc/manual/html_node/Asynchronous-I_002fO.html.

IV. Operating Systems

This chapter overviews research literature in the area of Operating Systems models which is relevant for the S(o)OS Project. The works surveyed in this chapter constitute a snapshot of the current OS kernel architecture models, with a strong focus on the support for highly and massively parallel systems. These constitute the point of departure from which the research in the context of the S(o)OS project will be carried out.

1. Operating System Kernel Models

Various models of Operating Systems and kernels have been proposed in the literature. In what follows, an overview of the most significant works related to the S(o)OS project is provided.

1.a) Microkernels

Microkernels have been proposed in the research literature as an alternative to monolithic kernels [26][90][94]. In this architecture, a small code base provides essential services, such as physical memory management and protection, interrupt handling and task scheduling. This code (the microkernel) is the only one that executes with full privileges in the processor, whilst most of the classical services provided by a monolithic operating system, like network protocols, I/O device drivers, file systems, virtual memory management, are provided by special *server processes* running in user-space. In this way, the operating system is extremely modular, robust and secure. Only a small set of essential services is always present in the kernel, while all other services are optionally implemented as processes. In microkernel based OSes, all services interact with each other and with applications through message passing.

Also, the robustness and security of the OS is highly increased, because misbehaviour of a device driver, either malicious or accidental (due to common bugs), can only cause the malfunctioning of the corresponding OS subsystem and cannot cause the crash of the entire OS. Conversely, in a monolithic OS, every kernel-level device driver, developed in the hurry to support the latest (and widely sold) piece of hardware, may potentially crash the entire system.

Several microkernel architectures have been proposed in the research literature, most notably the Minix system by A. Tanenbaum Group[21] and the L4 family of microkernels [26]. However, such architectures did not have a great success in commercial OSes, mainly because of the performance penalties due to interprocess communication (IPC) mechanisms on single processor systems with memory protection.

It should be noted that most of the research on microkernels is focused on two aspects: security and use as hypervisor for running multiple virtual machines on the same hardware platform. Unfortunately, only a few researchers have investigated the performance and scalability of microkernels on multicore systems. In L4 [26], the basic IPC mechanism is designed to be highly optimised for uni-processors: the `send()` primitive hands off its time-slice to the server process without involving a call to the scheduler. In this way, the overhead of message-passing reduces to two contexts switches per IPC operation. However, such optimisations are not possible in multi-processor systems, because sending a message involves an inter-processor interrupt and a call to the scheduler, while the sender is forced to wait for the message reception

21 More information is available at: http://www.minix3.org

before continuing. Moreover, microkernels still have the problem of some internal shared data structure between cores, namely for interrupt forwarding and virtual memory. It is not clear how these data structures are handled in a multicore environment.

Uhlig [90] presented an adaptive mechanism for sharing data structures between microkernels running on different cores, which is a combination of coarse and fine locking and Remote Procedure Call (RPC). Locks can be enabled or disabled at run-time depending on the sharing profile presented by the application with an appropriate API. Also, remote resources (for example on a NUMA machine, resource residing on a memory with a longer access time) are treated differently from local resources. Uhlig reports examples for the Virtual Memory management. The idea is interesting, however the results depend on the specific hardware platform thus cannot be easily generalised to other architectures.

The use of microkernel based OSes has been investigated also in the domain of supercomputing on multi-processor systems, like happened with the Amoeba [92], Mach [94] and Chorus [93] OSes. These investigations were born from the observation that monolithic OSes used to carry on lot of unneeded functionality on each and every node of a parallel machine, introducing unneeded overheads. Also, the standard communication primitives used on traditional OSes used to be highly inefficient in the context of a multi-processor machine. On the other hand, the approach typical of the HPC domain used to consist in not having a real OS, but rather a small run-time environment provided in the form of libraries. This was too minimalistic and used to lack potentially useful capabilities. So, the adoption of a microkernel based OS was considered a good trade-off between these worlds [91].

A few commercial operating systems were inspired by the microkernel architecture. Windows NT borrowed some of the ideas from the microkernel environment [27], however NT should be considered a “hybrid” between a monolithic kernel and a microkernel. Also, the Mac OS-X kernel, a.k.a., XNU, derives from the fusion of the Mach and FreeBSD kernels. However, only some kernel-level primitives and paradigms of Mach were kept, whilst the microkernel architecture has been basically dropped.

QNX Neutrino [28] is a Real-Time Operating System with a structure similar to a microkernel, however, some of the essential scheduling services (like the pthread interface, mutexes and semaphores) are implemented directly by the kernel for efficiency reasons. The market of Neutrino (mainly embedded systems and networking systems like high speed routers) pushed the development of the multicore support early on, thus Neutrino was one of the first RTOSes to be available on embedded multicore platforms.

OSE[22] is another microkernel-based RTOS. OSE was developed by ENEA, a spin-off of Ericsson AB. In OSE, real-time tasks are implemented as processes that communicate among them mainly through message passing paradigm. Since memory protection is enforced between processes, the OSE kernel is known for its fault-tolerant and high availability features, and it is widespread in telecommunication applications.

1.b) Single-System Image

Single-System Image (SSI) Operating Systems have been designed in the context of cluster computing, for the purpose of making the usability and programmability of clusters easy. An SSI OS gives the application programmer the illusion that a cluster is a single computing system with a higher performance. The programmer writes parallel applications composed of many processes which communicate to each other by means of standard IPC mechanisms. Actually, the OS is capable of seamlessly distributing the

22 More information is available at: http://www.enea.com.

workload over a distributed set of *homogeneous* machines interconnected by a network, migrating applications and data as required in order to make an efficient use of the underlying physical resources.

Key features of a SSI OS comprise: migration of processes and threads, load balancing strategies, global distributed shared-memory, management of the cluster possibly supporting dynamic node addition, fault-tolerance, checkpointing and high availability capabilities, scalable distributed File-System and high-performance networking primitives, special security features needed to manage users, groups and permissions in the cluster.

The SSI Operating System concept has been implemented for example in Kerrighed [13], openMosix[23] (initially based on MOSIX [15], the project was officially closed in 2008) and OpenSSI[24]. All of them constitute variants of Linux, which add to the kernel the fundamental lacking features. A comparison among these approaches can be found for example in [14].

SSI systems aim to realise high-performance computing clusters preserving a “local” programming model that is unaware of the actual distribution of the load within the network. This allows parallel applications initially thought for multi-processor (or multi-core) systems to easily take advantage of the additional computing resources made available across the network, without any need to explicitly code the distribution logic. While being one of the main advantages of this kind of systems, it also constitutes its very limitation. The actually obtainable performance speed-up depends strongly on the communication patterns among the processes composing an application. However, the assumptions of the programmer about locality of data and processes are subverted when the application is deployed in an SSI cluster. The interaction overheads (now implying networking latencies) may sometimes nullify the potential advantages due to the increased available overall computing power, unless the application is carefully coded considering the deployment environment. This kind of problems may be mitigated by proper monitoring and migration strategies at the SSI kernel level, e.g., by trying to keep those processes which interact too frequently on the same machine .

1.c) Operating Systems for Multiple/Many Cores

Research on multiprocessor operating systems is active since a long time, much before the multi-core paradigm became so successful. Two broad classes of kernel organisation have been proposed by Lauer and Needham [156]: message-based and procedure-based approaches. In a procedure-based kernel, there is no fundamental distinction between a process in user space and kernel activities: each process performs kernel operations via system calls, and kernel resources are represented by shared data structures between processes. Conversely, in a message-based kernel, each major kernel resource is handled by a separate kernel process, and typical kernel operations require message exchanges. The procedure-based approach closely mimics the hardware organisation of Symmetric Multiprocessors (SMP) with Uniform Memory Access (UMA): this is the basic organisation underlying monolithic kernels. The message-based approach closely mimics the hardware organisation of a distributed memory multicomputer, and this is the basic organisation of microkernels. As noted by Chaves et al. [155], the choice between procedure-based and message-based structures has a pervasive impact on the rest of the operating system than any other single design decision. The authors then compare two different approaches to kernel-kernel communication in a NUMA machine without cache coherency: remote memory access versus remote invocation. In the first case, access to shared resources is

23 More information is available at: http://www.openmosix.org.
24 More information is available at: http://www.openssi.org.

performed by accessing remote memory using a remote locking mechanism; in the second case, it is achieved through invoking an operation on the remote node. The work is outdated, because of the advances in hardware architectures, however, some of the basic findings are of general validity: in particular, remove invocation is preferable for long operations, while remote memory access is preferable for short critical sections.

Many different papers [155][156] have insisted on the tension between lock-based communication and synchronisation versus remote invocation. Depending on the underlying hardware architecture and on the different structure of the operating system, and depending on the requirements of the applications (performance, security, scalability, etc.) sometimes the lock-based approach seems to be the most appropriate, sometimes the remote-invocation approach proves to be the best approach.

Andrew Baumann et al. recently proposed an OS model called Multikernel [9] which is largely inspired by the message-based organisation described above: each single processor and even core inside a processor can be seen as a node of a distributed system. One of the basic principles Multikernel relies upon is the absence of shared kernel-level data structures. Each core runs a kernel instance which is independent from the other cores. The various kernel instances communicate with each other by means of message passing and RPC-like interactions. All kernel-level information and status data that needs to be shared among multiple cores is therefore replicated between them, and kept synchronised by explicit protocols. This way all communications among cores and processors need to be explicitly coded, and this naturally leads to asynchronous communication patterns, largely used in distributed systems, which enhance the possibility for the system to parallelise and pipeline activities, rather than having cores stall waiting for implicit cache coherence protocols run by the underlying hardware. Interestingly, in the Multikernel view, the fact that the OS does not rely on shared data does not preclude applications to be developed with a shared memory paradigm.

Also, Multikernel envisions a hardware-neutral OS model, where for example the part of a CPU driver in charge of handling the communications between different cores, may actually take advantage of available low-level information about the cores topology and their interconnection infrastructure. For example, different hardware-level mechanisms may be exploited in order to send messages among cores sharing a L3 cache, as compared to the ones needed to send messages among cores that do not share such a cache, or reside on different processors (e.g., multi-cast trees, so well-known in the domain of distributed systems).

The Multikernel model has been implemented as of now as the Barrelfish[25] prototype, and preliminary measurements seem to be promising, especially on the side of scalability of certain critical operations involving all the cores (e.g., TLB shootdown). However, the experimental results available so far are to be considered as preliminary, in particular since many of the aspects addressed by the Multikernel concept have not been realised as yet, due to technological problems (such as the hardware independence etc.).

Yuan et al. proposed GenerOS [16], a variation of the Linux kernel explicitly addressing heterogeneous multi-core systems. In GenerOS, the cache contention on the same core due to different types of activities going on within a system is reduced by means of partitioning the activities among the available cores. Specifically, the GenerOS kernel envisions a partitioning of the available cores into three types: *application cores*, dedicated to running applications and specifically exclusively user-space code; *kernel cores* dedicated to running exclusively the kernel-space part of the system calls

25 More information is available at the URL: http://www.barrelfish.org.

invoked by the applications; *interrupt cores* dedicated to servicing interrupt requests. This kind of partitioning of activity type into separate cores is made available through a set of modifications to the kernel that allow system calls to execute on a core different from the application core invoking the functionality. Also, kernel cores run one or more *kernel servers*. Each kernel server is dedicated to one or more system calls, it waits continuously for requests of that particular set of system calls from the application cores, and serializes their execution by means of the so called “slim scheduler”, avoiding any need for context switches among requests from different applications.
The fact that kernel-space code is handed over to different cores than the ones where the main applications code is running, together with the serialisation of kernel-space system call executions by the kernel servers, causes a decrease of the contention in accessing the cache, when compared with a plain Linux system, as shown by the experimental results performed by the authors. However, the serialisation of system calls execution is somewhat against the current trend in the Linux kernel: from the ancient ages in which the kernel-space code was non-preemptible, in recent years a lot of effort has been dedicated just for increasing preemptibility of kernel code, which is well-known to reduce latencies and improve responsiveness of the system. Even if the presence of multiple cores may mitigate such problem, the situation is not expected to be tremendously different when high workloads are in place with a nearly saturated system. Therefore, more investigations would be needed in order to understand what is the impact of the proposed OS model on various application classes, especially on interactive and real-time ones. An issue of the GenerOS model is constituted by the proper OS configuration in terms of balancing among the various types of cores, as well as the number of kernel servers and how system calls are distributed across them. In the initial prototype, the authors used a static configuration, but they also observed that this is one of the troublesome to be faced in their proposed OS model.
Boyd-Wickizer et al. proposed Corey [21], an Operating System designed from scratch around the need for allowing applications to make an efficient use of massively parallel and multi-core hardware. The authors highlight that kernel-level shared data structures may cause unneeded overheads when accessed from multiple cores, even when the applications that are being run would not need to share any data. For example, process and file descriptor tables are potentially at risk of being contended among multiple cores, even when the applications running on them are accessing independent processes and files. Therefore, it is proposed to delegate the responsibility to decide what is shared across which cores as much as possible to the application. This is done via a specialised API allowing applications to define and control three main elements: *shares* are areas of scope either local to a core or global for a set of identifiers; *address ranges* are memory segments explicitly assigned to shares; *kernel cores* are cores dedicated to the execution of kernel code, i.e., interrupt handlers and the kernel-side part of system calls (which are handed over from the application cores to the kernel cores). Experimental results seem promising, in that a reduction of the overheads due to contentions on kernel-level data structures is achievable, at the cost of a little complexity for the application developer, who needs to properly set-up shares and address ranges.
Wentzlaff and Agarwal proposed the Factored Operating Systems (fos) [22], an OS model that builds on concepts taken from the distributed computing world, in order to reduce contention on kernel-level shared data structures. Specifically, the fos architecture foresees the partitioning of cores between applications and kernel services. Each kernel service is implemented by one or more specialised servers that run on kernel cores, and bits of service-based computing are reused for allowing each

kernel service in distributing its workload to the available kernel servers, similarly to load-balancing techniques in web servers. The fos Operating System is still under development, and it is being entirely redesigned from scratch, based on a microkernel structure, where the concept of relegating OS functionality within specialised servers that communicate by message-passing mechanisms to each other and to applications is already in place. Also, in fos, each kernel core runs a single kernel server that enqueues requests in an input queue and services them in a serialised, non-preemptible way. This removes (in the opinion of the authors of this approach) the need for having traditional temporal scheduling of kernel cores. Actually, it is foreseen to have a form of cooperative scheduling, by which a kernel core, while serving a request, can yield explicitly the core so that it can serve other requests, while it waits for some device and/or other kernel servers to respond. The way kernel servers service applications requests is planned to be based on stateless protocols, so that subsequent requests of the same application can be potentially handed over to different kernel servers for a better load distribution across kernel cores.

Interesting investigations in this area have also been carried out by Schubert et al. [1] [2][3][4]. These are summarised in D5.2 [86], as they are at the foundation of the research proposed in the context of the S(o)OS project itself.

All of the above mentioned approaches to the (re-)engineering of the OS kernel model for dealing with massively parallel systems are of utmost interest for the purposes of the S(o)OS project. However, these approaches are at a quite preliminary and conceptual stage, with only some of them having experimental prototype implementations. Therefore, these do not constitute consolidated approaches proved to be industrially viable, feasible and understandable for a wide audience. More research needs to be performed on this side, addressing scalability, efficiency and programmability issues for all of the sub-components of an OS kernel, and investigating on the achievable trade-offs between overall system and individual applications performance and responsiveness.

1.D) Operating Systems for GRIDs

Proposals have appeared in the literature for Operating Systems specifically targeted at supporting GRID systems. For example, Padala and Wilson proposed [10] a GRID OS that has the goal of providing a minimum set of services which are common to all GRID middleware infrastructures, still building on a traditional OS like Linux with as few changes as possible (in fact, additional features required at the kernel level are provided through Linux loadable kernel modules). The core functionality that needs to be added to the OS, according to the authors, is: high-performance I/O and networking, for example relying on copy-free communication primitives and fine-tuning of network stack parameters such as the TCP/IP window size; communication primitives with a better support for such mechanisms as MPI; resource management features allowing for resource discovery, allocation, monitoring; process management capabilities supporting for example global identifiers for processes, which may be used in the mentioned communication primitives for distributed IPC. The point that is made by the authors, and validated by the presented experimental results, is that implementing such services merely at the middleware level, outside the kernel, as commonly done in existing GRID middleware solutions, constitutes a bottleneck in the potentially achievable performance.

More recently, Puri and Abbas conceptualized [11] a GRID OS aimed to support GRID applications, by means of embedding within the OS itself such capabilities as: fault-tolerance and check-pointing, transparent access to distributed resources in a location-

independent fashion, load balancing by means of migration of processes and virtualization, and scalability. However, the paper remains at a very abstract level, and it does not discuss practical implications of the envisioned architecture, such as what is required to be supported at the kernel level and what can be delegated to the OS middleware.

The XtreemOS European Project[26] produced XtreemOS [17], a variation of the Linux OS enhanced with Grid capabilities. The XtreemOS extensions to Linux include: LinuxSSI, a single-system image version of Linux based on Kerrighed [13] which allows to register into the XtreemOS Grid a cluster of systems virtually seen as a single, more powerful machine; XtreemFS [18], a networked file-system supporting automatic replication and high-availability of data; process checkpointing; a middleware for management of Grid nodes and submission of tasks; XOSAGA [20], an application-level API for Grid applications which constitutes an implementation of the abstract language-independent SAGA [19] specification, with some XtreemOS specific extensions.

Starting from release 10.4, the Mac OS-X Operating System embeds a simple Grid management middleware called Xgrid [12], which is immediately available on all installations of the OS, if the corresponding service is activated. Xgrid has a three-tier architecture: *clients* submit jobs to *controllers*, which in turn hand them over to *agents* for the actual processing. Clients may submit jobs to the Grid by means of either a command-line tool, or a dedicated API which is part of the OS foundations API. In order to share large amounts of data among Grid tasks, it is possible to use one of the available distributed filesystems, such as NFS. Xgrid is targeted to an easy set-up of Grids with no strong requirements on the number of interconnected nodes and complexity of the submitted jobs. For example, only linear workflows are supported, whereas for more complex Grid settings one can install on the OS one of the other more complex Grid management middleware solutions. Mac OS-X 10.5 introduced Xgrid 2, with some enhancements on the job scheduling decisions, such as the so called Scoreboard, i.e., a customizable scoring script that may be provided by clients in order to drive the decisions made by controllers about what agent nodes to choose when multiple ones are available. The script may base its score on the availability of particular capabilities of the agent node, or the connection/connectivity conditions, etc. This is useful for increasing the performance of the deployed applications.

1.e) Operating Systems for HPC

As detailed in section V. on high performance computing, in classical cluster systems, each processor (in the sense of smallest compute unit) hosts its own operating system environment. HPC jobs typically run on the system in an exclusive way, in order to achieve maximum performance. This means that the respective execution environment does not have to deal with scheduling issues, scale management etc. Essentially, the operating system therefore primarily serves the same purpose as a virtualisation system, i.e. it abstracts from the underlying hardware and deals with I/O of the system, in particular for accessing shared resources and communication between threads, respectively processes. This means implicitly that many of the functions in a general purpose operating system are obsolete for HPC usage and can (should) be removed from the kernel, in order not to produce unnecessary overhead.

As noted, it is thereby of particular relevance for efficient parallel computing that the specific characteristics of the hardware are exploited to their maximum potential, such as the memory architecture of the systems. Therefore, the operating system is typically specifically adapted to the environment, so as to reduce performance loss due to misalignment.

26 More information is available at: http://www.xtreemos.eu.

Essentially, most HPC providers therefore reduce the operating system to an essential minimum and adapt the kernel to the specific environment. Obviously, Linux is the primary choice in such cases, due to its open source nature. Microsoft Windows and even Apple Mac OS-X have been demonstrated to work for HPC clusters, too[27], yet their performance and the overhead for adaptation typically does not fulfill the expectations – as such, there are for example 475 Linux / UNIX based systems on the Top 500[28] list (in which systems are ranked by their performance on the LINPACK Benchmark [87] [88]), but only 5 Windows based ones (and none for Mac OS-X)[29]. With the increasing heterogeneity and hence divergence between supercomputer setups and at the same time the growing scale of affordable high performance machines, it is likely that this distribution will change slightly in the near future (see D5.2 for details).

Currently, the most widely used **Linux distributions** are probably[30] :

Red Hat Linux[31] and SUSE Linux Enterprise[32] are widely used[33] mainly due to the range of system architectures they support (namely: x86 32 & 64 bit, Itanium IA-64, PowerPC 32 & 64 bit), even though official Red Hat Linux releases are comparatively rare (latest official release was in 2003).

The Scientific Linux[34] distribution is typically preferred over the Red Hat distribution, which is 100% compatible with Red Hat Linux. As opposed to the Red Hat version, Scientific Linux however is available for free and adaptations to individual systems are maintained by the community rather than by Red Hat. Even though reliability is decreased this way, the distribution adapts quicker to new systems. Scientific Linux supports the following architectures: x86 32 & 64 bit, Itanium IA-64.

CentOS[35] is another popular Linux distribution which bases on Red Hat Linux and is available free of charge. Like Scientific Linux it is mainly maintained by the community, yet the latest versions only support the x86 architectures thus making it less interesting in the future. At the time of writing, 7 machines in the top 500 made use of CentOS.

Even **UNIX based operating systems** are still in use on HPC clusters, as they generally scale quite well. With a few exceptions, they are mostly commercially distributed which makes them less attractive than Linux in particular in academic circles. The number of UNIX systems in the top 500 basically decreases, with the particular exception of AIX which ships with the IBM machines and supports in particular the PowerPC and the IA-64 architecture, making it attractive for the according clusters.

Though Lameter claims that Linux scales well enough for future large scale platforms [118], it must be noted that this is mostly true for the number of *processes* but not the number of *processors* [119][9]. The authors furthermore claim that due to messaging overhead, a microkernel approach is not a feasible alternative to Linux, yet as Barham et al. could show, the messaging approach scales better than the Linux monolithic structure [9].

27 More information is available at: http://hpc.sourceforge.net/.
28 More information is available at: http://www.top500.org/.
29 More information is available at: http://www.top500.org/stats/list/35/osfam.
30 More information is available at: http://www.clusterbuilder.org/software/operating-system.php.
31 More information is available at: http://www.redhat.com.
32 More information is available at: http://www.novell.com/linux.
33 At the time of writing this, 18 machines in the top 500 use either distribution.
34 More information is available at: http://www.scientificlinux.org.
35 More information is available at: http://www.centos.org.

2. SCHEDULING

One of the core features of an Operating System is its ability to multiplex the access to the available physical resources to multiple processes/threads that run at the same time onto the system. Some resources may be managed in an exclusive way, i.e., only one application at a time is granted access to it. Other resources, and particularly processor(s) and disks, are managed in a shared way, so that the competing applications alternate in accessing the resource(s) according to some scheduling policy. For the CPU, General-Purpose OSes usually provide Round-Robin based scheduling, where the ready tasks alternate each other after a certain time-slice which may be fixed or dynamically changing. GPOSes have usually scheduling policies designed so as to achieve a high overall system throughput, and to serve processes on a Best-Effort (BE) basis, i.e., no guarantees can be provided to the individual applications. On the other hand, Real-Time OSes have usually scheduling policies which are capable of providing precise scheduling guarantees to the competing applications. To this purpose, RTOSes undertake an admission-control phase, in which a new process is accepted into the system only if its timing requirements may be fulfilled, and the ones of the already accepted processes are not disrupted.

2.A) REAL-TIME SCHEDULING

The traditional real-time scheduling research area focuses mainly on hard real-time systems, where deadlines are considered to be critical, in the sense that deadline misses cannot be tolerated, because they lead to the complete system failure and possible catastrophic consequences (i.e. losses of life).
However, real-time theory and methodologies are gaining applicability in the field of soft real-time systems, where applications possess precise timing and performance requirements, but occasional failures in meeting them may be easily tolerated by the system, causing a graceful degradation in the quality of the provided service.
The real-time literature on soft real-time technologies for General purpose operating systems (GPOSes) is growing, and two major research branches to mention, with some relevance for the IRMOS project, are those relative to: multiprocessor scheduling, soft real-time scheduling, QoS control in distributed real-time applications and adaptive QoS control, detailed below.

SCHEDULING REAL-TIME TASK SETS ON MULTIPROCESSOR PLATFORMS

Even if the concept of multiprocessing has always been present in the real-time community, only recently it is receiving a significant attention, thanks to the increasing industrial interest in such platforms, and their consequent increasing availability. While the scheduling problem for uni-processor systems has been widely investigated for decades, producing a considerable variety of publications and applications, there are still many open problems regarding the “schedulability” analysis of multiprocessor systems. As pointed out by Liu in his seminal paper [193]: “*few of the results obtained for a single processor generalise directly to the multiple processor case: bringing in additional processors adds a new dimension to the scheduling problem*”.
Unfortunately, predicting the behaviour of a multiprocessor system requires in many cases a considerable computing effort. To simplify the analysis, it is often necessary to introduce pessimistic assumptions. This is particularly needed when modelling globally scheduled multiprocessor systems, in which the cost of migrating a task from a processor to another can significantly vary over time. The presence of caches and the frequency of memory accesses have a significant influence on the worst-case timely parameters that characterize the system. To bind the variability of these parameters,

often real-time literature focuses on platforms with multiple processors but with no caches, or whose cache miss delays are known. Also, the cost of preemption and migration on multi-processor systems is a very important issue that still needs to be properly considered in real-time methodologies. Some research in the domain of hardware architectures moves towards partially mitigating such issues. Recently, a few architectures have been proposed that limit penalties associated to migration and cache misses, for example the MPCore by ARM. Some researchers have recently proposed hardware implementations of some parts of the operating system, allowing one to reduce the scheduling penalties of multiprocessor platforms [236].

SCHEDULING ON MULTIPROCESSOR

When deciding which kind of scheduler to adopt in a *multiple processor system*, there are two main options: *partitioned* scheduling and *global* scheduling:

Partitioned scheduling

In a partitioned scheduler, there are multiple ready queues, one for each processor in the system, and it is possible to leave a processor in idle state even when there are ready tasks needing to be executed. Each queue is managed according to (well-known) uni-processor scheduling algorithms, and task migration is not allowed. The placement of tasks among the available processors is a critical step. The problem of optimally dividing the workload among the various queues, so that the computing resources be well utilised, is analogous to the bin-packing problem, which is known to be NP-hard in the strong sense [237][238]. This complexity is typically avoided using sub-optimal solutions provided by polynomial and pseudo-polynomial time heuristics. Example of policies used for this purpose are First Fit, Best Fit, Next Fit, First Fit with decreasing utilizations, etc. [241][242][243][244].

Partitioned scheduling is commonly used in real-life multi-processor systems, due to the fact that overheads due to continuous synchronisation and contention among cores typical of global scheduling are avoided, and that it allows for analysing a system similarly to what can be done in a distributed system with various single-processor elements.

Global scheduling

For task sets with highly varying computational requirements, instead of wasting computational power by repeatedly invoking complex load-balancing algorithms, it is better to use a global scheduler. This way, tasks are extracted from a single system-wide queue and scheduled onto the available processors. The load is thus intrinsically balanced, since no processor is idled as long as there is a ready task in the global queue.

A class of algorithms, called Pfair schedulers [245], is able to ensure that the full processing capacity can be used, but unfortunately at the cost of a potentially large run-time overhead. Migrative and non-migrative algorithms have been proposed modifying well-known solutions adopted for the single processor case and extending them to deal with the various anomalies [269] that arise on a parallel computing platform.

Complications in using a global scheduler mainly relate to the cost of inter-processor migration, and to the kernel overhead due to the necessary synchronisation among the processors for the purpose of enforcing a global scheduling strategy. Even if there are mechanisms that can reduce the migration cost, it could nevertheless cause a significant schedulability loss when tasks have a large associated context (i.e. data

overhead). Therefore, the effectiveness of a global scheduler is rather dependent on the application characteristics and on the architecture in use.

Hybrid schedulers

In addition to the above classes, there are also intermediate solutions, like hybrid- and restricted-migration schedulers. A hybrid-migration scheduler [250] limits the number of processors among which a task can migrate, in order to limit the number of caches in which the task image is present. This way, fewer cache misses are expected, and the cost of migration and context changes is lower. This method is more flexible than a rigid partitioning algorithm without migration, and it is particularly indicated for systems with a high number of processors (with tens of CPUs), where moving a task from a computing unit to a distant one would imply significant overheads.
A restricted-migration scheduler instead allows for task migrations, but only at job boundaries, i.e., before a job starts to be executed or at the end of its execution. In this way, the amount of information that must be transferred while migrating a task from one processor to another is likely to be less than the full migration case [251]. A similar method consists in using a global scheduler with preemptions disabled: this way, once a job starts executing on a CPU, it is not possible to preempt it until the end of execution, so that migration can never take place [252]. However, note that non-preemptable systems can incur in significant schedulability losses, due to potential delays caused by long chunks of code that can execute without being interrupted.

Soft real-time scheduling

Different scheduling algorithms have been proposed to support the specific needs of soft real-time applications. A first important class approximates the Generalized Processor Sharing concept of a *fluid flow* allocation, in which each application using the resource marks a progress proportional to its weight. Among the algorithms of this class, we can cite Proportional Share [260] and Pfair [245]. Similar are the underlying principles of a family of algorithms known as Resource Reservation schedulers [262]. In the Resource Kernels project [262], the resource reservation approach has been successfully applied to different types of resources (including disk and network). Also, it is noteworthy to mention that the current scheduler in the main-stream Linux kernel, known as Completely Fair Scheduler (CFS), is basically a variation of the Proportional-Share idea. The Resource Reservation framework has been adapted to partitioned multiprocessor systems in [258] and [259] for the Constant Bandwidth Server (CBS) and the Total Bandwidth Server (TBS) algorithms, respectively.

Design of distributed real-time applications

The problem of designing scheduling parameters for distributed real-time applications has received a constant attention in the past few years. In [264], the authors introduce a notion of transaction for real-time databases characterized by periodicity and end-to-end constraints and propose a methodology to identify periods and deadlines of intermediate tasks. A similar approach is taken in [265], in which activation periods of the intermediate tasks that comply with end-to-end real-time requirements are synthesized by an optimization problem. In [266], the same idea is applied to soft real-time applications. In this case the authors use performance analysis techniques to decide the bandwidth allocated to each task that attain a maximum latency and a minimum average throughput for the chain of computation.
Concerning modelling of timing requirements of real-time applications, usually models similar to synchronous data-flow networks [267] are used. As shown in [268], synchronous data flow networks lend themselves to an effective code generation

process, in which an offline schedule is synthesized that minimizes the code length and the buffer size. The models used in [270], [271] and [272] are also a special cases of synchronous data-flow, but, due to the inherently distributed and dynamic nature of the considered applications, the aim is not an optimized offline scheduling of activities, but rather an efficient on-line (run-time) scheduling mechanism. Finally, in [273] the problem of optimum deployment, over a physical heterogeneous network, of distributed real-time applications with computing and networking requirements subject to end-to-end response-time constraints is tackled by introducing a formalisation in terms of a Mixed-Integer Non-Linear Programming optimisation program. The authors propose both a deterministic and a probabilistic formalisation of the problem, with various possible optimisation goals. Once solved, the solution not only provides information about where to deploy the various real-time components, but it also provides the scheduling parameters to be used for configuring the local real-time schedulers over the CPUs, and the bandwidth figures to be reserved over the physical links.

In the context of the S(o)OS project, complex real-time applications may have several threads spread over multiple cores and processors within a many-core system. The problem of how to exactly perform the deployment of the various threads may be inspired to the above mentioned research works. However, for scalability reasons, distributed approaches in which sub-optimum solutions are found with a collaborative set of computations carried out by more cores/processors should be favoured over centralised approaches which may easily turn out to be infeasible on large scale systems.

3. Synchronisation

Synchronisation is an essential problem of concurrent programming, and it has received a great attention from the research community. The problems of concurrent access and of providing a consistent view of shared data structures can be solved in different ways, depending on the abstraction level and on the basic organization of the operating system and programming paradigm. The basic properties that correct concurrent programs must possess were described by Herlihy and Wing [165], and solutions have been proposed both for shared-memory and message-passing.

SHARED-MEMORY

In the shared-memory paradigm all processors can access the same memory, uniformly (UMA machines) or non uniformly (NUMA machines). In the following, we abstract from low-level mechanisms for providing shared-memory, as long as all the processors have a consistent and coherent view of the memory.

The basic mechanism to guarantee mutual exclusion is to use *locks*. In its basic form, every shared data structure can be associated with a a lock, which is just a Boolean variable that can be false (meaning that no process is accessing it) or true, meaning that a process is accessing it, so the other have to wait for the previous process to complete. Locks are usually implemented using special atomic operations, like *test-and-set, fetch-and-store* or *compare-and-swap*. A lot of work has been done to improve the basic locking mechanism. Mellor-Crummey and Scott [168] solved the problem of reducing contention on the shared bus by using separate spin variables for each processor on which each core performed busy waiting. Their solution, called MCS locks from the names of the authors, also resolves the typical problems with fairness and starvation under high contention scenarios. Many papers have proposed improvements on this basic mechanism, e.g., to introduce time-out [169], to reactively change the

lock behaviour [170], to balance overhead vs. latency using a mixed coarse-grain/fine-grain locking strategy [171].
Higher level concurrency control objects (e.g., semaphores) are built on top of low-level locks. The main difference is that when the resource is busy, a low level lock performs a busy wait (spin-lock) while a high level object will block the process and invoke the scheduler. Of course, this second option has a substantially higher overhead, as it involves a scheduler call and one context switch. Moreover, waking up requires an inter-process interrupt. Depending on the contention level, and on the duration of the critical section, it may be more convenient to perform non preemptive busy waiting or to block the process. Mukherjee and Schwan [175] proposed an adaptive locking protocol, that chooses which method to use (blocking or busy waiting) depending on the characteristics of the application (e.g., the typical duration of critical sections).
A different approach consists in making a local copy of the data structure (or of part of it), modify it, and later try to commit the changes in the global copy without disrupting the linearisability property [165]. This class of approaches is usually referred to as *lock-free* or *non-blocking* or *wait-free* [166]. In general, wait-free techniques are much more difficult to program than lock-based techniques, hence they are rarely implemented in actual systems; also, the commit phase may fail, and in such a case the operation needs to be repeated one or more times until the commit succeeds. Under high contention, it is difficult to bound the execution time of the operation. However, many wait-free algorithms have been proposed for common data structures, like priority queues [173], or stacks [174]. *Transactional memory* has been proposed as a hardware-level support for wait-free mechanisms [167]: the idea is to provide simple hardware mechanisms to support the implementation of wait-free algorithms.
For data structures where reading is more frequent than writing/updating, special mechanisms have been proposed to reduce contention, such as the Read-Copy Update (RCU) mechanism [172], widely adopted within the Linux kernel for accessing critical shared lists. With RCU, readers are allowed to just read (and walk through) the common shared list elements without any synchronisation; a writer modifies the list by atomically manipulating pointers so as to insert new items in the list or delete existing ones. In the latter case, one or more readers possibly accessing the deleted element will keep seeing the element, which can only be discarded at specific instants (called *quiescent states*) when no process is reading or writing it (e.g., idle instants, scheduler calls, etc.).
Which locking mechanism is more adequate? McKenney [181] provides a comparison of several techniques in the form of patterns, from different points of view: latency, memory bandwidth, memory size, granularity, fairness and read-write ratio. There is no clear winning strategy and every mechanism has its advantages and disadvantages. A similar comparison has been carried out by Anderson [162].

MESSAGE-PASSING

In the message-passing paradigm, each resource is assigned a server thread which exclusively performs operations on the data structures of the resource. Mutual exclusion is thus guaranteed by the fact that only one thread has exclusive access to data. Other threads (clients) must request the operation by issuing a remote invocation via an IPC. This organisation mimics distributed systems where nodes do not share memory. The MPI interface [176] is based on this paradigm.
From the point of view of the kernel in multiprocessor systems, it is worth comparing the performance of message-passing vs. shared memory communication. Chavez et al. [163] perform a comparison of shared-memory communication based on spin-locks vs. remote invocation in a NUMA multiprocessor architecture. Another comparison has

been proposed by Chandra et al. [180]. They highlight that the server can be implemented as an interrupt handler or as a thread. In the first case the overhead is minimal, however the implementation is more difficult as an interrupt handler is not subject to the kernel scheduler and cannot block. In the second case, the overhead is larger but the advantage is that the implementation is easier. Performance is highly dependent on the underlying hardware structure, memory hierarchy (UMA or NUMA, presence of cache coherency, etc.), however it can be generally said that message passing is more adequate for long critical operations on processors with high memory access delay, while shared memory is preferable on short critical section on local data structures. It is important to underlying that if two servers on two different resources are executed on the same node, their operations will be serialized, and this in some case may be an unnecessary restriction of parallelism. On the other hand, it is worth noticing that providing lock-based access to remote memory may be expensive and unnecessarily increase the level of contention.

Clearly, it is possible to mix shared memory locking and remote invocation [178][179]. Recently, Uhlig [177] proposed to use IPC or shared memory locking depending on the locality of the data structure with respect to the current node, and to use an adaptive locking mechanism depending on the level of contention.

4. Synchronisation and Scheduling

When multiple tasks need to access a shared resource or data structure, they need to synchronise each other, in order to avoid performing the access at the same time. The basic synchronisation means is constituted by a binary semaphore, which, if already taken, causes the process attempting to acquire the lock to be suspended by the scheduler, and be woken up later when the lock owner exits the critical section. However, a great research effort has been done in two very important domains: in the literature of real-time scheduling, it is important to ensure that the amount of time a process has to wait before acquiring a lock may be some how kept under control; also, in multi-processor and multi-core scheduling, if the acquired resource is held by a task on another processor, and the critical section is expected to be very short (like it happens quite often in the kernel of an OS), then suspending the current task and performing a context-switch might lead to unnecessary overheads, whilst other policies may be more convenient (e.g., spin-locking). Interestingly, the PREEMPT_RT branch of the Linux kernel has an option [276] for turning (almost) every spin-lock primitive used inside the kernel into a mutex.

In the following, we summarise the most important research works on these topics:

4.a) Shared Resources Protocols in Real-Time Scheduling

There are well-known efficient protocols to arbitrate the exclusive access to shared resources for real-time task sets scheduled on a single processor platform: Priority Inheritance (PI), Priority Ceiling Protocol (PCP) [253], Dynamic Priority Ceiling Protocol (DPCP) [254] and Stack resource Policy (SRP) [255]. Each one of these protocols allows for solving the “priority-inversion” problem, that takes place when a task is blocked by a lower priority one at the beginning of a critical section accessing a shared resource. The SRP ensures that a task can never be blocked once it started executing. This minimises the number of context switches, allows all tasks to be executed on a single stack, and prevents deadlocks.

The problem of designing shared resource protocols for real-time systems scheduled on more than one processor has been analysed in different papers. Most of them [66][67]

[69][68][76] are extensions of well-known approaches that have been developed for uniprocessor systems. Others, instead, mandate that each short global critical section be executed non-preemptively, reducing the occurrence of *remote blocking* [74][75], i.e., a task that blocks in the attempt to lock a global resource which is already locked by another task running on a different processor. We will hereafter summarise the main characteristics of these solutions, distinguishing between solutions for partitioned and global scheduling. Global schedulers have a single system-wide queue from which tasks are extracted to be scheduled on the available processors. Note that, with a global scheduler, a task might be preempted by a higher priority job, and later resume its execution on a different processor. This process is called *migration* of the task. With a partitioned scheduler, instead, tasks are statically assigned to processors, so that they are bound to execute on that processor and cannot migrate.

PROTOCOLS FOR PARTITIONED SCHEDULING

Adaptations of the above policies for partitioned multiprocessor real-time environments are given by the Multiprocessor PCP proposed by Rajkumar et al. in [256], and by the MSRP protocol proposed by Gai et al. in [257]. The Multiprocessor PCP is basically an extension to partitioned systems of the uniprocessor Priority Ceiling Protocol (PCP) [65] [66]. Since each global resource is controlled by one particular processor, the solution proposed in [66] is not so suitable for shared-memory multiprocessors. For this reason, an improved MPCP version [67] was proposed. According to this policy, local resources – i.e., resources that are shared only within tasks assigned to one specific processor – are arbitrated using the classic uniprocessor PCP. Global critical sections – accessing resources that are shared among more than one processor – are executed at a priority higher than any non-critical section of code. This allows for reducing the remote blocking experienced by the system, making it a function of critical sections only. Whenever a task cannot lock a global resource because it is locked by a different task, the task is blocked and it is inserted in a prioritized queue, using the normal priority as the key for the queue insertion. When the lock is released, the highest priority task in the queue can acquire the lock. Since a task is suspended whenever it is blocked on a shared resource, lower priority tasks can execute and lock some other shared resources. When the blocked task resumes execution, it might experience additional blocking due to the (global or local) shared resources meanwhile accessed by lower priority tasks. The blocking factors to be accounted for are therefore multiple and can be significantly large. Moreover, deadlock conditions are possible in case of nested critical sections unless a proper ordering of the lock operations is used.

To avoid these problems, an alternative approach is to prohibit lower priority tasks to lock any resource while a higher priority task is blocked in the same processor. Alternatively, the blocked task might *busy wait* until resource access is granted again, forbidding the execution of lower priority tasks. However, even if the blocking factors are significantly reduced, both approaches introduce some inefficiency, in that the processor is left unused whenever a high priority task is blocked on a global resource, increasing the interference on lower priority tasks. Schedulability tests and partitioning algorithms for MPCP with suspension and busy wait have been recently presented in [68].

Extensions of MPCP for partitioned EDF scheduling have been proposed by Chen and Tripathi [69]. Global critical sections were executed non-preemptively and they were not allowed to be nested with local critical sections. However, their method is valid only for the more restrictive periodic task model, and not for the sporadic one.

Among busy-waiting approaches, Gai et al. presented [70] the Multi-processor Stack Resource Policy (MSRP), generalizing to multi-processor systems the SRP protocol [71]. Global critical sections are executed non-preemptively, and access to global resources is implemented with FIFO-based spin-locks: tasks that are waiting for a global resource insert their requests on a global FIFO queue, without surrendering the processor until the resource is granted. To avoid deadlocks, global critical sections cannot be nested. A less efficient solution based on SRP for partitioned EDF scheduling has been proposed by Lopez et al. [72], assigning to the same processor all tasks sharing the same resource.

PROTOCOLS FOR GLOBAL SCHEDULING

The problem of mutually exclusive access to shared resources for global scheduling algorithms has only recently been analysed [73][74][75][76]. Devi et al. proposed [74] a straightforward modification to the global EDF scheduler, with a locking protocol based on the non-preemptive execution of global critical sections and with FIFO-based wait queues. Holman and Anderson designed locking protocols for Pfair schedulers [73]. Block et al. implemented the FMLP protocol [75], validating it for different scheduling strategies, namely, partitioned EDF, global EDF, and Pfair scheduling. Short critical sections are handled using spin-locks with FIFO-based wait queues, while long critical sections are arbitrated using a priority inheritance protocol similar to PCP [65] [66]. In all of the above approaches, the problem of accessing nested shared resources is either avoided – forbidding the nesting of critical sections – or it is solved by limiting the degree of locking parallelism – i.e., protecting nested critical sections with group locks. Very recently, Easwaran and Andersson proposed [76] two different protocols for the arbitration of the access to shared resources in multiprocessor systems globally scheduled with Fixed Priority. The first one (PIP) is a generalization to global scheduling of the Priority Inheritance Protocol [65][66]. The second one (P-PCP) implements a tunable trade-off between PIP and a ceiling-based protocol that limits the number of times a job can be blocked by lower priority tasks. In the same paper, schedulability tests based on response time analysis were presented for both protocols.

SHARED RESOURCES PROTOCOL IN THE LINUX KERNEL

The design criteria and performance metrics mainly adopted by Linux kernel developers is overall system throughput and fairness. However, although real-time behaviour and predictability have not been a primary concern until now, Linux embeds a few features that are commonly included in real-time kernels, and the synchronization sub-system does not constitute an exception to that.

In a Linux system, support for mutual exclusive access to shared memory areas is provided at both kernel and user level. In the latter case, this is achieved by means of system libraries (e.g., the `glibc` library). From the real-time analysis point of view, this means that the well-known phenomenon of Priority Inversion may occur, in such a system. However, the Linux kernel embeds some precautions against it, as detailed below.

Synchronisation inside the kernel: mutexes and RT-mutexes

Inside the kernel, critical sections can be protected mainly by spinlocks, mutexes and RT-mutexes. There are other means of regulating the access to sensible code, such as read-writer locks, RCU approaches, and some others, but describing them in details is out of the scope of this document.

Spinlock Spinlocks are busy-waiting, non-preemptable locks typically used for protecting small code sections of critical kernel paths. Being non-preemptable makes

them real-time safe to some extent – since the Non-Preemptive Protocol (NPP) is a known priority inversion control technique. However, a code section protected by a spinlock is required to be very short, to not “sleep” or block on any other lock and to not invoke any other code that does such.

Mutexes Mutexes are binary semaphores, and they are quite widespread, protecting many of the critical sections inside the kernel. Their use is subject to almost no limitation, but they do not deal with priority inversion avoidance.

RT-mutexes RT-mutexes are mutexes with support for Priority Inheritance (PI), i.e., the owner of a RT-mutex inherits dynamically the priority of the task with highest priority blocked on the same rt-mutex, if such priority is higher.

In the Linux implementation of PI, the concept of *pending ownership* is supported as a performance optimization to allow uninterrupted workflow of a (high priority) task repeatedly locking and releasing the RT-mutex.

When an rt-mutex is released, the highest priority waiting task is woken-up and assigned its /pending ownership/, i.e., it becomes the *pending owner* of that rt-mutex. Hence, it will try to lock the mutex as soon as it is scheduled. If nothing happens between these instants, the pending owner will be able to lock the rt-mutex, becoming its effective owner.

However, it might happen that some other task – very likely with a higher priority – tries to lock the rt-mutex before the woken-up task is scheduled again. In that case, the priority of the newcomer and of the pending owner are checked, and the lock will be granted to the higher priority one.

RT-mutexes are optimised for fast path operations, i.e., both acquiring and releasing them reduces to a `cmpxchg` instruction (atomic compare and exchange, nowadays supported by most of the available hardware architectures) in the uncontended case.

If the RT-mutex is contended (the so-called slow path), locking means: if the RT-mutex has not an owner but it has got a pending owner it can be locked only if the priority of the pending owner is lower than the one of the locking task; if the RT-mutex has an owner or stealing the pending ownership failed, the locking task is blocked and the priority of the RT-mutex owner is updated accordingly.

Conversely, unlocking of a contended RT-mutex means: wake up the blocked task with highest priority and give him the pending ownership of the RT-mutex; update the priority of the unlocking task accordingly.

In mainline Linux, the only subsystem which uses RT-mutexes is the futex interface, used to offer to user space application locking and synchronization mechanisms as required by the POSIX standard [78], which include priority inheritance and priority protection (more on this later).

On the other hand, in PREEMPT_RT – a patch maintained by a small developer group led by Ingo Molnar, with the aim of making the kernel suitable for very low latency and real-time applications – things are quite different: In fact, when this patch is applied, most of the spinlocks and mutexes are turned into RT-mutex. Since this enormously shortens the duration of in-kernel non-preemptable sections and enforces priority inversion avoidance, this might negatively impact the overall system throughput, but it also reduces the performance and predictability gap between Linux and classical real-time kernels available in industry or academia by far.

In order to allow for atomic management of core kernel-level data structures, small regions of code still exist where preemption is not allowed, and thus they are still dealt with by means of classical spinlocks.

Synchronisation support for user-space: futexes

In order to provide user-space application developers with mutual exclusion and synchronization primitives the standard system (e.g., POSIX) libraries need some support from the kernel. One way in Linux is *Fast Userspace muTEXes* (futexes):
They consist of aligned in-memory integer values that can be shared between different tasks (mainly threads, but also processes). Such values can be incremented and decremented by atomic operations directly from user-space, while the kernel is only involved when a contended case requires arbitration and some task has to fall asleep. This means that synchronization mechanisms implemented on top of futexes are very fast and efficient, at least in the fast path. A futex slow path is instead implemented by means of a specific system call which includes a parameter to indicate what kind of operation the caller wants to perform, such as: `FUTEX_WAIT`, `FUTEX_WAKE`, `FUTEX_REQUEUE`, `FUTEX_FD` and also `FUTEX_LOCK_PI` and `FUTEX_UNLOCK_PI`, which rely on RT-mutexes to handle the contended case with priority inheritance in place.

Synchronisation in user-space

Since POSIX 1003.1 1995 the so-called Pthreads API includes locking operations, to achieve mutual exclusive access to shared data, such as `pthread_mutex_lock` and `pthread_mutex_unlock`, and synchronization operations, to wait on events and conditions such as `pthread_cond_wait`, `pthread_cond_timedwait`, `pthread_cond_signal` and `pthread_cond_broadcast`.
Interestingly, in the API, mutexes may be enriched at creation time with such "protocols" as `PTHREAD_PRIO_INHERIT` or `PTHREAD_PRIO_PROTECT`. In the former case, PI-aware futex operations are used inside the library implementation, in the latter one normal futex operations are utilized, and when a task locks a PRIO_PROTECT-ed mutex, it starts executing at the higher between its priority and the priority ceilings of all the mutexes it owns, regardless of whether there are or not other tasks blocked on any of these mutexes. Such behaviour is implemented inside the library without involving the kernel.

5. VIRTUALIZATION

There are many possible approaches to virtualisation that differ by a number of technical details related to how hardware and peripherals are emulated, whether the guest Operating System is aware or not of running in a virtualisation environment (a.k.a., para-virtualisation and full-virtualisation, respectively), and if they exploit special capabilities of modern CPU architectures designed specifically for virtualisation. 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 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 [79], [82], [83] and [85] (an interesting survey [84] 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[36]:

- 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).

6. Run-Time Support and Middleware

Common desktop machines have to run multiple applications *locally* in a concurrent (time-sharing) fashion – as opposed to that, distributed systems, in particular high performance computers, are typically focused on executing specific application types with a high degree of communication. Whilst the desktop landscape is expected to change in the near future (see Deliverable D5.2 [86]), currently the actual execution environment of the individual areas differ considerably (see also section IV.).

The set of support services which are described in what follows may be either available as additional services that build on top of the Operating System, or they may be more tightly integrated with the Operating System core services and kernel, as it is also evident from the above discussion about specific types/models of OS proposed in various application contexts.

CLOUD SYSTEMS

In cloud systems, one must distinguish between the type of service offered: infrastructure (IaaS), platform (PaaS) or software (SaaS). In the latter two cases, the environment focuses on providing one specific service (type) and the according run-time support typically consists of an execution engine (such as the Google Docs App[37]) that is available on all systems. Main task therefore consists in distributed data management which poses different requirements on consistency management and transportation. There are no standard solutions though: for example, Amazon relies on eventual consistency, eBay executes a plain merge, data is replicated in the background etc. [95][194]. Due to the specialisation, the system can focus on a subset of requirements and find appropriate solutions.

Of more interest in this context are the general purpose goals pursued by Infrastructure as a Service providers, such as Amazon EC2[38], Microsoft Live[39] etc. They are either realised through virtualisation engines or middleware components (Amazon EC2, OpenNebula[40]) or using their own dedicated operating system (MS Azure[41], Google Android[42]). Each compute instance thereby is effectively a full blown server system with the full instance of the operating system / engine / image running (cf. Grid)

36 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.
37 More information is available at: http://code.google.com/appengine/docs/whatisgoogleappengine.html.
38 More information is available at: http://aws.amazon.com/ec2/.
39 More information is available at: http://www.live.com.
40 More information is available at: http://www.opennebula.org/.
41 More information is available at: http://www.microsoft.com/windowsazure/.
42 More information is available at: http://www.android.com/.

GRID(S)

Similar to cloud systems, most Grid environments provide their own middleware (Globus Toolkit 4[43], gLite[44] etc.) which is sometimes embedded in the operating system (ExtreemOS[45], .NET Framework[46]). Effectively each endpoint (Grid node) hosts its own full-fledged grid system that deals with communication and instance management. As opposed to clouds, Grids are mostly general purpose oriented – accordingly, the focus of existing middleware and frameworks rests on standard interfaces and communication support, rather than efficiency. The respective frameworks *do* scale comparatively well with respect to communication efficiency, but only to a limited degree with respect to instances, as they expect at least one full instance to be available per node.

Scheduling in both grids and clouds relies on a more or less centralized instance responsible for the respective service in order to ensure manageability. There are some approaches towards distributed scheduling [120] which however do not have to consider consistency across instances, so that the scheduling really just deals with a "scale-out on demand". For further information see e.g. [195][196][197][198]

HIGH PERFORMANCE COMPUTING

Current HPC usage takes a special position in the field of computing: with its focus primarily on efficiency, HPC jobs do generally not concurrently share resources. In other words, processes run exclusively on their respective computing environment (core) and across multiple compute units, only one such job has to be executed. Implicitly, the environment has comparatively simple scheduling and hardly any time-sharing tasks to perform.

With the classical HPC infrastructure, consisting of multiple single-core processors, each compute unit had sufficient cache available to host a full environment including the relevant code and data, so that only little memory swaps (and hence communication overhead) occurred. With the multi-core systems, resources are shared per processor, so that additional overhead builds up which is currently not catered for in the systems.

The environment is accordingly very bare-bone and offers little capabilities additional to abstracting the hardware and managing system calls. It is generally expected that the developer in some way has to cater for data segmentation, code distribution, concurrency and consistency etc. as there are no other processes running at the same time. In most cases, however, the according capabilities are actually build and provided by the compiler (cf. Programming Models).

Both compiler and execution environment are therefore explicitly adapted to the (hardware) environment they are running on, to make the most use of the communication and setup specifics. Higher-level details, such as memory layout generally need to be respected by the developer too, though, in order to achieve maximum performance. According effort is vested into converting the operating system (typically Linux) and the compilers front-end (mostly building up on LLVM) to the environment.

The heterogeneity of new cluster systems poses according problems.

Summary:

- Most scale and heterogeneity management takes place on the middleware layer
- Middlewares and frameworks are too inefficient for HPC usage

43 More information is available at: http://www.globus.org.
44 More information is available at: http://glite.web.cern.ch/glite/.
45 More information is available at: http://www.xtreemos.eu/.
46 More information is available at: http://msdn.microsoft.com/netframework/.

- Cache limitations and system heterogeneity make “one environment per unit” impossible
- Much effort is vested into adaptation of the run-time support to the respective system

V. Parallel and Distributed Programming Models

Optimum exploitation of parallelism in applications – in particular when efficiency is crucial – is an NP hard problem [61][62][63] and therefore generally requires an experienced human user to execute the according parallelisation tasks. Before the introduction of multi-core processors back in 2006, parallel development was mostly restricted to users of supercomputers (parallel clusters), as they were the only ones to draw benefits from this approach. Accordingly, parallel and the more common (sequential) programming models diverged slightly: with the latter looking in particular at simplicity, parallel models aimed particularly at efficiency to make the most of the system capabilities.

Recently, multi-core computing are also omnipresent in the domain of home computing (nowadays it is becoming increasingly difficult to buy a single-core machine). Therefore, in the world of common programming models that need to be at reach of most developers, parallelism is becoming an increasingly important issue. Unfortunately, when adding parallelism to an application, a number of issues arise which need to be properly addressed, in order to preserve a correct operation of the program. These range from how to properly split the functionality across concurrently running threads, to how to properly synchronize them for accessing shared data structures and to how to avoid race conditions and deadlocks.

1. Common Programming Models

Compared to parallel programming, sequential programming models have made quick progress and adapted perfectly to the mixed needs of developers, hardware and operating system, so that now most programs are fairly portable and easy to learn, whilst they show performance suitable to the environment needs – obviously, higher efficiency comes at the cost of complexity and higher portability at the cost of efficiency. We will spare the reader a detailed overview over the most relevant current languages, such as C/C++, Java, C#, etc. So far there have been little efforts though to seriously take over parallel concepts into the common programming language world – mostly for the problem of maintaining simplicity whilst supporting parallelism.

1.a) Process- or Thread-Based Parallelism

The most classical approaches to dealing with parallelism in common development consists in splitting the program up into individual threads that exchange data with each other at dedicated communication points. Multiple libraries extend common languages (mostly C) with such low-level capabilities for dealing with parallelism. Most of them build up on the POSIX standard [78] which embeds a set of well-known UNIX library calls for the C language for coding parallelism by means of multiple processes (e.g., by using the fork() system call) and synchronisation mechanisms (such as signals, semaphores and FIFOs). However, multiple processes have different virtual memory spaces, what forces the programmer to use message-passing among processes, or shared-memory segments in memory (e.g., `shmget()` and `shmat()` system calls of the same standard), both suboptimal solutions. In the context of a single parallel application, multiprocessing is deprecated in favour of multithreading, largely supported by nowadays Operating Systems, where multiple concurrent threads of execution share the same memory space and the context switch among threads is lighter than among processes.

For example, **POSIX Threads** constitutes a library allowing C programs to manage multiple threads of parallelism and manage synchronisation among them. The concept of threads is already well known and will therefore not be elaborated here. For more details, check out Barney's tutorial [89]. Pure thread-building is however fairly complicated and therefore in contradiction with maintaining simplicity in common programming languages. Many efforts building up on this principle therefore exploit the object-orientation capacities of C++ allowing to instantiate objects as threads and supporting their communication through dedicated libraries – this is for example the case in KAAPI[47]. Many more modern languages natively incorporate threading for parallelism and synchronisation, being designed for a higher level of programming abstraction - this is for example the case for Java and C# from the .NET framework.

The main problem with pure threading support consists in the comparatively weak fine-tuning and alignment to the infrastructure: they produce substantial messaging and protocol overheads, as well as leading to delays due to misalignment of communication between threads.

Due to the high degree of complexity to actually split up a task in potential threads that actually exploit the multi-core environment - many libraries and extensions therefore build up on these capabilities by incorporating higher-order functionality that can be parallelised by the compiler (e.g. Intel's TBB[48]) or at run-time (in particular Microsoft Parallel Extensions and the Java Concurrency Utilities). This also allows the framework to take more care of thread alignments according to predefined templates.

Microsoft Parallel Extensions[49] introduce so-called “tasks” as a concept similar to threads that act in particular on database queries and according loops by splitting the query into tasks according to the size of the loop. By using a task manager, these thread-like instances can be spread out across the infrastructure according to availability. Obviously, the performance gain depends on their individual work load as opposed to the messages they have to exchange – short, small tasks will obviously not be very beneficial.

The developer can even write his / her own tasks by explicitly initiating and executing them, much like threads. The main difference being that .NET lends from the influence of functional programming in its development, so that tasks are actually closer to lambda operators and can be treated accordingly[50]. As will be discussed in more detail in D5.2 in the context of development trends in programming languages, the functional (declarative) approach towards programming benefits from parallelism which procedural (imperative) approaches have more difficulties with (see e.g. [121][122]). It should be noted in this context that still less people make use of declarative than of imperative programming languages [51,52], non-regarding these advantages – this may change once parallel programming finds more common usage.

Similarly, in the Java domain the **Concurrency Utilities[53]** provides thread management features to Java. As opposed to their .NET counterpart, the concurrency utilities primarily deal with the *user-initiated* declaration, instantiation and execution of threads and does not aim at automatic scaling support, for example in the case of database queries. Accordingly, the programming effort is comparatively higher, as the developer has to ensure proper synchronisation and coordination.

Whilst the average efficiency achieved through these extensions will exceed the one by pure thread management for the average developer (not used to parallelism), they at

47 Kernel for Adaptative, Asynchronous Parallel and Interactive programming - http://kaapi.gforge.inria.fr/
48 Threading Building Blocks - http://software.intel.com/en-us/intel-tbb/
49 More information is available at: http://msdn.microsoft.com/concurrency
50 More information is available at: http://msdn.microsoft.com/en-us/library/bb397687.aspx.
51 More information is available at: http://www.devtopics.com/most-popular-programming-languages/
52 More information is available at: http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html
53 More information is available at: http://java.sun.com/j2se/1.5.0/docs/guide/concurrency/overview.html

the same time restrict usage to specific, typical parallelisable functions, which are not always employed in the average application. What is more, with little (to no) expertise by the developer, there is a high risk of over using these functions in cases where the implicit messaging overhead will exceed the parallelisation benefit, e.g. for large loops over small operations etc.

Frequent attempts are therefore directed at semi-automatic parallelisation which will maintain the maximum of sequentiality and simplicity – however, as noted, automatic parallelisation is NP hard and therefore not generally the best approach towards covering the efficiency issue.

On a related note, one of the languages with a good native support for concurrency and synchronisation is the Ada language [277]. However, Ada is generally used in the context of embedded and real-time applications, thus it lacks a powerful and rich run-time support such as found in modern development environments.

1.b) Distributed Programming

Recently, with the advancements in Web Services and Service-Oriented Architectures, standard messaging has become widespread as a basis for distributed (as opposed to parallel) computing, being greatly advanced and simplified for the user. Since the introduction of the Grid and the success of Web Services, many commercial and community based enhancements to common programming models have been developed that enable the user to communicate with remote service instances via automated, standard-based messaging (see also sections IV. 1.d) and in particular IV. 6.). The basic idea behind (web-based) distributed programs is more related to dynamically linked libraries than to parallel threads and / or processes, but can be exploited in a similar fashion, taking the additional communication overhead (typically SOAP based) and latency (Ethernet) into consideration.

Given the interconnect problems, distributed, service-oriented environments can in particular be exploited to “outsource” tasks that local resources cannot provide for, i.e. employing remote resources following the principles of utility computing. This has been extensively researched (and actually employed) in the context of so-called “***Virtual Organisations***” (see e.g. [102][103][182]). In these cases, each participant in the collaborative environment will provide (host and execute) one ore multiple tasks in the form of services that can be invoked remotely. In its most basic form, a workflow defines the order of tasks to be executed, i.e. services to be invoked – this can essentially be regarded as a higher level algorithm or program logic. A typical description logic to this end is WS-BPEL (Web Services Business Process Execution Language) [183].

Obviously, such workflows can incorporate parallel steps where multiple services are invoked at the same time, i.e. follow parallel strands. The classical business process specification does not foresee full distribution and hence no direct communication between individual services, so that the workflow engine host (typically the user) will act as an intermediary for all communication. Notably, there are concepts for full distribution of the workflow to separate participants, thus allowing direct communication (see e.g. [184]) – however, there is very little uptake of these approaches, mostly due to the lack of control and supervision.

A business process of this kind therefore shows principal similarity with a threaded application in so far, as any process can effectively be represented by a workflow. With the basic idea behind the Grid being the dynamic outsourcing of tasks and potentially even migration of code on the fly, the approach can principally be used to dynamically scale out an application, thus distributing the actual logic and executing it in parallel.

This requires an according platform in the respective environments, respectively pre-installation of the logic and invocation on the fly. There are multiple approaches along that line, ranging from higher level extensions, such as Parallel C#[54] to enabling HPC parallel programming models such as MPI over the Ethernet (see e.g. [185][186][188]) – see also section V. 2. on HPC programming models below.

The main problem with distributed programming models as opposed to parallel programming concepts consists in the implicit latency between communication endpoints, which is generally much higher in the environments for distributed program execution, as in those for parallel execution. Typically, the prior acts over the internet, whilst the latter requires Infiniband or higher (see also section II. 4.). Due to these communication constraints, distributed computing deals more with separate tasks with little communication dependencies between them. In its simplest form, a distributed process consists of a sequence of individual tasks that each gather input from a preceding task and pass it on to the next.

Accordingly, the main point to this type of environment is not so much efficiency in light of communication and synchronisation, but work offloading. It can thereby exploit the specific capabilities of services and / or resources as they are available – this means that specific characteristics of the overall process can be addressed on task-level in this environment [103][187] (see also service discovery protocols in section III. 2.).

2. HPC Programming Models

As the programming model is only as efficient as the underlying compiler, compilers for parallel programming are typically extremely specialised to the underlying hardware - they typically look for specific patterns in the code that are best suited to the system characteristics (e.g., for loop unrollment).

Over time, in particular the compilers improved more and more through community effort – therefore, as a general rule, “older” programming models typically lead to better performance figures than more modern ones, for even though newer ones typically address outstanding issues and potentially are better suited for modern hardware architectures, they still do not compete with the years of experience in development of the “classical” languages.

Accordingly, programming languages in the HPC domain change only very rarely and the typical “maintenance” duration of a language is more than 10 years – i.e. the community will use almost solely a specific programming language for almost 10 years, before new models are considered. During this time, the new models are adopted by compiler developers and experienced developers who provide the necessary support for the language.

So far, MPI and OpenMP on top of C have been the prevalent programming models in High Performance Computing with some remainders of Fortran still being notable. Notably, these languages are effectively parallel extensions to the original programming language, so that e.g. Fortran has been extended with explicit parallel elements that indicate functions that essentially show no dependencies and can hence be executed in parallel [123]. Along the same line, MPI and OpenMP extensions exist for both C and Fortran.

As has been noted, the primary problem of parallel programming consists in controlling synchronisation and message exchange which not only takes away most of the performance if not done accurately, but which also poses the most problems to the developers: in classical shared memory architectures or where cache is maintained

54 More information is available at: http://www.parallelcsharp.com/.

coherent by the hardware (like in Intel current multi-core processors), synchronisation maintenance effectively means locking memory until the synchronisation point, after which concurrent access is again allowed. Obviously, this may cause deadlocks and inconsistency if not programmed carefully, but the developer does not actually have to cater for how the memory is updated to achieve coherency. In large distributed clusters, this means however that the system needs to provide some hardware support to maintain (distributed) coherency.

The **Open Multi-Processing (OpenMP)**[55] programming interface originated in 1997 and it relies strongly on the fact that the system provides some form of shared memory. It provides the means to decompose the work very much like programming threads, so that, e.g., a loop can be unrolled into parallel threads. OpenMP does not support the decomposition of data and offers little (to no) control to the user in this direction, which often causes problems when mapping the code and data structure to the system, so that efficiency can be seriously affected. Since messaging is implicit, OpenMP may produce overheads that affect in particular small loops [124][125]. As opposed to MPI though, OpenMP (like most shared-memory models) allow easier transfer of serial code to the parallel environment.

The **Message Passing Interface (MPI)**[56] model (originated 1992) builds on distributed memory machines where communication between threads / processes has to occur by exchanging messages. MPI itself just specifies the operations to exchange messages synchronously and asynchronously between the parallel instances in different forms (broadcast, scatter etc.) It does *not* standardise the actual message format or links to one specific implementation, so that system-specific capabilities and restrictions can be employed by the compiler.

Over time, MPI has accumulated additional features that go beyond just message passing – whilst these operations suit experts' needs, it makes the standard very difficult to master for beginners. In fact, it requires a lot of experience to achieve efficient load balancing and reduce the messaging overhead [124]. As will be discussed in more detail below, programming MPI is also not very intuitive to the average user, and it is generally more difficult to port serial code to MPI [124]. Also, MPI is quite difficult to debug, even with the support of tools such as MARMOT [126]. Performance of MPI is directly affected by the communication model of the system and the message overhead produced by inexperienced programmers – in shared memory models, MPI generally shows bad performance compared to OpenMP due to this overhead [127].

Due to the specifics of OpenMP (shared-memory) and MPI (message-based), it is not unusual to actually employ a hybrid programming approach, where OpenMP is employed in a local shared memory environment (cores) and MPI to pass data over the network (nodes).

Both MPI and OpenMP are still the most used programming language in HPC, non-regarding their age (more than 10 years at the time of writing this). More recently (since roughly 2003) an alternative programming standard has emerged: **Partitioned Global Address Space (PGAS)**[57] which essentially exposes a “virtually” shared global memory that allows access from and across all threads / processes. It essentially partitions a global address space and assigns it to individual compute units where essentially threads (cf. Pthreads) are executed – obviously, access to local blocks is fastest, but the threads are allowed to access remote memory too, where the compiler has to take care of how remote access (message passing or shared memory) is realised.

55 More information is available at: http://openmp.org/wp/.
56 More information is available at: http://www.mcs.anl.gov/research/projects/mpi/.
57 More information is available at: http://pgas.org/.

PGAS is still in progress, even though the first implementations appeared more than 5 years ago. The Fortran implementation (CAF – Co-Array Fortran[58]) shows best performance so far, even comparable to MPI [128] – this is again due to the fact that there is more knowledge in the efficient parallelisation of Fortran than in other languages, e.g., in C. Accordingly, C (UPC – Unified Parallel C[59]) implementations (such as by the Berkeley University[60] or HP[61]) (still) show worse performance than CAF [129].
We will discuss PGAS again in the context of development trends (D5.2).

3. Transactional Memory

Transactional Memory (TM) has recently been proposed as a parallel programming paradigm to take benefit from modern multicore architectures. In contrast to traditional paradigms like lock-based techniques, TM uses speculative execution of transactions for simplicity reasons: semantics is preserved under transaction composition.
The power of the TM paradigm is twofold. First it lies in its abstract nature: no need to know the internals of shared object implementations - it suffices to delimit any critical sequence of shared object accesses using transactional boundaries. This avoids the most common difficulties related to parallel programming known as data race detections, deadlocks, livelocks.
Second, the code produced using TM is straightforwardly extensible: there is no need to break existing code abstractions and one can reuse a parallel library without modifying it or precluding concurrency. For instance, a TM- based set abstraction that is implemented from a binary tree data structure can be enriched to provide a collection abstraction without compromising atomicity or parallelism.
Besides making code development scalable, the adoption of TM thus makes parallel programming simple enough for average programmers and ensures reusability of parallel software components as libraries.
The idea of the transactional memory (TM) originally comes from [147] before a hardware [144] and a software implementations were proposed [154]. More recently, other hardware transactional memories have been suggested [137][150] but keep being limited by the existing physical constraints, like the cache size. Attempts to cope with these issues exist [132][133][152], however, software extensions seem necessary to keep the appealing simplicity of the TM and to face heterogeneity of future architectures.
Operating systems for many-core architectures, as investigated in the S(o)OS project, will have to face the inherent parallelism available on the chip, considering both communications among cores through shared-memory , and communications among machines through message-passing. First, we discuss the achievements in the context of shared memory architectures (STMs), second we present the recent achievements in the context of message passing (DTMs).

STMS

The first purely software TM (STM) [154] considered that the locations accessed by a transaction were known prior to its execution. In order to support dynamic transactions where the transaction control flow could depend on the value returned by the latter memory access, alternative software TMs have naturally been proposed [142][146]. Whereas these TM implementations relied only on locks and universal primitives to

58 More information is available at: http://www.co-array.org/.
59 More information is available at: http://upc.gwu.edu/.
60 More information is available at: http://upc.lbl.gov/download/index.shtml.
61 More information is available at: http://h21007.www2.hp.com/portal/site/dspp/menuitem.863c3e4cbcdc3f3515b49c108973a801?ciid=a53e82a454348110VgnVCM100000275d6e10RCRD.

synchronize concurrent transactions, new time-based TMs were proposed to improve performance by validating a transaction depending on the time instants at which it could commit [138][139][140][153].
Yet, some parallel programs tuned by expert programmers could reach higher concurrency than the traditional transactional model shared by all these “easy to program with” TMs. Consequently, several research work [141][143][145][148][151] extended the classical transactional model in order to account for the semantics of the application and increase concurrency. Nesting models [151][143] consider high-level operations as nested transactions, so that they can either commit or abort (and roll-back) only subparts of a transaction. Transactional boosting [145] and coarse-grained transactions [148] benefit from commutativity of application-level operations to avoid false-conflicts and enhance concurrency.
Only very recently, have new TM implementations shown better results than usual synchronization techniques [141][135] by relaxing consistency at read/write levels while retaining programming simplicity. We believe that the upcoming challenge is to integrate these TM mechanisms with future communication-aware DTMs without impacting on their efficiency.

DTMS

Several DTM implementations have recently been proposed to run on clusters. In [149] the authors present a DTM that uses speculation to execute the distributed transactions issued by one node as well as transactions running on a single node. Their approach exploits a conflict-aware scheduler that spreads the transactional request so that update transactions are executed by some master node locally, and read-only ones are executed on slave nodes. This limits the invalidation of read-only transactions as they can return an old snapshot without requiring slave nodes to maintain multiple versions. In contrast, the update transactions create a local transactional memory copy at the master node to buffer updates that remain invisible during the transaction execution. Upon commit, the update transaction acquires a global lock on the cluster and sends its modifications (as a “diff” between the two copies) to the other nodes of the system. After reception of the acknowledgement of all other nodes, the transaction can commit.
In [134] the authors have identified the existing transactional memory designs that are well-suited for clusters where the memory access time is non-uniform among the nodes. Among these designs, they confirmed the need for a node to acknowledge a transaction from a distant node prior to effectively applying its changes locally - as noticed in [149], this allows the distant transaction to commit early. In addition, they identified that making a read visible to other nodes (by for instance using concurrent-readers/exclusive-writer locks) and deferred updates (where all updates are bufferized until commit-time) can perform well when remote operations dominate the latency. D2 STM [136] aims at bringing fault-tolerance to the existing cluster-based STMs. Instead of acquiring a global lock on the cluster, D2 STM uses an atomic broadcast primitive that ensures the serialization of commits among concurrent transactions.
Tomorrow’s many-core architectures will have to support highly scalable services that neither rely on atomic broadcast nor on centralized control.

4. Compilers

Programming languages hardly ever act directly upon the hardware – instead a compiler has to translate the language into hardware instructions first. Since every infrastructure has its own specifics regarding instruction set, but also, more

importantly, regarding strengths and weaknesses in most cases it is the compiler that performs a major task in optimizing the code for the specific platform. There are multiple optimization techniques employed by different compilers, ranging from exploitation of memory hierarchies and operation reordering to identification of typical functions and analysis of code dependencies [189]. As such, for example the Intel compiler employs three main methods for optimisation:

(1) in the *profile-guided optimization (PGO)* method, the compiler is provided with the results from a test run of the code with a selected input set (a profile) – this information can be used to identify frequently executed program code. The compiler typically uses this data to reorder the blocks, thus reducing cache misses . Notably, the representative input sets have to be provided by the user – if the actually used data leads to different code behaviour, profile-guided optimization may actually reduce code performance (this method is employed by Intel, GCC, Microsoft Visual C++).

(2) *High-level optimizations (HLO)* actually imply a set of optimization techniques performed on the source-code (or an intermediary higher-level code, cf. below). This includes steps such as loop unrollment, loop fusion etc. Due to the high degree of complexity [189], the according analysis may take most of the compilation time and the actual gain is highly dependent on the executing architecture, restricting the portability of the code (even on this higher level)

(3) Similarly, *interprocedural optimization (IPO)* steps actually incorporate a set of techniques but on a broader code scope than (2), i.e. these steps try to analyze the whole program. In particular, the IPO strategies tries to reduce or eliminate recurring procedures by inlining the respective code, thus reducing overhead for branching and potential memory swaps. This technique typically also includes elimination of unused code and parameters etc. These sets of techniques are typical for most C compilers and can be found e.g. in GNU GCC, Intel Compiler and Microsoft's C compiler.

There are overall more than 60 well-known compiler optimization strategies, which shall not all be listed here – for a more comprehensive overview over all strategies, please refer to Wikipedia[62,63]. We can thereby specifically distinguish between hardware-specific and general optimisation strategies, whereas the latter (such as IPO and partially HLO) focus on steps that improve (or should improve) the execution performance non-regarding the platform on which the code is running, and hardware-specific strategies (such as PGO and some HLO) utilize specific characteristics of the hardware (such as rd-length in vector processors) for reducing unnecessary steps and delays in the code. Obviously, the latter reduce portability of the code enormously.

Generally, the lower the level of the programming language, the more hardware-bound the according code – e.g. C is less hardware-specific than Assembly code. Most modern programming languages build up on existing ones, such as C# basing on C++ and parts of Java. It is therefore not unusual for a compiler to traverse down this hierarchy when compiling the source code, thus producing intermediary versions which become increasingly hardware-specific. Many modern compilers build on this principle, thus employing compiler logic and knowledge that has already been build up in “older” languages (such as C and Fortran). Typically, this means that the compiler actually executes multiple passes (***multi-pass compiler***) in order to convert the source code into respective “lower-level” languages and execute the according compilation on the respective code.

In such cases, most passes perform so-called “*source-to-source*” compilation steps, where actually no machine code is generated, but only a conversion takes place, where e.g. higher-level specific commands which integrate multiple commands from a lower-

62 More information is available at: http://en.wikipedia.org/wiki/Compiler_optimization.
63 More information is available at: http://en.wikipedia.org/wiki/Category:Compiler_optimizations.

level language are converted into their according representative in that respective language. This step may also involve according high-level optimisation steps that are particular to the source language. The final step in such a hierarchical compilation approach always consists in producing the actual machine code.

Notably, we also speak of *"multi-pass"* compilation when the compiler produces assembly code right away but passes over the code multiple times in order to iteratively improve performance, identify dependencies etc. As opposed to that, a *"single-pass"* compiler generates the (final) assembly in a single pass over the source code. Whilst single pass compilers show much compilation speeds (as they only have to execute transformation once), they also typically generate less optimal code, as some optimisation points can only be identified after one pass annotating of the code.

The other major compiler technology next to the ones described above consist in ***Just-in-Time (JiT) Compilers..*** The basic idea behind these types of compilers consist in executing the actual conversion of the code at the moment of actual execution. Typically this applies to a whole code block at once, though in the worst case this could be applied to one command at a time – in these cases we speak of "*Interpreters"* or "*Virtual Machines"* though. In the context of this document, we treat them as special cases of JiT compilers, as the implications from this distinction are of little relevance in this context, even though we can note that interpreters and virtual machines typically show much lower performance.

In theory, JiT compilers are very similar to interpreters insofar as they translate a function the moment it is invoked, yet with the performance advantage that e.g loops do not need to recompile (interpret) the according set of operations every time, but only with the first iteration. According to this approach, the compilation would unload once it is removed from the memory – in other words, the compilation is not persistent. Obviously, this approach increases portability significantly, since hardware-specific optimisation and adaptation tasks are performed only at program execution time, and thus only once the code is actually in the according environment. However, even though this means in principle that more hardware-specific optimisation steps can be executed, the impact from the constant recompilation has a major performance impact. As already noted above, optimised compilation takes up a lot of time, in particular if IPO steps are executed that have to analyse the whole overall program structure.

In order to improve performance and reduce the recompilation effort, most JiT compilers take a hybrid approach, in which they create an intermediary code in a first transformation step, similar to source-to-source conversion in a multi-pass compiler (see above). More time can thus be vested into this first pass, with a focus on holistic optimization steps, such as pursued by the IPO steps, which take up most of the compile time. At the same time, this intermediary code still shows a maximum degree of portability and platform-independence, as the according steps are mostly performed in the last compiler steps (cf above).

Microsoft has improved the JiT technology with its release of the .NET framework and in particular the C# programming language significantly: the .NET framework generates a memory map at first invocation, keeping frequently used sections in cache, thus reducing overhead for recompilation and memory swaps [161].

Some compilers support programming model extensions dedicated to writing parallel applications, where the main issues are segmentation, distribution and catering for communication. There are different approaches to this end (see also Section V. 2.): with MPI, the programmer needs to focus on the communications between the application components and threads; with PGAS, the programmer may rely on a

shared-memory abstraction and process threads; with StarSS, the focus is on the definition of data dependencies; with OpenMP, the programmer may pretend to program sequentially, but he/she may add special directives along the code, instructing the compiler on how to create automatically the parallelisation logic; etc. All these extensions still require the programmer to define the details relating to parallelism.

Even though data and code segmentation are an important part of the performance on multi-core systems, there are currently no compilers or programming models that understand such annotations (for code and data distribution) effectively.

Finally, it is worth mentioning that, for those languages that support native language-level extensions for parallel programming, the compiler has the responsibility to generate the logic for handling multiple flows of control (usually creating new threads) and the synchronisation among them (for example, the a native support for concurrency and synchronisation is present in the Ada language, and partially present in the Java language). However, usually the language-level constructs for parallel programming aim to be simple to use and to be understood by the programmer, thus rarely they manage to provide the programmer with optimum and extremely efficient solutions. For example, language-level constructs usually do not include optimised read-write locks or other sophisticated locking strategies.

VI. BENCHMARKING

In order to estimate the performance of a system, direct measurements are typically not possible: too much depends on the actual usage and system load under stress. This is particularly true for computational performance, where typically reference applications are used as a means of performance indicator. In the following we will provide a quick overview over the main benchmarks used for estimating performance and their usefulness for the S(o)OS purposes.

HIGH PERFORMANCE COMPUTING

The "Top 500"[64] ranks high performance machines according to their performance – there is a general agreement, that any machine within the Top 500 list may thus safely be called a "supercomputer" (cf. section II. 4. on High Performance Computers). The performance rating is achieved through usage of an Linpack adaptation for distributed-memory computers (HPLinpack) – a reference implementation (HPL) existing by Jack Dongarra et al. [23]. HPL essentially solves random dense linear systems with double precision on the distributed machines. It bases on MPI and a set of predefined libraries (BLAS or VSIPL) to perform a set of calculations and then assess the execution time.

Even though it is often claimed that the order in the top 500 would not be affected by replacing the benchmark with another one, the results are clearly affected by the vendor-specific matrix-matrix operations which can thus be tuned for better performance on the Linpack benchmark. Whilst it is only fair that the vendor can make use of specific capabilities of the machine that only he knows (cf. Section II. 4.), it nonetheless means that the test is not objective (which it effectively cannot be). Furthermore Linpack is making strong use of broadcasts rather than explicit synchronisation points. Overall, Linpack focuses on specific mathematical functions which essentially drive a 100% processor load, but tells little about the mixed functionality requirements of typical applications, in particular from the more common areas, such as home usage (cf. D5.2)

Alternative benchmarking approaches are examined amongst others by the PRACE project[65] in order to reflect the actual usage patterns better and thus provide more meaningful performance figures. At the time of writing this, the PRACE project has only identified a set of relevant applications and no specific means of benchmarking though.

GENERAL COMPUTING BENCHMARKS

The Standard Performance Evaluation Corporation[66] provides a standardised set of benchmarks for specific hardware and computation aspects, such as the SPEC CPU2006 benchmark set[67] that tests the CPU performance for integer and double-precision floating point operations. The set of benchmarks do therefore not cover the full capability requirements of typical applications, but are particularly used for hardware developers and testers.

In order to provide usage-wide benchmark results, the most typical approach consists in agreeing on reference applications and comparing their specific characteristics against each other in different environments. The most common benchmarks following this direction are 3dMark [24] and PCMark [25] from Futuremark Corporation – both benchmarks effectively implement default applications in the area of graphics (3dMark)

64 More information is available at: http://www.top500.org.
65 Partnership for Advanced Computing in Europe - http://www.prace-project.eu/
66 More information is available at: http://www.spec.org.
67 More information is available at: http://www.spec.org/cpu2006/.

and home usage (PCMark) and compare the execution performance with other (known) system results. The resulting performance indicators are effectively relative and thus only meaningful, if the user is able to interpret them in comparison to different hardware settings.
As discussed in more detail in D5.2, the actual system requirements per application differ strongly according to the usage area – this is most visible in gaming versus office applications, but can also be noted in stream editing etc. No benchmark can address these differences equally – instead it will break down the overall requirements into a set of sub-features, such as average I/O access etc. PCMark introduced so-called test suites that reflect specific consumer behaviour, which combine the specific sub-features accordingly [25] - this provides a more meaningful performance indicator within the respective usage area.
Notably, since Windows Vista, Microsoft provides a reduced performance benchmark based on essential test parameters (memory access latency, floating point calculations per second etc.) - the according data is not representative for distributed (parallel) computing, but particularly useful for games, which partially already indicate their system requirements through this performance score.

CONCLUSIONS

Applications have different requirements towards the system, depending on their respective tasks and functional structure – as such, games require in particular high graphics throughput and use the CPU mostly for physics calculations; video and streaming applications require transcoding capabilities etc. (please refer to D5.2 for details). The performance of an application is therefore basing on the combined performance of the respective features needed in executing the processes. This follows the same principles as Amdahl's law [130]: speed-up of a distributed application in a parallel environment is determined by the ratio of the application parallel and serial aspects to the system parallel and serial capabilities; for the general application this means that its performance is determined by the ratio of the application algorithms to the system capabilities to support these algorithms.
Benchmark tests therefore do not necessarily have to be executed on the full application, but on its essential functional aspects. FutureMark identified aspects such as data encryption, data compression, video transcoding etc. as core test features [25] which leave the execution and implementation of the according features completely to the operating system. In the context of High Performance Computing, (cf. also section II. 4.), Asanovic et al. have identified the so-called 13 dwarfs as being the essential algorithmic kernels putting the actual performance demand on *distributed and parallel* computing [131]. There is, however, no benchmark suite explicitly dedicated to these dwarfs.
Note that in the specific context of S(o)OS, that does not aim at an implementation of a single full fledged operating system, but in particular at evaluating essential aspects of future highly scalable operating systems, no full benchmark suite is applicable. Instead, the benchmark must be capable of evaluating the influence of these approaches (such as a scheduling protocol) on to principle application execution. The benchmark(s) must therefore follow a line similar to PCMark and the 13 dwarfs and compare the effect of different algorithms on the respective core features (dwarfs or tests) – for example the impact of the scheduling protocols on parallel audio transcoding etc.

VII. Conclusions

In this document, a survey has been presented over the current state of the art in both industrial practice and academic research in the areas of parallel and massively parallel systems, and the support for such platforms at the Operating System and programming levels. From the presented discussion, it is clear that software development, both from the Operating System and from the application-level programming perspectives, struggles at following the hardware trends in the areas of massively parallel and distributing computing. The tera-scale future computing platforms of tomorrow, along with the envisioned developments in the areas of interconnection capabilities, parallelism level and application distribution, are going to constitute a challenging scenario for future ICT development. Methodologies for developing software need to be accordingly adapted to the new evolving scenario, starting from a re-engineering process that encompasses the entire software stack in a holistic way: from the foundations of Operating Systems kernel model and middleware services, to application-level software infrastructures and parallel programming paradigms. Only this rethinking of the software stack will allow for the needed departure from the classical sequential programming paradigms for moving towards novel distributed and parallel programming techniques on a large scale, in order to achieve an efficient use of the resources that will be available to the applications in the future. This deliverable, along with D5.2 “Definition of Future Requirements” [86], highlights the motivations underpinning the research that is being undertaken in the context of the S(o)OS project, and it serves as a basis over which the S(o)OS future investigations will be carried out.

REFERENCES

[1] L. Schubert, A. Kipp, B. Koller, and S. Wesner, "Service Oriented Operating Systems: Future Workspaces," IEEE Wireless Communications, vol. 16, 2009, pp. 42-50.

[2] L. Schubert and A. Kipp, "Principles of Service Oriented Operating Systems," Networks for Grid Applications, Second International Conference, GridNets 2008, P. Vicat-Blanc Primet, T. Kudoh, and J. Mambretti, Springer, 2009, pp. 56-69.

[3] L. Schubert, A. Kipp, and S. Wesner, "Above the Clouds: From Grids to Service-oriented Operating Systems," Towards the Future Internet - A European Research Perspective, G. Tselentis, J. Domingue, A. Galis, A. Gavras, D. Hausheer, S. Krco, V. Lotz, and T. Zahariadis, Amsterdam: IOS Press, 2009, pp. 238 – 249.

[4] L. Schubert, S. Wesner, A. Kipp, and A. Arenas, "Self-Managed Microkernels: From Clouds Towards Resource Fabrics," In Proceedings of the First International Conference on Cloud Computing, Munich, 2009.

[5] Intel, 'Intel White Paper. An Introduction to the Intel® QuickPath Interconnect'. http://www.intel.com/technology/quickpath/introduction.pdf (2009)

[6] Lameter, C.: Extreme High Performance Computing or Why Microkernels Suck, *in* 'Proceedings of the Linux Symposium'. (2007)

[7] Jeffrey Dean and Sanjay Ghemawat, MapReduce: simplified data processing on large clusters, Commun. ACM, Vol. 51 n.1, pp107-113, Jan 2008, issn 0001-0782

[8] Sanjay Ghemawat, Howard Gobioff, Shun-Tak Leung, The Google file system, Proceedings of the 19th ACM symposium on Operating systems principles (SOSP 2003), pp29-43, isbn 1-58113-757-5

[9] Andrew Baumann et al., The Multikernel: a new OS architecture for scalable multicore systems, SOSP 2009

[10] Pradeep Padala, Joseph N. Wilson, GridOS: Operating System Services for Grid Architectures, Proceedings of the International Conference on High-Performance Computing (HiPC 2003), FIXME

[11] Sanjeev Puri, Qamas Abbas, Grid Operating System: Making Dynamic Virtual Services in Organizations

[12] Hamid Hussain-Khan, Olivier Michielin, Xgrid, a “just do it” grid solution for non IT's, EMBnet.news, Vol. 11, Issue 3, September 2005

[13] Christine Morin, Renaud Lottiaux, Geoffroy Vallée, Pascal Gallard, David Margery, Jean-Yves Berthou, Isaac D. Scherson, Kerrighed and Data Parallelism: Cluster Computing on Single System Image Operating Systems, 6th IEEE International Conference on Cluster Computing, pp. 277-286, San Diego, California, September 2004

[14] Renaud Lottiaux, Benoit Boissinot, Pascal Gallard, Geoffroy Vallée, Christine Morin, OpenMosix, OpenSSI and Kerrighed: A Comparative Study, INRIA Technical Report N.5399, November 2004

[15] Amnon Barak, Shai Guday, Richard Wheeler, The MOSIX Distributed Operating System, Load Balancing for UNIX, Lecture Notes in Computer Science, Vol. 672, Springer-Verlag, 1993

[16] Qingbo Yuan, Jianbo Zhao, Mingyu Chen, Ninghui Sun, GenerOS: An Asymmetric Operating System Kernel for Multi-core Systems, IEEE International Parallel & Distributed Processing Symposium, Atlanta, USA, April 2010

[17] Christine Morin, Jerome Gallard, Yvon Jogou, Pierre Riteau, Clouds: a New Playground for the XtreemOS Grid Operating System

[18] Felix Hupfeld, Toni Cortes, Bjoern Kolbeck, Jan Stender, Erich Focht, Matthias Hess, Jesus Malo, Honathan Marti, Eugenio Cesario, The XtreemFS architecture – a case for object-based file system in Grids

[19] Tom Goodale, Shantenu Jha, Hartmut Kaiser, Thilo Kielmann, Pascal Kleijer, Gregor von Laszewski, Craig Lee, Andre Merzky, Hrabri Rajic, John Shalf, SAGA: A Simple API for Grid Applications. High-level application programming on the Grid , Computational Methods in Science and Technology, 12(1), 7-20, 2006. Online at: http://wiki.cct.lsu.edu/saga/.

[20] XtreemOS Project Deliverable D3.1.6 : Second Prototype of XtreemOS Runtime Engine , Version 1.0.3, January 5th, 2009

[21] Silas Boyd-Wickizer, Haibo Chen, Rong Chen, Yandong Mao, Frans Kaashoek, Robert Morris, Aleksey Pesterev, Lex Stein, Ming Wu, Yuehua Dai, Yang Zhang, Zheng Zhang, Corey: An Operating System for Many Cores, 8th USENIX Symposium on Operating Systems Design and Implementation

[22] David Wentzlaff and Anant Agarwal, Factored Operating Systems (fos): The Case for a Scalable Operating System for Multicores, ACM SIGOPS Operating System Review: Special Issue on the Interaction among the OS, Compilers, and Multicore Processors, April 2009

[23] Petitet, A.; Whaley, R.C.; Dongarra, J. & Cleary, A.: HPL - A Portable Implementation of the High-Performance Linpack Benchmark for Distributed-Memory Computers. Innovative Computing Laboratory (2008). Available at: http://www.netlib.org/benchmark/hpl/

[24] Korpi-Anttila, J.; Aalto, T.; Haapasalo, J.; Sjöholm, J.; Pietilä, K. & Kerminen, A.: 3DMark Vantage Whitepaper, Rev. 2. Futuremark Corporation (2008). Available at: http://www.futuremark.com/pressroom/companypdfs/3DMark_Vantage_Whitepaper_v100_Rev2.pdf?m=v

[25] N. Renqvist and P. Timonen. PCMark Vantage Whitepaper, V1.0. Futuremark Corporation, 2007. Available on-line at: www.futuremark.com/pressroom/companypdfs/PCMark_Vantage_ Whitepaper_v1.0_(PDF)

[26] Jochen Liedtke "On µ-Kernel Construction". Proc. 15th ACM symposium on Operating Systems Principles (SOSP), December 1995, pp. 237–250.

[27] Microsoft TechNet, “Windows NT 4.0 Workstation Architecture”, http://technet.microsoft.com/en-us/library/cc749980.aspx

[28] QNX Software Systems, “The QNX Neutrino Microkernel”, http://www.qnx.com/developers/docs/6.3.2/neutrino/sys_arch/kernel.html

[29] Alcatel-Lucent Bell Labs announces new optical transmission reocrd and breaks 100 Petabit per second kilometre barrier, September 28 2009, Paris, Press release.

[30] Jensen, M.; Nielsen, R.H.; Madsen, O.B.; , "Comparison of Cost for Different Coverage Scenarios between Copper and Fiber Access Networks," Advanced Communication Technology, 2006. ICACT 2006. The 8th International Conference , vol.3, no., pp.2015-2018, 20-22 Feb. 2006

[31] Gordon Haff, “Latency Matters”, Research Note, Illuminata, Inc, September 2002.

[32] David Patterson, “Why Latency Lags Bandwidth and What it Means to Computing”, High Performance Embedded Computing (HPEC 2004), Invited Talk, October 2004

[33] William J. Dally, “From Hypercubes to Dragonflies, a short history of interconnect”, IAA workshop, July 2008.

[34] Intel QuickPath Architecture, May 2010, available on-line at: www.intel.com/technology/quickpath/whitepaper.pdf.

[35] Hao Yu; I-Hsin Chung; Jose Moreira; , "Topology Mapping for Blue Gene/L Supercomputer," SC 2006 Conference, Proceedings of the ACM/IEEE , vol., no., pp.52-52, Nov. 2006

[36] J. Lawrence and Xin Yuan, "An MPI tool for automatically discovering the switch level topologies of Ethernet clusters," IEEE International Symposium on Parallel and Distributed Processing (IPDPS 2008), pp.1-8, 14-18 April 2008

[37] Infiniband Trade Association, http://www.infinibandta.org, as in May 2010

[38] Meddeb, A., "Why ethernet WAN transport?," Communications Magazine, IEEE , vol.43, no.11, pp. 136- 141, Nov. 2005

[39] Kim, J.; Balfour, J.; Dally, W.J.; , "Flattened Butterfly Topology for On-Chip Networks," Microarchitecture, 2007. MICRO 2007. 40th Annual IEEE/ACM International Symposium on , vol., no., pp.172-182, 1-5 Dec. 2007
[40] Ksentini, A.; , "Enhancing VoWLAN service through adaptive voice coder," Computers and Communications, 2009. ISCC 2009. IEEE Symposium on , vol., no., pp.673-678, 5-8 July 2009
[41] Rosenberg et al, SIP: Session Initiation Protocol, RFC3261
[42] Moskowitz et al, Host Identity Protocol, RFC 5201
[43] MPI: A Message-Passing Interface Standard, version 2.2, Message Passing Interface Forum, September 4, 2009
[44] Chi Zhang; Xin Yuan; Srinivasan, Ashok; , "Processor affinity and MPI performance on SMP-CMP clusters," Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on , vol., no., pp.1-8, 19-23 April 2010
[45] Walters, J.P.; Chaudhary, V.; , "Replication-Based Fault Tolerance for MPI Applications," Parallel and Distributed Systems, IEEE Transactions on, vol.20, no.7, pp.997-1010, July 2009
[46] Plan 9 Remote Resource Protocol, 9p2000 http://ericvh.github.com/9p-rfc/rfc9p2000.html, May 2010.
[47] Narayan, S.; Peng Shang; Na Fan; , "Network performance evaluation of Internet Protocols IPv4 and IPv6 on operating systems," Wireless and Optical Communications Networks, 2009. WOCN '09. IFIP International Conference on , vol., no., pp.1-5, 28-30 April 2009
[48] Khurri, A.; Kuptsov, D.; Gurtov, A.; , "Performance of Host Identity Protocol on Symbian OS," Communications, 2009. ICC '09. IEEE International Conference on , vol., no., pp.1-6, 14-18 June 2009
[49] A compact operating system for building cross-platform distributed systems, http://www.vitanuova.com/inferno/, as of May 2010
[50] Ionkov, L.; Van Hensbergen, E.; , "XCPU2 distributed seamless desktop extension," Cluster Computing and Workshops, 2009. CLUSTER '09. IEEE International Conference on , vol., no., pp.1-9, Aug. 31 2009-Sept. 4 2009
[51] Eric Newcomer, “Understanding Web services: XML, WSDL, SOAP, and UDDI”, Addison Wesley , ISBN-10: 0201750813 , 2002
[52] The NetBios Guide, http://www.netbiosguide.com/, as in July 2010
[53] Gsottberger, Y., Shi, X., Stromberg, G., Sturm, T.F., and Weber, W., Embedding low-cost wireless sensors into universal plug and play environments. In Wireless Sensor Networks. First European Workshop, EWSN 2004. Proceedings. (Lecture Notes in Comput. Sci. Vol.2920), pp. 291 - 306, 2004.
[54] Universal Plug and Play Forum, Universal Plug and Play Device Architecture, v. 0.91, http://www.upnp.org, Mar. 2000.
[55] C. Bettstetter and C. Renner, “A Comparison of Service Discovery Protocols and Implementation of the Service Location Protocol”, Proceedings of the Sixth EUNICE Open European Summer School: Innovative Internet Applications, EUNICE 2000, Twente, Netherlands, September, 2000.
[56] Cheshire, S. and Steinberg, D. Zero Configuration Networking: The Definitive Guide. O’Reilly Associates, ISBN 0596101007, December, 2005.
[57] Apple Inc, Bonjour in Mac OS X, http://www.apple.com/macosx/features/bonjour/, as in July 2010
[58] Stuart Cheshire, Marc Krochmal, Internet-Draft: "DNS-Based Service Discovery," 8 March 2010, Apple Inc. http://files.dns-sd.org/draft-cheshire-dnsext-dns-sd.txt, (June 2010).
[59] Schmidt, Douglas C.: “Reactor: An Object Behavioral Pattern for Concurrent Event Demultiplexing and Event Handler Dispatching”, Pattern Languages of Program Design, Addison-Wesley, 529–545, Eds: Coplien, James O., and Schmidt, Douglas C., 1995
[60] Pyarali, Irfan, Harrison, Tim, Schmidt, Douglas C., and Jordan, Thomas D.: “Proactor - An Architectural Pattern for Demultiplexing and Dispatching Handlers

for Asynchronous Events", The 4th Pattern Languages of Programming Conference, September 1997

[61] Koulamas, C.P., 1996. Scheduling two parallel semi-automatic machines to minimize machine interference. Computers and Operations Research 23 10, pp. 945–956.

[62] Scalable Computing: Practice and Experience. Scientific international journal for parallel and distributed computing. Published as Parallel and Distributed Computing Practices (ISSN 1097-2803), Vol. 4 No. 2, 2001, Book review

[63] Finding Least Constrained Plans and Optimal Parallel Executions is Harder than We Thought (1994) , by Christer Backstrom, Christer Backstrom, In Proc. 2nd European Workshop on Planning, 1993.

[64] R. Rajkumar, L. Sha, and J. Lehoczky, *Real-time synchronization protocols for multiprocessors,* in Proceedings of the Ninth IEEE Real-Time Systems Symposium, pp. 259--269, IEEE, 1988.

[65] L. Sha, R. Rajkumar, and J. P. Lehoczky, *Priority inheritance protocols: An approach to real-time synchronization*, IEEE Transactions on Computers, vol. 39, no. 9, pp. 1175--1185, 1990.

[66] R. Rajkumar, *Synchronization In Real-Time Systems -- A Priority Inheritance Approach*. Boston: Kluwer Academic Publishers, 1991.

[67] R. Rajkumar, *Real-time synchronization protocols for shared memory multiprocessors*, in Proceedings of the International Conference on Distributed Computing Systems, pp. 116--123, 1990.

[68] K. Lakshmanan, D. de Niz, and R. Rajkumar, *Coordinated task scheduling, allocation and synchronization on multiprocessors*, in Proceedings of IEEE Real-Time Systems Symposium, Washington, DC, USA, 2009.

[69] C. M. Chen and S. K. Tripathi, *Multiprocessor priority ceiling based protocols*, tech. rep., College Park, MD, USA, 1994.

[70] P. Gai, G. Lipari, and M. di Natale, *Minimizing memory utilization of real-time task sets in single and multi-processor systems-on-a-chip*, in Proceedings of the IEEE Real-Time Systems Symposium, IEEE Computer Society Press, December 2001.

[71] T. P. Baker, *Stack-based scheduling of real-time processes*, Real-Time Systems: The International Journal of Time-Critical Computing, vol. 3, 1991.

[72] J. M. Lopez, J. L. Diaz, and D. F. Garcia, *Utilization bounds for EDF scheduling on real-time multiprocessor systems*, Real-Time Systems: The International Journal of Time-Critical Computing, vol. 28, no. 1, pp. 39--68, 2004.

[73] P. Holman and J. H. Anderson, *Locking under pfair scheduling*, ACM Trans. Comput. Syst., vol. 24, no. 2, pp. 140--174, 2006.

[74] U. C. Devi, H. Leontyev, and J. H. Anderson, *Efficient synchronization under global edf scheduling on multiprocessors*, in ECRTS '06: Proceedings of the 18th Euromicro Conference on Real-Time Systems, Washington, DC, USA, pp. 75--84, 2006.

[75] A. Block, H. Leontyev, B. B. Brandenburg, and J. H. Anderson, *A flexible real-time locking protocol for multiprocessors*, in RTCSA '07: Proceedings of the 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, Washington, DC, USA, pp. 47--56, 2007.

[76] A. Easwaran and B. Andersson, *Resource sharing in global fixed-priority preemptive multiprocessor scheduling*, in Proceedings of IEEE Real-Time Systems Symposium, Washington, DC, USA, 2009.

[77] Dario Faggioli, Antonio Mancina, Fabio Checconi, Giuseppe 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

[78] IEEE Standard for Information Technology – Portable Operating System Interface (POSIX), IEEE Std 1003.1, 2004 Edition. Available online at: http://www.opengroup.org/onlinepubs/009695399
[79] Ian Pratt, XEN and the Art of Virtualization, 2006
[80] *Framework for Real-Time embedded Systems based on Contracts (FRESCOR)*, European Project No. FP6/2005/IST/5-034026, http://www.frescor.org.
[81] Michael González Harbour and Miguel Tellería de Esteban . Architecture and contract model for integrated resources II . FRESCOR Deliverable D-AC2v2 . 2008.
[82] Virtualization Overview, VMWare Whitepaper, available at the URL: http://www.vmware.com/pdf/virtualization.pdf.
[83] *Windows Server Virtualization – An Overview*, Microsoft Corporation, May 2006, http://download.microsoft.com/download/3/2/2/32212eab-a431-4cd4-8567-cf951b1322de/Virtualization.doc.
[84] Robert P. Goldberg, *Survey of Virtual Machine Research*, IEEE Computer Magazine, 7(6):34—45, 1974
[85] 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
[86] S(o)OS Project Deliverable – D5.2 Definition of Future Requirements, July 2010
[87] J. Dongarra, J. Bunch, C. Moler and G. W. Stewart, LINPACK Users Guide, SIAM, Philadelphia, PA, 1979.
[88] A. Petitet, R. C. Whaley, J. Dongarra, A. Cleary, HPL - A Portable Implementation of the High-Performance Linpack Benchmark for Distributed-Memory Computers, September 2008
[89] Blaise Barney, POSIX Threads Programming, Lawrence Livermore National Laboratory, available on-line: https://computing.llnl.gov/tutorials/pthreads
[90] Volkmar Uhlig, "The mechanics of in-kernel synchronization for a scalable microkernel," Operating Systems Review, 41:4, 2007, pp 49-58
[91] Andrew S. Tanenbaum, A Comparison of Three Microkernels, Journal of Supercomputing, Vol. 9, 1995
[92] Andrew S. Tanenbaum, M. Frans Kaashoek, Robbert Van Renesse, Henri E. Bal. The Amoeba Distributed Operating System – A Status Report. Computer Communications, Vol. 14, pp.324–335, 1991
[93] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrmann, C. Kaiser, S. Langlois, P. Léonard, W. Neuhauser. Overview of the CHORUS Distributed Operating Systems. Computing Systems Vol. 1, pp-39–69, 1991
[94] Richard Rashid, Daniel Julin, Douglas Orr, Richard Sanzi, Robert Baron, Alessandro Forin, David Golub, Michael Jones. "Mach: A System Software kernel," Proceedings of the 34th Computer Society International Conference (COMPCON 89), February 1989
[95] L. Schubert, K. Jeffery, B. Neidecker-Lutz, and others, "The Future of Cloud Computing," 2010. Available at: http://cordis.europa.eu/fp7/ict/ssai/docs/cloud-report-final.pdf
[96] C.G. Willard, A. Snell, L. Segervall, "Cloud Computing Opportunities in HPC", HPC Wire 2009. Available at: http://www.hpcwire.com/specialfeatures/cloud_computing/features/Cloud-Computing-Opportunities-in-HPC-68773637.html
[97] HPC Wire, "Special Feature: HPC in the Cloud", 2010. Available at: http://www.hpcwire.com/specialfeatures/cloud_computing/
[98] W. Gentzsch, "Grids or Clouds for HPC?", HPC Wire 2009. Available at: http://www.hpcwire.com/specialfeatures/cloud_computing/features/Grids-or-Clouds-for-HPC-67796917.html
[99] I. Foster, C. Kesselman, "The grid: blueprint for a new computing infrastructure", Elsevier, 2004

[100] F. Berman, G.C. Fox, A.J.G. Hey (eds), “Grid computing: making the global infrastructure a reality”, Wiley, 2003
[101] R. Buyya, M. Baker (eds), “Grid Computing – GRID 2000”, In: Lecture Notes in Computer Science, Springer, 2000
[102] Saabeel, W., Verduijn, T.M., Hagdorn, L., Kumar, K. “A model of virtual organisation: a structure and process perspective.” In: Electronic Journal of Organizational Virtualness, 2002. Available at: http://www.virtual-organization.net
[103] L. Schubert, S. Wesner, and T. Dimitrakos, "Secure and Dynamic Virtual Organizations for Business," Collaborative Product and Service Life Cycle Management for a Sustainable World - Proceedings of th 15th ISPE International Conference on Concurrent Engineering (CE2008), P. Cunningham and M. Cunningham, IOS Press, 2005.
[104] J. Haller, L. Schubert, and S. Wesner, "Private Business Infrastructures in a VO Environment," Exploiting the Knowledge Economy: Issues, Applications, Case Studies, P. Cunningham and M. Cunningham, IOS Press, 2006, pp. 1064-1071.
[105] G. Mateescu, “Overcoming the processor communication overhead in MPI applications,” In: Proceedings of the 2007 Spring Simulation Multiconference - Volume 2, 2007
[106] T. Leng, R. Ali, J. Hsieh, V. Mashayekhi, R. Rooholamini, “An Empirical Study of Hyper-Threading in High Performance Computing Clusters,” Linux Clusters: The HPC Revolution – Technical Paper, 2002. Available at: http://www.democritos.it/activities/IT-MC/cluster_revolution_2002/
[107] P.H. Worley, “Impact of Communication Protocol on Performance,” Oak Ridge National Laboratory, 1999. Available at: http://www.ornl.gov/~webworks/cpr/v823/rpt/104898.pdf
[108] F. Putrya, “On-chip interconnect impact on the performance of multicore processors with NUMA architecture,” European-Russian Semiconductor Technology Conference 2009 on Networking. Available at: http://www.eu-ru.info/index-Dateien/ERSTC2009_PDF/Session6/ERSTC2009_Session6_MIET_Putrya.pdf
[109] X. Du , X. Zhang , Z. Zhu, “Memory Hierarchy Considerations for Cost-Effective Cluster Computing,” IEEE Transactions on Computers, Vol. 49, 2000
[110] B. Barney, “Introduction to Parallel Computing,” Online publication, available at: https://computing.llnl.gov/tutorials/parallel_comp/
[111] M. Müller, “SPEC OpenMP Benchmarks on Four Generations of NEC SX Parallel Vector Systems”, In: OpenMP Shared Memory Parallel Programming, Vol 4315/2008, pp. 145-152, Springer, 2008
[112] K. H. Tsoi, W. Luk, “Axel: A Heterogeneous Cluster with FPGAs and GPUs,” In: Proceedings of the 18th Annual ACM/SIGDA international Symposium on Field Programmable Gate Arrays (Monterey, California, USA, February 21 - 23, 2010). FPGA '10. ACM, New York: 2010
[113] E. Focht, “Heterogeneous Clusters With OSCAR: Infrastructure,” In: Proceedings of the 4th Annual OSCAR Symposium, 2006. Available at: http://www.csm.ornl.gov/oscar06/proceedings/
[114] J. Hill, M. Bull, A. Simpson, “Identification and Categorisation of Applications and Initial Benchmarks Suite,” Partnership for Advanced Computing in Europe (PRACE). Available at: http://www.prace-project.eu/documents/PRACE-Applications.pdf
[115] J. Matthews, S. Trika, D. Hensgen, R. Coulson, K. Grimsrud, “Intel® Turbo Memory: Nonvolatile disk caches in the storage hierarchy of mainstream computer systems,” In: Trans. Storage 4, 2, 2008
[116] U. Drepper, “What Every Programmer Should Know About Memory,” Online publication, 2007. Available at: http://people.redhat.com/drepper/cpumemory.pdf
[117] D. Hackenberg, D, Molka, W.E. Nagel, “Comparing cache architectures and coherency protocols on x86-64 multicore SMP systems,” In: Proceedings of the

42nd Annual IEEE/ACM international Symposium on Microarchitecture (New York, New York, December 12 - 16, 2009). MICRO 42. ACM, New York: 2009

[118] C. Lameter, “Extreme High Performance Computing or Why Microkernels Suck,” In: Proceedings of the Linux Symposium, 2007. Available at: http://kernel.org/pub/linux/kernel/people/christoph/ols2007/performnce-vs-microkernel-paper.pdf

[119] L. Schubert, S. Wesner, A. Kipp, and A. Arenas, "Self-Managed Microkernels: From Clouds Towards Resource Fabrics," In Proceedings of the First International Conference on Cloud Computing, Munich: 2009.

[120] Z. Dadan, W. Xieqin, J. Ningkang, J, “Distributed Scheduling Extension on Hadoop,” In: Proceedings of the 1st international Conference on Cloud Computing (Beijing, China, December 01 - 04, 2009). Lecture Notes In Computer Science, vol. 5931. Springer-Verlag, Berlin: 2009

[121] Z. Budimlic, A.M. Chandramowlishwaran, K. Knobe, G.N. Lowney, V. Sarkar, V., L. Treggiari, “Declarative aspects of memory management in the concurrent collections parallel programming model,” In: Proceedings of the 4th Workshop on Declarative Aspects of Multicore Programming (Savannah, GA, USA, January 20 - 20, 2009). ACM, New York: 2009

[122] G. Blelloch, N. Glew (eds.), “DAMP '07: Proceedings of the 2007 workshop on Declarative aspects of multicore programming”, ACM, New York: 2007

[123] J. Rei, “The Advantages of Fortran 90,” In: Computing, Vol. 48, p. 219-238. Springer, New York: 1992

[124] L.A. Smith, “Mixed Mode MPI / OpenMP Programming,” Online publication. Available at: http://www.cslab.ntua.gr/courses/pps/files/Mixed_Mode_MPI-OpenMP_Programming-Tutorial.pdf

[125] G. Jost, H. Jin, D. an Mey, F.F. Hatay, “Comparing the OpenMP, MPI, and Hybrid Programming Paradigms on an SMP Cluster”, NAS Technical Report NAS-03-019, November 2003. Available at: http://www.nas.nasa.gov/News/Techreports/2003/PDF/nas-03-019.pdf

[126] B. Krammer, K. Bidmon, M.S. Müller, M.M. Resch, “MARMOT: An MPI Analysis and Checking Tool,” In: Proceedings of Parallel Computing 2003, PARALLEL COMPUTING: Software Technology, Algorithms, Architectures & Applications, pp. 493-500. Elsevier, 2004.

[127] B. Vanvoorst, S. Seidel, “Comparison of MPI implementations on a shared memory machine,” In: Proceedings of the 15th IPDPS 2000 Workshops on Parallel and Distributed Processing, pp. 847-854. Springer, 2000.

[128] J.K.Reid, J. M. Rasmussen, P. C. Hansen, “The LINPACK Benchmark in Co-Array Fortran,” Online. Available at: http://www2.imm.dtu.dk/documents/ftp/tr00/tr14_00.pdf

[129] C. Coarfa, Y. Dotsenko, J. Mellor-Crummey, F. Cantonnet, T. El-Ghazawi, A. Mohanty, Y. Yao, D. Chavarrıa-Miranda, “An Evaluation of Global Address Space Languages: Co-Array Fortran and Unified Parallel C,” In: Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. ACM, New York: 2005

[130] G. Amdahl, “Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities,” in: AFIPS Conference Proceedings (30), pp. 483–485. 1967. Available at: http://www-inst.eecs.berkeley.edu/~n252/paper/Amdahl.pdf.

[131] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, K. A. Yelick, “The Landscape of Parallel Computing Research: A View from Berkeley”, EECS Department - University of California, Berkeley, Technical Report No. UCB/EECS-2006-183, 2006. Available at: http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.pdf

[132] C. Scott Ananian, Krste Asanovic, Bradley C. Kuszmaul, Charles E. Leiserson, and Sean Lie. Unbounded transactional memory. In HPCA ’05.

[133] Jayaram Bobba, Neelam Goyal, Mark D. Hill, Michael M. Swift, and David A. Wood. Tokentm: Efficient execution of large transactions with hardware transactional memory. In ISCA ’08.

[134] Robert L. Bocchino, Vikram S. Adve, and Bradford L. Chamberlain. Software transactional memory for large scale clusters. In PPoPP ’08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming, pages 247–258, New York, NY, USA, 2008. ACM.
[135] Nathan G. Bronson, Jared Casper, Hassan Chafi, and Kunle Olukotun. Transactional predication: High performance concurrent sets and maps for STM. In PODC ’10: Proceedings of the 29th Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing, 2010.
[136] Maria Couceiro, Paolo Romano, Nuno Carvalho, and Lu´ Rodrigues. ıs D2STM: Dependable distributed software transactional memory. In PRDC ’09: Proceedings of the 15th IEEE Pacific Rim International Symposium on Dependable Computing, pages 307–313, 2009.
[137] Dave Dice, Yossi Lev, Mark Moir, and Daniel Nussbaum. Early experience with a commercial hardware transactional memory implementation. In ASPLOS ’09.
[138] Dave Dice, Ori Shalev, and Nir Shavit. Transactional locking II. In DISC ’06: Proceedings of the 20th International Symposium on Distributed Computing, September 2006.
[139] Aleksandar Dragojevi´, Rachid Guerraoui, and Michal Kapalka. Stretching transactional memory. In PLDI ’09: Proceedings of the ACM SIGPLAN 2009 Conference on Programming Language Design and Implementation, 2009.
[140] Pascal Felber, Christof Fetzer, and Torvald Riegel. Dynamic performance tuning of word-based software transactional memory. In PPoPP ’08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, 2008.
[141] Pascal Felber, Vincent Gramoli, and Rachid Guerraoui. Elastic transactions. In Proceedings of the 23rd International Symposum on Distributed Computing (DISC’09), volume 5805 of LNCS, pages 93–107. Springer-Verlag, Sep 2009.
[142] Tim Harris and Keir Fraser. Language support for lightweight transactions. In Object-Oriented Programming, Systems, Languages, and Applications, pages 388–402. Oct 2003.
[143] Tim Harris and Srdjan Stipi´. Abstract nested transactions. In The 2nd ACM SIGPLAN Workshop on Transactional Computing, 2007.
[144] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In Proceedings of the Twentieth Annual International Symposium on Computer Architecture, 1993.
[145] Maurice Herlihy and Eric Koskinen. Transactional boosting: A methodology for highly-concurrent transactional objects. In PPoPP ’07: Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, 2008.
[146] Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer, III. Software transactional memory for dynamic-sized data structures. In PODC ’03: Proceedings of the twenty-second annual symposium on Principles of distributed computing, pages 92–101, New York, NY, USA, 2003. ACM.
[147] Thomas F. Knight. An architecture for mostly functional languages. In Proceedings of ACM Lisp and Functional Programming Conference, pages 500–519. Aug 1986.
[148] Eric Koskinen, Matthew Parkinson, and Maurice Herlihy. Coarse-grained transactions. SIGPLAN Not., 45(1):19–30, 2010.
[149] Kaloian Manassiev, Madalin Mihailescu, and Cristiana Amza. Exploiting distributed version concurrency in a transactional memory cluster. In PPoPP ’06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming, pages 198–208, New York, NY, USA, 2006. ACM.
[150] Kevin E. Moore, Jayaram Bobba, Michelle J. Moravan, Mark D. Hill, and David A. Wood. Logtm: Log-based transactional memory. In HPCA ’06.
[151] J. Eliot B. Moss. Open nested transactions: Semantics and support. In Workshop on Memory Performance Issues (WMPI 2006), February 2006.
[152] Ravi Rajwar, Maurice Herlihy, and Konrad Lai. Virtualizing transactional memory. In ISCA ’05.

[153] Torvald Riegel, Pascal Felber, and Christof Fetzer. A Lazy Snapshot Algorithm with Eager Validation. In DISC '06: Proceedings of the 20th International Symposium on Distributed Computing, pages 284–298, September 2006.
[154] Nir Shavit and Dan Touitou. Software transactional memory. In Proc. Of the 12th Annual ACM Symposium on Principles of Distributed Computing (PODC), pages 204–213, 1995.
[155] Eliseu M. Chaves Jr., Prakash Das, Thomas J. LeBlanc, Brian D. Marsh, Michael L. Scott, "Kernel-Kernel communication in a shared-memory multiprocessor", Concurrency - Practice and Experience, 5:3, 1993, pp. 171-191
[156] H.C. Lauer and R.M. Needham, "On the Duality of Operating System Structures", ACM SIGOPS Operating System Review 13:2 (April 1979), pp. 3-19
[157] K. Dowd , C. Severance, "High Performance Computing (RISC Architectures, Optimization & Benchmarks," O'Reilly , 1998
[158] G. Hager, G. Wellein, "Introduction to High Performance Computing for Scientists and Engineers", CRC Press, 2010
[159] A. Grama , G. Karypis, V. Kumar, A. Gupta, "Introduction to Parallel Computing", Addison Wesley, 2003
[160] K. Koch, "Roadrunner Platform Overview," Roadrunner technical seminar series, 2008. Available at: http://www.lanl.gov/orgs/hpc/roadrunner/pdfs/Koch%20-%20Roadrunner%20Overview/RR%20Seminar%20-%20System%20Overview.pdf
[161] J. Mayo, "C# Unleashed," Sams Publishing, 2001
[162] [And90] Anderson, T.E. "The performance of spin lock alternatives for shared-memory multiprocessors", IEEE Transactions on Parallel and Distributed Systems, 1:1, pp. 6-16, 1990
[163] [Chav93] Eliseu M. Chaves Jr., Prakash Das, Thomas J. LeBlanc, Brian D. Marsh, Michael L. Scott, "Kernel-Kernel communication in a shared-memory multiprocessor", Concurrency - Practice and Experience, 5:3, 1993, pp. 171-191
[164] [Lauer79] H.C. Lauer and R.M. Needham, "On the Duality of Operating System Structures", ACM SIGOPS Operating System Review 13:2 (April 1979), pp. 3-19
[165] [Her90] M.P. Herlihy, and J.M. Wing "Linearizability: A correctness condition for concurrent objects", ACM Transactions on Programming Languages and Systems (TOPLAS), 12:3, pp. 463-492, 1990
[166] [Her91] M. Herlihy, "Wait-free synchronization," ACM Transactions on Programming
Languages and Systems, vol. 11, no. 1, pp. 124–149, Jan. 1991.
[167] [Her93] Herlihy, M. and Moss, J.E.B., "Transactional memory: Architectural support for lock-free data structures", Proceedings of the 20th ACM annual international Symposium on Computer Architecture, 1993.
[168] [Mell91] J.M. Mellor-Crummey, M.L. Scott, "Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors", ACM Transactions on Computing Systems, 9:1, 1991, pp. 21-65
[169] [Scott] M.L. Scott, W.N. Scherer "Scalable queue-based spin locks with timeout", Proceedings of the eighth ACM SIGPLAN symposium on Principles and Practices of Parallel Programming, 2001
[170] [Lim94] Lim, B.H. and Agarwal, A. "Reactive synchronization algorithms for multiprocessors", Proceedings of the 6th International conference on Architectural Support for Programming Languages and Operating Systems, 1994
[171] [Unrau] R.C. Unrau, O. Krieger, B. Gamsa, M. Stumm, "Experiences with locking in a NUMA multiprocessor operating system kernel", OSDI, pp. 139-152, 1994
[172] [McKen98] P.E. McKenney, and J.D. Slingwine, "Read-copy update: Using execution history to solve concurrency problems", Parallel and Distributed Computing and Systems, pp 509-518, 1998
[173] [Sun05] Sundell, H. and Tsigas, P. "Fast and lock-free concurrent priority queues for multi-thread systems", Journal of Parallel and Distributed Computing, 65:5, pp. 609-627, 2005, Elsevier
[174] [Hen09] Hendler, D. and Shavit, N. and Yerushalmi, L. "A scalable lock-free stack algorithm" Journal of Parallel and Distributed Computing, 2009, Elsevier

[175] [Muk93] Mukherjee, B. and Schwan, K. “Improving performance by use of adaptive objects: Experimentation with a configurable multiprocessor thread package”, Proc. of the Second International Symposium on High Performance Distributed Computing (HPDC-2), pp 59-66, 1993
[176] [MPI] Gropp, W. and Lusk, E. and Doss, N. and Skjellum, A. “A high-performance, portable implementation of the MPI message passing interface standard”, Parallel Computing, 22:6, pp. 789—828, 1996, Elsevier
[177] Uhlig, V., “The mechanics of in-kernel synchronization for a scalable microkernel”, ACM SIGOPS Operating Systems Review, 41:4, 2007
[178] Kranz, D. and Johnson, K. and Agarwal, A. and Kubiatowicz, J. and Lim, B.H. “Integrating message-passing and shared-memory: early experience”, ACM SIGPLAN Notices, 28:7, 1993
[179] Heinlein, J. and Gharachorloo, K. and Dresser, S. and Gupta, A. “Integration of message passing and shared memory in the Stanford FLASH multiprocessor”, Proceedings of the sixth international conference on Architectural support for programming languages and operating systems, 1994
[180] Chandra, S. and Larus, J.R. and Rogers, A., “Where is time spent in message-passing and shared-memory programs?”, Proceedings of the sixth international conference on Architectural support for programming languages and operating systems, pp. 61—73, 1994
[181] McKenney, P.E., “Selecting locking primitives for parallel programs,” Communications of ACM, vol. 39, pp. 75—81, 1996
[182] I. Foster, C. Kesselman, and S. Tuecke, “The Anatomy of the Grid - Enabling Scalable Virtual Organizations,” International Journal of Supercomputer Applications, vol. 15, 2001, p. 2001.
[183] OASIS, “Web Services Business Process Execution Language Version 2.0”, 2007 – available at http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.pdf
[184] P. Muth, D. Wodtke, J. Weissenfels, A.K. Dittrich, and G. Weikum, “From Centralized Workflow Specification to Distributed WorkflowExecution,” Journal of Intelligent Information Systems, vol. 10, 1998, pp. 159-184.
[185] Masaki Tatezono, Naoya Maruyama and Satoshi Matsuoka , “ Making Wide-Area, Multi-site MPI Feasible Using Xen VM”, in: Lecture Notes in Computer Science, Frontiers of High Performance Computing and Networking – ISPA 2006 Workshops, 2006
[186] S. Lakshminarayanan, S. S. Ghosh and N. Balakrishnan , “ Implementation of MPI over HTTP “, in: Lecture Notes in Computer Science, High-Performance Computing and Networking
[187] M.D. Wilson, L. Schubert, and A.E. Arenas, "The TrustCoM Framework V4," 2007. Available at: http://www.eu-trustcom.com/trustcom/wp-content/uploads/2007/08/D63_trustcom_framework.zip
[188] E. Gabriel, M. Resch, T. Beisel, and R. Keller, “Distributed Computing in a Heterogeneous Computing Environment,” Proceedings of the 5th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface, Springer-Verlag, 1998, pp. 180-187.
[189] K. Kennedy, R. Allen, “Optimizing Compilers for Modern Architectures: A Dependence-based Approach,” Morgan Kaufmann, 2001
[190] NVIDIA, “NVIDIA’s Next Generation CUDA TM Compute Architecture: FERMI”, NVIDIA Corporation Whitepaper, 2009. Available at: http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf
[191] S. Che, M. Boyer, J. Meng, D. Tarjan, J.W. Sheaffer, and K. Skadron, “A performance study of general-purpose applications on graphics processors using CUDA,” Journal of Parallel and Distributed Computing, vol. 68, Oct. 2008, pp. 1370-1380.
[192] M. Garland, S. Le Grand, J. Nickolls, J. Anderson, J. Hardwick, S. Morton, E. Phillips, Yao Zhang, and V. Volkov, “Parallel Computing Experiences with CUDA,” Micro, IEEE, vol. 28, 2008, pp. 13-27.

[193] C.L. Liu and J.W. Layland, “Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment,” J. ACM, vol. 20, 1973, pp. 46-61.
[194] W. Vogel, “Eventually Consistent – Revisited,” online publication – available at: http://www.allthingsdistributed.com/2008/12/eventually_consistent.html
[195] A. Kipp, S. Wesner, L. Schubert, R. Piotter, H. Schwichtenberg, C. Thomson, and E. Karanastasis, "A new approach for classifying Grids," 2007, p. 13.
[196] M.D. Wilson, L. Schubert, and A.E. Arenas, "The TrustCoM Framework V4," 2007.
[197] L. Schubert, S. Wesner, and T. Dimitrakos, "Secure and Dynamic Virtual Organizations for Business," Exploiting the Knowledge Economy: Issues, Applications, Case Studies, P. Cunningham and M. Cunningham, IOS Press, 2005.
[198] I. Foster, C. Kesselman, and S. Tuecke, “The Anatomy of the Grid - Enabling Scalable Virtual Organizations,” International Journal of Supercomputer Applications, vol. 15, 2001, p. 2001.
[199] Hennessy, J.L and Patterson, D.A., “Computer Architecture: A Quantitative Approach, 4th Edition”, Morgan Kaufmann, ISBN-10: 0123704901, 2006
[200] Intel, “Intel Core i7-800 Processor Series and the Intel Core i5-700 Processor Series, Based on Intel Microarchitecture (Nehalem)” http://download.intel.com/products/processor/corei7/319724.pdf
[201] Intel, “Intel Advanced Encryption Standard (AES) Instruction Set – Rev 3” http://software.intel.com/en-us/articles/intel-advanced-encryption-standard-aes-instructions-set/
[202] ARM, “The ARM Cortex-A9 Processors” http://www.arm.com/files/pdf/ARMCortexA-9Processors.pdf
[203] Texas Instruments, “TMS320C6713B Floating-point digital signal processors” http://focus.ti.com/lit/ds/symlink/tms320c6713b.pdf
[204] Texas Instruments, “TMS320C6452 Digital Signal Processor” http://focus.ti.com/lit/ds/symlink/tms320c6713b.pdf
[205] Analog Devices, “TigerSHARC Embedded Processor” http://www.analog.com/static/imported-files/data_sheets/ADSP_TS201S.pdf
[206] Analog Devices, “Getting Started with ” http://www.analog.com/static/imported-files/tech_docs/GettingStartedwithBlackfinProcessors.pdf
[207] Texas Instruments, “OMAP 4 mobile applications platform” http://focus.ti.com/lit/ml/swpt034/swpt034.pdf
[208] Tensilica, “Xtensa Architecture and Performance” http://www.tensilica.com/products/literature-docs/white-papers/xtensa-architecture.htm
[209] Burns, G.F. and Jacobs, M. and Lindwer, M. and Vandewiele, B., “Silicon Hive's Scalable and Modular Architecture Template for High-Performance Multi-core Systems” http://www.siliconhive.com/Flex/Site/Download.aspx?ID=1856
[210] Heysters, P.M. and Rauwerda, G.K. and Smit, L.T., “A Flexible, Low Power, High Performance DSP IP Core for Programmable System-on-a-Chip” In the proceedings of IP/SOC 2005
[211] PACT, XPP-III Processor Overview – White Paper, http://www.pactxpp.com/main/download/XPP-III_overview_WP.pdf
[212] Cosoroaba, A. and Frederic, R. (Xilinx), “Achieving Higher System Performance with the Virtex-5 Family of FPGAs”, White Paper: Virtex-5 Family of FPGAs, http://www.xilinx.com/support/documentation/white_papers/wp245.pdf
[213] Actel, “Design Security in Nonvolatile Flash and Antifuse FPGAs” http://www.actel.com/documents/DesignSecurity_WP.pdf
[214] NVIDIA, “NVIDIA's Next Generation CUDA Compute Architecture: Fermi”, Whitepaper, http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf
[215] NVIDIA, “NVIDIA Tesla – GPU Computing Technical Brief”, http://www.nvidia.com/docs/IO/43395/tesla_technical_brief.pdf
[216] Abachi, H. and Walker, A. 1996. Network Expandability and Cost Analysis of Torus, Hypercube and Tree Multi-Processor Systems. In *Proceedings of the 28th*

Southeastern Symposium on System theory (SSST '96) (March 31 - April 02, 1996). SSST. IEEE Computer Society, Washington, DC, 426.

[217] Ajima, Y., Sumimoto, S., and Shimizu, T. 2009. Tofu: A 6D Mesh/Torus Interconnect for Exascale Computers. *Computer* 42, 11 (Nov. 2009), 36-40.

[218] im, J., Balfour, J., and Dally, W. 2007. Flattened Butterfly Topology for On-Chip Networks. In *Proceedings of the 40th Annual IEEE/ACM international Symposium on Microarchitecture* (December 01 - 05, 2007). International Symposium on Microarchitecture. IEEE Computer Society, Washington, DC, 172-182.

[219] Boris Grot, Joel Hestness, Stephen W. Keckler, and Onur Mutlu, "Express Cube Topologies for On-Chip Interconnects" *Proceedings of the 15th International Symposium on High-Performance Computer Architecture (**HPCA**)*, pages 163-174, Raleigh, NC, February 2009.

[220] Berejuck, M. D. and Zeferino, C. A. 2009. Adding mechanisms for QoS to a network-on-chip. In *Proceedings of the 22nd Annual Symposium on integrated Circuits and System Design: Chip on the Dunes* (Natal, Brazil, August 31 - September 03, 2009). SBCCI '09. ACM, New York, NY, 1-6.

[221] Bjerregaard, T. and Mahadevan, S. 2006. A survey of research and practices of Network-on-chip. *ACM Comput. Surv.* 38, 1 (Jun. 2006)

[222] K. Banerjee, S. Souri, P. Kapur, and K. Saraswat. 3-d ics: A novel chip design for improving deep-submicrometer interconnect performance and systems-on-chip integration. *Proc. IEEE*, 89(5):602- 633, May 2001.

[223] Weldezion, A. Y., Grange, M., Pamunuwa, D., Lu, Z., Jantsch, A., Weerasekera, R., and Tenhunen, H. 2009. Scalability of network-on-chip communication architecture for 3-D meshes. In *Proceedings of the 2009 3rd ACM/IEEE international Symposium on Networks-on-Chip* (May 10 - 13, 2009). NOCS. IEEE Computer Society, Washington, DC, 114-123.

[224] Goossens, K., Dielissen, J., and Radulescu, A. 2005. Æthereal Network on Chip: Concepts, Architectures, and Implementations. *IEEE Des. Test* 22, 5 (Sep. 2005)

[225] S. Kumar, A. Jantsch, J.-P. Soininen, M. Forsell, M. Millberg, J. Öberg, K. Tiensyrjä and A. Hemani, "A network on chip architecture and design methodology," in Proceedings of IEEE Computer Society Annual Symposium on VLSI, April 2002, pp. 105–112.

[226] M. Millberg, E. Nilsson, R. Thid and A. Jantsch,"Guaranteed bandwidth using looped contain- ers in temporally disjoint networks within the nostrum network-on-chip," in Proceedings of IEEE Design, Automation and Testing in Europe Conference (DATE), 2004, pp. 890–895.

[227] Chen, W. and Gehringer, E. F. 1988. A graph-oriented mapping strategy for a hypercube. In *Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications: Architecture, Software, Computer Systems, and General Issues - Volume 1* (Pasadena, California, United States, January 19 - 20, 1988). G. Fox, Ed. C^3P. ACM, New York, NY, 200-209.

[228] Gaurav Kumar Singh, Mythri Alle, Keshavan Vardarajan, S K Nandy, 2010. "Ranjani Narayan A Generic Graph-Oriented Mapping Strategy for a Honeycomb Topology," International Journal of Computer Applications, Number 21 - Article 17

[229] Montanana, J. M., Koibuchi, M., Matsutani, H., and Amano, H. 2009. Balanced Dimension-Order Routing for k-ary n-cubes. In *Proceedings of the 2009 international Conference on Parallel Processing Workshops* (September 22 - 25, 2009). ICPPW. IEEE Computer Society, Washington.

[230] Chiu, G. 2000. The Odd-Even Turn Model for Adaptive Routing. *IEEE Trans. Parallel Distrib. Syst.* 11, 7 (Jul. 2000), 729-738.

[231] Xie, L. and Xu, D. 2009. The Two-Level-Turn-Model Fault-Tolerant Routing Scheme in Tori with Convex and Concave Faults. In *Proceedings of the 2009 Sixth international Conference on information Technology: New Generations* (April 27 - 29, 2009). ITNG. IEEE Computer Society, Washington, DC

[232] Chou, C. and Marculescu, R. 2007. Incremental run-time application mapping for homogeneous NoCs with multiple voltage levels. In *Proceedings of the 5th IEEE/ACM international Conference on Hardware/Software Codesign and System*

Synthesis (Salzburg, Austria, September 30 - October 03, 2007). CODES+ISSS '07. ACM, New York, NY.
[233] C. L. Chou, U. Y. Ogras, R. Marculescu, "Energy- and Performance-Aware Incremental Mapping for Networks-on-Chip with Multiple Voltage Levels", in IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems (TCAD), vol. 27, no. 10, Oct. 2008, pp. 1866-1879.
[234] Chen-Ling Chou; Marculescu, R., "User-Aware Dynamic Task Allocation in Networks-on-Chip", Design, Automation and Test in Europe, 2008. DATE '08, Page(s): 1232 – 1237
[235] D. N. Jayasimha, B. Zafar, Y. Hoskote, “On-Chip Interconnection Networks: Why They are Different and How to Compare Them”, Intel, 2006.
[236] J. Starner, J. Adomat, J. Furunas and L. Lindh. Real-Time Scheduling Co-Processor in Hardware for Single and Multiprocessor Systems. Proc. Of the 22nd EUROMICRO Conference, 1996, Los Alamitos, CA, USA.
[237] P. Kuacharoen, M. Shalan, and V. Mooney, A configurable hardware scheduler for real-time systems, Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms, Jun 2003, pp. 96–101.
[238] N.C. Audsley and K. Bletsas, Fixed priority timing analysis of real-time systems with limited parallelism, Proceedings of the IEEE Euromicro Conference on Real Time Systems (Catania, Italy), IEEE, Jul 2004.
[239] M. Garey and D. Johnson, Computers and intractability : a guide to the theory of NP-completeness, W. H. Freeman and company, NY, 1979.
[240] J. Leung and J. Whitehead, On the complexity of fixed-priority scheduling of periodic, real-time tasks, Performance Evaluation 2 (1982), 237–250.
[241] S. K. Dhall and C. L. Liu, On a real-time scheduling problem, Operations Research 26 (1978), 127–140.
[242] A. Burchard, J. Liebeherr, Y. Oh, and S. H. Son, New strategies for assigning real-time tasks to multiprocessor systems, IEEE Transactions on Computers 44 (1995), no. 12, 1429–1442.
[243] S. Lauzac, R. Melhem, and D. Mosse, An improved rate-monotonic admission control and its application, IEEE Transactions on Computers 58 (2003), no. 3.
[244] J. M. Lopez, J. L. Diaz, and D. F. Garcia, Utilization bounds for EDF scheduling on real-time multiprocessor systems, Real-Time Systems: The International Journal of Time-Critical Computing 28 (2004), no. 1, 39–68.
[245] Sanjoy Baruah, Neil Cohen, Greg Plaxton, and Donald Varvel, Proportionate progress: A notion of fairness in resource allocation, 1996
[246] Michele Cirinei and Theodor P. Baker, Edzl scheduling analysis, ECRTS (Pisa, Italy), July 2007.
[247] Anand Srinivasan and Sanjoy Baruah, Deadline-based scheduling of periodic task systems on multiprocessors, Information Processing Letters 84 (2002), no. 2, 93–98.
[248] New schedulability tests for real-time tasks sets scheduled by deadline monotonic on multiprocessors, Proceedings of the 9th International Conference on Principles of Distributed Systems (Pisa, Italy), IEEE Computer Society Press, December 2005.
[249] Bj¨orn Andersson and Eduardo Tovar, Multiprocessor scheduling with few pre-emptions., RTCSA, 2006, pp. 322–334.
[250] John M. Calandrino, James H. Anderson, and Dan P. Baumberger, A hybrid real-time scheduling approach for large-scale multicore platforms, Proceedings of the Euromicro Conference on Real-Time Systems (Pisa), 2007.
[251] Sanjoy Baruah and John Carpenter, Multiprocessor fixed-priority scheduling with restricted interprocessor migrations, Journal of Embedded Computing (2005)
[252] Sanjoy K. Baruah. The non-preemptive scheduling of periodic tasks upon multiprocessors, Real-Time Systems: The International Journal of Time-Critical Computing. February 2006.
[253] Priority inheritance protocols: An approach to real-time synchronization - Sha, Rajkumar et al. – 1990

[254] Dynamic priority ceilings: A concurrency control protocol for real-time systems - [Min-Ih Chen and Kwei-Jay Lin] – 1990
[255] Stack-based scheduling of real-time processes - Baker – 1991
[256] Synchronization in multiple processor systems (context) - Rajkumar – 1991
[257] Minimizing Memory Utilization of Real-Time Task Sets in Single and Multi-Processor Systems-on-a-chip - Paolo Gai, Giuseppe Lipari, Marco Di Natale – 2001
[258] Sanjoy Baruah, Giuseppe Lipari, "Executing aperiodic jobs in a multiprocessor constant-bandwidth server implementation", Euromicro Conference on Real-Time Systems (ECRTS 04), Catania (Italy), June 2004
[259] S. Baruah and G. Lipari, "A Multiprocessor implementation of the Total Bandwidth Server", International Parallel and Distributed Processing Symposium (IPPDS 04), Santa Fe', April 2004
[260] Ian Stoica and Hussein Abdel-Wahab and Kevin Jeffay and Sanjoy, K. Baruah and Johannes E. Gehrke and C. Greg Plaxton, A Proportional Share Resource Allocation Algorithm for Real-Time, Time-Shared Systems, Proceedings of the IEEE Real-Time Systems Symposium, Dec. 1996
[261] S.K. Baruah and N.K. Cohen and C.G. Plaxton and D.A. Varvel, Proportionate Progress: A Notion of Fairness in Resource Allocation, Algorithmica, 1996
[262] Raj Rajkumar and Kanaka Juvva and Anastasio Molano and Shuichi Oikawa, Resource Kernels: A Resource-Centric Approach to Real-Time and Multimedia Systems, Proceedings of the SPIE/ACM Conference on Multimedia Computing and Networking, January 1998
[263] Luca Abeni and Giorgio Buttazzo, Integrating Multimedia Applications in Hard Real-Time Systems, Proceedings of the IEEE Real-Time Systems Symposium, December 1998, Madrid, Spain
[264] Neil C. Audsley and Alan Burns and Mike F. Richardson and Andy J. Wellings, Data Consistency in Hard Real-Time Systems, Informatica (Slovenia), Vol. 19 n.2, 1995
[265] Gerber, R. and Seongsoo Hong and Saksena, M., Guaranteeing real-time requirements with resource-based calibration of periodic processes, IEEE Transactions on Software Engineering, Jul 1995
[266] Dong-In Kang and Richard Gerber and Manas Saksena, Parametric Design Synthesis of Distributed Embedded Systems, IEEE Trans. Computers, Vol. 49, n.11, 2000
[267] Walid A. Najjar and Edward A. Lee and Guang R. Gao, Advances in the dataflow computational model, Parallel Computing, Vol. 25 n. 13-14, 1999
[268] Shuvra S. Battacharyya and Edward A. Lee and Praveen K. Murthy, Software Synthesis from Dataflow Graphs, Kluwer Academic Publishers, 1996
[269] Björn Andersson and Jan Jonsson. Preemptive Multiprocessor Scheduling Anomalies. International Parallel and Distributed Processing Symposium. Fort Lauderdale, California, April 2002.
[270] L. Palopoli and T. Cucinotta, Feedback scheduling for pipelines of tasks, Proc. of 10th conference on Hybrid Systems Computation and Control 2007 (HSCC07), April 2007, Pisa, Italy
[271] Tommaso Cucinotta, Luigi Palopoli, *"QoS Control for Pipelines of Tasks Using Multiple Resources,"* IEEE Transactions on Computers, Vol. 53, No. 3, pp. 416--430, March 2010, IEEE Computer Society Digital Library
[272] Steve Goddard and Kevin Jeffay, Managing Latency and Buffer Requirements. in Processing Graph Chains Computing Journal, Vol.44 n. 6, 2001.
[273] Kleopatra Kostanteli, Dimosthenis Kyriazis, Theodora Varvarigou, Tommaso Cucinotta, Gaetano Anastasi, "*Real-time guarantees in flexible advance reservations*," in Proceedings of the 2nd IEEE International Workshop on Real-Time Service-Oriented Architecture and Applications (RTSOAA 2009), Seattle, Washington, July 2009.

[274] Tommaso Cucinotta, Fabio Checconi, Luca Abeni, Luigi Palopoli "*Self-tuning Schedulers for Legacy Real-Time Applications*," in Proceedings of the 5th ACM European Conference on Computer Systems (EuroSys 2010), Paris, April 2010.

[275] Ting Yang, Tongping Liu, Emery D. Berger, Scott F. Kaplan and J. Eliot B. Moss . Redline: First Class Support for Interactivity in Commodity Operating Systems . 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2008). San Diego, December 2008.

[276] Paul McKenney. *"A realtime preemption overview"*. Available on-line at: http://lwn.net/Articles/146861/.

[277] Abraham Silberschatz. *On the synchronization mechanism of the ADA language*. ACM SIGPLAN Notices, Vol. 16, No. 2, pp. 96 – 103. 1981.