First Set of OS Architecture Models

Project Deliverable D5.3 v1.0

Tommaso Cucinotta, Giuseppe Lipari (SSSA)
Rui Aguiar, João Paulo Barraca, Bruno Santos and Javad Zarrin (IT)
Jan Kuper and Christiaan Baaij (UT)
Lutz Schubert and Daniel Rubio Bonilla (USTUTT-HLRS)
Vincent Gramoli (EPFL)

Due date: 31/07/2011
Delivery date: 21/08/2011

SEVENTH FRAMEWORK PROGRAMME

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

Version History

Version	Date	Change	Author
0.1	24/05/11	First TOC	Tommaso Cucinotta
0.2	02/06/11	Second TOC and preliminary contributions	Tommaso Cucinotta, all
0.3	08/06/11	Refined TOC and contributions re-arrangement	Tommaso Cucinotta
0.4	10/06/11	Merged HLRS contributions	Tommaso Cucinotta, Lutz Schubert
0.5	17/06/11	Merged refined contributions and partial TOC re-arrangement.	Tommaso Cucinotta, all
0.6	22/06/11	Merged chapters restructuring by HLRS	Lutz Schubert
0.7	24/06/11	Merged EPFL & SSSA contributions	Tommaso Cucinotta, Vincent Gramoli
0.8	30/06/11	Merged OS Requirements Tables	Tommaso Cucinotta, all
0.9	01/07/11	OS Requirements fixup	Tommaso Cucinotta, all
0.10	04/07/11	Clarified meaning of OS Requirements	Tommaso Cucinotta
0.11	07/07/11	Merged dependencies summary and various restructuring of sections	Tommaso Cucinotta, all
0.12	02/07/11	Further restructuring; extended dependencies table; added overarching architecture view	Lutz Schubert, all
0.13	18/07/11	Refined code analysis section	Lutz Schubert
0.14	28/07/11	Added scenario & requirements restrictions	Lutz Schubert
0.15	02/08/11	Extended architecture integration section; integrated various extensions	Lutz Schubert, all
0.16	09/08/11	Extended API discussion	Lutz Schubert
0.17	21/08/11	Various fix-ups, addressing final comments & typos	Tommaso Cucinotta, Lutz Schubert
0.18	21/08/11	Final fix-ups, proof-reading, typos, punctuation, bibliography fixes.	Tommaso Cucinotta
0.19	22/08/11	Last-mile fixes by EPFL & SSSA	Tommaso Cucinotta
1.0	23/08/11	Final version ready for submission to the EC	Lutz Schubert

EXECUTIVE SUMMARY

This document reports the results of the preliminary investigations about general Operating System (OS) architecture models that will make it easier for developers to code applications on massively parallel and distributed systems as expected to be available in 10-15 years in the future. The discussion focuses first on a small set of target application scenarios which are useful to highlight particularly critical requirements posed by the applications on the OS, as arising in the context of S(o)OS. These requirements are mainly related to scalability issues of nowadays OSes and run-time environments. Then, the OS architecture model is sketched out in terms of subcomponents, their interconnections and interdependencies and behaviour. However, as this constitutes a first preliminary investigation that will be refined in the future version of this deliverable D5.4, the whole set of details of an OS architecture are not deeply discussed, but rather the focus is on those aspects that are critical for the project objectives. A recurring discussion in this context relates to kernel- versus user-space location of the OS/kernel components, which we however want to avoid in this first iteration, for the sake of concentrating on the OS capabilities in the first instance. Indeed, the definition of a whole OS architecture is actually outside the scope of the S(o)OS project. The main focus of this document (and of the refined version that will be released later) is the one to identify critical architectural elements and subcomponents that, constituting major bottlenecks in nowadays operating systems, need to be reviewed and rethought for the purpose of being able to face with future massively parallel and distributed systems so as to expose the available computing power to a broad range of developers, ranging from average developers to highly experienced ones.

TABLE OF CONTENTS

List of Figures

List of Tables

ABBREVIATIONS

Abbreviation	Description
ABI	Application Binary Interface
API	Application-Programming Interface
ccNUMA	Cache Coherent NUMA
CPU	Central Processing Unit
FS	File System
GPU	Graphics Processing Unit
GPGPU	General-Purpose GPU
HTM	Hardware-based Transactional Memory
IPP	Integrated Performance Primitives (by Intel)
ISA	Instruction-Set Architecture
MPI	Message-Passing Interface
NoC	Network-on-a-Chip
NUMA	Non-Uniform Memory Architecture
OS	Operating System
PGAS	Partitioned Global Address Space
QoS	Quality of Service
RD	Resource Discovery
S(o)OS	Service-oriented Operating Systems
SCC	Single-chip Cloud Computer (by Intel)
SSI	Single-System Image
STM	Software Transactional Memory
TM	Transactional Memory

TERMS

Term	Definition
Kernel	Core part of an OS implementing basic fundamental functionality that serve as a basis for realising all of the other OS services.
Node, Machine, Host	All synonyms denoting a physically separated autonomous computing element (a PC, a node inside an HPC rack, etc.), which is connected to other elements via network adapters.
Thread	Single thread of execution, sharing the same memory space as all the other threads belonging to the same process.
Process	A process is composed of one or more threads. Processes on the same OS do not share the same memory space, however they can use shared memory segments if they use explicitly the corresponding OS services.
Task	Synonym of thread.
Code Segment	A piece of a program that can be extracted from a process, and executed on a different core than the one where the rest of the process is being executed. For example, a sequential execution involving a loop over a large number of items can be split into code segments to be executed on different cores in parallel for enhanced performance.
Scheduling	The algorithm by which it is decided which physical resources are allocated to which program execution units and when. For example, scheduling code segments, threads and processes over the available cores and processors, scheduling the network packets over the available network adapters, etc.
Scalability	Property of a system to not suffer of major performance penalties as its size grows.
Component, Module, Service	An OS component, or OS module, or OS Service, is a piece of software implementing a specific OS functionality.
Driver	A specific component of an OS supporting a specific hardware capability, peripheral or peripheral type.
Operating System	The run-time environment that mediates the interactions between an application and the underlying hardware, simplifying the task of writing applications for a variety of physical platforms.
Middleware	Software layer mediating the interactions between the applications and the core part of an OS. An OS itself may include middleware components for the purpose of providing application developers with high-level and powerful management interfaces.
Service, Primitive	OS functionality or OS component(s) implementing a particular functionality.

I. INTRODUCTION

With the movement from single-core machines to large-scale heterogeneous multi- and many-core processors, the classical development and even execution paradigms changed completely: not only does the average developer has to face challenges similar to the ones in the HPC domain, but also HPC developers have to cater for a much more complex environment, requiring even higher expertise than before. Whilst the general tendency seems to consist in shifting the burden to the developer, the S(o)OS project aims at significantly reducing this burden through increased support by the operating system.

This means in particular that the OS must provide a method to abstract the underlying hardware and enable a common execution model on top of it, yet still be able to exploit the respective capabilities in the best fashion for the program. At the same time, it must be enabled to deal with the large amount of resources, at least without affecting negatively the performance of the deployed applications. Current operating systems fail with both of these respects.

1. ORGANISATION OF THIS DOCUMENT

This document is organised as follows.

In Chapter II. , basic concepts around the need for designing novel operating systems are shortly recalled, starting from summarising key hardware characteristics that future massively parallel systems will posses, and proceedings to highlight the shortcomings of nowadays OSes as well as the potential for overcoming them in a novel S(o)OS operating system. We outline how we foresee S(o)OS to behave in the context of a generic application lifecycle, thereby highlighting the main benefits and challenges.

In Chapter III. , we list the general OS requirements which are critical for the project, and whose importance is highlighted by focusing on a couple of application scenarios particularly critical for the project. These help in shaping the requirements towards a next-generation class of OSes which are able to overcome such limitations.

In Chapter IV. , we depict a preliminary set of OS architecture models, detailing what are the set of possible OS components, services and functionality and what is their layout in terms of distribution and interactions across a massively parallel physical system.

In Chapter V. , we focus on a subset of OS services which are particularly critical in the context of many-core and massively parallel systems. These include resource discovery and matching, including the support for heterogeneous hardware, scheduling, and others.

In Chapter VI, we discuss the relationships and interactions between these services, thus developing the integrative view on the S(o)OS architecture. We thereby classify the components and align them to the general architecture discussions in chapter IV.

In Chapter VII. , we provide a preliminary discussion on issues specifically related to the API and the programmability of applications in the context of an OS with the enriched functionality and the architecture model as described in the previous chapters.

Finally, conclusions and the research roadmap for the following S(o)OS activities are sketched out in Chapter VIII.

II. S(O)OS BASIC CONCEPTS

In this chapter we overview the key concepts at the basis of a novel operating system architecture for massively parallel and distributed systems. This is done by recalling first what are the hardware trends that massively parallel systems will see in the future, then summarising the shortcomings of nowadays OSes and key innovations that allow S(o)OS to overcome them.

1. HARDWARE TRENDS

As detailed in the S(o)OS Deliverable D5.2 "Future Requirements" [15], a key characteristic of future many-core platforms consists in the increasing heterogeneity of the runtime environments, which will not only incorporate a large set of completely different resources, but, what is more, will differ strongly from one another. In other words, it must be expected that no two computing infrastructures will be the same in the future. Also, it must be generally assumed that the current development trends in processor and system manufacturing will continue and even grow in complexity. It is in particular notable that the era of the so-called "frequency race" has come to an end. This implies that future processor generations will no longer be faster (in terms of clock speed), but integrate more processing units instead, i.e. they will process more operations at the same time. Due to the nature of this form of processing, the classical (sequential) programming paradigm does not apply any more, and future code needs to be parallelised in order to still gain in performance. As manufacturers compete with each other to realise large scaling processors with high theoretical peak performance, this development has become known as the "core race".

However, it cannot be expected that all code is parallelised optimally – instead, many algorithms are essentially sequential in nature, or at least in parts (see [45]). Since the executing frequency can no longer be improved, manufacturers therefore try to develop specialised processing units that are explicitly built to perform specific actions well. Mathematical coprocessors and graphical processing units are historical examples of such specialised processors, that became very well known.

Modern processors already combine multiple specialised units, such as floating point, vector and general purpose units. In order to make best use of these resources, the developer has to organise the code accordingly, respectively adjust the algorithms to meet the hardware specifics. Not only does this imply additional effort for the developer, it also implies that portability of code becomes more and more problematic, as individual processors incorporate different specialised units.

Since there are no clear indicators either for the software requirements or the usage of future processors, nor for the optimal combination of specialised and general-purpose units under these circumstances, future processor generations will essentially be "experimental" in nature. In other words, manufacturers will take different approaches to iteratively improve the quality / performance of their processors. At the same time, the diversity of use cases, such as mobile, office, home and high performance computing, leads to a wide range of requirements that different manufacturers approach differently. To the normal divergence on grounds of lacking knowledge comes thus the divergence due to the variety of use cases.

Currently, there are processors emerging that incorporate fast and energy-saving cores allowing the system to switch between "performance" and "energy-saving" modes by switching between either core type, depending on workload and circumstances. On the other end of the spectrum, Intel is examining how to incorporate multiple homogeneous cores in a cloud like fashion on a single chip, whilst AMD is investigating a more hierarchical approach where segments share specialised floating point units. For more details, please refer to S(o)OS Deliverable D5.2 "Future Requirements" [15].

1.A) ABSTRACTING THE HARDWARE

The divergence of application domains, and implicitly of hardware types, causes multiple problems not only on the program development side, but also on the operating system, as we will elaborate in more detail in the next section. In order to optimally deal with resource types, all details of the respective resource(s) should be known to the compiler / operating system / middleware. However, neither do manufacturers want to publish all this information, nor is the impact of these details onto code execution, let alone programming, fully understood. Efficient programming is typically mostly constrained by this lack of knowledge regarding the resource details.

What is more, these details would further restrict the portability of the code. Accordingly, it is neither feasible, nor sensible to act upon the whole resource details. Instead, ideally an abstraction level can be found that not only provides enough details for essential adaptation steps, but that also covers future developments to a wide degree, so that no constant re-adaptations of this information schema is needed.

From the most generic level of both optimisation and description, we can distinguish between essential resource "classes" along the lines of[1]:

- memory/storage units;
- processing units;
- connection elements.

Each of these elements in turn have further characteristics that specify the concrete behaviour and low-level optimisation details – however, on the highest abstraction level, these three classes suffice to provide essential information about the potential degree of parallelisation, the distribution and adaptation of threads / processes and the communication impact. It will be noted that this description is closely related to von Neumann's architecture, which forms the basis of all modern processor architectures. Von Neumann's model is however insufficient to capture different levels of hierarchical storage over different communication layers, as well as multi-core units within a single processor. We therefore build upon a "new" von Neumann architecture as depicted in Figure 1, where multiple generic Processing Units (PU) and multiple generic Memory Units (MU) are

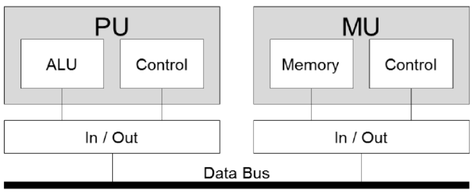


Figure 1: "new" von Neumann architecture

interconnected through proper interconnection elements (Data Bus in the picture).

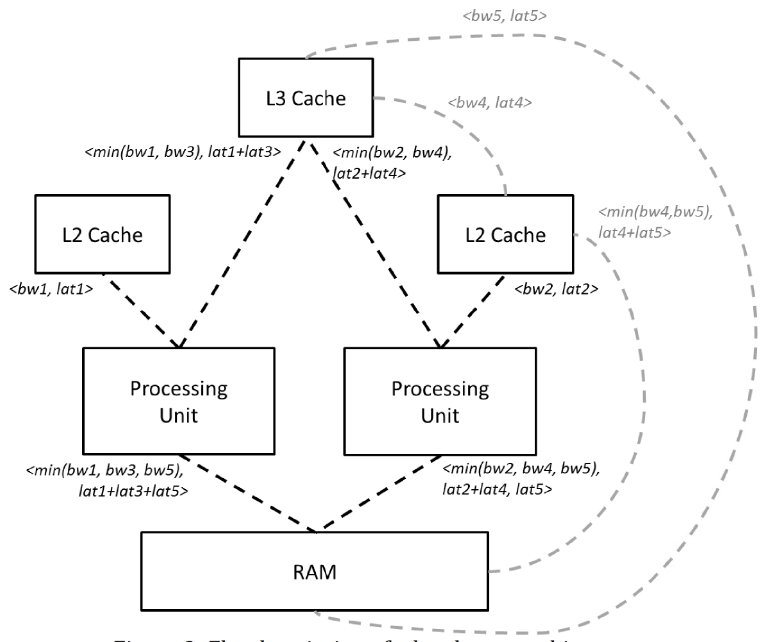
This architecture allows us to equally cover local cache, as well as even Internet-provided storage, and single-core desktop machines, as well as large-scale cluster machines etc. Even though S(o)OS does not try to address all these varieties in the same degree, the principles of S(o)OS can thus be extended to cover further architectures, such as cloud infrastructures, or even future processor architectures. We claim thereby that developments such as 3d stacked memory are implicitly covered by this structure, too: any of these architectural choices will either be hidden to the developer, such as using shadow caches, in which case they subsume under the specific characteristics of a unit in the new von Neumann architecture, reflecting the special functionalities aimed at their respective development in the first instance. Or the choices will be explicitly exposed to the developer, in which case the respective choices are simple units in the new von Neumann architecture.

[1] Note that we focus exclusively on hardware elements that are needed to support computing. For example, these classes do not cover peripherals.

According to this architecture, all units are connected via some form of communication link. In other words, processing units and memory (such as cache) are not connected directly, but via an interconnect that exhibits all the characteristics of a regular messaging route, i.e. are subject to latency and bandwidth restrictions. The model allows for multiple units of different type to be connected along the same communication link, or hierarchically using multiple routes with different connectivity and properties. It is secondary for this model that external devices (sensors, displays etc.) are equally connected via such a communication interconnect.

The model does not prescribe the type of either processing or memory unit – that is, they can equally well cover concrete low-level units, such as L1 cache and processing cores, as well as more abstract units, such as virtual files, internet-provided disks, tables, many-core processors, virtual hosts etc. In general, a processing unit is any entity that can execute computational tasks on data, and likewise a memory unit is any resource that can store data. It thereby does not matter whether the unit itself is again built up by multiple processing or memory units, as long as the developer does not have to cater for it either – or more generally spoken, as long as the details are of no consequence for development.

The architecture thereby takes mainly a (software) developer's / optimiser's perspective – i.e. whilst it may not be suitable for the hardware developer, it provides a software developer aiming for high efficiency in his code execution with all essential information. Even though classical programming development neglects such aspects as latency to L1 cache, this information in combination with the size of this cache and the bandwidth of its connection proves vital for optimisation tasks and for identifying scalability issues. In heterogeneous environments, it furthermore provides the necessary data to identify the best distribution of dependent code segments. It may be noted in this context that hardware-specific adaptation steps are still necessary, even though subsumed in this model – for example, when two processing units have different ISAs.



<bw5, lat5>
L3 Cache
<bw4, lat4>
<min(bw1, bw3), lat1+lat3>
<min(bw2, bw4), lat2+lat4>
L2 Cache
L2 Cache
<min(bw4,bw5), lat4+lat5>
<bw1, lat1>
<bw2, lat2>
Processing Unit
Processing Unit
<min(bw1, bw3, bw5), lat1+lat3+lat5>
<min(bw2, bw4, bw5), lat2+lat4, lat5>
RAM

Figure 2: Flat description of a hardware architecture.

Even though S(o)OS does not try to address all these varieties in the same degree, the principles of S(o)OS can thus be extended to cover further architectures, such as cloud infrastructures, or even future processor architectures. With this architectural concept, it is possible to depict the whole layout of an infrastructure, down from the lowest processing units up to the highest-level systems. We can thereby choose between a flat or a hierarchical description of the infrastructure (cf. Figure 2). For more details about hardware descriptions, please refer to S(o)OS Deliverable 2.2 [31].

1.B) S(O)OS INFRASTRUCTURE CHARACTERISTICS

Though the new architectural structure allows for a wide variety of infrastructures to be covered, S(o)OS restricts itself to a selected subset of architectures in order to keep the effort feasible and improve the quality of the results. With S(o)OS aiming specifically at dealing with large-scale problems and improving their programmability, the primary application domain consists in high performance computing, as the according code has typically been explicitly developed for a high degree of scalability.

As S(o)OS deals with improving the parallel execution of code by exploiting inherent and explicit concurrency, it is relevant to estimate timing behaviour of code segments given specific conditions. A domain where the timing behaviour of software is actually critical is the one of real-time (RT) systems, where timing is at least as important as functional correctness. Furthermore, a notable class of RT applications exists where the high computational workloads can meet the in-place tight timing constraints only thanks to proper parallelisation and distribution techniques (we could think of these as belonging to a hybrid HPC-RT class). Accordingly, a secondary application domain for S(o)OS consists in time-sensitive applications – ranging from interactive to soft RT use cases. Both HPC and RT applications benefit from a highly predictable duration of code segments which should ideally be nearly constant, or with minimum variability. However, when this is not possible (as common in e.g., multimedia processing), HPC development cares particularly about the average-case performance for maximising the achievable throughput, whilst for RT development the focus is on the worst-case duration, or some percentile of its probabilistic distribution, for the purpose of respecting the in-place timing constraints, such as deadlines.

In particular S(o)OS thereby addresses the increasing need for interactive HPC use cases. Implicitly, the major infrastructure type consists in HPC and cluster machines that incorporate a high amount of resources with dedicated, high-performing interconnects. It is notable in these infrastructures, that their general hierarchy is comparatively flat, i.e. processors are connected as directly as possible (via I/O) without having multiple levels on top of them, such as servers, servers farms and world wide web (i.e., intranet and internet). Furthermore, even though HPC systems become more and more heterogeneous, their diversity is way lower than the heterogeneity in the Internet. In other words, HPC systems have a high degree of homogeneity, such as multiple GPU cards of the same type, multiple identical processors, etc. Similarly, the interconnect is typically very homogeneously arranged in order to maintain equal latency and bandwidth behaviour.

Further to this, it can be expected that the code already exhibits characteristics of parallelism and concurrency and that the algorithm is not essentially sequential in nature. The according programs typically also exhibit a minimum of OS interactions, thus making the dependencies clearer. As the major aim of HPC consists in execution performance, jobs are generally executed in an isolated fashion, both task and hardware wise. In other words, the OS does not need to handle multi-tasking aspects and generally does not have to cater for security. In all these cases, the according capabilities can be built up on this base model.

In general, the concepts of S(o)OS are by no means restricted to the given use cases – in fact it will be noted from the detailed descriptions below that the same principles can be applied to cater for other application domains, such as more general purpose home and office usage or distributed environments, such as clouds, or social networks. S(o)OS does not explicitly address them so as to not diverge from its main objectives (scalability and heterogeneity), by having to cater for additional concerns, such as security, task isolation, reliability etc.

A NOTE ON RELIABILITY

Since reliability is a major concern also for HPC applications, we must elaborate this latter issue in more detail: we can distinguish between different types of reliability, in particular between (a) reliable execution of the application and (b) reliability of the OS itself.

Reliable execution of type (a) refers to fault tolerance at runtime, i.e. that neither problems on the infrastructure or the application itself lead to an execution failure. This covers a vast research area in itself and can be mainly separated into the following main approaches: (1) execute the code in a managed framework, (2) replicate the application, i.e. execute it multiple times in parallel or (3) checkpoint the application in time and roll back when need be. Even though S(o)OS can be compared in parts with a managed framework, the according tasks (monitoring and adaptation) serve different purposes and are not executed everywhere. Similarly, replicating application execution is only sensible when the code size does not exceed a certain degree of scale, as otherwise this leads to bad resource utilisation in large scale environments such as envisaged by S(o)OS – more advanced algorithms try to identify the critical code segments, yet this is (for now) outside the scope of this project. Since S(o)OS aims to be general purpose, reliability would also have to cover dynamicity issues, which are generally unsolved as yet.

The most typical form of application level reliability in HPC environments consists in checkpointing at regular intervals and rolling back the application state if an unexpected error occurs. Notably, the same error may occur again if the code is buggy in itself. S(o)OS will indirectly support this approach by providing all essential means for checkpointing and rollback, yet either a middleware or the application itself will have to actually employ these capabilities according to the respective need of the use case.

As opposed to this, S(o)OS will address reliability on the level of the operating system itself, i.e. trying to ensure that the whole system does not fail with problems in a single node. This document does not explicitly address this issue though, as it will be subject to future work in the project.

2. The Role of the OS

The S(o)OS project proposes to address the problem of scalability and heterogeneity through a novel OS architecture approach which builds up from the lessons learned in Service-Oriented Architectures and Distributed Systems. All these domains are signified by a large degree of scalability, but at the same time by a comparatively low degree of communication and connectivity. As opposed to that, typical HPC scalable applications have a high demand of data exchange and thus require a strong interconnect. Naive programmers now implement a program in such a fashion that the degree of data exchange between threads is very high. What is worse, some parallel programs exhibit tendencies to act like a set of processes that require constant OS support – in these cases, OS communication strongly interferes with the interprocess communication, and OS execution stalls multiple processes at the same time.

From the application perspective, we can therefore distinguish different types of dependencies across threads, code segments and OS system functions that have different impact on the execution performance depending on their way of implementation. In general we can state that the “tighter” and more frequent a certain form of dependency (and implicitly communication) occurs, the higher the impact of any delay due to the implementation choices. For example, if the access to registers were delayed e.g. through a message-based access implementation, the code would hardly be executable any more. On the other hand, the impact of delaying a web service invocation even further is minimal from the total application execution perspective.

Due to the extended distribution features in S(o)OS which may involve separating an application at “arbitrary” points, the operating system needs to cater for different types of dependencies in different forms of communication in a fashion that is mostly invisible to the application, and in particular to the developer – i.e. the developer should not have to cater for choosing the communication means for all different setups. For example current HPC developers have to consider the distribution of threads across cores and processors to implement a well-adjusted hybrid mix of OpenMP and MPI communications in order to achieve the best performance. Accordingly, the communication choices may fail in different infrastructures.

We can distinguish in particular the following levels:

- **Register access**: the lowest level of information access in a program and at the same time the one with the highest frequency of occurrence. Register access is completely inherent to the program code and generally not interceptable, nor is it sensible to increase the access delay. We can generally assume that this level is unchanged.
- **Memory access**: in particular in shared memory applications, this level forms the most crucial form of inter-process state and data sharing. Depending on the implementation choice and the programming model used, this type of access is realised in completely different ways. Memory access in general is very frequent in any form of application. The degree of memory sharing depends very much on the type of application and its coupling.
- **System calls & events**: system calls are handled by the OS, leading to a non-negligible amount of cache swaps and to implicit communications on memory level. The amount of invocations is comparatively high, but lower than memory access.
- **Remote procedure calls and inter process communication**: the highest level of communication consists in explicit invocations of remote processes or services via dedicated messages. This type of communication is typically explicitly catered for in the code and delays are considered in the overall execution time.

Notably, there are further types of dependencies and communication on intermediary levels which however expose similar properties. What is of main importance here is that the highest levels are typically catered for by a middleware, whereas the lowest level is left up to the hardware. The intermediary levels are either completely neglected or left to the programmer (sometimes helped by the compiler), who faces a higher development effort and may (involuntarily) specialise the code for a specific infrastructure.

The S(o)OS approach tries to cater for all these levels in three major ways: (1) selecting, injecting and adjusting the communication according to the current deployment and distribution; (2) distributing code and threads according to their dependencies and hence communication requirements / restrictions; and (3) distributing and replicating operating system level support according to these dependencies.

3. Main S(o)OS Principles

S(o)OS main concept builds on exploiting concurrency implicit and explicit to application code: as communication forms the major obstacle towards highly efficient and large-scaling code execution, the major task in parallelisation optimisation consists in reducing *any* communication between parallel processes / threads. Communication is encoded in the degree of dependencies between different parts of the code, thus including not only direct invocation and explicit messaging, but also common memory or data access and thus state sharing. Any two parts of the code can be considered “concurrent” to one another, if there is little to no dependency between these two. Ideally, the developer exploits this concurrency at development time and enables execution of a maximum number of independent threads in parallel. Typically however, dependencies cannot be avoided and a clever developer must attempt to reduce their impact to a minimum.

S(o)OS supports this task by analysing the code behaviour with particular respect to its cross-dependencies, i.e. potential points of concurrency. It will thereby not only consider low- or high-level communication dependencies, but in particular the intermediary dependencies with the operating system itself. It will thus not only support identification of the main segmentation points in the application itself, but even of the operating system itself.

The main principle of S(o)OS consists in splitting up the operating system into a series of essential functionalities that are atomic in nature, i.e. they should not further be subdivided, but provides the minimal set of capabilities whilst still maintaining maximum performance. It has been shown that [29] message-based communication between distributed kernels has a potential for avoiding performance bottlenecks of the OS when scaling to many-core architectures. By clever distribution and exploitation of concurrency, the messaging overhead can be massively reduced, thus improving the overall performance. This is supported at the same time by extending parallel execution beyond the point of "just" work / data distribution, to concurrent execution that no current programming model supports properly. This allows reducing the impact from Amdahl's law by even parallelising the sequential code part to a certain degree.

The essential OS functions therefore become "services" to the code (application or other OS segments). Similar to the principles of Service-Oriented Architectures, the individual services can be easily replaced with others, as long as they offer the same interfaces and the same functionalities. The service-oriented approach allows replacement of individual modules / services easily without affecting the whole OS. In other words, hardware specific adaptations, as well as application-specific services, can be realised easily by providing only the relevant modules in an adaptive fashion. This implies that the OS has to identify and deploy these modules though. The architectural concept of S(o)OS is described in more detail in section IV. 2.

In the following we will exemplify how an application behaves on such an OS.

4. Application Behaviour

The lifecycle of a parallel and distributed application spans the whole range from development to actual execution on the destination platform. This covers in particular the following steps:

I. Development / Programming

Most applications are essentially sequential in nature, i.e. the underlying algorithm is not per se suited for a parallel, multi-core or multi-processor environment. Automatic parallelisation only reaches bad execution performance, so that in principle the developer has to think about parallelising the algorithm itself. Supporting tools for this are limited to low-level actions, rather than parallelisation of the overarching logic such as:

- parallel libraries that offer specific functions in a parallel fashion - in particular mathematical libraries that provide functions e.g. for parallel matrix calculation;
- libraries or programming extensions to support communication or data exchange across threads (OpenMP, MPI, PGAS etc.).

In order to parallelise the application, the developer has to think about means to distribute the actual execution over the available cores, processors and nodes. This implies that he/she has to either split up the data and work among the processing units, and thereby has to respect the different communication modes in a heterogeneous environment.

II. Compilation

Even if the developer has parallelised the code and potentially optimised it for the destination platform, the compiler will still execute further parallelisation tasks and optimisation steps. When supporting languages or tools were used, the compilation step will replace the according commands with the respective parallel command pattern, such as replacing the OpenMP pragmas with the according thread creation commands, or replacing the shared memory invocations with messaging in PGAS. Further to this, the compiler will perform specific optimisation steps dedicated to the respective hardware by identifying typical patterns and replacing them with the corresponding code patterns that suit the hardware best.

As a consequence of the compilation step, the resulting code is fixed to a specific platform, i.e. the executed optimisation and adaptation steps make the code most suited for the single selected infrastructure, but restricts its portability to platforms that are almost identical.

III. Submission and reservation
In HPC scenarios, the user submits the application execution to the job queue of the machine. As part of this, the developer will specify the required amount of nodes and time for execution of the code. Generally, it is currently not possible to specify more details, such as communication needs, quality restrictions or hardware needs. With the heterogeneity increasing, it slowly and more frequently becomes possible to specify the destination hardware, but generally it is assumed that the developer has put thoughts into this prior to selecting the machine.

The queues order the submissions on basis of their requirements and availabilities in the platform. In some cases, the queues will also respect priorities basing on customer type / role. The actual execution does not begin before the job queue has reached the according submission. Once the respective position in the queue is reached, the binary package is deployed.

IV. Deployment
Of the reserved resources, one instance is selected as the main node for execution. On this node, the actual job is deployed (stored) and invoked. Typically, the application itself is responsible for creating and deploying additional threads and to ensure their availability through unique identifiers.

In HPC scenarios, the nodes are reserved for the execution of a single application for the reserved time, and cannot be used for other purposes. In most cases, however, the resources are shared between multiple applications. In general, the hosting environment already needs to contain and run a middleware that enables deployment and communication (i.e. an operating system). In IaaS environments, the user can deploy full environments that are typically hosted in the form of virtual machines.

V. Execution
The actual execution of parallel code is essentially identical to running multiple independent processes in a distributed environment, whereas each process (thread) is either explicitly invoked by the main thread or executed right after deployment. The differentiation between sequential and parallel code parts thereby poses the major problem for efficient code execution in large-scale environments (cf. Amdahl's law).

The main task for the middleware to realise parallel execution consists however in synchronisation and state maintenance across threads, as well as communication between them. This means that some level of the execution framework needs to be able to identify and distribute shared data – generally, rather than identifying specific data, the framework will distribute and maintain all data that is specified as “shared” in one way or another in the source code (s. e.g. PGAS). More low-level approaches require the user to explicitly send and synchronise all according variables. As a consequence, the communication overhead is typically high - and typically way higher than necessary.

VI. Profiling & Monitoring
During execution, the behaviour of an application is monitored with respect to resource usage, performance etc. so as to create a behaviour profile. Behaviour profiles give an indication of the software system requirements, including such aspects as memory size, computing and network

bandwidth requirements. Such information is helpful for the application developer to further improve the code performance, or to the run-time environment to adaptively schedule applications for maximum performance. Notably, this is very important for interactive and real-time applications (e.g., adaptive feedback-based scheduling). Profiling and monitoring information can also be used in order to react to faults or to adjust energy consumption.

Profiling information covers aspects such as:

- what is the duration and execution time of certain code segments, including the overheads due to scheduling, synchronisation and communications;
- what are the actual delays experienced as due to the communications among the distributed application components;
- what are the latencies experienced across various critical monitoring points of the application, including the end-to-end latency;
- what are the sustainable data throughput for various critical parts of an application execution, and where are the bottleneck.

Even though performance profiling, modelling and analysis are critical steps for real-time applications, they constitute sometimes important steps also in HPC scenarios. Indeed, often an HPC application needs to check whether the various parts of the computations carried out in a distributed fashion are exactly balanced (i.e., they exhibit similar execution times) or not, so it is important to keep under control the execution time of the various code segments.

5. S(O)OS APPLICATION LIFE CYCLE

S(o)OS extends this application life cycle by explicitly adding means for the purpose of handling scalability and heterogeneity. In other words, by steps that are typically left up to the developer or the middleware into the operating system and adjusting the latter so as to cater for the implicit changes on the architecture.

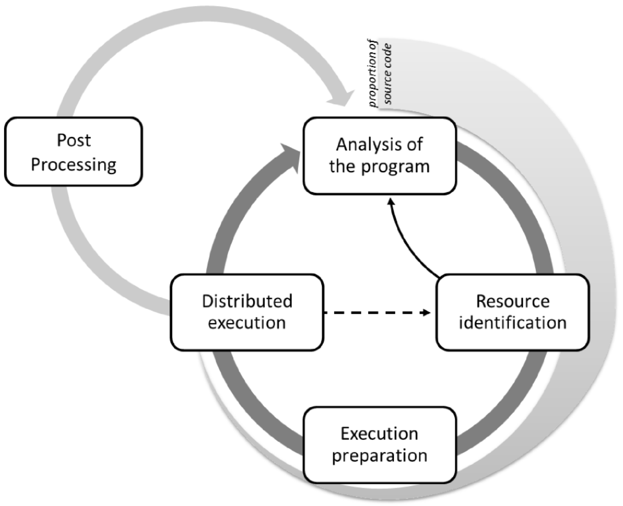


Figure 3: S(o)OS' extended application lifecycle

Similar to the "classical" lifecycle, we can identify the following main phases:

- development/programming;
- compilation;
- code analysis;
- resource identification;
- execution preparation;
- execution;

- post-processing.

Notably, as the operating system incorporates more dynamicity and adaptability than classical models, the phases become more interdependent. Execution may thus “jump” between phases, depending on the context triggers (cf. Figure 3). For example, profiling during execution may indicate that the application shows vectorial behaviour in parts, thereby enabling deployment on a vector processor – this automatically triggers redeployment of the code, given that the expected gain outweighs the cost for adaptation.

In the following sections we will explain these phases in more detail.

5.A) DEVELOPMENT

Program development in S(o)OS can be enriched with **high-level code annotations** that provide information about the dependencies of code executions (and hence principle concurrency) which could be exposed in the form of algorithmic descriptions ("code purpose", if you want); further annotations of that type could explicitly steer the behaviour of the S(o)OS components, such as specifying the best resource types for a specific task, which could support resource discovery in making a selection.

On the other hand, S(o)OS will expose **low-level code annotations** that are related to explicitly controlling the behaviour, such as dependency scope of an individual parameter, messaging specifics (send as block, send individually etc.), segmentation sizes etc. This will improve the performance of S(o)OS and hence the code execution, if the developer has enough knowledge about code behaviour and destination platform. Low-level annotations hence aim at expert (HPC) developers.

Similarly, real-time application developers will mainly benefit from a high-level declarative approach by which they will provide explicitly the timing constraints of their applications, in addition to the dependencies among code segments. The adaptive capabilities of the OS will manage to parallelise and deploy the application in a way that is suitable for meeting the in-place timing constraints. On the other hand, expert RT developers might prefer to use a lower-level control of scheduling and parallelisation, for improving predictability of a particular application execution on a particular hardware platform.

5.B) COMPILATION

Compilation in S(o)OS could be executed in different forms: since the OS (tools) will perform the final adaptation steps towards the destination platform, the code may not actually be converted into full machine code, but into an intermediary form. Also, due to the dynamic behaviour of the OS, the compilation should lead to an additional set of meta-information (such as dependencies) which could be used for analysis and deployment (and execution) purposes.

Accordingly, we can see at least the following options:

- **source-to-source**: the compiler converts the code into another code form that is closer to the actual S(o)OS specifics, e.g. C with low level annotations;
- **source-to-intermediary**: the compiler generates a "close-to-machine" code that can be easily converted into machine code, but still contains relevant information about the last conversion steps;
- **JiT compilation** (relates to source-2-intermediary): the actual compilation is executed on-the-fly, i.e. as soon as information about the destination platform and the code distribution is available;
- **recompilation on the fly**: the compiler generates full machine code but provides enough information that allows detection of the appropriate source (or intermediary) code position to trigger recompilation if need be (see debugging symbols). In other words: during distribution, the OS detects that the destination platform differs from the existing compiled code version - it

therefore detects the according source code and requests the compiler to compile the according segment with different options (different destination);

- **conversion on the fly**: the most difficult way to treat heterogeneity consists in adapting the machine code on the fly to different platforms. This has no impact on the compiler - instead, compilation will be directed towards a generic platform, that allows easy adaptation to new systems. In general, this is not possible (e.g. different ISA) - however, low-level adaptations could be executed this way, depending on circumstances and purpose. For example, the same code can always be executed on a system with an extended (rather than a restricted) ISA; also, certain patterns can be identified to use these extended operations; adaptations such as data restructuring for a different cache organisation may also be possible etc.;
- **multiple destination platforms**: the compiler will create multiple branches of selected code areas with an additional annotation or branching mechanism to select the appropriate branch once the execution environment is known. Accordingly, non-executable or non-optimised code would take up memory space, but would not threaten execution, if the parameters can be read correctly. Some compilers already exploit these features in order to address a scope of (similar) platforms in the best fashion (e.g. Intel IPP[2], Cray Compiler Environment). S(o)OS may use this information already during deployment to only select the appropriate branch.

Realistically, S(o)OS will take an approach that combines these options on different levels. Most likely seems the compilation to different destination platforms, whereby the individual destinations will be represented as generic and abstract as possible. The resulting code should have clear entry points for low-level adaptations, such as loop unrollment etc. through annotations during compilation process and according meta-information. These low-level adaptations are then executed on the fly, but only when necessary. Finally, when the destination deviates too much, or more complex adaptations are required, S(o)OS will trigger recompilation of the according segment.

It must be noted in this context that the choice of approach depends on multiple additional factors, the major one relating obviously to performance: not only should the approach realise efficient code that exploits the hardware features to its best, but the process itself should not conflict with the execution performance. Recompilation, adaptation etc. therefore should only be considered if the expected gain outweighs the cost for the according process – this, however, implies that the cost / benefit can be estimated beforehand. As this must base on the execution time of the respective code-segment, this requires a means to estimate the timing behaviour and compare it accordingly.

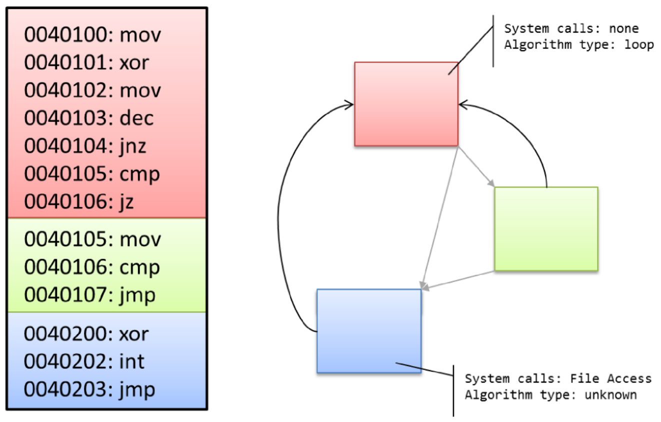
5.C) CODE ANALYSIS

The goal of code analysis consists in identifying potential points of parallelisation and, more importantly, the degree of concurrency implicit in the code. To this end, code analysis can take place on multiple levels:

- **on source code**: source code provides the level of concurrency as (intentionally or non-intentionally) provided by the developer. It furthermore provides the level of concurrency that is independent of the restricted memory architecture of the destination platform (i.e. registers instead of variables etc.) and is therefore on a higher level allowing for easier and more coarse segmentation. However, it does not contain any information about the actual code behaviour once fed with specific data (data-specific behaviour) and therefore cannot identify the size of parameter-bound loops;
- **on machine code**: the machine code level provides the low-level dependency information, including the memory related behaviour of the actual execution. On the one hand, this allows for more fine-grained and execution related segmentation. It also allows to watch the actual

[2] http://software.intel.com/en-us/articles/intel-ipp/

data-specific behaviour. On the other hand, it provides too much detail and makes segmentation more difficult.



```
0040100: mov
0040101: xor
0040102: mov
0040103: dec
0040104: jnz
0040105: cmp
0040106: jz
0040105: mov
0040106: cmp
0040107: jmp
0040200: xor
0040202: int
0040203: jmp
```

Figure 4: Basic blocks in an assembly code snippet and associated control graph.

The result of code analysis consists in a dependency graph. In the case of a highly scalable and mostly embarrassingly parallel application, this graph will be comparatively "sparse", e.g. you can identify loops that have high connectivity internally, but very little across iterations. Even though the source code based graph will look similar, the relation of amount of edges internal to and across the loops will be less clear (no count).

The resulting meta-information (graph and more) will enable a preliminary assessment with respect to the type of required resources (see next step).

5.D) RESOURCE IDENTIFICATION

Once the code has been analysed regarding its main dependencies, the resulting graph can be analysed with respect to the specific hardware features that would suit the execution best. This involves aspects such as:

- big segment => large code heap
- data from external source => minimal cache hierarchy
- high interactivity => close to user
- data dependency => little latency
- small independent loops => vector
- etc.

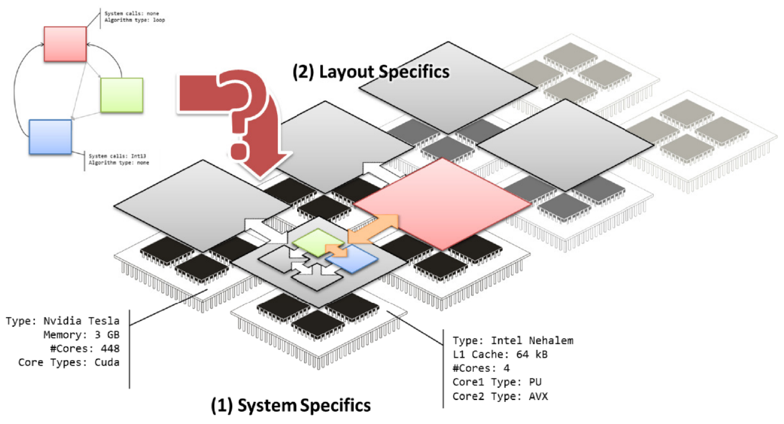


Figure 5: Layouting the code according to the infrastructure specifics

From this information, resource discovery is initiated, which tries to identify the most suitable resources fulfilling the requirements best. Notably, this does not mean that the code is restricted to this type of resources (cf. compilation above), but that it will execute less efficiently in other environments.

For example in the case of a ray-tracing application, the best resources would be graphics cards with shader units (GPUs) and large memory. Since it is not clear from the analysis that this is a graphics related task, the actual request will look for vector units, rather than GPUs with at least 3 pipes but the more the better, and high memory capacity. Communication will be secondary, but still exist at some point with a single unit, thus preferring a star shaped layout. The resource specific information can be enhanced through profiling the application behaviour, as well as the resource performance given a specific code type (see execution and post-processing).

5.E) EXECUTION PREPARATION

Prior to execution, the code will actually have to be adapted to the specifics of the destination platform - as already discussed in the development/programming step, this can take multiple forms. Within the context of this application, we can safely assume that a sufficient amount of graphics cards are actually available. Major point is rather to find the most suitable CPU for the execution of the main thread. We can furthermore assume that the developer knew that he/she was preparing a graphics related application and therefore specified GPUs as destination, which helps resource identification and code adaptation - this is not a general assumption though.

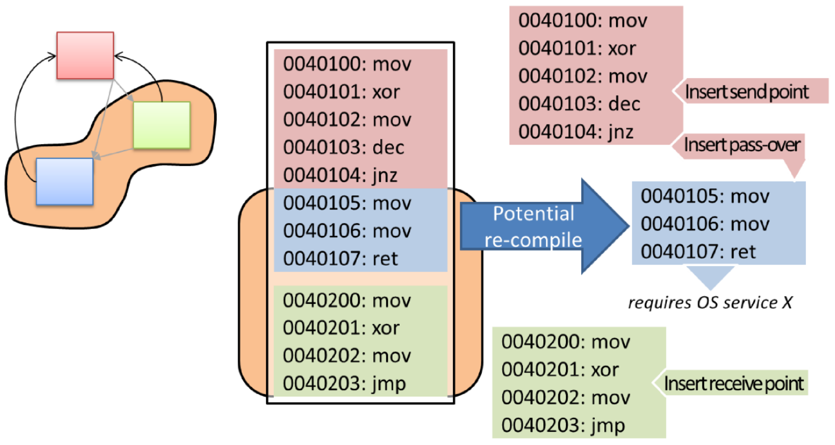


Figure 6:Different code segmentations and the impact on communication

If the code does not contain a GPU branch, it will have to be recompiled due to ISA compliance issues. However, only the individual render iterations need to be recompiled, whilst the rest is mainly CPU related and does not require recompilation.

Once the destination resources are known, they will be reserved and prepared, i.e. all necessary OS modules for maintaining communication, loading code, executing rendering etc. will be loaded, along with the actual code. Notably, the actual code and the accompanying data will potentially only be uploaded at the time of spread-out. Further to this, all communication routes have to be established and set up. This follows the communication dependencies (i.e. data source and sink) as identified in the graph during code analysis. Here again, the communication is mostly star shaped, requiring mostly direct routes to the main thread.

Notably, in an ideal case the S(o)OS environment will also investigate at this point the different potential schedules for all code segments to achieve the highest throughput / resource utilisation. We can generally assume that there are less vector pipelines than pixels and hence loop iterations in our example. Accordingly, each unit will have to execute multiple iterations itself - either leading to multiple full loop iterations one after another, or to a single operation on a range of data, storing them individually. It is potentially possible that there are say 1.5 times more pixels than vector units, leading to half the resources being idle half of the time. If we leave timing (12 fps or similar) aside, the free resources could be used to execute the next render iteration even after the sequential synchronisation part - potentially even purely speculative, if the interactions with the main thread are not fully known.

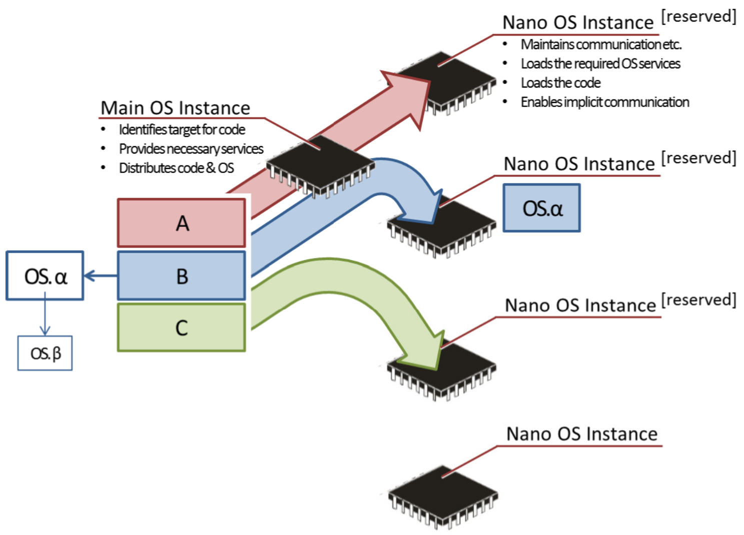

Nano OS Instance [reserved]
Maintains communication etc.
Loads the required OS services
Loads the code
Enables implicit communication
Main OS Instance
Identifies target for code
Provides necessary services
Distributes code & OS
Nano OS Instance [reserved]
OS.α
A
B
C
OS.α
OS.β
Nano OS Instance [reserved]
Nano OS Instance

Figure 7: Deploying code segments in according resources

5.F) Distributed Execution

The actual execution involves off-loading code according to spread outs, initiating and maintaining communication, as well as ensuring state availability and consistency. Thereby, "threads" should be as minimal as possible and effectively just involve code deployment and execution.

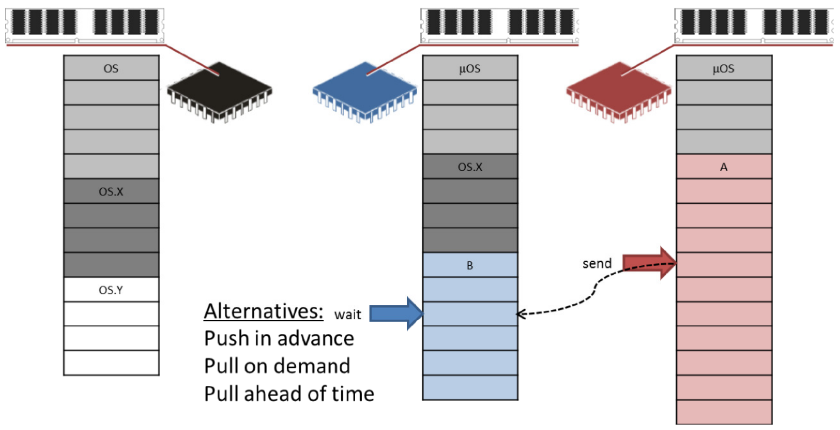


Figure 8: Memory allocation of code segments, share data and OS services

Furthermore, during execution additional performance and dependency data may be gathered that may trigger re-analysis (step 1), if too many discrepancies are identified. Notably, rearranging the code has to be reassessed against potentially incurring costs.

In our example, the main thread will prepare the data for rendering (e.g. calculating some interactions or the camera position) and then spread out the actual rendering threads. Since there is no communication between the threads, but only with the main thread, the main data dependencies exist during spread-out and aggregation (synchronisation), where in both cases data is stored in the main thread. However, the individual renderers should not communicate all individual calculations back to the main thread, as this would lead to too much overhead.

Since ray-tracing are "static" applications, their behaviour does not change a lot during execution, i.e. no re-arrangement of code should be necessary.

5.G) POST PROCESSING

In general, after execution the resources have to be freed for other processes again – this includes in particular freeing the memory, but in the case of distributed execution also the resource reservations have to be removed again, so as to make sure that the resources available for other processes. In addition, communication routes and configurations can be released again and the potentially deployed OS modules be removed, so that the resource can be discovered and initiated again (see section II. 5.d)).

However, S(o)OS will allow for storing information about the monitored/observed code behaviour, so that it can be reused for future executions. The recorded monitored data includes code-specific information, such as dependencies and algorithm types, as well as hardware-specific information. Such information may be greatly useful for deploying the same application again in the future with a better performance.

The analysis results can be stored at multiple points in the execution process, but should at the latest be stored during post-processing. It may be noted in this context that code-related analysis results may also be stored as extended annotations in the source code, if the latter is accessible.

III. Use Cases and Requirements

It has already been discussed in preceding sections how the concepts of S(o)OS aim towards supporting a generic hardware and software model. In other words, the concepts behind S(o)OS are designed so as to be generally independent of the underlying infrastructure and the type of application to be executed on top of it. This is due to the fact that S(o)OS principles behind identifying and addressing software requirements, respectively behind identifying and exploiting hardware capabilities are generally applicable to all currently known software and hardware models.

However, it must be noted in this context that not necessarily all applications actually expose characteristics that S(o)OS can exploit: some strongly sequential algorithms for example cannot efficiently be parallelised and distributed and thus will not benefit from S(o)OS specific support. S(o)OS specifically aims for supporting scalable application development and execution for and on large scale heterogeneous infrastructures. The project-related research and development will therefore specifically focus on scenarios and applications that can explicitly use the S(o)OS supporting features, thereby adding to the evaluation of the project success.

Accordingly, the most suitable use cases relate to applications with high-scale and specific time constraints, as detailed in what follows.

1. Specific Use Cases

Rather than considering the whole scope of the application scenarios as presented in D5.2 [15], S(o)OS will focus its efforts on use cases that can actually contribute to the S(o)OS objectives. In other words, that exhibit specific requirements towards hardware capabilities and expose behaviour characteristics that allow principally for segmentation and distribution.

With the overarching objectives of S(o)OS in mind (i.e. the capability to support large-scale heterogeneous environment), we can particularly identify two domains that specifically require support in the according directions and at the same time expose characteristics of relevance for S(o)OS:

1. **High Performance Computing**: HPC applications are generally explicitly written in a way that increases the level of concurrency, thus allows higher scalability with as little communication overhead as possible. HPC machines tend towards higher and higher heterogeneity, though at the same time, an according system typically offers a large number of each resource type.
2. **Real-Time Applications**: Though real time applications as such do not strive for parallelisation, there is an increasing demand for handling distributed real time applications. Of specific interest for S(o)OS are thereby the timing constraints of such applications which bears resemblance to the synchronisation estimation (i.e. execution timing) in distributed applications. In other words, by exploiting the timing constraints applicable in real-time scenarios, the distributed synchronisation behaviour can be improved further.

1.a) Scalable HPC Application

In the following we look at a generic HPC application that combines aspects of embarrassingly parallel and weak scale, which primarily shows aspects of vertical scale. Renderers and ray-tracers exhibit such characteristics, even though they can be considered almost only embarrassingly parallel. However, in combination with interactive controls or physics that shape the behaviour of the renderer, dedicated sequential tasks have to be executed which request for explicit synchronisation. Other applications exhibiting similar characteristics (i.e. vertical scale with both strong and weak coupling), are Grid -based calculations, such as aerodynamics, molecular flow.

Also, heavyweight 3D renderers (that need parallelisation) are commonplace in interactive real-time applications (think of virtual or augmented reality). The main difference between the deployment of a rendering engine in an HPC scenario and in a RT one is that, in the former case it needs to run as fast as possible, in a batch-computing fashion, whilst in the latter case it has normally a clearly stated deadline within which it needs to terminate, and a faster computation it is not necessary nor required.

In what follows, the main characteristics of a scalable HPC application are described, with particular reference to:

- spread out;
- synchronisation and communications;
- aggregation;
- example behaviour;
- post-processing.

SPREAD OUT

At dedicated points in the process, the main node will thus request the OS to instantiate a given number of processes or threads and to deploy them in the infrastructure. Generally, the developer does not designate specific nodes to these threads and the OS or middleware will have to take care of this step. Each spawned process / thread is assigned with a specific identifier which allows other threads to communicate directly with each other. Frequently, however, the main thread is used as a communication point, i.e. the main thread collects and distributes data - this follows the principle of shared memory machines, where all information is equally available to all threads.

Threads are effectively fully stand-alone processes where dedicated communication points and mechanisms ensure availability of data, respectively their distribution.

It must be noted here that creation and deployment of separate processes or threads is a relatively expensive operation, so creating threads with a very small workload may result in work distribution and synchronisation overheads that turn out to be higher than the potential speed-up in parallelising the workload, achieving a worse performance than a sequential execution would.

COMMUNICATIONS

Besides for dedicated synchronisation points, where all processes / threads get to share the same data, the developer can specify dedicated communication points which allow specific exchange of data between selected threads. In particular in cases of work (rather than data) distribution, this is necessary whenever the tasks show data dependencies. In most cases the communication is asynchronous, i.e. it does not expect a reply as in the remote process invocation case. Since communication implies a specific dependency between threads (even if possibly just in the form of an event), the receiving process is stalled until the data becomes available - accordingly, in the ideal case the data is sent prior to the point of requirement in the receiving thread, so that the according data is already available in the message queue. However, since this depends very much on the execution speed of both threads, this is hardly ever catered for. Similarly, hardly any programming model allow other communication models, such as an "early reception" or "pull model" etc. Again, this is simply due to the fact that the data will not be available early. In general, the data will be distributed as early as possible, thus - it is assumed - reducing any delay. Messaging overhead or bandwidth problems are not considered thereby.

It must be respected that the actual communication (protocol etc.) differs strongly depending on the system and the type of sender / receiver. We must therefore distinguish between:

- core-core communication
- processor-processor communication

- node-node communication
- (and all according intermediaries)

Hardly any programming model / compiler currently respects this mix but will instead select one communication protocol, non-regarding the overhead this implies. Modern developers therefore tend to hybrid OpenMP / MPI development.

SYNCHRONISATION

Following the shared memory model, most implementations choose the full distribution of data over all threads, as this is simplest. In general, this is an implication from the "data distribution" approach towards parallelisation. Synchronisation can either be realised as a point-2-point communication between all threads, or - more typically - as a communication with the main thread which receives and then distributes all data again.

A synchronisation point is implicitly a break point in a thread, as it will have to wait until all threads have finished their execution and provided the data. Similar to communication, this can lead to major delays in the overall execution. As the unit cannot perform other tasks during synchronisation, the resources may be under-utilised.

In both cases, communication, as well as synchronisation, the full communication processes have to be implemented in each thread's code, thus increasing the respective overhead.

AGGREGATION

After all threads have performed their tasks, the main node will aggregate results through a final synchronisation step. Once the units have been freed again, they can be used for the next iteration of spread out.

EXAMPLE BEHAVIOUR

In the specific case of a renderer, the main thread consists of hardly anything more than preparing the next iteration of distributed processes. Notably, synchronisation over a main thread is important to ensure that the frame rate of image displaying is maintained, as otherwise an individual unit may already process the data of the second frame, whilst all other units are still working on the first one – whilst this could be beneficial for resource utilisation, no current system is capable of dealing with this out-of-band data provisioning behaviour, in particular since an individual resource may run completely out of sync.

The individual render threads do hardly share any data, but instead receive their full information at the spread-out time and provide it back again at synchronisation and hence aggregation.

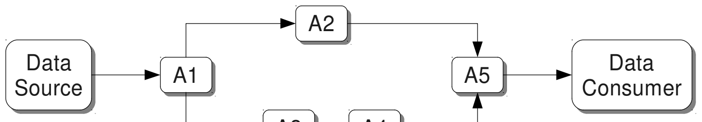
In the case of grid-based simulation, however, the individual threads will send information directly to other threads, as they pass over the "grid boundaries". A grid-based simulation, such as molecular behaviour, implies that specific particles or information units can move fairly freely over the dedicated space. Each grid thereby represents a part of this space and therefore only calculates all its designated particles in one time step. Once a particle crosses the boundary of a grid, i.e. moves into another grid, the according information has to be communicated to the processing unit being responsible for the respective "receptive" grid unit.

POST PROCESSING

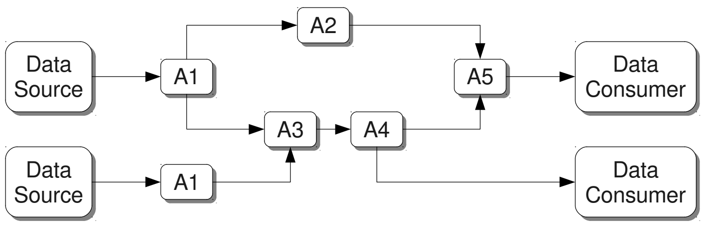
Once the main process has finished execution, the data is stored in the storage space of the user and the resources are freed for the main queue again, so that new jobs can be started. All assigned nodes are released again.

1.B) SCALABLE DISTRIBUTED RT APPLICATION

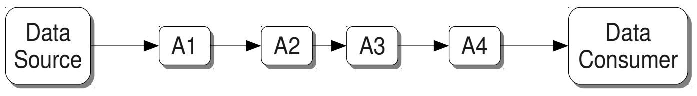
A distributed real-time application is composed of a number of computing activities being interconnected in a Direct Acyclic Graph (DAG) like fashion, as shown in Figure 9 (a). In the simplest case, the whole application is activated periodically by some data made available from a data source providing input, and having to be delivered to some destination peer/consumer after the whole chain (graph) of computations. For example, an interactive video editing or video streaming application would fit in such a model, in which the data source might be a camera providing continuously A/V frames to be encoded on a node local to the camera, then possibly processed remotely and finally delivered to other users on the network, or back to the same user. In such an application, it is commonplace to find a multiplexing logic separating the audio processing path from the video processing path, then joining the two paths later for further transmissions.



(a)



(b)



(c)

Figure 9: Real-Time Application composed of a DAG (a) with possibly multiple data sources and consumers (b), or the simple case of a linear workflow of activities (c).

One particular topology that we may want to consider is the simple case in which we have a pipeline of activities, as exemplified in Figure 9 (c). However, in the most general case, we may have multiple sources and multiple destinations, as shown in Figure 9 (b).

The application needs to respect the in-place real-time constraints:

- maximum **end-to-end latency** D, from when a new data frame is available from the source, to when the output of the computations is delivered to the final destination, at least with a given probability; D is normally sub-second (< 1s, e.g., a few hundreds of ms);
- in the case of multiple sources and/or consumers (Figure 9 (b)), independent end-to-end latency constraints can be expressed for multiple paths;
- **communication throughput** for all the links among activities need to be sustainable; links may exhibit heterogeneous requirements, e.g., like before/after compression for multimedia processing;
- data is coming from the source in the form of periodic/sporadic "frames" with a minimum known **inter-arrival period** of T – or, alternatively, we might have a probability distribution (model) for the incoming inter-arrival times.

Note that the individual activities in the DAG explicitly provided by the programmer may on their own be parallelisable. This means that the programmer will generally code them in such a way that it is possible to take advantage of available massively parallel hardware. Alternatively, the programmer might use a mainly sequential way of programming, but the S(o)OS run-time will be able to automatically try to realize speculative parallelisation so as to take advantage implicitly of the available massively parallel hardware also for the individual activity.

EXPECTED OS BEHAVIOUR

The application developer expects that S(o)OS helps (at run-time, as well as at application development time via proper interfaces and tools) in understanding how to exactly deploy the application on massively parallel hardware (e.g., how many parallel threads, how they are deployed, on which cores with what capabilities, etc.) so as to meet the timing constraints.

S(o)OS expects the application to cooperate (either explicitly or implicitly) in understanding whether or not its own timing constraints are being fulfilled, so as to be able to undertake corrective actions at run-time (i.e., changes in the deployment/scheduling decisions w.r.t. the initial configuration).

As compared to the HPC scenario, the real-time one mainly distinguishes because:

- there is no need to "take any free resource so as to go as fast as possible", but it is sufficient to merely "take the resources/cores that are needed to meet the timing constraints at least most of the time";
- S(o)OS will take care of handling more instances of this application, and/or of other applications with similar characteristics, so that they manage to run all at the same time, whilst the HPC scenario normally foresees a single application running over a clearly identified set of physical resources.

2. High-Level OS Requirements

Novel architectures, which embed more and more cores, and associated user requirements will pose new challenges that future OSes must address. The inherent additional concurrency of such contexts will shed more lights on problems related to consistency, including synchronisation of distributed entities, system recovery from failures, or even security.

High-Level OS Requirements

In the following, we review shortly the key high-level requirements on the software stack (and particularly the OS) for future massively parallel systems, as derived from the discussion in the S(o)OS deliverables D5.1 State of the Art [33] and D5.2 Future Requirements [15]. These high-level requirements have been identified in:

1. Scalability
2. Efficiency
3. Predictability
4. Usability
5. Adaptiveness
6. Heterogeneity
7. Modularity
8. Energy Management
9. Security
10. Robustness
11. Portability
12. Mobility
13. Compatibility

First, we describe shortly each one of these high-level requirements, in order to provide an unambiguous interpretation of these terms. Then, we connect them with more concrete and detailed low-level requirements on the OS which constitute a driver for the initial OS architecture that is sketched out in the following chapters.

SCALABILITY

Scalability is the ability for a system to make an efficient use of the available resources despite its size growing. Considering future massively parallel platforms, this means that the available parallelism degree can actually be exploited by applications, avoiding any major bottleneck (e.g., as due to excess of contention on shared resources) that may cause overheads (e.g., communication/synchronisation ones) to grow uncontrolled limiting the achievable performance.

We must distinguish between different types of scalability:

- **Horizontal scale** is a form of *replication*. This means that the application itself does not use multiple resources in order to increase its performance, respectively the precision of its data - instead, the full application is replicated onto multiple resources. Horizontal scale is typical for cloud application use cases, and it is needed to increase the application availability to a wide users community.
- **Embarrassingly parallel scale** belongs to the area of *strong scale*. In other words, the performance (or result precision) increases proportionally with the number of resources vested in the application. Embarrassingly parallel tasks are denoted by a strong independence of the individual working threads / processes, i.e. by little communication and messaging across. A particular application domain for embarrassingly parallel applications comes from the eScientists community, where the same calculation has to be applied to multiple parameters in order to get the full result scope. Implicitly, in some cases embarrassingly parallel relates strongly to horizontal scale. The main scalability aspect, however, relates to vertical scale.
- **Vertical scale** is the typical form of HPC application, where one single application is spread over multiple resources in order to increase performance or data precision. Generally, the individual worker threads share common data, which requires them to communicate across the infrastructure. Frequently, parallelised applications belong to the area of *weak scale*, i.e. the performance scales only to a certain point of saturation with the number of resources employed (cf. Amdahl's law [45]).

EFFICIENCY

The OS should allow for the efficient exploitation of the available physical resources. This means that the core OS components need to be efficient and highly optimised, and that they should not limit the actually exploitable concurrency degree (compare with the problems raised in the past by the Linux big kernel lock). Being the expert of the hardware, the OS should play the role of mediation with the applications in such a way that applications developers may exploit OS services that abstract away from the details of the underlying hardware, still with the possibility to make an efficient use of the available computing power, without wastes.

EASE OF USE

The OS should hide the complexity of the underlying hardware, and allow programmers to develop applications at a sufficiently high abstraction level. This allows for deployment of the developed applications on a number of different hardware systems (connecting with portability, see below), and also leaves the freedom to the OS to deploy various parts of the same application onto different and possibly heterogeneous resources possibly available in the same system (e.g., CPUs, GPUs, etc.)

HETEROGENEITY

Another essential requirement is the support for heterogeneity of the resources in a cluster or even inside the same system and chip (e.g., GPUs mixed with CPUs), in terms of hardware architecture. In this regard, heterogeneity should be supported more dynamically compared to what commonly done in nowadays OSes, where the code is compiled statically for one possible target, with perfect awareness of the available optimisations as due to the target ISA. Differently from the current programming approach which is static with respect to the destination platform (ranging from compilation to execution), the S(o)OS approach will allow to exploit a wider scope of destination resources more easily and efficiently.

ADAPTIVENESS

An important requirement is the one of providing sufficient support for adaptiveness of the applications, and the OS itself being sufficiently adaptive. This means that the OS and kernel should be capable of adapting its internal structure and configuration to the needs of the dynamically varying workload. Indeed, it may happen that the optimal way to realise certain functionality at the OS level, such as scheduling and deployment of tasks or specific OS functionality (e.g., system calls) to resources be dependent on the actual workload as imposed by the mix of the running applications. For example, when deploying HPC workloads or real-time workloads the OS might detect it and change its internal behaviour accordingly.

Finally, another aspect of adaptiveness is related to robustness and fault-tolerance. Whenever in a large many-core chip a failure of one or more units occurs, the OS should be capable of detecting it and re-configuring itself accordingly.

PREDICTABILITY

Predictability in the timing behaviour of the various parts of the software stack is also an important requirement for S(o)OS. Indeed, concurrent executions are inherently unpredictable due to the unexpected behavior of asynchronous executions of concurrent computational entities. A large-scale and highly concurrent OS should be able to cope with such an unpredictability to guarantee QoS to the user.

Both HPC and Real-Time workload scenarios may benefit of predictable execution. On one hand, in HPC having predictable duration of code segments allow for exploiting better the available computing resources, because tightly coupled and often-communicating code segments will waste less time on waiting for completion of (and messages from) each other. On the other hand, predictability in time duration constitutes a strong requirement in RT scenarios, in order to support properly a stable performance of the applications, in terms of providing the necessary QoS and timing guarantees (e.g., in terms of throughput, responsiveness, etc.).

MODULARITY

Modularity is the ability to write modular and composable code, both at the kernel level and at the application level. Improving composability involves a number of issues, like: having the possibility to use or not certain modules depending on the local application needs, without any impact on the performance due to the unused modules; ensuring the interfaces and communication paradigms of software components are compatible (e.g., synchronous versus asynchronous, binary representation of data despite the components may be running on heterogeneous hardware, etc.).

SECURITY

Security issues arise in modern operating systems, middleware components, services and applications for a number of different reasons. A comprehensive overview of them would be outside the scope of the present document. However, some of these issues have been due to certain inherent difficulties in coding, and particularly some of them have been attributed to the difficulty in writing bug-free parallel and distributed code and protocols.

For example, consider a modification to the file-system consisting of updating a file by first checking its associated rights. If an attacker concurrently modifies the file system name space between the access control check and the `open()` operation, this may result in overwriting a sensitive system file. OpenSSH (before version 1.2.17) suffered from a similar issue from a socket race exploit that allowed a user to steal another's credentials[3].

ROBUSTNESS

The OS should have support for robustness and fault-tolerance, for dealing with failures. First, in a massively parallel many-core system, it is foreseeable that hardware failures may occur. For example, in a many-core chip with thousands of computing units, if one of those units has a malfunctioning or does not work as expected, it is not acceptable nor needed to turn off the entire chip, as long as the units that work are still reachable and usable. Therefore, the OS should possess the needed sensing capabilities (exploiting of course hardware diagnostic features that also need to be present), and undertake the necessary corrective actions (i.e., dynamically turn off the malfunctioning core, or deciding to ignore a broken/malfunctioning memory bank etc.).

Second, in a massively parallel system it is foreseeable that software failures may happen across the chip, similarly to what happens in a large distributed data centre. Possible bugs at the OS and kernel level may show up more easily, once thousands or even more kernels are instantiated within a single chip and cooperate very tightly. It should be possible to safely restore the OS/kernel status to a safe and healthy one, whenever such failures are detected.

ENERGY MANAGEMENT

Energy management constitutes an important requirement on OSes for future massively parallel systems. First, the OS will have to support intelligent configuration of the hardware in such a way to save energy whenever possible, both for properly supporting mobile devices (see Mobility below), and for "green-computing" purposes. In particular, the sensibility of the HPC world to green-computing issues is gaining more and more interest, recently.

For example, it is important that the OS is capable to detect when there are nodes inside a system or cores, interconnect and memory elements inside a chip, that are not used, so as to be able to shut them down. Also, the OS needs to properly exploit idle and deep idle states of the computing units whenever available from the hardware, in order to let the system save energy for example while waiting for messages to be received from the network.

Also, it must be noted that future many-core systems targeting unprecedented performance will suffer of major issues related to dissipating heat. In order to properly cope with this problem, one of the envisioned trends in system design is the one of allowing for temporary overheating of parts of the chip, in order to boost computations for a certain time, then falling back to a normal safe operation that allows for cooling. For example, the Intel TurboBoost[4] technology is practically a hardware-driven over-clocking of the core controlled by hardware by means of a feedback loop involving local thermal sensors.

In the future, manufacturers of systems based on 3D chips design will push even further such an approach by relying more and more on software, for the proper management of overheating situations [42][43][44].

[3] More information can be found in the SSH FAQ: http://www.employees.org/~satch/ssh/faq/.
[4] More information is available on the Intel website: http://www.intel.com.

PORTABILITY

In nowadays HPC programming practice, developers often focus on a single type of target hardware (i.e., the system over which the application is going to be deployed), and also OSes provided by HPC systems manufacturers are sometimes optimised for specific underlying hardware capabilities. However, as already mentioned, one of the goals of S(o)OS is also the one to provide a sufficient abstraction level that allows application developers to program on a wider range of hardware systems (this follows the requirements coming from ease of use). Therefore, the OS will support a wide range of possible hardware devices, something that makes it easier to deploy applications on data centres with heterogeneous systems, as well as the deployment on completely different centres.

Also, the OS will have to support a wide range of devices and peripherals, as commonly seen in nowadays OSes. Especially if the OS will be adaptable to be used both on client-side ends (e.g., a mobile device – see also the Mobility requirement), and on server-side ones (e.g., HPC resources remotely available to the client), then the range of supported hardware devices and peripherals will have to be wide. This is possible in S(o)OS thanks to the modularity of the OS, that allows various features that can be made available as modules only where needed, and be actually loaded into the OS only when needed.

MOBILITY

The development of applications for future massively parallel systems, and in particular of real-time and interactive ones in which users interact on-line with a massively parallel application (e.g., distributed session for editing huge multimedia data as required in the film post-production industry, or on-line gaming with heavyweight physics simulations, etc.), will likely need to support both mobile, possibly lightweight hosts, under the direct control of users, and heavyweight, massively parallel ones available remotely. S(o)OS will support such scenarios, allowing the lightweight device to hand-over computations to the remote side whenever needed and possible, in order to save energy (see also the requirement on Energy Management).

COMPATIBILITY

Another OS feature that is of secondary importance due to the disruptive innovation nature of S(o)OS is the one of backwards compatibility. Of course, it would be nice to have, in a future OS, a compatibility layer that allows developers of nowadays applications to easily port their applications to the new run-time environment, still taking advantage of the new OS features and capabilities. For example, a POSIX-compliant (or Unix-compliant) software layer may be available for such purposes, enhancing the chances for developers to write portable code across a number of heterogeneous Oses.

2.A) DISCUSSION

Here we summarise the major key points in how S(o)OS will be able to tackle the above described high-level requirements.

SCALABILITY

A direct implication of the scalability requirement onto the design of the OS is the need for avoiding any type of centralisation in resource management, supporting dynamic behaviours and tolerating failures.

Ensuring scalability of various OS components that are not embarrassingly parallel is far from being trivial due to inherent contention. For example, in a concurrent environment a file system should support multiple modifications from various cores at the same time. Such a concurrency is allowed for example by transactional soft updates [12] of FreeBSD, but not by BSD 4.2 [1]. In Linux, a recent

VFS scalability patch[5] addresses bottlenecks arising when many cores are performing filesystem operations at the same time. As a further example, when considering the process scheduler of the OS, it is important to avoid a centralised data structure, that would quickly become a bottleneck for the whole system when the number of cores grows, or at least scalable data structures should be used [50].

Also, while accessing data structures shared across multiple cores, it may often happen that the modifications can actually be made concurrently, because they deal with different independent parts of the data. The transaction programming abstraction can easily enhance scalability in such cases, as opposed to alternative techniques, generally using locks to handle the contention between concurrent accesses. Roughly speaking, the analogy between lock-based and transaction-based programming lies typically in the mapping of critical sections (wherein correctness validation is prior to execution) to transactions (wherein execution is performed in a speculative manner before correctness validation).

Even though the transaction bookkeeping and additional aborts and restarts can penalize performance, transactions have shown impressive scalable results in various applications [13] due to the lack of overly conservative protections.

A critical aspect in supporting scalability of software is that, in the nowadays programming practice, scale is limited to the amount of threads explicitly specified by the developer. With the S(o)OS approach the amount of used resources will be more dynamic, whereby the actual number can exceed the amount of explicit threads through implicit concurrent execution, whenever possible.

EFFICIENCY

One of the main implications of the efficiency requirement is that the OS itself should be scalable in the number of cores, threads, processes, applications, without suffering of major performance bottlenecks and overheads becoming overly predominant when the number of managed items grows higher and higher. Also, the OS services should allow developers to write scalable applications. Therefore, the efficiency requirement partially overlaps with the one of scalability. However, efficiency means not only that the performance can scale with the number of resources, something referring to the complexity by which performance grows for example in the number of cores, but also that the absolute achieved performance should not be too far away from the theoretical one.

Also, an implication of efficiency is that the OS may dynamically re-allocate resources whenever in need. For example, if delays are predictable then they can be exploited for executing other tasks, and free resources can be provided to other processes.

EASE OF USE

One aspect of easing the use of a platform is that the OS may offer a single system image (SSI) vision for using all, or a subset of, the available resources, especially whenever such resources are handled in a distributed-system fashion (i.e., by different OS instances running within the same node, board and/or chip). This would allow for instance to launch an application execution from anywhere with the possibility to be able to access the whole application data set, without taking care of the real execution location. This may be useful also at an in-chip level, where it may basically realise in software a coherent view of the memory, that is not realised in hardware due to the potential performance and scalability drawbacks. However, such a feature is well-known to potentially lead to scalability and efficiency issues, if the programmer does not use it properly, as it easily happens in SSI clusters realised on a set of distributed nodes. For example, using a programming model similar to P-GAS may mitigate such problems. Also, when considering realisation of an SSI-like view over

[5] For more information, see for example: http://lwn.net/Articles/401738/.

clusters of cores in the same chip, it must be noted that the mentioned performance drawbacks may show up less often, as inside the chip it may be extremely easy to move memory pages across different OS instances running in different cores and tiles.

One of the key issues is to design proper trade-offs in the OS between programmability and efficiency. For example, in order to reduce contention with a high number of cores, the OS can adopt more fine-grained locking strategies. A commonly known move along this direction was the one of the key improvement from the 2.4 to the 2.6 Linux kernel series, where some critical sections using the big kernel lock were shortened, by releasing and re-acquiring the lock more frequently when possible (e.g., `kmem_getpages`), and the later progressive removal of all the big kernel lock usages, shifting towards more fine-grained locking strategies. However, despite the performance increases, this may easily lead to difficulty in maintaining the code in later developments. For example, the Linux kernel source file `linux/mm/filemap.c` starts with a header containing 50 commented lines to explain in which order the different elements are locked.

Due to the support through code analysis and segmentation, as well as through explicit code annotations, programmability should be simplified so that no low-level annotations, such as required by MPI or OpenMP should be necessary any more. Instead, the OS related tools will analyse the code for concurrency and parallelisation aspects, and the OS will take care of the low-level communication and state maintenance aspects, thus relieving the developer from this burden. Instead, high-level annotations will steer the OS behaviour for improving the analysis, segmentation, distribution and deployment of the code segments.

HETEROGENEITY

One possibility to support better heterogeneity is to allow developers and/or compilers to annotate the (machine) code with the requirements on the underlying hardware capabilities. This information may be exploited when the tasks are submitted to the distributed OS in order to ensure the execution with the most appropriate resources.

Differently from the current programming approach which is static with respect to the destination platform (ranging from compilation to execution), the S(o)OS approach will allow for exploiting a wider scope of destination resources more efficiently. For example, it would be desirable to have an approach resembling on-the-fly recompilation (e.g., just-in-time compilation) of code segments, whenever needed to potentially improve performance. The OS will possess the needed degree of adaptiveness and dynamicity that allow for the maximum flexibility in allocation of tasks and generally code segments to available resources.

The principle is related to Grid / WS service discovery, where providers are identified according to the task needs. In S(o)OS, we advocate the analysis of the code for its specific needs towards the hardware and the identification of the most suitable resources, rather than restricting the code to a certain precise platform.

ADAPTIVENESS

The OS should be adaptive in that it may monitor (via proper interfaces and collaboration with the application) the performance being experienced at run-time, and undertake corrective actions on its internal behaviour (e.g., on the scheduling and deployment side).

For example, consider those novel OS approaches for many-cores [33], advocating the use of dedicated kernels and cores for specific OS functionality (such as interrupt management or system calls execution). One of the issues of such approaches is the one of dimensioning properly the OS. Indeed, the correct configuration in terms of the number of execution units dedicated to each specific OS functionality may depend on the actual workload type, that needs to be dynamically sensed at run-time.

Concerning the support for adaptiveness at the application level, the direct implication on the OS architecture is the need for providing to applications the sufficient “sensors” so that applications can adapt to the dynamically sensed run-time conditions. Furthermore, whenever multiple adaptation loops are present at multiple levels of the software stack (application, middleware, OS services), they might need to be designed so as to operate in a coordinated way [51], as opposed to an independent operation that might sometimes lead to undesired or sub-optimal results.

PREDICTABILITY

In a S(o)OS environment, the code execution will be more predictable. This is achieved both by exploiting proper tools made available by the OS (e.g., for profiling the execution times of applications, task bodies, code segments, functions), and by run-time mechanisms (such as scheduling) designed around adaptive policies that are fed by data coming from properly designed monitoring services.

When considering transactional memory support, it must be noted that transactions and more generally speculative executions add another layer of unpredictability due to the unexpected impact of conflicts. A conflict may cause a transaction to abort, roll-back and restart a large number of steps. In other executions, the conflict would simply be avoided due to the particular interleaving of transactional accesses. Conversely, transactions are complemented with a contention manager that usually arbitrates adequately between conflicting transactions. In RT scenarios, a future OS should provide deadline-driven contention management, that is capable of favouring those tasks whose completion is more urgent over the others.

SECURITY

One of the ways to deal with security at the OS/kernel design level is the one to enhance the programmability of parallel and concurrent applications and services. For example, providing the support for easier programming models (e.g., transactional memory) or letting the OS expose synchronisation primitives which are easier to use and understand by the developers, would easily decrease the likelihood of having security issues related to bugs in parallelisation and synchronisation logic.

However, it should be noted that such approaches may non-necessarily be undertaken across the whole OS and kernel. Indeed, parts of the kernel which are performance-critical, may still need a number of optimisations and complex synchronisation logic that simply cannot or should not be simplified too much, lest the achievable performance or scalability.

ROBUSTNESS

For example, whenever a kernel failure occurs, it is again non-acceptable and probably unneeded to cause a major shut-down and reboot of the whole system/chip. Instead, the specific core over which the kernel failed may be shut down and rebooted safely, especially if the memory that was accessible by that core was not the whole memory available in the chip, but merely a subset. For example, a similar approach was attempted also on Linux [41] when running on multi-core machines, where it was shown how to keep an application running without failures, despite a kernel failure on one of the cores of the machine.

Furthermore, whenever dealing with a stronger notion of fault-tolerance in a distributed environment, of course checkpointing is of crucial help for OSes to ensure that a system always act on a consistent state of the kernel. Checkpointing is already used in common OSes to make sure that a partial system upgrade does not let the system in an inconsistent state. For example, Microsoft OSes already checkpoint key structures, like registries and some system files, or sometimes the entire file system, to cope with software update problems [8].

WAFL file system [11] ensures that an old bunch of file system data gets updated atomically by recording them in a tree structure whose root is updated in a single step. The Sprite LFS [14] are log-structuring file systems that write all filesystem data in a continuous stream, the log. Although it makes information retrieval more complicated, it is easy to mark the point in the log up to which data have been committed and stored and it benefits from consecutive writes without disk seeks. Finally, journaling (i.e., log enhanced) FS keep old and new versions of incomplete updates on the disk. Only after commit, the new version is copied back to the original location as in Linux ext3.

Apart from the filesystem, when dealing with other generic parts of the OS/kernel, one possible way to deal with failures is also a transactional programming model, as opposed to a lock-based one. Indeed, transactions provide failure-atomicity, as it may be easier for the run-time to simply discard the uncommitted changes of a code segment that failed before the commit phase, as opposed to having to checkpoint and restore the previous version of the objects state. However, checkpointing may still be used jointly with transactions, because the failure of a complex software component may involve the need for rolling back a number of commits made by that component during the failure, something that can be easily accomplished by restoring a previous checkpoint known to be safe.

2.B) PRIORITISATION OF THE OS REQUIREMENTS

The high-level requirements on the operating system as detailed above are not all equally important for the purposes of the investigations carried out in S(o)OS. Therefore, those requirements have been categorised into the ones that are considered “mission-critical” for the project and thus they have a high priority for us, the ones that are considered nice optional additions (low priority), and the ones that are actually out of scope for S(o)OS. This prioritisation of the high-level OS requirements is summarised in Table 1.

Requirement Id	Requirement Name	Requirement Priority
HR1	Scalability	High
HR2	Efficiency	High
HR3	Heterogeneity	High
HR4	Usability	High
HR5	Adaptiveness	Low
HR6	Predictability	Low
HR7	Modularity	Low
HR8	Energy Management	Out of scope
HR9	Security	Out of scope
HR10	Robustness	Out of scope
HR11	Portability	Out of scope
HR12	Mobility	Out of scope
HR13	Compatibility	Out of scope

Table 1: Prioritisation of high-level OS requirements.

2.C) LOW-LEVEL REQUIREMENTS ON THE OS

In addition to the above described high-level requirements, a number of concrete low-level requirements have been identified as a result of the analysis conducted in Section VII.2 of D5.2 [15]. These can be summarised (and somewhat aggregated) as:

1. support for **horizontal scale** (multiple replicas of the application to support many users), including instantiation of the application;
2. support for a **message-passing** programming model at the application level;

3. **implicit data coherency**, i.e., support for globally coherent shared memory (for ease of programming);
4. support for the realisation by the (expert) programmer of **explicit data coherency** protocols, for increased efficiency/scalability;
5. support for **low-latency** workloads (for real-time applications);
6. support for **high-throughput** workloads (for HPC applications);
7. proper scheduling of CPU, network packets, I/O requests, supporting predictability and QoS awareness (**QoS-aware scheduling**);
8. **predictability** in the duration of **OS** components, including scheduler and memory allocator;
9. **predictability** in the time behaviour of **applications** (e.g., proper scheduling of applications/tasks on proper nodes so that the performance is predictable);
10. proper on-line **monitoring API**, so as to enable adaptiveness at the application level;
11. support for **asynchronous I/O** services (pre-load next data frame while processing the current one);
12. support for **explicit parallelisation** and for **synchronised execution** in a distributed environment;
13. support for **DFG-like applications**, and support for attaching performance requirements on DFG nodes and arcs;
14. support for expert developers allowing to **query underlying hardware capabilities**; for example tuning the parallel algorithms so that the needed data fits into the cache size (i.e., the application needs precise information on the underlying hardware), and/or the required latency/bandwidth fits within the available capabilities; in addition or alternative, allowing developers to explicitly state what are the requirements on underlying hardware (including cache size, bandwidth and latency requirements), so that the scheduling logic can select proper resources, among the heterogeneous available ones;
15. support for **embarrassingly parallel applications**: high bandwidth, latency unimportant;
16. support for tightly coupled parallel applications (**HPC**) and vertical scale: more importance to the control of latency and in general predictability (and little variation) of code execution time;
17. **distributed execution** of both the applications and the OS, avoiding centralised contention sources, e.g., globally shared data structures or services within the whole many-core system – they become easily bottlenecks;
18. support for **disabling time-sharing** if needed (HPC case), to minimize overheads;
19. support for **code segmentation**, and on-the-fly instantiation/migration of segments, with minimum overheads;
20. support for some form of “**sand-boxing**”, for enhanced security;
21. support for direct access (by expert developers) to **hardware acceleration** (e.g., use of GPUs, FPGA, SIMD-like extensions, processor pipeline control when available, data pre-fetching, etc.): this may contrast with ease of use, but it may be needed for increased and leading-edge performance;
22. **Ease of installation/deployment;**
23. **Ease of maintenance/administration;**
24. **Resource sharing** (as opposed to partitioned resources), such as via multi-tasking (i.e., time-sharing scheduler), particularly important for RT workloads;
25. **application-level reliability** through proper support for fault-tolerance mechanisms, such as checkpointing;
26. Support of a wide range of available **peripherals;**
27. **Stability of the OS API/ABI** over time.

The table below connects each of these concrete low-level requirements with the high-level requirements described in the previous subsection. Each mark in the table means that there is a direct relationship between the corresponding high-level and low-level requirements, however the exact implications of this are not described in this section (more information is provided throughout the document).

		HR1 Scalability	HR2 Efficiency	HR3 Heterogeneity	HR4 Usability	HR5 Adaptiveness	HR6 Predictability	HR7 Modularity	HR8 Energy Mgmt	HR9 Security	HR10 Robustness	HR11 Portability	HR12 Mobility	HR13 Compatibility
LR1	Horizontal scale	X	X								X			
LR2	Message-passing	X											X	
LR3	Implicit data coherency				X									
LR4	Explicit data coherency	X	X											
LR5	Low-latency		X				X							
LR6	High-throughput	X	X	X										
LR7	QoS-aware scheduling	X	X				X	X						
LR8	OS predictability	X	X	X	X	X	X							
LR9	Application predictability						X							
LR10	Monitoring API		X			X	X							
LR11	Asynchronous I/O		X					X						
LR12	Explicit Parallelisation	X	X	X										
LR13	DFG-like applications				X		X							
LR14	Hardware capability querying		X	X			X							
LR15	Embarrassingly parallel apps	X	X											
LR16	HPC applications	X	X				X	X						
LR17	Distributed execution	X	X	X		X	X				X			
LR18	Disabling time-sharing		X					X						
LR19	Code segmentation		X	X	X	X		X						
LR20	Sand-boxing									X				
LR21	Hardware acceleration		X	X										
LR22	Ease of installation/deployment				X					X				
LR23	Ease of maintainance/admin				X					X				
LR24	Resource sharing		X	X										
LR25	Application reliability										X			
LR26	Peripherals				X							X		
LR27	Stability of API/ABI over time			X	X									X

Table 2: Relationship between low- and high-level requirements

IV. OS Architecture Models

In this chapter we discuss the major architectural concepts behind S(o)OS.

1. Background

In order to overcome the main limitations of nowadays OSes in terms of scalability in the number of handled resources (cores), the S(o)OS OS is designed around distributed computing principles (see the multikernel, FactoredOS and CoreyOS approaches discussed in [33]).

As already described in section II. 2. , the key factor thereby consists in building up a *modular and distributed* operating system, where each module represents an essential OS level functionality that is exposed to the program and the rest of the system in a fashion similar to a "service". This differs essentially from existing OS architectural concepts.

1.a) Current OS Architecture Concepts

The nowadays industrial practice in developing operating systems cannot probably be considered any more as following the traditional *monolithic* approach, however it is still strongly oriented towards *centralised* architectures that incorporate all essential functionalities within the main kernel, plus additional functionality that is dynamically loaded on-demand (e.g., device drivers for specific peripherals, or optional OS capabilities). The centralised architecture allows for tuning the OS so as to achieve good performance under certain specific scenarios (e.g., web servers, desktop), thanks to various optimisations that can be embedded within the kernel. However, it also leads to a number of design problems. For example, the OS may become rather bulky and in particular difficult to adjust: due to the large amount of internal dependencies between the OS functionalities, a single change may affect thousands of lines of code [29]. Additionally, all kernel-level code runs with the same privilege level, leading to instability issues and potential crashes of the whole system as a result of a buggy device driver for an unimportant peripheral (just to provide an example). These issues have been overcome by the introduction of *microkernel* architectures. In these, the kernel was reduced to the bare minimum set of functionality, implemented through a very small code-base that can be more easily verified, maintained and kept under control. All the additional (but still essential for the OS) functionality can be implemented as user-space modules forced to communicate to each other via a message-passing paradigm. This way, additional functionality is relatively easier to realise and comparatively more stable than in traditional monolithic kernels. Indeed, modern OSes incorporate some aspects of micro-kernel architectures (see [33]).

While the micro-kernel approach has been proved to increase greatly the robustness of the OS, it also showed serious performance limitations, what hindered the development and adoption on a wide scale of microkernel-based OSes in those contexts in which performance is important.

Additionally, traditional OSes have serious problems to adapt to the expected increase in heterogeneity even in desktop PCs, due to the difficulty in handling differences at the ISA level in available cores.

Further to this, nowadays OSes are stand-alone systems that can only interact with other systems through dedicated communication channels – typically on higher level, such as the middleware layer. This strategy is typically exploited in typical multi-processor HPC clusters, where each processor hosts its own operating system and cross-process communication is executed through explicit invocations, e.g. through MPI. This kind of set-up of the software stack can reach a considerable high scalability on nowadays HPC platforms.

However, with the foreseen introduction of many-core processors, multiple thousands of processing units are expected to build up a single processor, and accordingly share the essential resources, such as memory, hard-drive, I/O etc. The centralised structure of the OS can easily turn into a bottleneck, where the individual processes may block each other in attempts to accessing the OS functionalities which may block for example on spin-locks for accessing common data structures.

Therefore, the nowadays centralised structure of OSes and their kernels, which pretend to have a global/centralised view and management of the available underlying physical resources, cannot scale when the amount of cores goes to thousands or even more.

Even when considering micro-kernel architectures, these are still used with a single OS instance serving the multitude of physical resources available to a single node. Therefore, they run into similar communication issues and thus do not generally show better scaling behaviour.

Recently, *distributed-kernel* approaches have been proposed (such as the already mentioned multi-kernel), or the distributed Linux kernel as available on the Intel SCC) to address these issues. Instantiating a full OS kernel instance per core, the scalability is increased as due to the reduced contention on common resources. However, as highlighted in [33], a large number of issues are still unsolved. First, when instantiating different kernel functionality on different cores, it is not clear at all how to dimension the OS/kernel layout. Second, the distributed nature of the OS also leads to usability/programmability and potential efficiency problems (e.g., if memory needs to be copied from one core to another one, when simply a pointer could be passed by), and also the heterogeneous nature of modern multi-core architectures remains mostly unsolved.

1.B) General Concepts

One of the key drawbacks of the distributed-kernel approaches is on the side of programmability. Indeed, core to core communications are constrained to follow a message-passing paradigm, and the classical way applications deal with parallelism, i.e., spawning new processes or threads, do not work any more, because the only way to exploit the underlying parallelism is to have another instance of the application being deployed on a different core, i.e., OS node. These limitations can be overcome by reusing concepts from the world of Single-System Image (SSI) kernels. Indeed, an SSI-enabled OS is able to migrate a process to a remote physical node, handing to it all the needed data by exploiting the virtual memory mechanism. Similarly, an SSI-enabled distributed kernel may be able to hand over a new process to a different OS instance running on a different core in the same chip. However, differently from the classical SSI case, in a many-core chip the SSI mechanism may exhibit a much higher performance, being able to exploit the incredibly fast core to core communications available inside the chip. Not only, whilst in traditional distributed SSI systems memory pages of a migrating process need to be transmitted from one physical system to another, in a SSI-like distributed kernel within a many-core chip the memory may already be globally shared. Therefore, memory pages may be directly accessed after a much simpler cache flush operation (in case cache coherency will not be available in hardware), without any need for actually copying data.

However, the visibility of resources under an SSI model should not be brought beyond a certain limit. If, from one side, the clustering of more (closeby) cores together into a virtualised “single” SSI-like kernel may overcome on the software-side the limitations in programmability due to the lack of a completely coherent memory architecture across these cores, on the other side, as it is well-known, there is a risk of increasing the number of resources each kernel instance needs to be aware of.

Therefore, in S(o)OS we advocate a solution in which:

1. a few homogeneous cores close to each other which are sharing a coherent view of the memory (i.e., the same L2 cache in the same tile) can actually run a single kernel instance, similarly to nowadays multi-core, SMP systems;

2. a set of kernel instances which run on a cluster of cores/tiles that are close to each other (thus they exhibit very efficient communications), and which are homogeneous, can easily be clusterised to form a single virtualised OS/kernel by using SSI-like mechanisms (but with a far higher efficiency than traditional SSI);
3. different clusters of potentially heterogeneous cores within the same chip, or in different chips, run OS/kernel instances which are actually independent from each other, and that communicate exclusively by message-passing primitives (which may be highly optimised to exploit again the efficient in-chip communications).

Additionally, with the typical layout of S(o)OS on the various heterogeneous cores of the same many-core physical machine, it will be possible to have:

- multiple instances of the same OS/kernel, with a different subset of modules/components loaded and active;
- multiple instances of different OS/kernels, possibly obtained by compiling the same sources with different options: for example, GPGPUs computing units may host a minimum run-time environment needed to move data and code and perform the needed batch processing, whilst general-purpose computing units will feature more complex features like scheduling, context switching, interrupt and peripheral management, etc..

This way, each OS instance will have to deal with a limited number of local resources (cores and interconnections) and run-time entities (processes, threads, files, etc.), resulting in an improved efficiency and scalability of the kernel-level operations of the individual instance. On the other hand, resource management and scheduling across the whole system will need to be done similarly to how load-balancing techniques are designed in service-oriented and distributed systems nowadays.

The immediate drawback of the distributed architecture is that both data and code may sometimes need to be replicated, in order to allow different OS instances to coordinate among themselves for performing a distributed operation.

Note that, considering the span of the individual OS instance, previous distributed-kernel approaches proposed a single OS instance per-core. However, we believe that having each OS instance control a subset of cores may be even more advantageous- this ultimately depends on the availability of a coherent memory view at least across subsets of cores. For example, in a tile-based architecture, if each individual tile has a multi-core layout, then it is easy to think of these cores having the same coherent view of the memory, both of the local memory possibly available in the tile, and of the remote memory accessed through the Network-on-a-Chip (NoC).

The advantages of having an OS organised as a distributed set of multi-core OS instances rely in the lower memory overheads required to access common/shared data and code. For example, cores in the same tile will likely need to run similar OS services, and sometimes they will need to share the same data. A shared-memory model is still convenient to be exploited by applications and developers, in order to increase programmability and ease of use of the underlying architecture.

In the context of RT applications, global scheduling algorithms, applied to a limited set of cores, leads to an easier deployment of the RT tasks over the system and to a better exploitation of the underlying resources, without suffering of the scalability issues arising when the number of managed cores increases over and over. At the same time, the distributed scheduling algorithms, that need to coordinate among the various OS instances for deploying and load-balancing the active tasks, are simplified, as they need to deal with a lower number of deployment targets (OS instances), potentially achieving a better performance due to lower overheads for the distributed scheduling decisions. For example, deploying optimally real-time tasks over a multi-core machine is well-known to be an NP-hard problem, which becomes quickly intractable as the number of cores grows.

2. S(O)OS ARCHITECTURE CONCEPTS

Analysing the behaviour of a single distributed application it becomes obvious fairly quickly that the parallel threads / processes typically expose similar behaviour and thus similar OS needs. Even if the threads are not "just" replicated instances of the same task, hardly any thread requires the full OS system support for its execution, as e.g. it does not spawn further threads, does not require scheduling, does not access the hard-drive etc. Instead, they only execute certain subtasks and thus only call system subroutines. Implicitly, it is not necessary to provide the full OS instance per core, if the core is only to execute threads / processes with limited OS requirements.

No current OS architecture supports separation and distribution of the kernel itself though and instead require that either a full instance is available, or that a central instance has to be called. The essence of the S(o)OS approach consists in not only supporting the segmentation and distribution of application code, but also of the operating system itself. To this end, the operating system is split into individual modules that expose their capabilities not unlike services in a service-oriented architecture. The individual modules can act standalone and incorporate the capabilities to interact with other services in order to function properly. The modules are furthermore selected to suit the respective processes' / threads' needs, i.e. to support inter-process communication, access to storage etc.

As already described S(o)OS supports the whole application lifecycle by identifying relationships and requirements of code segments and distributing them across the infrastructure. As part of this process, the dependencies with the essential OS capabilities are identified and used to decide which functionalities to host parallel with the code, and which ones to make accessible via interconnect. Along that line the S(o)OS also has to maintain execution state consistency by introducing communication commands at the respective points, respectively by introducing according system calls into the process.

It will be noted first of all that the S(o)OS incorporates more functionalities than the core essential OS capabilities that are promoted by most microkernel approaches. In other words, S(o)OS incorporates functionality that is typically associated with the middleware, rather than the operating system. As noted in [47] there is a growing tendency to overcome programmability and management issues by adding further middleware layers on top of existing ones, so as to increase the capability scope. This approach not only leads to increasing misalignment between these layers, where some functionality is reproduced on multiple levels, but as a consequence also to a severe reduction of performance.

Meanwhile, certain middleware functionalities have been well established and for example Microsoft already exhibits tendencies to integrate the .NET framework into the OS architecture. A major aim of S(o)OS is to incorporate the capabilities that are essential for large scale distributed code execution over a heterogeneous infrastructure into the operating system, so as to increase their performance and expose their capabilities as programming extensions to the user. A recurring discussion in this context relates to kernel versus user space which we however want to avoid in this first iteration for the sake of concentrating on the capabilities in the first instance.

In the following we therefore treat typical middleware features, as well as kernel and user space as one thing, subsumed under "OS architecture".

2.A) GENERAL S(O)OS ARCHITECTURE

S(o)OS incorporates execution and hardware management capabilities to deal with handling application execution on top of large distributed heterogeneous infrastructures. To this end, the OS itself is split into essential *modules* (or *services*) that provide functions relevant to this distributed execution. Each of these modules can in turn communicate with each other via different means, making the OS itself a loose set of services that can potentially be distributed themselves. From the perspective of the application, the interactions on OS level thus become invisible in the sense of "virtualised".

The main goal is thereby to structure the OS modules in such a fashion that, from the application perspective, a complete virtual *local* environment may be provided. This includes not only resources and OS functionalities, but implicitly also state and data. Similar to the PGAS programming model, the developer gets an API that exposes a shared memory machine, even if the underlying infrastructure is a NUMA cluster machine. As opposed to PGAS though, the S(o)OS support will distinguish between different system architectures to make the best protocol decisions. To put it in current computing terms, if PGAS were employed, S(o)OS would translate data access to

- simple direct access in a ccNUMA or shared memory environment;
- OpenMP like actions in a many core NUMA environment;
- MPI like communication across processors; and
- web service like invocations across the Internet.

Locally, these may have to be combined if various system types partake in the definition of the data. Accordingly, the OS must be able to principally support all 4 methods, but does not need to load the modules into memory that are not used. This may lead to different OS structures associated with different instances of the code.

We can thereby distinguish between three principle types of modules:

1. **"optional" modules**: can be located anywhere in the infrastructure and are not always needed – these are the most dynamic and flexible types of services;
2. **"essentially local" modules**: have to be co-located with the executing code segments (classically: threads / processes). They provide essential capabilities, such as communication and memory management;
3. **"essentially central" modules**: are the least flexible and most rigid services. They form the main kernel which is implicitly responsible for the overarching functionalities, such as code segmentation and distribution, global scheduling etc. As will be detailed below, "central" does not necessarily imply "unique".

In accordance with the application behaviour in a S(o)OS environment the distributed execution support requires at least one instance to analyse, adjust and distribute code, as well as to organise the distributed execution, state maintenance etc. This central instance therefore also assigns and distributes the OS modules to the individual resources. This is comparable to the main thread in a parallel application, and is related to the fact that the essential nature of the application is sequential, thus starting from a central point.

We refer to this central instance as the "main OS instance". As noted it serves as the entry point for the application and may thus also serve as the interface to the user, though this is not necessary (see detailed description below).

This main OS instance can instantiate "satellite" instances that take over working aspects of the distributed code, ranging from actual execution of code segments, to exposing the respective resources to the application – this may particularly apply to single instance resources in a distributed environment, such as a central hard disk etc. "Remote instances" are thereby primarily designed to be minimal, i.e. to only provide the essential local capabilities in order to reduce the local memory load and thus reduce the amount of memory swaps. At the same time, it should contain enough of the optional capabilities to reduce the communication overhead to a minimum.

In the following, we will elaborate the respective instances in more detail. It should be noted in this context that, since the architecture follows a strong service-oriented approach, the architectural diagrams should not be confused with deployment diagrams. In fact, even the main OS instance can be segmented and deployed in multiple resources, but the additional communication overhead would reduce performance significantly. Therefore the following discussions are essentially to be interpreted along the line of “sensible recommendations”.

MAIN OS INSTANCE

The main OS instance builds the core entry point for an application. It is thereby responsible to identify the potential hosting resources, for analysing the code, adapting and distributing it. The main OS instance is therefore closest to classical operating systems, in the sense that it can principally execute the code in a stand-alone fashion, i.e. without the support of any remote instances.

This instance loads the application on request and therefore must be able to access the storage system and identify the correct source. To execute distribution, it also maintains some form of resource information about the system, though it may initiate a query upon need. Since the individual capabilities may also be located remotely and in order to deploy and communicate with code segments, the main instance must necessarily incorporate communication capabilities.

We will not list the full functionality scope at this place. The main issues to note here is that the main OS instance can be considered the “highest” hierarchical point in the distributed OS infrastructure, i.e. that can principally control all other instances and serves as a main communication point for any centralised actions. It therefore should maintain a general overview over all other instances, in the sense of location, status and availability.

Notably the infrastructure can host multiple main OS instances and share resources with other OS instances. This implies however, that multi-tasking capabilities are provided on the shared resources and that the process identification can be clearly associated with either instance. Even though this case is not explicitly addressed by S(o)OS it is nonetheless covered by its architectural principles.

Similarly, it is not necessarily implied that multiple applications running for the same user and even potentially interacting with each other all have to be hosted in the main instance. Instead, it may be sensible to instantiate one main OS instance per individual application and realise scheduling and other cross-application capabilities in a hierarchical fashion. This corresponds to the potentially hierarchical structure of an application where e.g. threads may spawn further threads at run-time, which thus have to be catered as a subinstance of the S(o)OS.

REMOTE OS INSTANCE

Remote instances execute only work aspects / segments of the overall application, i.e. they can generally be regarded as service hosts for application functions. These functions are effectively extracted code segments turned into services that are exposed to the main application thread as pure function invocations. From the application point of view, this distribution is either completely invisible (if the developer does not explicitly cater for it), or is only indicated through according code annotations, which are however not concretely defined yet. In either case, the OS handles the routing and synchronisation of the messages, unless additional constraints are in place.

The remote OS modules are selected with respect to

1. provide all essential functionalities to the local code segment's execution
2. reduce the communication overhead with other remote OS instances or the main OS instance
3. reduce the number of memory swaps / cache misses.

Therefore, remote OS instances are essentially very small in nature and do not offer the capabilities for standalone full application execution. Remote OS instances can furthermore be regarded as subordinated to the main OS, similar to a master-slave or client-server architecture.

OS COMPONENTS

As it has been noted in the preceding sections, the individual OS instances really consist of a set of modules, which in turn can be compared to services in so far as they expose a common interface that allows remote (or local) interaction among these modules. Following the modular / service-oriented approach, the OS service instances or modules are agnostic to the communication mode necessary for interaction with one another. The overall functionality / capability (in this case of the operating system) is composed as an aggregation of services, or more correctly of individual functionalities that are exposed as services, i.e. with a standard interface and on top of a generic communication infrastructure. It is hence irrelevant for the individual segments whether the respective services are local (i.e. do not require messaging) or remote (which potentially even requires routing support). However, the dynamic and adaptive nature of the scheduling made in S(o)OS will properly account for the different latencies that may be experienced when deploying the various modules in different ways over the physical topology.

In the web service / Grid case, the according communication management is handled by the middleware, though in the holistic case such as pursued by S(o)OS, the according support is not underlying the execution, but actually acts "on the same level". This however enables the messaging capabilities to distinguish better between different communication strategies, thus removing the overhead for local messaging, where invocation would be sufficient (see e.g. the conflict between OpenMP and MPI).

What is more, due to the interaction mode, the individual functionalities do not need to be implemented in exactly the same fashion, i.e. the modules can be specifically developed for the respective resource type. This allows for quick adaptation to new processor models and hardware types without affecting the rest of the OS.

In summary and in accordance with the grid and web service based concepts, these services or modules can be regarded as separate function or code entities that can be hosted anywhere and used transparently from another location. It must be noted that within the context of this document, we use "module", "service" and "component" synonymously unless otherwise stated.

2.B) S(O)OS LAYOUT CONCEPTS

As discussed above, the modular architecture implies that S(o)OS can be deployed across a distributed infrastructure without a specific deployment architecture. In *principle*, the OS can be fully distributed, i.e. with every module or segment being hosted on a different resource. However, it is obvious that this will fail for lack of communication support on the one hand and due to missing state support on the other: in order to coordinate the execution of the OS as a whole, some functionalities need to be made available at specific times across all instances. For example, communication needs to be generally available, whilst scheduling related functionalities are only necessary during execution of the respective resource.

GENERIC DEPLOYMENT ARCHITECTURE OF A DISTRIBUTED OS

Before investigating the specific deployment options for S(o)OS in more detail, it is necessary to examine the general distribution options in environments such as pursued by the project. In particular, we need to distinguish between centralised and distributed deployments:

In the **centralised case**, all core controlling and management instances are located at a central point that can be reached by all service and code instances in the environment (see Figure 13) – this has the advantage that the information source is always known and that reallocation of services / resources can be more easily propagated. At the same time, the central instance however becomes a bottleneck (as discussed in the context of monolithic kernels) and a single point of failure. Centralised systems generally do not scale well.

As opposed to that, in the **fully distributed** case, all instances have to host at least the base capabilities related to controlling and management (see Figure 11), so that they do not have to query a central instance for instruction and control information every time. Accordingly, the system becomes more failure resilient and in particular much more scalable. However, depending on the use case, the degree of distribution can quickly lead to a management related communication overhead if all nodes have to share the same information which may arise from different endpoints – in these cases the distributed case leads to a all-to-all communication across the whole network. In

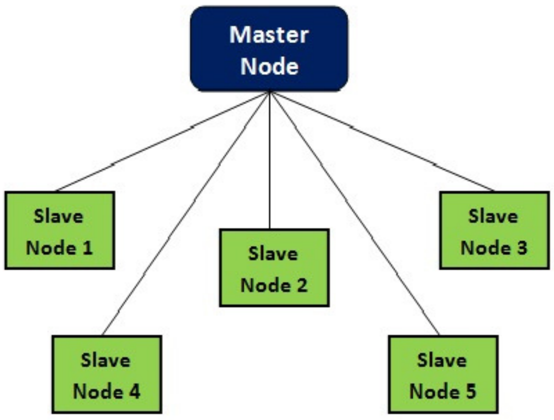


Figure 10: Centralised deployment

addition, it is more difficult to keep track of potential reallocations, as the location information either has to be distributed to all potential communication points, or some mechanisms for routing and updating has to be introduced.

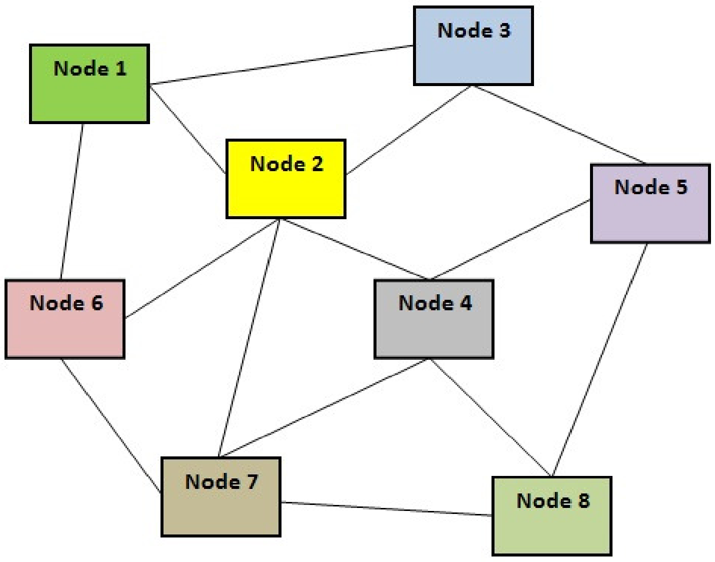


Figure 11: Fully distributed architecture

The respective models can be extended by a **hierarchical** substructure through which the relevant information is propagated in a hierarchical fashion, i.e. through locally connected layers that each in turn are connected to lower layers. This way, the information does not need to be propagated throughout the whole network and the communication overhead is restricted to the substructure. At the same time however, propagating the layers adds delay due to the messaging route, so that lower level will get access to the information later than higher levels (cf Figure 12 and Figure 13).

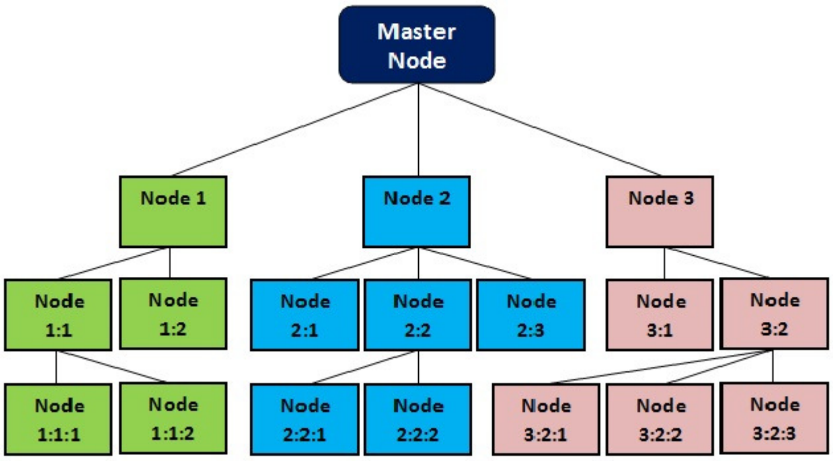


Figure 12: Architecture of a hierarchical centralised deployment

Hierarchical layouts are particularly useful when the actual dependencies between the hosted instances are non-uniform, i.e. when some nodes require only parts of the information and with different timing constraints. By exploiting this information and layouting the network accordingly, the communication overhead can be reduced to a minimum, whilst maintaining a maximum of adaptability and dynamicity.

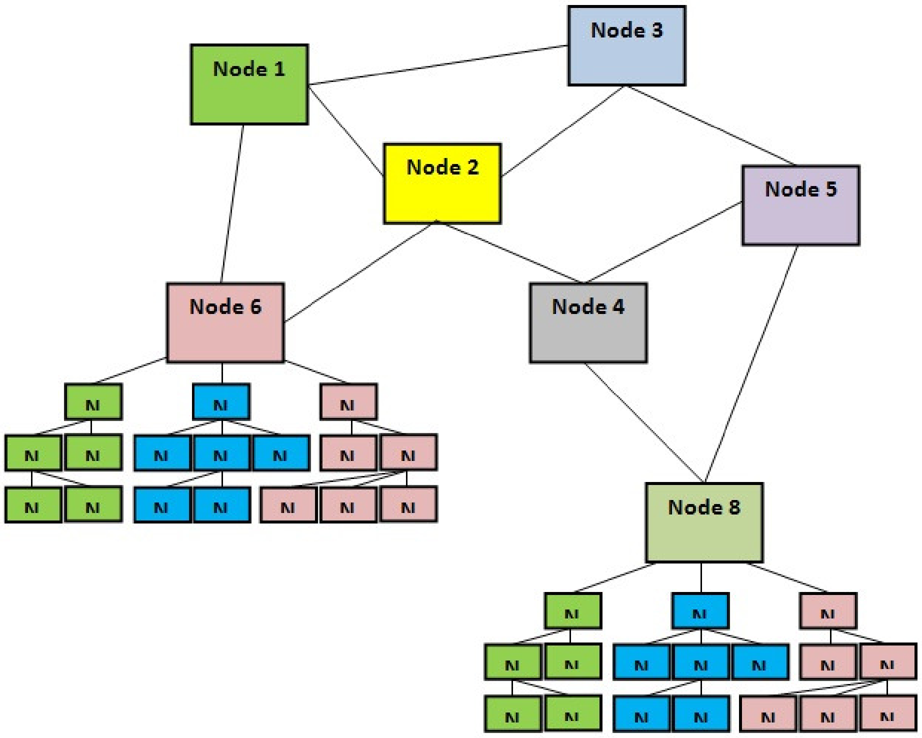


Figure 13: Distributed hierarchical communication architecture

In S(o)OS we can note that next to control and management data, the dependencies between the executing code segments play the major communication role, according to their data and state relationships. Just like in the HPC case, the according communication overhead can lead to severe performance reduction which should be avoided in either case. We can note thereby that the OS relationships are implied by the code dependencies, as – in the strictest case – the OS modules will not communicate with each other if the application does not (directly or indirectly) request it.

It thus becomes obvious that the best (code and OS) module distribution relates to a hybrid hierarchical model where the communication links reflect the dependencies between the individual code segments and their timing constraints. In other words, the deployment should strive to reduce the communication overhead whilst satisfying the timing constraints. For example, a local scheduler instance per tile / processor may be responsible to scheduling all code segments within that domain, but will itself be subject to a higher-level scheduler which triggers the respective code block as a whole.

Another typical example relates to resource discovery, where the individual resource description sources can encapsulate multiple levels of aggregated resources, such as a full processor, instead of the individual cores or a full rack, instead of the full node details. This higher level information can either integrate all lower-level resource descriptions, or abstract from it by providing a more holistic view on the information. Either choice affects the communication strategy as more information implies a higher amount of data (and according maintenance), whilst more abstract data implies that additional queries may have to be executed in order to acquire details that are potentially required.

Further distribution and deployment details of the modules according to the code behaviour and the infrastructure requirements will be subject to further in-depth research in the second phase of the project.

INTERACTION BETWEEN INSTANCES

As noted before and as elaborated in more detail in section V. 1. , the challenge for the distributed operating system consists in maintaining the state information and communication links between all instances of both code and operating system. The according overhead should thereby be reduced to a minimum so as to improve the overall execution performance – this is closely related to the classical problems in distributed execution efficiency. It has already been discussed above that normally some form of communication middleware takes over responsibility for injecting and maintaining the interconnects. Depending on the type of system, these interconnects are comparatively dynamic and well-adjusted to the infrastructure – however, due to the huge range of different interconnects and usage types, the respective middlewares are not interchangeable and start causing problems in hybrid infrastructures, or at least require the developer to invest more effort into coding the according application, thus at the same time making it more portable. One specific example that recently arose in high performance computing related to that consists in the mixed multi-core - multi-processor programming, which requires the developer to use both OpenMP and MPI [48].

By converging these different communication forms on the operating system level, the developer is relieved of this effort and allows the environment to decide which type of interaction to use on basis of the deployment and distribution in the infrastructure. As S(o)OS has to be aware of the code deployment (and hence hosting resources), it can derive the available communication paths from the resource description (see section V. 3.) and select (and enact) the according protocol. We must however, distinguish two types of communications that arise from segmenting and distributing the code: (1) explicit communication and (2) implicit communication.

Explicit communication is fixed in the source code by the developer. It must thereby be respected that if the developer uses a specific communication protocol at this level, it is difficult for the operating system to adapt this to the specific infrastructure, so that this form will not exploit the advantages of the OS communication support. Under certain circumstances, however, the communication invocations may be intercepted and replaced with an appropriate protocol.

More interesting however is implicit communication, which derives from the fact that the code may be segmented at location unintended by the developer so as to exploit the infrastructure best. Implicit communication is not encoded in the application and has to be identified and provided by the OS instead (cf. V. 1.) - this implies awareness of distributed state and dependencies across code segments. S(o)OS must identify these points and intercept / redirect state access, invocations etc. according to the distribution across the infrastructure and the implicit communication modes, without the code execution being affected – in other words, communication becomes invisible to the application.

What is more, the OS itself has to incorporate such communication strategies into its very own execution, as the individual OS modules may be effectively arbitrarily distributed across the infrastructure. Cross-module communication must therefore be enables in an efficient manner employing different communication modes whilst being invisible to the modules themselves – i.e. they should not be affected no matter whether the respective module is local or remote. This implies that the right communication modules need to be accessible without enacting additional protocols – in other words in local cache or at least memory. The choice of modules (i.e. protocol support) thereby depends on infrastructure layout and dependencies across resources.

Example: Virtual Memory

For the purpose of better understanding, we will describe the behaviour and usage of messaging on the OS level in the context of a typical program behaviour, namely accessing memory / data, without going into an in-depth discussion of memory management. What is important to notice in this case is that all typical operating systems expose a virtual memory address space for the executing application, but handles the actual source of the data internally. Frequently accessed data is thus maintained in cache, whereas rare, secondary data may be shifted to the hard-drive. Implicitly the "virtual" memory space of an application can exceed the available RAM, thus also ensuring that no memory conflicts across multiple applications / services / system functions occurs. Without the recent middleware support, this means that the essential OS functionalities are organised somewhat similar to the box with the black frame in Figure 14.

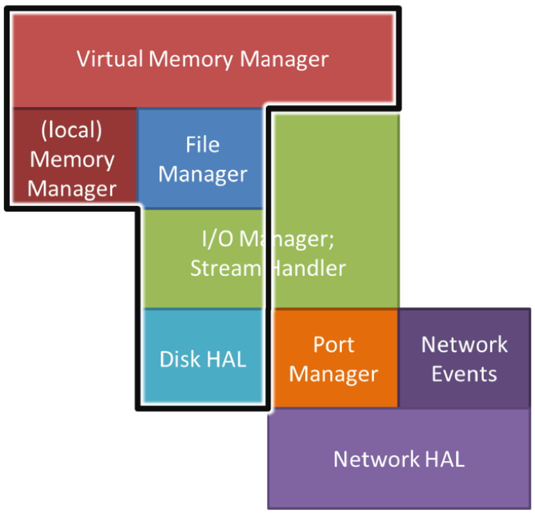


Figure 14: Architectural sketch for virtual memory management

Within S(o)OS this diagram gets extended by typical middleware support aspects, i.e. access to network exposed storage – directly or indirectly. This means in particular that additional network handling functionalities need to be added (Figure 14, full diagram).

Without going into details of the individual modules' functionalities, the main point of the S(o)OS messaging support is to redirect an access attempt to the appropriate resource – notwithstanding whether that's cache, RAM, hard-drive or a remote storage. For example, the local virtual memory management system might identify that the actual source is located in a remote processor, so it will convert the access attempt into a message based invocation to that remote instance. The communication manager (here represented by the Network HAL and Port Manager) will enact the appropriate protocol for the request.

At the reception side, the request triggers an interrupt that causes the local memory manager to be invoked and identify the actual source of the respective virtual address space (cf. Figure 15). The manager can deal with this access attempt like with a local request and take according consequences to adjust the memory's relevance, thus lifting it e.g. from hard-drive to RAM. What is more important in this context is that the requester does not need to know these details about the actual source, but just its location in the infrastructure. In the context of service-oriented architectures, we refer to this as “double-blind invocations” to denote that the location must be resolved twice in the communication process [49].

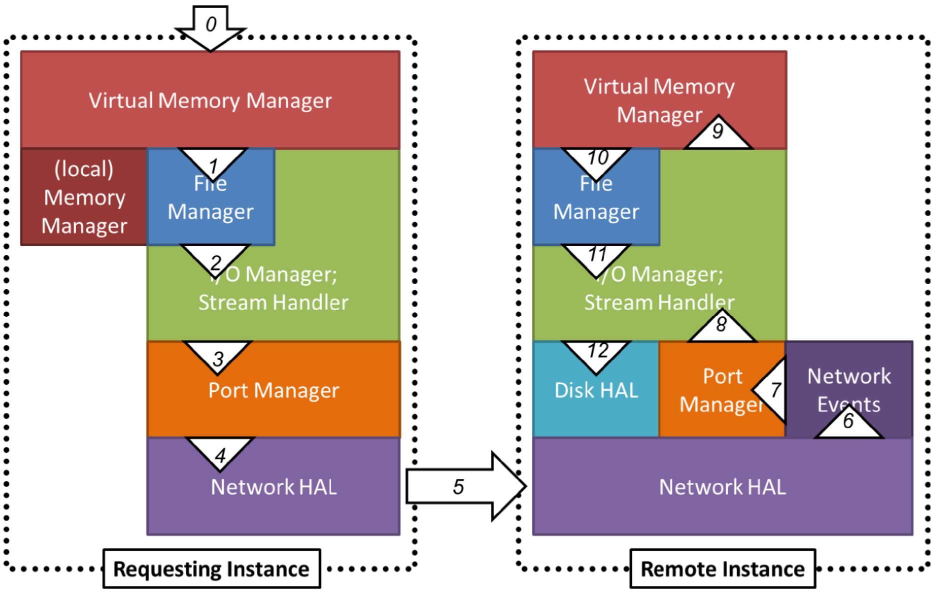


Figure 15: Routing a request over instances and modules - the numbers denote the sequence of steps

Notably, the destination identified by the requester does not necessarily have to be the actual data source, but instead might cause the remote memory manager to initiate the same network based access process. Accordingly, the request may be routed further according to the potential dynamicity of the distribution and deployment. This process extends the storage hierarchy along the line indicated with the new Von Neumann architecture model in section II. 1.a) but at the same time complicates estimation of timing behaviour, i.e. effective latency and bandwidth. Memory management must therefore respect the potential delays for access according to the dependency tables gained by behaviour analysis. In principle, similar methods for runtime data rearrangement can be employed, as for local memory management, i.e. moving frequently accessed memory closer to the requester etc. However, here also the same concerns for runtime rearrangement apply as for code (cf. section II. 5.).

V. Critical OS Capabilities

In this section we focus on some critical components of the OS design and architecture, discussing what options and alternatives are possible.

The discussion is particularly focused on:

- code segmentation and analysis;
- speculative parallelisation;
- resource description, discovery and matching;
- scheduling.

1. Code Analysis and Segmentation

The code analysis and segmentation related modules primarily fulfil two main tasks: (1) monitoring the code behaviour and (2) adapting the existing code to the destination resource specifics. The principle idea behind this set of components consists in the fact that parallelisation and more importantly distribution of code depend primarily on the communication (data and message exchange) density between the respective candidate threads (or segments). The code analysis examines the behaviour of the application in terms of these dependencies and relationships and applies extended (graph) segmentation principles on the so-gained information to identify potentially distributable parts of the code. This may imply transformation strategies in order to improve the dependency layout, or allow for more structured data exchange. The patterns gained this way can also be exploited to identify hardware requirements, restrictions and preferences, such as vectorisable code and size of loop unrollment.

The point of this set is not to automatize parallelisation, but to improve code concurrency for even higher degrees of parallel execution (better resource utilisation), and to support the developer in easily creating more efficient parallel code, by exploiting specific patterns in the dependency graph and feeding them back, exploiting them automatically and exposing them in form of annotations.

INTERNAL ARCHITECTURE PRINCIPLES

Given the typical architecture of modern processors, the supervising and analysis related functionalities can be achieved in multiple ways, since a multitude of information is required which can be gathered from different sources. In particular we can distinguish between (a) intercepting all relevant transactions, (b) building a managed or virtual environment, where all according operations are executed "safely" and finally (c) completely statical ("offline") analysis basing only on the available code and not its execution behaviour. Obviously, these approaches allow for different implementations, as they have different implications on the system requirements.

Figure 16 depicts the general principle of the code analysis & segmentation cycle: the four main conceptual tasks relate to

1. Analysing the code with respect to dependencies, relationships etc.
2. Generating a dependency graph according to the information gathered during analysis
3. Identifying potential segmentation points according to min-cut graph theory (i.e. to minimise the dependencies across the cuts)
4. Segmenting the code and injecting communication points to ensure distributed execution capabilities

During the analysis related tasks (1 and 2), the major point for the environment consists in gathering and providing relationship information with minimal impact on execution, yet with enough detail to improve performance. The segmentation part is denoted by identifying the best match towards the infrastructure in terms of hardware suited for the code requirements and layouts suitable for the communication needs.

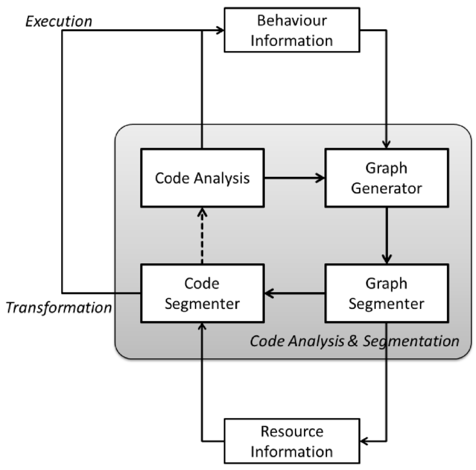


Figure 16: general architecture principles of code analysis & segmentation

The following sections discuss how the individual tasks can be realised technologically.

1.A) CODE BEHAVIOUR ANALYSIS & MONITORING

Analysing the code behaviour can be executed either offline, i.e. prior to code execution either by the compiler itself or a dedicated analysis engine, or online, i.e. whilst the code is actually running on a physical machine. The main difference between the two techniques relate to the type of information that can be gathered: whilst offline analysis primarily provides the information about "static" code relations - such as invocations, global and local data scope etc. - online analysis allows for gathering the "dynamic" code behaviour, i.e. even the data-dependent workflow and the dataflow in general. Notably, the relationship information gained from online analysis is subject to changes basing on the input data used, and therefore can potentially lead to different segmentations. One major task is therefore to extract the points of stability from this information.

Complete **offline analysis** implies that the analysis is executed independently of code enactment and therefore of any information or support from the operating system. The information (dependency graph etc.) could therefore be provided by a third party application, or even by a human user (see also programming models). Implicitly, there are no specific requirements from offline usage that are not covered in online usage.

In the case of **online analysis**, the monitoring / analysis related modules need be able to access and gather information about the current execution behaviour with minimal (ideally none) impact on code execution performance.

In order to generate the code information graph in the desired fashion, the following data should be gathered:

- dataflow dependencies:
    - memory accesses (for data flow dependencies);
    - I/O accesses;
    - register accesses;
- workflow dependencies:
    - instruction pointer access / changes.

Notably, in particular gathering low-level information, such as register access, impacts heavily on code execution and principally requires an intermediary visualisation layer to intercept the according operations in the first instance. In order to acquire this information, three possible options exist:

1) Virtual Execution Environment / Interpreter

The most straight-forward approach consists in executing the respective code in a completely virtual environment, where each register and memory access etc. can be interpreted / analysed prior to its execution. Due to the nature of this approach, performance is comparatively bad - what is more, distributed execution of parts of the code becomes more difficult in a completely virtualised environment, reducing performance even further.

Since we do not want to map between different environments in this context, but instead can assume that the code was compiled (or already adapted) for the platform it is being executed on, the virtualisation does not have to actually cater for execution of the operations, but just for their interpretation. This allows serious speedup of the execution.

The main task of the according interpreter would therefore consist in assessing the respective operation(s) for potential relationships / dependencies with other parts of the code. In all other cases, the operation can be executed directly on the chip. This requires however, that the context switches back and forth between OS (ISA interpreter) and the running code, thereby still impacting considerably on execution. It should / could be investigated whether execution on a second core could reduce the context switching overhead.

Components:

- **File and / or Memory Access:** to load the code operation wise (see external dependencies)
- **ISA interpreter/emulator ("code analyser")**: checks the current operation for any impact on dependencies (i.e. memory access etc.); simulates their execution
- **Dependency manager**: maintain all the dependencies information, such as last accesses etc. to relate them.
- **Graph Generator**: produces the dependency graph on basis of the interpreted commands and the context / state information

2) Behaviour Interception

The second approach consists in "just" intercepting access to memory, I/O, and registers. Whilst this obviously reduces the amount of information gathered, it still provides all necessary information for graph generation and increases the execution performance considerably. Further to this, most of the according modules / functionalities are already part of modern OS and HW architectures in order to allow separate execution environments (multi-tasking) and I/O control. However, intercepting register access is currently hardly possible without seriously readapting the code (see Option 3) or virtualising the environment (see Option 1).

As opposed to classical virtual memory and I/O managers, the extended version needed here would have to create access annotations to derive dependency information.

Components:

- **Extended Virtual Memory Manager (EVMM)**: monitors not only how often which memory part is accessed, but also by whom, how and when
- **Extended I/O Monitor (EIOM)**: like the EVMM monitors when, how and by whom external resources where accessed
- **Virtual Registers (VR)**: intercepts and analyses register accesses.
- **Graph Generator**: see Option 1

Effectively the components in this case are extensions to memory management and general messaging, with the addition of virtual register management.

3) Code Rewriting

As a final resort, the code can be rewritten at compilation time with the specific goal to identify all the potential points of dependencies - more specifically, by replacing all memory, register and I/O accesses with dedicated system calls to modules that annotate the respective memory space, register etc. The annotations can then serve the dependency analysis.

Whilst this model is effectively very similar to interpreting the individual operations, it increases execution performance considerably. It furthermore allows for a certain degree of distribution, as long as the according annotation modules are either locally available or via a communication route (in which case the respective module becomes a bottleneck).

Components:

- **Code Transformer (part of Code Segmenter and Code Adapter)**: converts dependency specific operations into OS module calls to EVMM, EIOM and VR
- **Extended Virtual Memory Manager (EVMM)**: similar to option 2, but on system call
- **Extended I/O Monitor (EIOM)**: similar to option 2, but on system call
- **Virtual Registers (VR)**: similar to option 2, but on system call
- **Graph Generator**: see option 1

1.B) Code Transformation and Adaptation

As a second part of exploiting the code analysis and segmentation, the metadata gathered by the code analysis step needs to be analysed with respect to their potential distribution in the infrastructure, and the best destination resources for execution. Generally, it cannot be assumed that the appropriate ("best") resource distribution exists and is available. In order to make best use of the existing resources, this task is therefore ideally constituted by three major steps:

1. identification of the resource layout and the generic capabilities of the resources in the infrastructure
2. segmentation of the code according to the layout and the potential adaptations - this implies selection of the resources to be used
3. adaptation of the code segments according to the destination platform

Thereby the degree of resource information changes drastically between these individual steps: at step 1, the resource information is just high-level to potentially steer a recompilation, or at least to enable selection of the right branch in a "multi-compiled" code, i.e. where the source has been compiled to multiple high-level destinations. Step 3 however requires the full details of the selected resources in order to execute the final optimisation and adaptation tasks.

The tasks involved in transformation and adaptation form primarily *functions* rather than *modules* - however, as the functions may have to be provided independently from one another, they may turn into *services* that are effectively the same as OS modules.

The main functions are the following:

- **Code (or Graph) Segmentation**: finds the best graph segmentation according to dependencies, concurrency and overlap. Depending on the mechanisms selected for communication and state maintenance, as well as potentially for online, distributed code monitoring, the segmentation will also have to insert communication points to maintain distributed information (see also code transformation)
- **Code Transformation ("Code Adapter")**: takes care of adapting the code to the destination resource. As discussed, this may take different forms, ranging from recompilation to pattern replacement. This module / function could also incorporate capabilities for injecting explicit communication operations and replacing access commands (see above).

- **Resource Requirement Identification (part of Code Segmenter and Performance Predictor)**: through pattern analysis in the dependency graph and on code level, specific resource requirements are identified, such as vector support, cache size, communication latency, precision etc. This information is fed to resource discovery.

1.C) External Requirements and Dependencies

Next to the internal functionalities, the process depends on information and capabilities provided by "external" modules, i.e. modules that fulfil other purposes than strictly code analysis and segmentation related. Notably, some of the functionalities / modules presented in the preceding section are actually adaptations, respectively extensions of existing OS functionalities, such as virtual memory management or I/O monitoring.

Further to the external requirements, the code also has impact on other modules, respectively in order to fully exploit the functionalities that code analysis & segmentation aims at (namely the exploitation of implicit concurrency of the running code), other modules need to provide certain capabilities, that come implicit from the segmentation. For example, state maintenance across segments can exploit the dependency information gathered during behaviour analysis.

REQUIREMENTS TOWARDS OTHER MODULES (DEPENDENCIES)

As noted, the code analysis and segmentation capabilities require a set of information and functionalities from other modules in the operating system. The following lists these functionalities without proclaiming specific modules to handle them, as this depends on discussion across the project:

- **Resource Information**: the actual code segmentation and adaptation depends heavily on the capability to gather information about the available resources, their location and their specific usage details, such as instruction set, cache size etc. The information may be gathered in a multi-step procedure, where different queries are posed depending on the current task in the segmentation procedure.
  Data needed: available resources; resource capabilities & specifics (such as cache size etc.); infrastructure layout (including latency, bandwidth etc.)
  Related components: Resource Manager
- **Monitoring**: in order to execute code behaviour analysis, the system status of multiple parameters related to memory, I/O, registers, instruction pointer are required - the source for this information depends on the setup option chosen (see above for details)
  Data needed: memory access, I/O access, register access, IP movement (all with timing and / or frequency, and type of access (read, write, size)); expected performance given a specific algorithm
  Related components: Virtual Memory Manager, I/O Manager, Monitoring system, State Manager, Dependency Manager
- **Execution Control**: in order to execute code in a managed, respectively controlled fashion, the system needs to be able to control scheduling, process management etc.
  Related components: Scheduler, Process Manager, Code Adapter

Impact On Other Modules

The information gathered from code analysis and segmentation can only be effectively put to use, when other components exploit the according information correctly. As opposed to the requirements section above, this section therefore does not describe which information are required from other modules, but which information is provided to other modules and how they should use it.

- **Migration**: obviously, in order to actually execute the segmentation, i.e. to distribute the code across the environment, the OS must be able to migrate code segments to a different location. Depending on the time of execution, the code may be currently running, about to be run, or paused. Generally, we must assume that migration is only possible during locally paused execution. This does not mean that the whole application must be halted - in fact, currently unused segments can be migrated at any time.
  As part of migration, all dependencies needs to be maintained - this does not only mean maintenance of the communication (see below), but also maintenance of state (see also below) and availability of the relevant OS modules. This latter part could imply either moving them with the code, having them available on the destination resource, or inserting according communication links. The relevant information (which module, when and how) is encoded in the segmentation graph generated during code behaviour analysis.
  Data provided: dependencies (including rough timing and type of content
  Modules affected: Code Migration
- **Distributed execution**: the main purpose of code analysis and segmentation consists in enabling concurrent, distributed execution of various code segments forming the application. By segmenting an application at "unintended" locations, i.e. where the developer did not cater for the actual distribution, context and state information becomes corrupted. Distributed execution needs to provide means that enable a "virtual" execution environment where state, communication etc. is all maintained. This means mostly that the OS modules need to provide means to route communications and memory / register accesses or provide the according information in a different way (see communication)
  Modules affected: execution management
- **Communication**: communication, as intended by the programmer, i.e. such as writing or reading to a device, loading from the internet etc. but also explicit messaging to other threads or processes, must be explicitly handled by the operating system so as to route it to the correct endpoint. The endpoint may furthermore be potentially dynamic, i.e. vary over time. What is more, the connection TO the endpoint may imply different communication protocols (ranging from intercore to TCP/IP), which needs to be catered for by the OS. In other words, the communication means must be transparent to the application. The dependency graph provides the general communication information (i.e. which code segments communicate in the first instance), thus helping the maintenance of routes etc. Generally, however, the OS needs to provide a fall-back mechanism in case the graph does not depict the according dependency (e.g. because it was never used before).
  Data provided: dependencies (communication links between segments)
  Modules affected: general messaging
- **Distributed State Maintenance**: any application / process maintains its own environmental state, i.e. content of register, memory, stack, heap etc. Through segmentation and distribution, this context information gets disrupted and therefore needs to be maintained at different locations at the same time. Obviously, different mechanisms are possible to execute state maintenance, such as: push data on availability, pull data on necessity, block push / pull, push / pull ahead of time.
  The according communication mechanisms need to be artificially introduced into the code in order to ensure that state is maintained properly. Alternatively, any state related request needs to be intercepted by the OS (see code behaviour analysis above for details). Again, the dependency graph gained by code behaviour analysis provides information about the state dependencies and hence helps in adjusting the communication points.
  Data provided: dependencies (communication links between segments)
  Modules affected: State manager, dependency manager

1.D) ARCHITECTURE SKETCH

The overall integration of Code Analysis and Segmentation into the operating system takes a slightly higher-level stance than most of the other components. As described in more detail in chapter VI. , concurrency increasing related mechanisms are not essential for the working of an operating system, though they are relevant for the specific goals of S(o)OS (cf above). Accordingly, even though the respective components tightly integrate with the other OS components, this does not imply that the latter require the former for functioning, but rather the other way round.

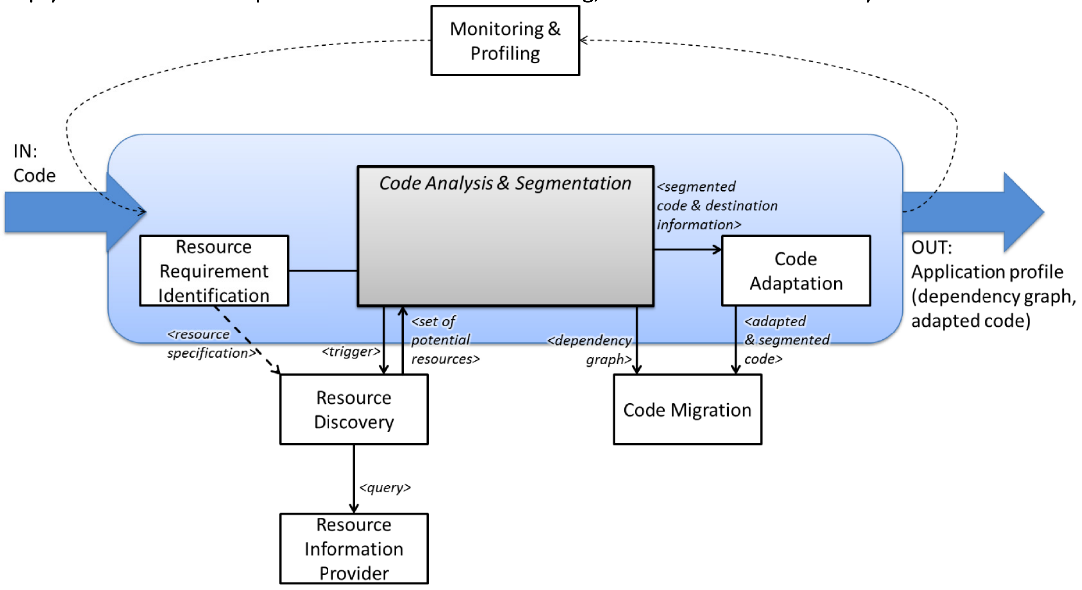

Monitoring & Profiling
IN: Code
Code Analysis & Segmentation
<segmented code & destination information>
Resource Requirement Identification
Code Adaptation
OUT: Application profile (dependency graph, adapted code)
<resource specification>
<trigger>
<set of potential resources>
<dependency graph>
<adapted & segmented code>
Resource Discovery
Code Migration
<query>
Resource Information Provider

Figure 17: Conceptual behaviour of the code analysis & segmentation related components (and their dependencies)

Figure 17 depicts the general behaviour of code analysis and segmentation in the context of the related components and functionalities. The figure thereby focuses on the offline analysis behaviour, i.e. without considering the additional issues for execution – the dotted arrows denote the additional feedback loop for online monitoring. As can be seen, the system essentially takes a (compiled or uncompiled) code and generates a set of additional information for steering execution. It can also generate adapted code on basis of this information, if so desired.

2. BINARY CODE ADAPTATION

DISTributed Adaptable Executable (DISTAE) is the software layer that will allow the OS the portability of programs among different heterogeneous computing units of the system and run the different parts of the code simultaneously in a distributed and/or heterogeneous environment.

2.A) MOTIVATION

Previously a program execution depended mainly on the compatibility with the OS which could essentially run on any hardware infrastructure. But at the current moment, the trend in hardware infrastructure development is to increase the divergence. The new semiconductor fabrication technologies do not allow to continue increasing the performance of a single processing unit as fast as it has been possible in the last decades. Manufacturers have only been able to increase the processor computational power by adding new specialised units, creating new ISAs (Instruction Set Architecture) or new extensions to the current ISAs.

But at the same time the improvements in miniaturisation are leading to the new era of the multi-core and may-core processors; implementing many heterogeneous cores in the same silicon die [24].

This causes that the operating systems cannot run on any platform any more, making the execution of programs even more complex. And in the case that they can execute, then it is very difficult to obtain a substantial performance gain of each different hardware architecture, which makes necessary to adapt the code to the available resources in each moment.

The overall complexity is increased even more in distributed systems as the heterogeneity [25] has a tendency to increase and it is added to the normal complexity of this kind of systems. And when it is a dynamic system, this means that the configuration and availability of the computing resources change during the execution, the programs have to be modified and adapted during execution time to be able to execute and to make a better use of the platform in which it is running to achieve a reasonable performance.

In this chapter we are going to analyse the means that are necessary in the OS architecture for DISTAE to be implemented and work correctly in a distributed and heterogeneous system. We will describe the minimum OS functionality needed and optional functionality, that if not strictly necessary, simplifies the implementation and/or improve the execution performance.

2.B) SET OF FUNCTIONALITY THE OS MUST PROVIDE

For the DISTAE layer to work, the OS will have to provide these modules and services:

CODE ANALYSER & SEGMENTATION

This module has the task of monitoring the behaviour of the code of a program and decide how to segment it.

CODE DISTRIBUTOR & SCHEDULER

This module decides how to map the different code segments of the program extracted by the analyser to the resources available in the system.

CODE COMPILATION AND ADAPTATION

The compiler can produce different products depending on the desired result. The output can be:

- **Generic source code:** The original source code is cut in segments and wrapped into extra code that will deal with the communications, data types conversions and synchronisation. Each segment is packed and then distributed to the computing units where it is finally compiled into the local binary code.
- **Intermediary code:** The source code is cut in segments, wrapped intro extra code that will deal with the communications, data types conversions and synchronisation. Then each segment is compiled into intermediary code. This code is called RTL (Register Transfer Language), it has already been parsed and has a first level of generic optimisations. Later it is packed and sent to the mapped processing unit, where the compiler will finish the compilation from intermediary code to binary local code.
- **Binary code:** The original source code is cut in segments and wrapped into extra code that will deal with the communications, data types conversions and synchronisation. Then it is compiled into binary called in these different ways:
    - Uni-path package: The package is only compiled and optimised for one architecture, this means that portability is reduced, but latency is improved as there is no need to make a final adaptation on the destination processing unit.
    - Multi-paths package: The package contains different binary files, for different architectures. The size of the package is increased, which affects negatively the latency, but at the same time increasing the portability.

- Binary package translation: This is a technique for making a binary uni- or multi-path package compatible with an architecture that has no binary path in the package. This is done by translating the binary code. The most common way is create a virtual machine, translate the binary instruction intro high level language code, and then compile it into a binary code. This is for example the technique used by the TCG (Tiny Code Generator) of Qemu. The disadvantages are that the performance and latency are worse, in exchange of being able to execute in a processing unit for which it was not prepared.

INTER SEGMENT COMMUNICATION

Different architectures have a different way of representing the data internally. That is the reason when code from different architectures is interacting all data types should be translated before and after getting into the communication channel. The data representation usually affects this items:

- Endianness:
    - Word Endianness
    - Bit Numbering
- Memory Address Space
- Memory Alignment

The OS should also handle and define the communication channel. It could be that two segments are running on different cores of the same CPU (in this case the can communicate using a shared memory model or message passing, without needing to translate the data types) or in two different processing units connected over the Internet with very different architectures (then the OS will need to handle the communication and the translation of the data types). All this should be done transparently to the application.

2.C) OPTIONAL EXTENSION

PROFILING

Software profiling is an optimisation technique. It is a form of dynamic analysis that measure memory use, function calls (frequency and time), usage of particular instructions... Once the profile is realised, the information is analysed to find how to optimize the application. This information is very useful as the behaviour of the application can be difficult to analyse statically as there is no indication of the input parameters.

ADAPTED/COMPILED CODE STORAGE

The adaptation and compilation of code might take a considerable time. This specially affects the execution of small running time segments, as the compilation time can be higher, drastically reducing the performance.

That is the reason it makes sense to have some local storage with already adapted and compiled code, for segments that are going to be used many times. This has no advantages for first time execution, but in subsequent executions, the latency can be greatly reduced.

COMPILATION CACHE

Some times the distributed segment might need to be recompiled because of many reasons like introduce optimisations derived from profiling analysis or to update the source code of the application. Often this changes are small and most of the already compiled code, stored in object files, is valid. The use of tools that caches de compilation of object code so that it can be reused on the next time, and avoiding to recompile the whole code can greatly speed up recompiling time. This tool can be implemented by hashing different kinds of information that should be unique for the compilation and then using the hash sum to identify the cached output[27].

DISTRIBUTED COMPILING

There are some cases in which a processing unit can not adapt and compile a segment package or it is very slow in doing it. To solve this situations distributed compilation can be used in which another processing unit that is better suited for code adaptation and compilation generates the binary code for the target architecture. This implies to have cross-compilers installed [26].

2.D) CODE DEVELOPMENT

There are different programming models that can be used in the implementation depending on what the OS provides and what the programmer wants to achieve, but that in principle interact with the code analyser and segmentation:

- **Annotations**: The programmer can write annotations for expanding code, giving indications about the behaviour of the program that can not be guess by static analysis. This annotations should suggest which chunks of the code could be segmented for distribution and which others should be kept as a single unsegmented unit.
- **Automatic**: If there is a lack of annotations then the code analyser with the profiling information if available and according to the characteristics of the system and the code/application will decide how to segment the code.

2.E) INTERACTION WITH OTHER OS MODULES

SAM has effect on other modules:

- **Resource Discovery**: The resource discovery module finds the different processing units and its characteristics which will affect the adaptation of the code based on it unique peculiarities.
- **Code analyser & Segmentation:** How the code analyser decides to segment the code influences the adaptation. Depending on the characteristics of the segments it might need to be better to compile it locally or using distributed cross-compiling. Also different compiler binary optimisations (flags) might have different results depending on the code segment.
- **Scheduler:** The scheduler algorithm influences the adaptation in a few ways. Depending on if the segment is going to be run multiple times on that processing unit, it can keep the already adapted code for further reuse. If the other segments with what the segment is communicating are in different cores of the same processor, then the adaptation of the communicating later will be different as if they have to communicate using another kind of network.

3. RESOURCE DESCRIPTION

A resource description component stores information of (part of) the hardware system state. It potentially holds information about the entire system, part of the system, or only the hardware that is used by the OS instance the resource description component is running on. It can contain both static and dynamic information about the hardware state. It is comparable to the System Knowledge Base found in the Barrelfish OS [29].

3.A) MOTIVATION

In order to decouple low-level heterogeneous resources from higher levels and to aid the development of programming models that hide the complexity of future of architectures from developers, the execution environment of applications has to be made resource aware.

The resource description component can assist other OS components to meet the requirements set out for S(o)OS. The resource information captured by this component can be used to improve efficiency, as code can be tailored as much as possible to the specific characteristics of the hardware. Including dynamic information in the hardware model will also improve potential scalability, as saturated shared resources can be identified, which can be subsequently avoided when executing other threads and/or code segments. The resource description component also aids in the support for heterogeneous platform, by making node-specific information far more explicit, and making commonalities between nodes (e.g. both are 4-wide SIMD) apparent. This will make certain optimisations more platform-agnostic, and hence increase heterogeneity support.

The Resource Description functionality is needed by the Code Analysis and Segmentation one to provide information about the system state. It also provides resource information to the Resource Discovery component.

3.B) MINIMALLY REQUIRED OS FUNCTIONALITY

In a minimal setup of the resource description component, only the static resource information is considered. In the minimal setup configuration, a system information "database" can be pre-configured and loaded at boot time of the component. A minimal setup would only require the following OS functionality:

OS FUNCTIONALITY: VIRTUAL FILE SYSTEM

A virtual file system is an abstraction on top of the different file systems loaded on the running system. It allows applications to access storage devices in a uniform way, at the same time it allows to simulate storage devices (like RAM-disk) and to modify a file many times and only write it once in the storage device, after confirming the data is not going to be modified any further.

A VFS is needed to get access to the pre-configured resource information at boot-time, and potentially write any modifications to the component configuration.

3.C) OPTIONAL OS FUNCTIONALITY

The previously described set of functionalities is required to implement a minimal setup of the resource description component. We should also consider however additional OS functionalities that are not strictly necessary, but would either simplify the implementation of the resource description component, or allow it to have additional functionality. Do note however that such functionality does not have to be implemented as a separate module per se, but could be implemented as part of the resource description component, requiring only the minimal OS functionality.

EXTENSION 1: DYNAMIC SYSTEM DATABASE

In the minimal setup, a system database was preconfigured and loaded from a storage device. An extension to this scenario is to dynamically built this database at either boot-time or run-time (which could be saved to a storage device). This would allow the resource description component to deal with a dynamic environment in which system resources can appear (e.g. hot-swap), or disappear (e.g. failure). Such an extension would require the following OS functionality:

OS FUNCTIONALITY: RESOURCE IDENTIFICATION

Can acquire hardware-specific information of the local node through the likes of CPUID-like instruction or alike. Can acquire hardware-specific information of remote nodes through PCI information query (`lspci`) or alike.

EXTENSION 2: DYNAMIC INFORMATION

Besides static information of resources, such as cache size, instruction set, architecture type (VLIW, SIMD, etc), system information that is more dynamic in nature could also be stored. Such as cache and/or memory usage, or bandwidth and latency of a link scheduled by a best-effort mechanism. Such an extension would require the following OS functionality:

OS FUNCTIONALITY: SYSTEM MONITOR

Provides dynamic information about the system: cache-usage, stall-frequency, link usage, etc.

EXTENSION 3: INFORMATION SHARING

There are several cases where there might be more than one resource description component running on the system: the component is replicated for performance reasons, the system is dynamic as a result of mobile devices, etc. In these cases it might well be possible that separate instances have different information about the system state. Information could be exchanged to provide a larger or more precise view of the system. Such an extension would require the following OS functionalities:

OS FUNCTIONALITY: MESSAGING

A messaging functionality provides the processes a means to communicate either within an OS instance or across OS instances.

OS FUNCTIONALITY: SERVICE DISCOVERY

A service discovery functionality allows processes or components to find other components that provide a certain functionality (e.g. resource description).

3.D) IMPLICATIONS OF USE-CASE SCENARIOS

Although the application requirements certainly influence the resource information that might be required (and most likely also provided), the environment in which an application is run has equally large effects. The need for extension 2, “Dynamic Information”, is considered mostly application dependent (as opposed to environment dependent). Both the HPC and RT application scenarios will be evaluated according to effects of the most likely execution environment of the application.

SCALABLE HPC APPLICATION

In the HPC scenario, the full details of the system on which an application is run are most likely known. Meaning that a minimal setup, where the system “database” is preconfigured, is a feasible deployment option for this scenario. Depending on the reliability of future hardware, the HPC use-case might require extension 2 “Dynamic System Database”, as nodes might drop out of the system.

SCALABLE DISTRIBUTED RT APPLICATION

In the RT Scenario, the application will most likely be executed on many different system configurations (desktop machines), so extension 2 “Dynamic System Database”, is most likely required for the resource description component to work properly. In case of the application running on a mobile device, it might be worthwhile to offload parts of the computations to a more powerful resource. In this case resource description components running on different machines might exchange information regarding each others configuration, hence requiring extension 3 “information sharing”.

3.E) HIERARCHICAL RESOURCE DESCRIPTION

Rather than repeating all information on resource description and discovery, this section only provides some additional notes with respect to how *hierarchies* could be exploited to improve discovery and segmentation - this could affect timing, as well as quality.

CONCEPTS / PRINCIPLES OF HIERARCHICAL DESCRIPTIONS

Every system is composed of multiple resources, which in turn - in particular nowadays - incorporate multiple functional units, which may be further broken down into execution units etc. For example, a High Performance Cluster System can be broken down as follows:

1. Cluster
2. Racks
3. Nodes

4. Processors
5. Cores / Processing Units
6. Individual ALU elements (multipliers, ...)

Even more accurately, the description would span a tree where each branch holds the parts that form the higher level abstraction, e.g. Node => Network on Board; Processors; RAM; I/O etc.

At each level, different information about the system can be gained which can hence be matched to different requirements. For example, for running a large scale application, the information about the total number of cores available would be gathered on cluster level - on the other hand, the network on chip information etc. may not be of interest. On the other hand, a small scale application with timing constraint will not look at the cluster level etc. This distinction becomes even more relevant, the further down the segmentation advances, i.e. the lower the code abstraction level and the more details can / have to be respected.

Obviously, higher level resource descriptions can expose lower level information upon query, but not the other way round. However, the general purpose consists in reducing the size of the information through abstraction.

DISCOVERING RESOURCES USING HIERARCHICAL DESCRIPTIONS

As noted in the preceding section, the hierarchical description concept allows for mapping more abstract requirements to a destination platform. Hardware abstraction is closely related to the scope of program investigated:

- looking at the program as a whole, the hardware requirements relate mostly to the whole system
  e.g. required libraries, types of resource (gpu, cpu,...)
- Application kernels have typical hardware requirements on an intermediary level
  e.g. shared memory, vector processors etc.
- at the lowest level, I.e. across a few operations, the hardware requirements become very specific, such as
  size of the cache, vectorsize, latency etc.

Along that line, even though the requirements become more specific at lower level, they at the same time become less binding. I.e. the requirements have impact on performance, not on executability in the first instance.

Regarding the actual discovery process, the hierarchy can be exploited to reduce the size of information gathered with each query - this however implies that the results cannot really be stored locally (requester side) and that it is more complicated to store the full information (provider side).

4. RESOURCE DISCOVERY AND MATCHING

In the future, computing nodes with thousands of cores may be connected together to form a single transparent computing unit which hides the complexity and distributed nature of the many core systems from applications. In such many core environments resource discovery is an important powerful building block to exploit highest capabilities of all distributed resources. It aims to adapt the application's demands to the resources potentials by discovering and finding resources with deep understanding of the resource specifications according to application resource requirements.

The resource discovery functionality is required by the “Code Analysis and Segmentation” to find the available different processing units and exploit their characteristics information. It also uses Resource Description Provider to get information about the resources and employs Resource Requirement Identifier to get information about the code specific resource requirements.

4.A) RESOURCE DISCOVERY

Resource discovery encompasses locating, retrieving and promulgating resource information in large networked environments. Resource discovery supports both exact marching and partial query which the generated queries can be ranged from the simple key-based queries to the complex range-based queries. It uses hierarchical chord distributed hash tables to maintain resource information which consist of the detailed descriptions of all the processors in distributed environment. DHTs are essential robust components to provide storage and lookup services in peer to peer networks. Despite the existing master/slave data replication architectures approach DHTs are more reliable and provide performance guarantees. In multi-core environment we need to avoid exchanging unnecessary resource discovery information between nodes. It helps to reduce overload in the network during discovery procedure. To achieve this purpose we described resources in multi layer resource description which implements using hierarchical chord DHTs.

Categorising resource specifications includes computational and communicational properties and behaviors in there layers from more abstract information to more detailed characteristics. In resource discovery we are considering resource description to avoid generating overload for the discovery traffic. By using hierarchical resource description model we just transmit necessary information and resource description according to the status of the discovery procedure. Hierarchical Distributed Hash Tables maintain all the benefits of flat DHTs and support extra valuable features for bandwidth usage of efficient caching. In many core systems we have physical hierarchy of core-cpu-node, therefore HDHTs are the most efficient design for adaptation to the underlying physical structure. In DHTs we store key-data pairs by assigning keys to different processing units (cores) in many core system, In fact the Queue Management System components which are distributed service directories employ H-Chord DHTs to keep resource description for all the cores in a chip. Each QMS maintain the description values for all the keys for which it is responsible and for each core there is only a unique key. Chord specifies how keys are mapped to cores, chips or nodes, and how a node can find a data for a specific key or the range of keys by first locating the nodes which are responsible for those particular keys. In resource discovery, DHT nodes in Distributed Computing Environment communicate with each other by using Anycast messaging mechanism which is a robust scalable messaging mechanism for managing and locating resources. It allows a node to send a query to a group of nodes that are using the same Anycast address. The receivers of the message could be only one or several group members where the receiver nodes satisfy the query requirements for metrics such as latency, availability and load balancing between the target servers. Generally, an Anycast system must provide scalability, location awareness and load awareness for all the nodes group. This means that it should be able to recognize the closest resource to the query sender in the network. The system also should avoid overloading group members in the case of high demand for a particular resource, therefore system must be aware of resource utilisation in Anycast group.

The general architectures for Resource Discovery are Structured (Directory Based), Unstructured (Directory Less) and Hybrid.

STRUCTURED

In a structured architecture we can understand systems on certain nodes of which the resource information is fixed and pre-configured. Structured architecture is classified into Centralised and Distributed. Resource descriptions are pre-configured on specific nodes or controlled by resource discovery system.

A-Centralised

RD: In centralised or directory system, the information about the resources which are provided by all other nodes is stored on pre-configured or auto-configured nodes in a limited number. In this system nodes or the directory, periodically update the registered resource information (resource description) in central repository.

OS: With centralised resource discovery, the OS is characterised by:

1. Using the centralised OS architecture.
2. Running one instance of OS on each resource.
3. Only "Head-Node" OS instance has resource description component.
4. All resources should have resource discovery component.
5. All resources should have application description component to extract the application resource requirements.
6. All resources should have Query Generator Component to generate queries.
7. Resource discovery components can generate queries into resource description component on "Head-Node" OS instance.
8. Resource matching or resource allocation component maps the queries according to the resources.
9. Resource reservation component is required for Multimedia applications.

B-Distributed

RD: In decentralised systems, the distribution of resource information is controlled by RD system and the information is almost distributed between all the participant nodes in a predefined manner. Overlays of this type commonly make use of Distributed Hash Tables (DHT). Decentralised systems can be extremely effective for searching resources using unique identification.

OS: With distributed resource discovery, the OS is characterised by:

1. Using the distributed OS architecture.
2. running one instance of OS on each resource.
3. All resources have their own resource description component.
4. All resources should have resource discovery component.
5. All resources should have application description component to extract the resource requirements of application.
6. All resources should have Query Generator Component to generate queries.
7. Resource discovery components can generate queries into the resource description component on other OS instances.
8. Resource matching or resource allocation component maps the queries according to the resources.
9. Resource reservation component is required for Multimedia applications**.**

UNSTRUCTURED

Distribution of the resource information among the nodes is not followed by a predefined or controlled mechanism. In spite of the decentralised structured system, the system built as unstructured or classical peer to peer (UP2P) supports partial matching and also complex querying.

HYBRID

The classical P2P service discovery architecture do not scale well in the large networks, that is why it should be replaced with the peer to peer overlays that utilize a hybrid architecture. The hybrid solution tries to combine the advantages offered by the classical structured and unstructured service discovery systems. It discovers the resources/cores that are needed as fast as possible and supports exact and partial matching.

C-Hybrid

RD: locating resources in hierarchical levels: core-chip-node-network using DHT and Anycast messaging.

OS:

1. Using the distributed OS architecture.
2. running one instance of OS on each resource.
3. each group (group of cores in chip or node level) of resources have one Resource Description Provider component on header-node.
4. All resources should have resource discovery component.
5. All resources should have application description component to extract the application resource requirements.
6. All resources should have Query Generator Component to generate queries.
7. Resource discovery components can generate queries into the local resource description component
8. Resource discovery queries can be forwarded to the remote Resource Description Provider component.
9. Anycast messaging is used to disseminate queries to the remote resource description components.
10. The history of successful queries would be recorded on each individual resource storage.
11. Resource matching or resource allocation component maps the queries to the resources
12. Resource reservation component is required for particular applications such as Multimedia applications.

4.B) REQUIRED OS COMPONENTS

- **Resource Discovery**: locating the most appropriate available resources for applications queries. It is initiated by an application query, continued by finding the resource, and ended by mapping the query to the resource. To do these procedures we need to deploy the models/components for application description , resource discovery, resource matching and resource description.
- **Resource Description Provider**: A component to offer static and dynamic information about the resources
- **Resource Requirement Identifier**: Resource discovery protocol is required to be supported by an application description model with the ability to estimate the resource requirements of applications before scheduling decisions are made.
- **Code Analysis and Segmentation**: The resource discovery should have an initial trigger. Code Analysis and Segmentation component starts the discovery procedure and after that, it will get the list of discovered resources.
- **Query Generator**: Provide mechanism to generate query according to the application description
- **Messaging**: Support Anycast messaging
- **Storage Component**: Provide support to maintain Distributed Hash Tables and Resource Cost Tables.

4.C) IMPACT ON OTHER OS COMPONENTS

- **Resource Reservation Component**: Provides mechanism to reserve resources for particular applications which means that resources are only freed when the process has finished.
- **Load Balancing Component**:. When the resource discovery step is fulfilled, the resource allocation procedure is requested to match the most suitable resource for execution of the

application according to several policies. As far as it is impossible to plan the real cost of an execution at the resource allocation step, load unbalances might appear. Therefore Load Balancer component is required to distribute overload of each of the nodes among other alternative potential resources.

- **Resource Matching Component**: Mapping of the applications to the resources

DEPENDENCIES

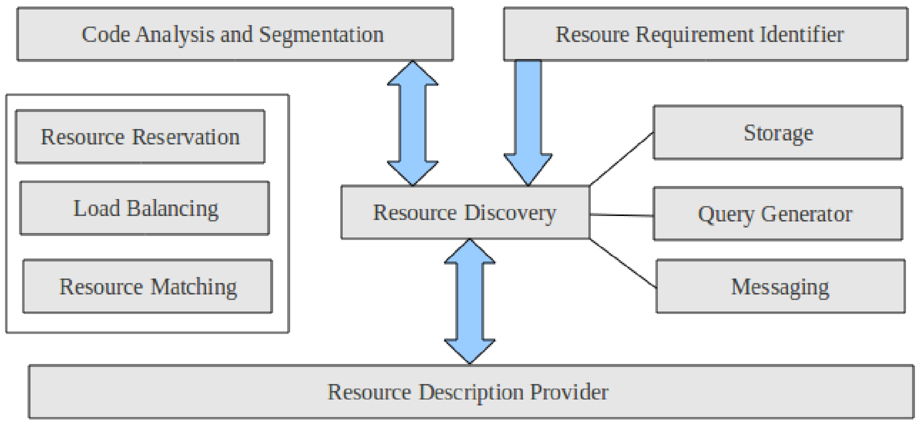


Figure 18: Interaction of resource discovery with other OS components.

Other OS components related to Resource Discovery are described below.

Related OS Component	Description
Resource Description Provider	A component to offer static and dynamic information about the resources
Resource Requirement Identifier	Resource discovery protocol is required to be supported by an application description model with the ability to estimate the resource requirements of applications before scheduling decisions are made.
Code Analysis and Segmentation	The resource discovery should have an initial trigger. Code Analysis and Segmentation component starts the discovery procedure and after that, it will get the list of discovered resources.
Query Generator	Provide mechanism to generate query according to the application description
Messaging	Provide support for Anycast messaging. Anycast is a robust scalable messaging mechanism which can be implemented for resource location-awareness (resource discovery) and resource load-awareness (load balancing)
Storage	Provide support to maintain Distributed Hash Tables and Resource Cost Tables.

5. SCHEDULING

Scheduling refers to the capability of the run-time environment to decide where and when to schedule individual code segments, threads of execution or whole processes. We will refer to these generically as scheduling entities from here on, unless otherwise stated.

As detailed above, the S(o)OS layout in terms of how many OS/kernel instances manage how many resources/cores, may include multiple possibilities. As a consequence, the scope of a scheduling algorithm (i.e., in terms of where software can be deployed) may be:

- one single core;
- a subset of the cores in the same physical system handled by the same kernel (e.g., traditional multi-core or multi-processor platforms or memory-coherent multi-core tiles in tile-based ones);
- a cluster of cores in the same or different processors (i.e., in the case of an SSI-enabled distributed kernel);
- a whole many-core processor, or a whole physical system;
- multiple physically separated systems interconnected through networking adapters.

Clearly, the OS component realising the scheduling functionality may possess various different architectures, in the above cases. For example, for temporally scheduling tasks on the same core, the scheduler needs to reside on the same core and operate very quickly. For load-balancing the workload among a cluster of cores, the scheduler needs to be distributed over the cores to be scheduled, and it will not operate too frequently, due to the inherent overheads in the core to core communications. Furthermore, for load-balancing among processors and cores of multiple physical systems interconnected via networking adapters, the scheduler will mainly be realised at a middleware level, and exploit hierarchical arrangement of scheduler and sub-schedulers as appropriate, possibly resembling the physical topology of the distributed network.

The target cores over which scheduling entities may be deployed are, from the viewpoint of a scheduler with a precise scope:

- either homogeneous, if all the resources in the scheduler scope are symmetric;
- or heterogeneous, if they are not.

Other key tunable characteristics of a scheduler are:

- whether or not the scheduler is *preemptive*, i.e., it allows the currently executing entity to be interrupted for executing another (more urgent/important) entity; in non-preemptive mode, it is responsibility of the thread itself to invoke explicitly the scheduler in order to allow other threads to execute, if and when needed;
- whether or not the scheduler is fair, i.e., it allocates (temporally and/or spatially) computing power to segments, threads, processes, applications, users in a (possibly weighted) fair way.

In order to operate correctly, a scheduler needs to have available a number of essential information:

- some meta-data about the resources in its scope (i.e., their provided capabilities);
- some meta-data about the requirements of the scheduling entities to deploy (required capabilities);
- some way to (efficiently) match required capabilities and provided ones (limiting a scheduler scope may help in making such operation more efficient).

In future many-core systems, it is envisaged that the scheduling decisions will be possibly undertaken:

- by the hardware (e.g., what done by GPGPUs)
- by the OS;
- by properly designed middleware, in a limited fashion;
- by the application (see next point), in a limited fashion.

The scheduler behaviour can be controlled up to a certain degree through proper APIs:

- applications can drive the deployment logic (where and when I want my threads to run), if (expert) developers really want/need it;
- applications can discriminate among the relative importance of various threads (or of applications):
    - either through priorities;
    - or through relative weights (i.e., weighted fair scheduling).

Some (expert) developers might want to exploit such capabilities to customise the scheduling behavior. For example, as an extreme, we might have finely-tuned code that has been optimised to be executed on a given hardware: nearly all scheduling decisions have been pre-investigated and tuned by the developer; the OS is just instructed to obey to the developer-supplied scheduling and deployment decisions. On the other hand, it is expected that most of the developers will not deal with such sophisticated capabilities, and therefore most developers will actually rely on the OS self-adaptation mechanisms for most of the scheduling decisions.

We focus on this last case, which constitutes the most interesting and challenging one from an OS perspective. Scheduling decisions not specified/constrained by the applications/developers are left to the OS, according to:

- the configured scope/system-wide goal, or “cost” to optimise (see below);
- possible specific goals provided by the application(s), for example satisfying certain precise timing requirements (as needed for RT workloads).

Furthermore, in order to allow the scheduler to dynamically act and re-act, the OS needs:

- a way for understanding which deployment decision is better (see possible goals as mentioned above); this is normally achieved on the basis of the defined “cost” function that drives the scheduler logics;
- a way to predict the performance speed-up (e.g., in terms of execution time and/or throughput of a code section or application) due to changing scheduling decisions.

5.A) PERFORMANCE PREDICTION

The prediction of the performance of specific code segments, or application threads, or whole processes or applications, may be helped by reusing monitoring data coming from past observations of the same code over the past. However, it is rarely achievable to have a comprehensive set of such past data, encompassing the high variety of possible run-time conditions that may arise. On one hand, application-level parameters, such as the resolution of a video streaming application, will highly influence the performance experienced by the same code segment. On the other hand, even when operating on the same data set, software will behave differently if deployed on heterogeneous processing units. Finally, due to interferences on the in-chip shared resources, such as the NoC interconnect in tile-based systems, independent applications running on different cores may still affect each other on the performance side.

One possible way to deal with this problem is through the use of proper mathematical models, that may help in assessing (even roughly) what is the expected performance of a code segment under certain run-time conditions which have not been observed/monitored yet.

Therefore, critical challenges for S(o)OS scheduling, and particularly for deployment and load-balancing decisions, are:

- performance prediction in presence of heterogeneous computing;
- performance prediction in presence of heterogeneous NoC latencies;
- performance prediction in presence of multiple values of application-level parameters influencing the performance.

The second point is further detailed in the experimental results detailed next.

5.B) EXECUTION TIMES ON TILE-BASED SYSTEMS

In order to highlight one of the critical issues in programming future many-core systems, namely the variability of the execution times in tile-based architectures, we performed a set of experiments on the Intel Single-chip Cloud Computer. This is a 48-core Intel Architecture many-core experimental processor prototype chip, mainly built to study many-core CPUs, their architectures and the possible techniques to program them. The 48 cores on the SCC are based on the Intel P54C processor, and are distributed in 24 tiles, with two cores per tile. As shown in Figure 19, the tiles are interconnected by an on-die two-dimensional 6-by-4 mesh network using Direction-Ordered Routing (DOR). The chip has four memory controllers (labelled in Figure 19 as MC0, … MC3) each capable of addressing 16GB of external memory and, thus, a total of 64GB of globally shared (but incoherent) memory. Each tile has a small fast local memory as well (16KB, 8KB per core), referred to as the Message-Passing Buffer (MPB), that can be accessed by all cores on the chip, and which is mapped to a new memory type needed to support message passing. Each P54C core has also L1 (16KB for data and 16KB for instruction) and L2 (256KB) cache memories. Cache coherency is not supported, so each core needs to explicitly flush cache lines, in order to actually share memory with the other cores.

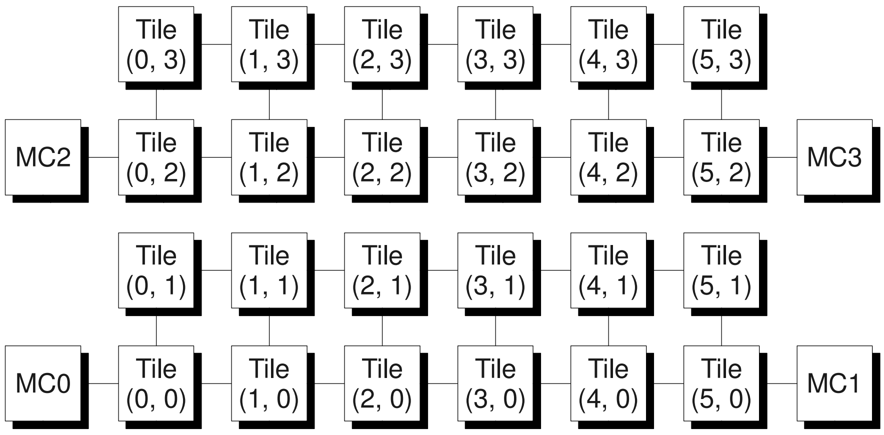


Figure 19: Layout of the 24-tiles mesh network within the Intel SCC chip. Cores have IDs increasing along the X tile coordinate, then the Y coordinate, e.g., Tile (0, 0) includes cores 0 and 1, Tile (5, 0) cores 10 and 11 and Tile (5, 3) includes cores 46 and 47.

The default software set-up on the Intel SCC has a modified version of Linux by Intel, with a completely distributed kernel where each core runs its own instance of a single-processor Linux OS. Also, it includes a user-space library, called RCCE, which can be used to access special features of the chip, such as fast core to core message passing primitives (these do not involve the OS kernel to operate). Alternatively, core to core communications via the special message-passing local memory can be realised by exploiting a special driver built into the kernel that allows to see a virtual network interface on each core OS, so that each core can communicate with each other core via standard networking primitives (e.g., TCP/IP).

We performed a set of latency measurement experiments to determine the latency involved in network-on-a-chip accesses to local message buffers and off-die private memory. We used the RCCE library routines for synchronisation and message passing. Note that, for all the experiments reported below, the traffic within the NoC was almost completely under the control of our profiling application, thus we posed ourselves in the most predictable conditions.

Network-on-Chip Latency

The NoC latency was determined by measuring the round-trip time for sending a message and replying to it, using the RCCE library, between each pair of cores.

The 3D graph in Figure 20 reports on the Z axis the round-trip times of sending a 32 byte message packet 100,000 times from core 0 to each other core, whose tile coordinates are reported on the X and Y axis.

Comparing also with the other collected round-trip times, obtained using each other core as source, we can conclude that, as expected due to the DOR routing, message-passing round-trip times are in a linear relationship with the routing distance between the communicating cores. Note that the

RCCE primitives for sending data through the MPB do not use merely the MPB memory itself, but they need to read/write the data to send/receive from another application-supplied buffer which needs to be allocated in the main off-chip memory. This means that, in theory, the times for performing data transmissions across cores via the RCCE message-passing primitives do not include merely the time for exchanging data through the MPB, but also some access times to the main memory. However, in the case of this experiment, such data was so small that it fitted entirely into the cache, so the access times towards the main memory affected in a negligible way the outcome of our measurements.

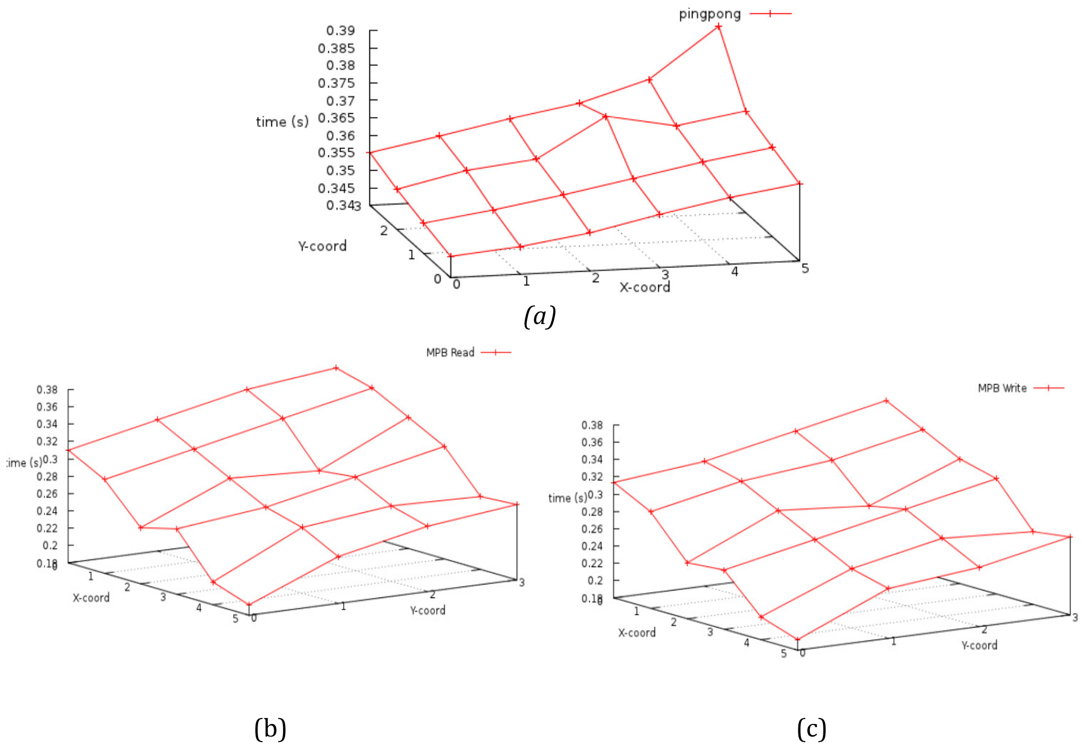


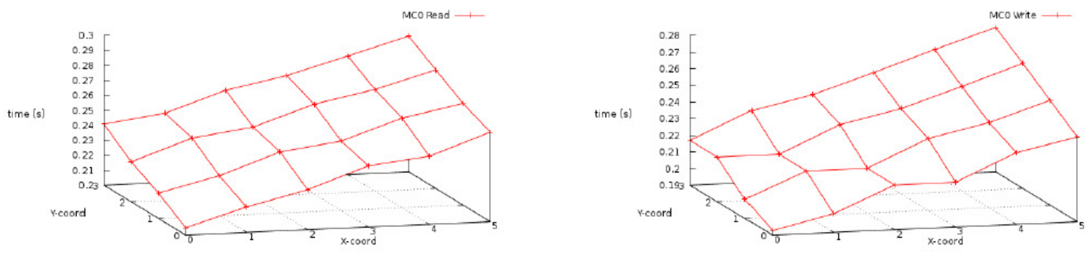
Figure 20: Times for: (a) round-trip message-passing between the first core and each other core; (b) read times from core 11 to each other core's MPB buffer; (c) write times from core 11 to each other core's MPB buffer.

Off-chip memory access latency

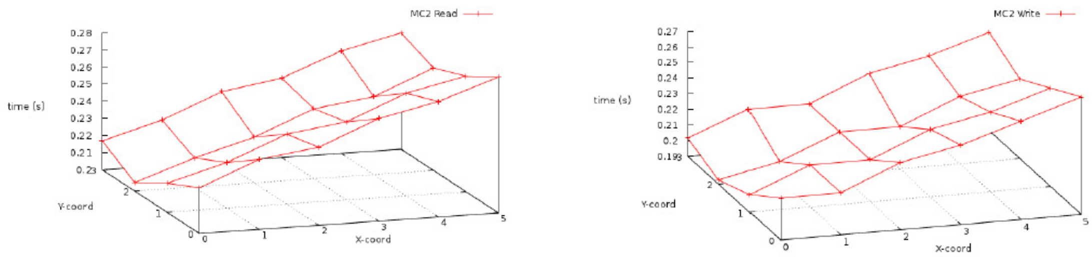
The access to off-chip memory by the core pass through the NoC and are routed to the appropriate edge tile depending on which physical memory bank the required byte resides on.

We performed an experiment allocating pages from each of the four banks of off-chip memory and measuring the time to read and write from/to these pages from each core. The results are shown in Figure 21, where the top pictures refer to accessing the first memory controller (MC0), whilst the bottom ones refer to accesses towards the third memory controller (MC2). Note that, while the first and second memory controllers are attached in the chip close to the corner tiles with coordinates (0, 0) and (5, 0), the closest tiles to the third and fourth controllers are not the corner tiles on the last Y coordinate (Y=3), but rather the two border tiles on Y=2, i.e., the tiles with coordinates (0, 2) and (5, 2). This is why, in Figure 21.(b), the minimum-time edge is located along Y=2, and not Y=3.

Summarising, we can conclude that the access times for the off-chip memory is again, as expected, linearly dependent on the routing distance between the tile of the core performing the access and the memory controller where the target memory locations reside. This is also highlighted in Figure 22, where the same access times (average values among all the cores) to the off-chip memory reported above have been plotted as a function of the routing distance between the accessing core tile, and the target memory controller.



(a)



(b)

Figure 21: Times for reading (left pictures) and writing (right pictures) 1MB from/to the off-chip memory from each core (coordinates on X and Y axes), when the memory pages are allocated in the first (a) and third (b) memory controllers.

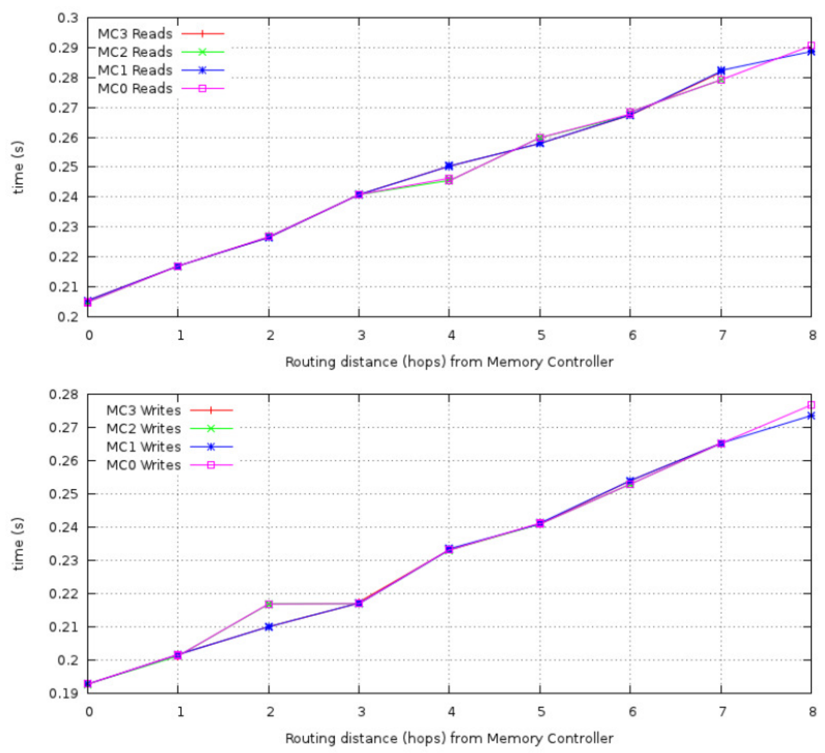


Figure 22: Read (top) and write (bottom) times as a function of the routing distance between each core and the target memory controller.

Loaded Conditions

As stated, the above experiments have been conducted under very predictable conditions, in which the NoC in the chip was (almost) merely affected by the traffic imposed by each core separately.

However, in a realistic scenario, the cores will be injecting packets into the NoC at the same time, causing variability of the memory access times of each other.

In order to highlight this phenomenon, we built an experimental scenario making use of a synthetic pipelined application exploiting the full set of available 48 cores in the chip, in order to process data coming from some source in chunks. The idea is that, as a data chunk becomes available, instead of applying all the needed computations on a single core, the first core only applies a subset of the needed computations, then it passes the data to the next core in the pipeline for further processing, and so on for all the 48 cores. This has the advantage that the first core becomes much earlier ready to process the next data chunk, as compared to having to apply the full set of computations, i.e., a pipelined implementation is able to increase its throughput in terms of processed chunks per second. However, the latency of the application will depend on how much time it takes for each single data chunk to traverse the whole set of computations made by all the pipeline stages.

Of course, our application is synthetic and a real one will hardly be capable of splitting the computations in exactly 48 quite balanced operations. However, we wanted to experiment with what happens when the whole set of available cores is imposing traffic on the NoC, so we used a single 48-stages pipelined application to this purpose.

The experimental results obtained on the Intel SCC are reported in Figure 23, where we report on the X axis the progressive identifier of the processed chunk, and on the Y axis the processing time as experienced by 3 out of the 48 computing stages.

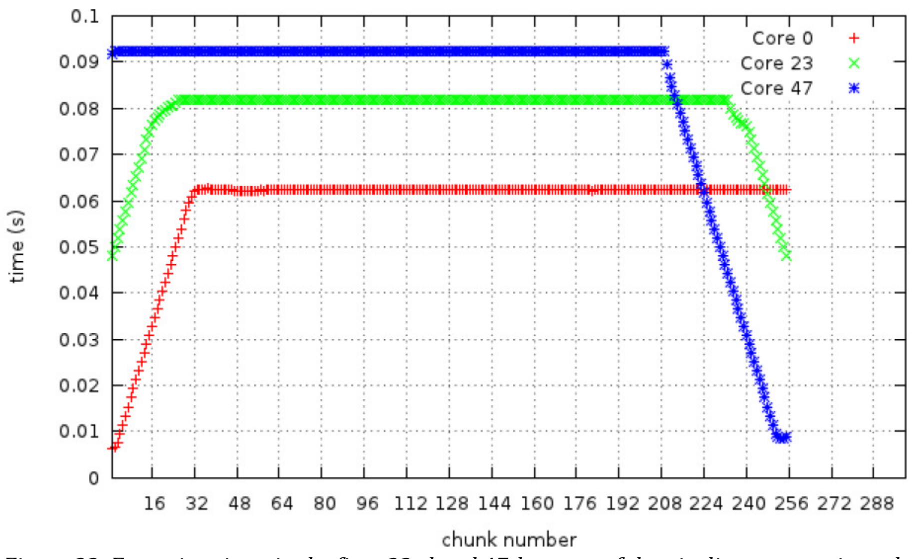


Figure 23: Execution times in the first, 23rd and 47th stages of the pipeline, as experienced on the various chunks processed through the pipeline, in the uncached shared memory case.

Note that, at the beginning of the experiment, the pipeline is idle, whilst as the experiment proceeds the pipeline fills more and more, thus there are more and more stages working concurrently, up to a point in which the pipeline is completely filled. Later, close to the end of the experiment, the source does not provide data chunks to process any more, so the pipeline stages gradually go back to the idle mode, starting from the first stages and going towards the last ones.

As it can be seen, the first pipeline stage, deployed on core 0 (red curve in Figure 23), experiences growing execution times during the transitory as the pipeline fills up, due to the increasing traffic imposed on the NoC by the other pipeline stages that gradually “wake-up”. This continues up to the “steady-state” condition, where the execution time remains constant, till the last processed chunk, during which still we have a completely fully operational pipeline. For the last pipeline stage, deployed on core 47 (blue curve in Figure 23), the execution times behave exactly in the dual way, i.e., they are constant from the first processed chunk, that reaches the stage when the pipeline is already fully operational, then they gradually decrease close to the end of the experiment, while the pipeline gradually goes idle. An intermediate stage like the one deployed on core 23 (green curve in Figure 23) experiences both behaviours.

The above measured data were relative to a pipeline using uncached shared memory for accessing the data, so the interferences among the various cores were particularly amplified.

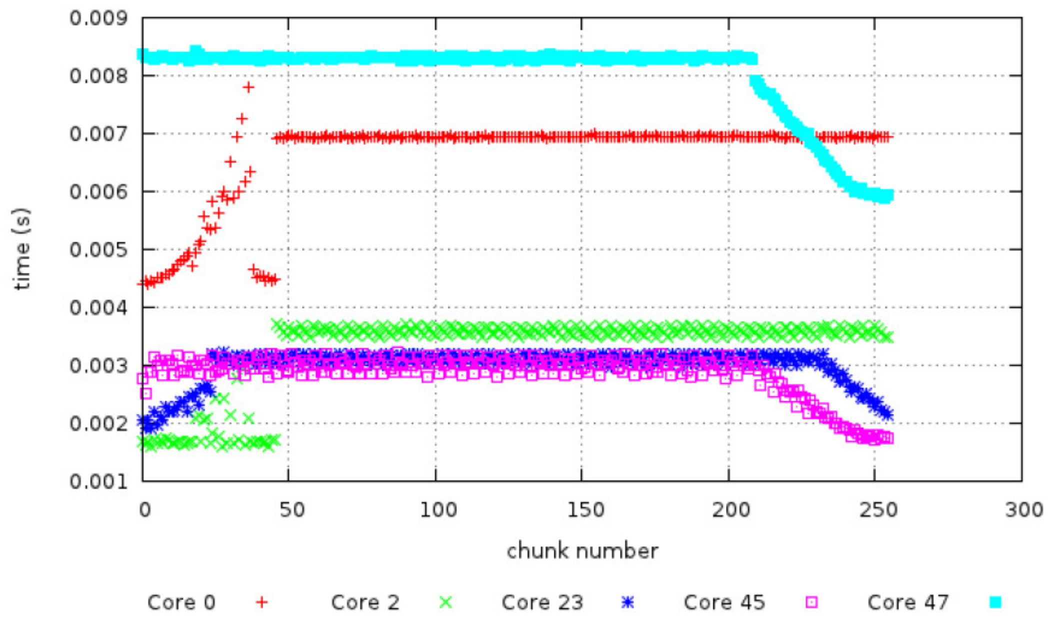


Figure 24: Execution times in various stages of the pipeline, as experienced on the various chunks processed through the pipeline, in the cached shared memory case. Cores 0 and 47 show generally higher times since these cores additionally have the task of reading data from input buffers into the pipeline and of writing data from the pipeline into the output buffer.

In Figure 24, the same figures are reported for the case in which the pipeline stages were using a cached shared-memory implementation. As can be seen, the general trend in execution time variability follows a similar pattern to the previous case. However, the variability is greatly reduced, thanks to the mediation of the cache in accessing the memory. Still, we can see that the execution times can potentially double, when switching from an empty network-on-a-chip to a completely loaded one (in the previous case, the first stage was undergoing a multiplication by 6 of its own execution time). However, the exact inflation factor is of course highly depending on the actual pattern of access to the memory of the various tasks, as well as on how much the traffic imposed on the NoC by the various cores overlap with each other, in their path towards the memory controllers.

5.C) DEPLOYING TASKS IN TILE-BASED SYSTEMS

As highlighted with the experiments above, tile-based systems exhibit different communication latencies depending on the routing distance between communicating cores. In the synthetic pipelined application described above, we played with a few deployment options, in order to evaluate the impact of the deployment logics on the performance of the application.

First, we considered a trivial (or careless) deployment of the 48 pipeline stages to the 48 cores, placing the pipeline stages over cores in order of progressive core ID (i.e., the first stage on core 0, the last one on core 47).

Then, we considered a smarter deployment aimed at reducing core to core latencies. Therefore, looking at Figure 1, we deployed the pipeline in a snake-like fashion starting from the bottom-left towards the bottom-right tile, then proceeding from right to left on the Y=1 tiles, then again from left to right, etc. Precisely, the used cores ordering was: 0, 1, 2, …, 10, 11, 23, 22, …, 13, 12, 24, 25, …, 34, 35, 47, 46, …, 38, 37, 36.

We considered a pipeline processing an entirely available data-set of various sizes (1, 2, 4 and 8 MB), in chunks of a fixed size of 256KB, and we measured the execution time of the experiment under the 2 deployment options just described. Also, each experiment was repeated using cached computations in private memory and core to core communications based on the RCCE message-passing primitives, and using directly shared cached memory so as to avoid the unneeded extra copies of data. Results are reported in the table below. As it can be seen, the advantages coming from a proper deployment of the application onto the system are merely noticeable with the private memory implementation, with a speed-up below 0.21%, however they are substantial with the shared cached memory implementation, with a speed-up between 6% and 10,2%.

Data Size (MB)	**Private memory**		**Shared cached memory**	
	Trivial	**Smart**	**Trivial**	**Smart**
1	3.678	3.670	0.920	0.868
2	4.155	4.151	1.256	1.143
4	5.128	5.117	2.252	2.042
8	7.060	7.054	4.745	4.313

5.D) Code Distribution Costs and Criteria

In all distributed execution, we pursue some quality-related goals with the execution. As such, in most HPC cases the primary concern is *performance*, i.e. to achieve the maximum amount of operations in minimum time - this does not necessarily imply that each segment must execute to maximum performance. On the other hand, in real-time scenarios, the primary concern relates to meeting the time-constraint of the execution, whereas this can either apply to individual segments / functions of the code, or even to the overall behaviour of the application as a whole. Notably, HPC applications can be considered a specific type of time-constrained use case, where timing should generally be reduced to a minimum.

Furthermore, for distributed real-time applications, the stated scheduling goal of the application is to respect its temporal constraints, that may be expressed in form of:

- end-to-end deadline(s): more paths may be associated with e2e deadlines, if desired;
- throughput at the various communication (virtual) links among the application DFG components.

Differently from the HPC use-case, target figures for such metrics are clearly identified by the application developer, and/or communicated by the application to the OS, so that the latter may know quite well when the currently employed scheduling and distribution decisions are fulfilling the application requirements (or what deviations are being experienced).

The according goals need to be maintained and addressed at the OS execution runtime, i.e. need to be mapped to application specific criteria. As S(o)OS is not investigating into program optimisation per se, the primary concern here relates to the assignment of appropriate resources so as to meet these goals. In the most general case, the OS needs to identify the resources that allow for the best execution of the respective code segments. This may involve code adaptation so as to make the most of the available resources. Following this general case, the main task is to find *for each segment* the resources that are best suited for the respective code behaviour.

More specifically, however, the individual segments stand in a specific relationship to the overall quality criteria, that may not imply a direct one-to-one mapping. For example, a parallel thread with shorter runtime than any other thread to be executed at the same time is not performance critical. Also, if a thread has to wait for an input from another thread, it may adjust its execution time to this other thread rather than to maximum performance. Similarly, timing constraints may only apply to certain subsections, of which again some segments are "optional", so that the constraints only need to be applied to the according segments. These differences need to be taken into account when selecting the appropriate resource and performing the code adaptation task.

A general solution consists in making all segments execute at maximum performance. We must however assume that:

- not enough resources of all required types are available - in fact there may be certain resources that the same application (or multiple applications in fact) compete over (this is generally true for limited resources in the system, such as I/O devices);
- multiple options are available for any given segment, i.e. that the segment is not restricted to a specific resource type but can be executed in different resources at different costs. Note that I/O resources such as keyboard or monitor form exceptions to this rule, but they are not considered "executing" resources.

It is noteworthy to mention that scheduling operates in tight relationship with the segmentation process of the Code Analysis and Segmentation OS component. This may dynamically parallelise a sequential piece of code, identifying code segments to be deployed somewhere else. However, whether or not it is worth to activate the corresponding scheduling/deployment option needs to be carefully evaluated, depending on the expected migration cost associated with the operation. The process can be simplified by exploiting hierarchical "encapsulation" of criteria and system performances, so as to allow for an iterative mapping approach.

MIGRATION & "LIVE COST"

A special case of comparing the deployment options consists in runtime re-adaptation and/or migration. Migration is principally only necessary if the quality criteria are not met by the current scheduling and deployment scenario. In particular for HPC applications, this can be considered generally the case *if there is a chance that the performance can be improved with a different code distribution* - implicitly, the cost for redistribution needs to be assessed whenever:

- new resources have been added that have not been assessed yet (or which combination with existing resources may lead to changes);
- new resources have been identified;
- new information about the resources is available;
- the current execution "quality" is lower than the expected quality (i.e. the assessment was wrong or not enough information was available at the time);
- more information about the requirements of the code is available.

During live analysis the cost for code transformation and migration has to be considered as an additional impact factor on the overall quality - in particular with respect to time: depending on the situation, the execution may have to be halted (or some form of live migration might be possible) in order to adapt the according segment(s) and move it/them to the new destinations. If none of the respective segments are active at the time and will not be needed during migration time, the re-adaptation can occur in the background. It is likely, however, that the according segments will be required in some form during the adaptation and migration process, thus reducing the overall performance and hence the time-related quality criteria. There may also be segments that would benefit from migration to another platform but which have little impact on the overall performance, e.g. if they are only executed once (or one more time), so that the overhead for adaptation and migration exceeds the gain.

The main question to be answered prior to migration is hence whether the (potential) loss in performance (introduced by the according overhead) is outweighed by the performance boost introduced in the new environment.

COST ASSESSMENT

In both live migration and offline analysis, we hence require some information about the general execution performance with respect to the specified quality criteria, so as to compare the distribution options with one another and thus form a selection.

In addition to that, live migration requires that the cost for migration is assessed so as to respect this information in the redistribution consideration. In other words, to estimate whether redistribution is worth execution in the first instance.

Generally we assume that cost analysis can be executed in the background, i.e. that the analysis itself does not affect the execution behaviour. However, this heavily depends on the frequency by which such analysis is conducted and its accuracy (influencing the amount of collected data, as well as the associated overheads).

EXECUTION COST

The primary concern consists in estimating the relationship between selected environment and its impact on the code execution performance. There are two major obstacles to this issue:

- the runtime behaviour of a given code segment is very difficult (to impossible) to assess. This is partially due to the impact of data specific behaviour that e.g. determines the size of a loop within the segment;
- the relationship between hardware and execution performance is generally unknown.
  Even though most processors are specifically generated for a specific type of code behaviour, this does not imply that a general statement about execution performance for any given code can be made.

To overcome these obstacles, there are multiple general approaches which we could use to give indicators (rather than concrete values):

- **worst-case performance**: taken from the real-time domain, it is possible to give worst-case performance estimations for specific code types. This does not provide information about the execution time experienced most of the times, or on the average;
- **algorithmic kernels**: if the code "function" can be classified, some general execution performance observations on specific hardware types can be made. This can be refined further with runtime monitoring information (data specific behaviour may impact on this estimation);
- **profiling**: assess the code execution time on basis of historical data. Whilst this respects data specific behaviour, it requires that the code has been monitored for quite some time; it also does not provide information about code execution performance on unknown platforms;
- **code annotation**: next to automated information, the developer can provide additional information that can help assessing execution time. This includes in particular identification of kernel types (i.e. algorithmic behaviour) and data space limitations (I.e. indication of the general data behaviour, thus providing information about the expected number of loops).

Note that the primary focus of these criteria rests on performance / execution time, as from the general project objectives.

USING COST INFORMATION

As detailed in the context of code analysis and segmentation, the expected execution time per code segment will be used for selecting the best destination platform, but also for supporting the distribution and "parallelisation" decision.

The general process involves comparing all potential options in terms of destination resources and parallel execution. The selection will base on the best match of the criteria against the calculated expected behaviour. By exploiting natural constraints and the dependency graph generated during the analysis, the according search space can be reduced significantly.

COST FACTORS

The following is just an initial list of criteria that participate in specifying the overall execution cost - here particularly related to execution time:

- latency
- bandwidth
- dependencies
- clock rate / fops
- pipeline
- specific code requirements towards hardware
    - cache size
    - vectorisation
    - precision

These will be further refined in the second phase of the project.

MIGRATION COST

For live migration, it is insufficient to "just" identify a better distribution / adaptation to the platform, as the migration and adjustment process may delay the overall execution more than is gained by this re-adaptation. In other words, even though the code may run faster, the total time is even higher, i.e., the job effectively runs slower. Even worse, it may happen under certain conditions that the OS may decide to constantly shift the code, thus never actually executing it.

Thus, effectively, the live migration should only be executed if the gain is higher than the cost, or in other words, if the sped-up gained by the adaptation is way higher than the *total* impact of migration in terms of delay. This includes direct delays (stopping the process, recompiling it if necessary, setting up the destination and migrating the process + context into the destination), as well as indirect ones (delays in communicating with the process, delays in synchronisation etc, i.e., wait time). As already discussed in the context of cost assessment in general above, these costs and in particular their indirect part is difficult to gather.

In general we can formulate that migration should take place if the remaining execution time unchanged is far greater than the remaining execution time adapted plus the adaptation time.

This is still an approximation for multiple reasons:

- multiple segments may be in the middle of execution during adaptation;
- adaptation time does not only affect the respective segments, but also all dependencies - this must be encoded in the "indirect" adaptation time of dependent segments.

5.E) REQUIRED OS COMPONENTS

The set of OS components/modules required by the Scheduler component can be listed as follows:

- I/O (network and/or disk) for accessing data;
- interrupt management (for I/O);
- process/thread management
- IPC communications:
    - in-chip IPC
    - off-chip IPC
- in-chip scheduler
- off-chip scheduler
- dynamic load balancer
- performance predictor/mapper for heterogeneous hardware
- performance monitoring

- resources saturation-level monitoring
- code migration
- location-independent access to application data (SSI-like)

Run-time interactions application<->OS are needed to:

- retrieve data if not readily available, i.e., loading from disk or remote storage, or fetching from a non-reachable memory area - are asynchronous interfaces useful ?
- buffer management (copy-less/copy-free I/O & networking stacks)
- dynamically scale out (and in, in case it is needed): a proper interface should allow one or more application threads to split the in-progress computations into more segments/threads to be (re-)deployed in a parallel fashion, or to group more segments/threads into a sequential fashion if the previous splitting is causing too overhead
- understanding whether or not the application is suffering of "unbalancing problems" (i.e., if the application spin-waits, it may be hard for the OS to understand it is waiting for another thread to provide the output)

Additionally, for RT applications, the OS needs to provide additional capabilities:

- time management functionality;
- timers management functionality;
- temporal scheduling (if a single application thread activated periodically cannot saturate a core, it is worth to pack more of them on the same core).

As stated, further run-time interactions application<->OS are needed to understand whether or not the application timing constraints are being fulfilled (i.e., the OS may just have no clue about whether or not a deadline was missed).

6. In-chip Communications

On a tile-based many-core system, we foresee a number of possible communication and synchronisation mechanisms among cores:

1. if there are subsets of cores sharing a coherent view of the memory, for example cores within the same tile, then we foresee the use of a single OS instance for such a small number of cores, and the standard communication and synchronisation mechanisms as available on nowadays OSes will work among them; however, for cores crossing the memory-coherency boundaries, we will be in one of the following other cases;
2. if there is a fast local memory, for example within the tiles, then it will be exploited to realise fast core to core communication primitives;
3. if cores are sharing a global, but non-coherent, view of the main memory, then it is possible to share memory buffers across cores simply by passing pointers; the mediation of a cache memory in accessing the globally shared memory means that the program needs to explicitly flush and invalidate the cache (either the whole cache, or the affected/needed lines), in order to ensure “manually” a coherent view of the memory. For example, after having filled a buffer, before sending the pointer to the receiver core, the sender core must flush any pending writes. Similarly, before reading the data, the receiver needs to ensure the corresponding cache lines are invalidated. An alternative may also be the one to disable the cache, in order to ensure that the main memory properly reflects the applied changes to a memory region by a core towards the other cores.

For example, the prototype Intel SCC platform by Intel allows for the above communication mechanisms 2 and 3, but not 1. Indeed, the two cores in each tile have independent and non-coherent caches. However, a local memory within each tile, referred to as the Message-Passing Buffer (MPB), allows for fast core to core communications. These are made available to application developers through a set of primitives that directly exploit the MPB from the user-space, without intermediation of the kernel. Actually, a kernel-level driver allows also for using the MPB memory for realising fast TCP/IP virtual networking communications among cores, even though the performance is expected to be greatly reduced in such a case. Finally, on the Intel SCC all of the cores have a global but non-coherent view of the whole memory available across the chip, including both the main memory and the local MPB buffers in each tile. This allows for exchanging large data segments across cores by simply sending pointers in shared memory around, ensuring proper flushing and invalidation of the involved cache memories. Also, the fast MPB primitives can be exploited for sending these pointers, realising very efficient communications.

A comprehensive evaluation of the mentioned communication mechanisms over the Intel SCC is under way, and it will be presented in a future evaluation deliverable. It is worth to mention that, also thanks to our interactions with the Intel support teams, it was possible to identify further bugs in the kernel-level drivers supporting cache flushing, which have been fixed only recently[6].

7. Component-based Asynchronous Kernel

The component-based asynchronous kernel is framework for implementing components for distributed operation and interaction among cores, the system is treated as a loosely coupled system, components can be migrated among cores or even machines. Another fundamental feature is the asynchronous pattern, which is fundamental in a network environment. The objective is to present a framework to implement components in a transparent way to the user without concern with message passing and to enable migration in loosely coupled system (e.g.: cloud computing). From the point of view of an application the component-based model is transparent to the application.

PROVIDED FUNCTIONALITY

- components can be moved across cores, nodes, machines.
- components can interact using message-passing.
- components are abstract, allowing heterogeneous implementations.
- components can interact in a location-independent fashion.
- components can represent processes, threads, files, devices, etc.
- component interfaces are asynchronous.

7.a) Motivation

We consider future many-core systems as a network of independent cores with no cache coherence or even memory sharing at certain levels. We also expect this systems to be heterogeneous. In a network system, cores interface with each other using message passing. However, this can lead to cumbersome programming patterns, where the user has to explicitly program with messages instead of method calls. We intend to use abstract component interfaces, making the implementation transparent to the user. Thus, a component instance can have:

- a generic implementation,
- a special purpose hardware implementation,
- an implementation for message passing when it is required to interface with another core.

Networks are naturally asynchronous, i.e., sending a message may require a reply, and the sender will have to wait for it. Thus, an non blocking model is fundamental to avoid the overhead of context switch of a blocking model. However, operating systems have been programmed using sequential

[6] Fore more information, refer to: http://marcbug.scc-dc.com/bugzilla3/show_bug.cgi?id=195.

(synchronous) language patterns. Asynchronous methods calls take a completion handler (function object or lambda) to invoked when the operation completes, this can happen immediately or later. No assumptions can be made has to when the completion handler is executed, it will be asynchronously. We argue that a fully asynchronous kernel which does not block should be the principle of an operating system for future many-core systems, which have embraced a network approach in order to scale.

7.B) REQUIRED OS COMPONENTS

The component-based kernel is a framework to implement kernel components. However, it requires some components on its own to provide a working solution. The provided asynchronous interface requires low-level support from asynchronous primitives. The abstraction provided by the framework allows for communication using message-passing and for the migration of component instances, including data managed by a component (e.g., an address space). This requires a communication component to provide an abstraction over different mediums and levels of communication, and to support optimal communication, i.e., to exploit the capabilities of communication medium. Finally, a component manager is required to manage component instances lifetime and migration.

ASYNCHRONOUS SUPPORT PRIMITIVES

The **async** primitive provides low level support for asynchronous completion handlers. This primitive contains a stack for storing completion handlers and the required mechanism to dequeue the completions handlers and execute them. When an asynchronous method is called, a completion handler function object is passed as an argument. The completion function object is stored in the **async** primitive stack, it is called when the operation completes with the result of the operation.

The **async** primitive can be put in wait state which allows it to referenced latter. An asynchronous read operation in hard disk would take a considerable amount of time in terms of CPU processing. Thus the **async** primitive would be put in wait state and a reference to it would be stored by hard disk driver to be reference latter. The function would return immediately allowing for other things to be processed. When the issued hard disk operation completes an interrupt is generated, the hard disk driver would then use the stored reference for the **async** primitive and notify it of the completion which results in the invocation of the completion handlers.

COMPONENT COMMUNICATION MODULE

Kernel components need to communicate at various levels and different mediums, it also needs to exploit the capabilities of each medium in order to use the most optimal communication mechanism. Thus a “network interface” abstraction must offer several communication mechanism which take into account the communication requirements and characteristics summarised below.

The following is a list of possible communication levels:

- core → communication among cores within the same chip.
- chip → communication among cores of different chips/dyes, but same board/chip.
- node → communication among cores located on separate boards.
- rack → communication among cores located on separate racks.
- grid → communication among cores located on separate computers, but same network.

Depending on the communication level characteristics and capabilities, several mechanisms can be exploited:

- Shared memory (with or without cache coherence)
- DMA

- Lazy Copy (move data to a memory location that offers lower latency or exploit the core memory management unit to lazy copy in a safe manner)
- Compression (can reduce latency and overhead, e.g. over Ethernet links)

The best communication mechanism will depend on the requirements and characteristics of a given scenario:

- message passing
- object migration
- address space migration, copy/cloning

For example, message passing should be short in size, packet oriented, require very low latency and guaranteed delivery. On the other hand, for address space migration it's a fixed page size, it may not necessarily need a guaranteed delivery as the system can always reissue the request when it actually touches the page in question.

One of the requirements for kernel communication model is location independent addressing. However, this is handled at higher level. Thus at this level addressing is handled with physical addresses. The "network interface" should be able to handle unicast, anycast, multicast and broadcast addressing schemes.

COMPONENT MANAGER

The component manager is responsible to manage the component instances: lifetime, creation, destruction and migration. Each component exposes a type information that allows the component manager to allocate memory for the component instance, select the implementation, construct the instance and destruct it when its lifetime ends. The type information also provide an handler for message-passing support, to handle a component specific message. A component type information also exposes information that provides the component manager a clear view of resource dependencies between the threads of a process for migration purposes.

The lifetime of an instance is managed with a reference count, the reference count keeps track of local and remote references.

7.C) IMPACT ON OTHER OS COMPONENTS

The component-based asynchronous kernel is a framework for the kernel components implementation, thus it has an (indirect) impact in all components.

8. TRANSACTIONAL MEMORY

8.A) KERNEL-LEVEL TRANSACTIONS

The isolated nature of transactions requires to combine them with locks to support network I/O in a large scale system. Hence the system should provide additional guarantees like strong atomicity, which requires the non-transactional accesses to be serialised with respect to transactional accesses.

A preliminary work, TxLinux [3], considered the combination of locks and hardware transactions into a cooperative transactional spinlock, (**cxpsinlock**), that presents promising results with the MetaTM hardware transactional memory system. The idea is that a transaction that executes an I/O, restarts and switches to a pessimistic mode by acquiring a lock and entering a critical section before issuing the I/O. The scheduler and the HTM are tuned to nearly eliminate priority and policy inversion. More recently, TxOS [2] has been proposed as a transaction-based OS that provides transactions at the system level, but that can be nested inside user-level transactions.

8.B) Modularity in Locking Schemes

Existing purely lock-based systems suffer from the lack of modularity of locks and their great complexity. In one comprehensive study on Linux bugs [9], 346 of 1025 bugs (34%) are due to synchronisation while another study [10] found 4 confirmed and 8 unconfirmed deadlock bugs in the Linux kernel v2.5.

Deadlock bugs are inherent to using fine-grain locks in various modules. These locks consist protect smaller regions of memory than coarse-grain locks, hence being less prone to false-conflicts and enabling higher concurrency between concurrently contending threads. Nevertheless, their multiplicity tends to lead to deadlocks: two threads waiting to acquire the lock owned by the other.

An example of deadlock may happen if for example when renaming a file to `file3` to `file4` while another rename in the opposite order. Consider the Google File System (GFS) that locks files (parent file first and child file second) on the pathname using fine-grain read/write locks to modify them [5].

```
read_lock (/dir)              read_lock (/dir)
write_lock (/dir/file1)       write_lock (/dir/file2)
update file1                  update file2
unlock (/dir/file1)           unlock (/dir/file2)
unlock (/dir)                 unlock (/dir)
```

modifying `file1`	modifying `file2`

This result would lead to a deadlock if the first rename from `file3` to `file4` by removing `file3` and inserting `file4` while the second rename remove `file4` before inserting `file3`. This scenario indicates the modularity limitations of GFS as any other module locking a file should respect the order in which any file of the same level must be protected, in other modules.

```
        root                        root
        /  \                        /  \
     dir1  dir2                  dir1  dir2
     /        \                  /        \
  file1  •  file2             file1  •  file2
```

renaming `file2` into `file1`	renaming `file2` into `file1`

In contrast, transactions are deadlock-free thanks to their ability to abort. They do not wait when encountering an already acquired lock, but rather abort and roll-back by releasing all the locks.

8.C) File System Use-Case: I/O and Syscalls Problems and Solutions

The systems calls are generally atomic and isolated from the rest of the systems and the evolving needs of user always require the system to provide more complex services. I/O and system calls are different. While a transaction may create a new file, the kernel data structures (sys call) may be updated without necessarily the disk to be written (I/O). FreeBSD has adopted soft update transactions to allow a **rename(x,y)** to run concurrently with a **rename(y,x)**. This concurrency is necessary to batch multiple FS update into a single disk write. Typically, this avoids re-writing a single block several time to disk---as in the case in the BSD Fast File System (4.2BSD)---even though the block must be guaranteed to be written after all its predecessors are safely on disk. However, this is not yet failure atomic and thus fault-tolerant: the disk is not guaranteed to recover from a failure as the resulting state may not be consistent.

WAFL file system ensures that an old bunch of file system data gets updated atomically by recording them in a tree structure whose root is updated in a single step. The Berkley LFS and Sprite LFS are log-structuring file systems that write all filesystem data in a continuous stream, the log. Although it makes information retrieval more complicated, it is easy to mark the point in the log up to which data have been committed and stored and it benefits from consecutive writes without disk seeks. Finally, journalling (i.e., log enhanced) FS keep old and new versions of incomplete updates on the disk. Only after commit, the new version is copied back to the original location, as in the Linux ext3 file-system.

MetaTM uses an eager version management while TxOS uses lazy version management. In other words, MetaTM stores new values in place and the old value is copied into an undo log, whereas TxOS uses shadow copy of kernel objects redirecting all accesses of the current transaction to the created shadow copy. This shadow copy plays the role of a redo-log that records all changes before they get reported the current kernel object get replaced by this shadow copy. Thanks to redo-logging and the I/O buffers to defer to I/O execution to the commit-time of the transaction. Concurrent transactional accesses to the kernel object are detected using extra metadata information associated to each kernel object.

8.D) APPLICATION USE-CASE: REAL-TIME STREAMING APPLICATION

The transaction abstraction plays an important role in the design of a real-time streaming applications. Priority-inversion, which consists of a higher priority thread waiting for a lower priority thread, is a common problem in lock-based system that require complex priority inheritance to ensure that a blocking thread gets temporarily the priority of the thread it blocks.

TM embeds contention management that remedy this issue by ensuring that among two conflicting transactions that is provided the higher priority wins. However, most of the TM contention managers relies on fixed priorities like timestamps taken at the beginning of a transaction to let the older transaction resumes, hence it is difficult to adapt such priorities during the course of the transaction. For the sake of QoS, a task with a high priority should avoid using its recent timestamp as its priority and should rather exploit the priority assigned by the OS scheduler.

A more subtle problem, often referred to as the policy inversion problem, stems from the combination of real-time tasks with non-real-time tasks in the same OS. Real-time tasks are not simply considered as tasks with a higher priority than non-real-time tasks, the real-time task will always be scheduled before any other non-real-time tasks. The policy inversion problem can be avoided only if the contention manager is aware of the task policies (in addition to the task priorities). For real-time task, one could reasonably considered that the priorities and policies are assigned at the granularity of transactions, so that neither the policy nor the priority changes in the course of the transaction.

8.E) RELATIONS WITH OTHER COMPONENTS

Scheduler	The transactional memory may be given information from the scheduler component in order to evaluate the policy and priorities of transactions. It can also give in return information about critical tasks that have been aborted a large number of time reaching a threshold indicating that other conflicting transaction must abort in favor of the current one.
Messaging	To achieve scalability of transactions, the transactional memory component should call the messaging component to emulate read and writes through message-passing. This is for example a requirement for our Distributed Transactional Memory.
Resource Discovery	The transactional memory component should be aware of the component in order to back-off a transaction instead of re-starting it right after and overloading the available resources.

9. Speculative Execution

Speculative Execution, or Speculative Access to Memory (SAM), refers to the execution of parts of an application out of the logical sequential order, that means, before the execution of the conditional operations and the resolution of the data dependencies. It is a technique for eliminating performance bottlenecks imposed by control flow, or unresolved data dependencies by static analysis. In this chapter the implications, consequences and requirements of implementing SAM as an OS service are described.

9.a) Motivation

Computing industry relied on increasing the clock frequency and on uniprocessor architectural enhancements to improve performance. The situation has changed nowadays, small architectural improvements continue, but power an thermal issues have made the approach of increasing clock frequency non feasible any more. Semiconductor fabrics continue developing the manufacturing technology and according to the still valid Moore's law, the number of transistor per unit area double every few months. Processors designers use the additional transistor density to place multiple cores in the same die. This creates a situation in which only multi-threaded applications will take advantage of the new available resources and increase performance. To be able to produce parallel programs is the main concern of the new multicore era [18].

Many applications exhibit irregular access to memory, a complex control flow and dynamic loop limits [21]. Normal code parallelisation and optimisation need to take a pessimistic approach to guarantee correctness of execution due to data and control dependencies. Being able to resolve and remove data dependencies in run-time can dramatically improve parallelisation. SAM can potentially widen the scope of single threaded applications that can be parallelised by resolving data dependencies at run-time that otherwise would prevent parallelism from being extracted. SAM is a parallelisation technique that is applied to regions of code that might contain a great amount of parallelism but cannot be statically proven to preserve the sequential behaviour under parallel execution. Speculative worker threads execute code out of the sequential order even when these may contain a true data dependencies. The master thread keeps the correct sequence of state and control flow. While the speculative worker threads, may consume and produce "dirty" values, that is the reason it is necessary to track the inter threads dependencies and memory accesses to revert to a safe point and restart the computation when a dependency violation happens.

We are going to analyse the means that are necessary in the OS architecture for SAM to be implemented and work correctly. We will describe the minimum OS functionality needed and optional functionality, that if not strictly necessary, simplifies the implementation and/or improve the execution performance. This information will be obtained from two different SAM implementation perspectives; as an OS module, and as an independent user-space library.

9.b) Minimum Set of Functionality the OS Must Provide

Parallel Execution Model

For SAM to be implemented, as an OS module, or as an user-space library, the OS must provide with at least a model of parallel execution with communication between the different parallel execution units. These can be:

- **Shared memory (Threads):** In this model the concurrent components share the memory address space. The communication is done by altering the components of the shared memory locations and requires some kind of locking mechanism for coordination.

- **Message passing (Processes):** In this model the concurrent components do not have a shared memory address space, so they have to communicate sending messages to each other (sockets, pipes, Transactional Memory...). Reasoning with message-passing model is usually easier than with memory shared models, which leads to less errors while programming.

Both models can be used for implementing an speculative execution service, although the programming implications of using threads or processes parallelisation affect the design and implementations of the programs. At the same time they have different performance characteristics: the memory and task switching overhead is lower in a message passing system, but the overhead of message passing itself is greater than for a procedure call. These differences are often overwhelmed by other performance factors [22].

Minimum Functionality

For enabling speculative execution, the parallel programming language must provide a minimum set of functionalities, independently of which of both, shared memory or message passing models is used. At least this operations are needed:

- **Creation:** A way to create the speculative threads/processes and set up their initial parameters and environment.
- **Join:** A speculative thread/process must wait for the sequential execution controller to accept its memory changes and merge them before finishing and destroying its private environment.
- **Communication:** Some mean for communicating data between the running processes/threads is needed, to propagate the new calculated values that were not set up at the moment of creating the process/thread. Although this is not an strictly necessary functionality, almost all programs will have to share data (if they were not sharing data then there would not be data dependencies, and speculative execution would not be needed).

VIRTUAL FILE SYSTEM

A Virtual File System (VFS) is an abstraction layer on top of the different file systems loaded on the running system. It allows user applications to access the storage devices in a uniform way, at the same time it allows to simulate storage devices (like RAM-disk) and to modify a file many times and only write it once in the storage device, after confirmation the data is not going to be further modified.

A VFS is needed for the OS to provide speculative writing to files (as the file read/write operations might be part of the speculative execution). For this it would be needed to take snapshots of the files, copy the "virtual file" and the file descriptors in a process/thread should point to the different versions of the "virtual files" accordingly. When speculative writing is accepted, then finally it should be written to disk (or the non-speculative VFS).

It is not impossible but it is difficult to make other kind of I/O devices support speculative operations. It depends on the possibility of making snapshots, rolling back and to reproduce the result of I/O operations.

9.C) OPTIONAL EXTENSION

Previously were described the set of functionalities needed for SAM to be implemented but we should also consider other OS functionalities that not being strictly necessary would simplify the complexity of the implementation as well as improving the performance. It must be also clarified that some of them are needed, but that can be implemented using the minimal OS functionalities when not directly provided by the OS.

PARALLEL EXECUTION MODEL

Optional functionality for the parallel execution model:

- **Cancel:** Kill a process/thread without accepting any changes, and without being restarted automatically;
- **Pause:** Suspend the execution of a process/thread till a resume command is sent;
- **Resume:** Continue the execution of a process/thread that has been previously paused;
- **Restart:** Start the execution of the process/thread from the beginning (equivalent to cancel and create with the same initial parameters);
- **Checkpoint creation:** Create a copy of the status of the execution of the process/thread (parameters, private environment...) to be able to roll back in case a data dependency violation is detected;
- **Roll back:** Roll back a process/thread to a previously created checkpoint when there was a data dependency violation;
- **Priority:** A way to prioritize some process/thread over others, giving the speculative segments priorities according to the logical sequential execution reduces the data dependency violations and the roll backs;
- **Execution synchronisation means:** like mutexes/semaphores/... in shared memory model (for synchronizing and coordinating the access to shared resources or block a process/thread when it is clear a data dependency will happen), or blocking messages/barriers in the message passing model.

VIRTUAL MEMORY

Optional Virtual Memory management functionalities, depending on the parallel execution model:

- For a shared memory model:
    - Ability to create partial private thread environments, this means to stop sharing a memory region (memory page) so that the modifications by each thread do not interfere with the others
    - Memory page versioning: make snapshot (copy) of a set of memory addresses (or pages), to be able to roll back
    - Merge memory pages: Speculative threads work on their private environment, when their work is accepted their environment must be merged to the sequential non-speculative execution, joining memory regions with same virtual addresses that are mapped to different hardware addresses
- For a message passing model:
    - **Memory page exchange**: to speed up sending memory regions by message passing
    - **Memory page versioning**: make snapshot (copy) of a set of memory addresses (or pages), to be able to roll back
    - **Variable Merging**: Merge variables that represent the same data element in different speculative environments, but that might be mapped to different virtual memory addresses.

SPECULATIVE FILE SYSTEM

Extra functionalities for the Speculative VFS:

- Partial "virtual" copy of a file segment: big files might cause to overuse the system main memory as many full copies of this file might be in memory (one for each speculative process/thread), so the VFS should divide the files into regions – pages – (similar to the system memory), and only keep copies of the chunks of the file that are being modified
- For speculative operation on distributed file systems the OS should be modified to support communication of changes to other instances of the OS accessing the same files.

9.d) Automatic SAM

In many cases we will want to execute speculatively an application, but the programmer did not use any annotations or library/system calls. Then we can enable automatic speculative execution to extract the possible parallelism. This speculative parallelism can be obtained mainly from:

- **Code Segmentation:** Often many segments extracted by the code analyser can be executed speculatively;
- **Loops:** Start the execution of the next iteration of the loop when the current one has not finish executing. Works better in loops with regular structures, and long execution time. Usually there are nested loops, so the speculation can be done in different levels, inner and outer loops;
- **Branches:** Start executing different branches simultaneously till it is clear which is the correct one.

Profiling and complex estimations will help to improve the results of the speculative execution scheduling/partitioning performed by the static analyser [20][21].

9.e) Code Development

Depending on the SAM implementation and the OS services available, there are different possibilities on how the programmer, user and/or the system itself, can speculatively execute the code. This means if it should be enabled or not, which segments of code should execute speculatively and to which degree.

There are different programming models that can be used in the implementation depending on what the OS provides and what the programmer wants to achieve:

- **Annotations:** The programmer can write annotations for expanding code, giving indications about the behaviour of the program that can not be guess by static analysis. This can help – through the compiler – the code segmentation analyser, the scheduler and the OS in general, to decide which parts to execute speculatively under specific circumstances.
- **Library Functions calls:** Direct calls to the speculative library, the programmer decide when, how and what to execute speculatively, not giving the OS to decide on how the speculative execution should behave.
- **OS System calls:** The application requests a service from an operating system's kernel that it does not normally have permission to run. This is similar to library function calls, but in this case SAM is implemented in-kernel, and not as a user-space system library.
- **Automatic Speculation:** The system decides automatically to execute speculatively some parts of the application as described in Section V. 9.d) .

Nevertheless, in all cases, the OS can decide to not enable speculative execution, or start the number of speculative threads that it considers suitable according to the resources available and the other applications that are being executed.

9.f) Interaction with other OS modules

SAM has effect on other modules:

- **Scheduler:** In general, scheduling affects the performance as it controls the use of the computing resources, having a direct impact on the throughput and latency of the applications [19]. In particular, a good speculative scheduling algorithm can result in significant improvements in performance as compared to scheduling without taking in account the speculative execution because it should give higher priority to the non-speculative parts and applications. At the same time the speculative segments should be prioritised according to their possible data dependencies, this means that if a segment A can affect segment B, then segment A should have a higher scheduling priority than B, minimizing the number of data dependency violations.

- **Transactional Memory:** The shared memory implementation of SAM could be build on top of TM, using TM as the mechanism for controlling the concurrency, obtaining all the benefits delivered by TM.
- **Code analyser & Segmentation:** How the code analyser decides to segment the code influences the speculative execution. When the speculative execution is realised without the support of the programmer it will rely on the segmentation done by the code analyser, executing speculatively (some, all or none) of the code segments.

9.G) SAM HARDWARE SUPPORT ASSISTED BY THE OS

The support for hardware-level speculative execution can be added to current processor, with some modifications that affect mainly the MMU (Memory Management Unit), the caches behaviour and the cache coherency system [23]. This support consists in adding to the hardware the ability to keep speculative memory state separate from non-speculative state, a mechanism for committing speculative state under the control of the OS, a method of detecting data hazards (RAW – Read After Write dependencies) and flushing the speculative state, and a mechanism for notifying the OS when a hazard is detected so that recovery can take place.

VI. OS ARCHITECTURE

Whilst the individual modules / components and algorithms address the specific requirements posed towards a distributed execution framework such as envisaged by S(o)OS (see Chapter III. and [52]), it does not implicitly form an operating system as a whole. As has already been discussed in detail in chapter IV. , there is no concrete deployment architecture of the operating system, due to its modular and service-based approach, which principally allows that any module may be deployed any number of times on different resources in the infrastructure.

However, as has also been discussed, in order to actually enable distributed functioning of the OS as a whole, certain capabilities need to be available at specific times in order to provide the (implicitly) needed functionality. It is for example insufficient to simply host the memory management related components onto each resource, assuming that this would enable a distributed memory system. Instead, for the actual execution, communication support needs to be provided, wherever the request could / should leave e.g. the processor boundaries (assuming that cache is handled by the hardware). Accordingly, we can distinguish between:

1. **Essential modules**: include any components or modules that need to be on a resource (during a specific phase) in order to make that resource available and usable for the operating system in the first instance. For example, any resource needs to be identifiable and reachable. Note that this does not imply that ALL resources have to host the specific capabilities, depending on the hardware setup (e.g. cores in a multi-core processor are generally visible without additional software support, and nodes only need to host external interconnect support once etc.).
2. **Dependent modules**: are essential for a specific module to actually enable the respective functionality. Besides for general invocation dependencies, in this case "dependency" means that the respective module needs to be co-located in the hosting environment and cannot be deployed in another location. One typical example is memory management, which is required by almost all execution engines, but not e.g. for resource discovery, unless under specific circumstances. Since essential modules are implicitly dependent modules, these will not be repeated in this context.
3. **Floating modules**: are independent from any other module in terms of location, i.e. they can be placed on any resource without requiring other instances (with the exception of essential modules).

In this classification, we can regard essential modules as the underlying "framework", and dependent and floating modules as components acting "on top" of the according functionality. This classification determines the segmentation and distribution restrictions of S(o)OS and implicitly form the base deployment model.

1. OS FUNCTIONALITY SETS

In line with the three different module base types, the operating system can be regarded as "layered" in the sense of building up on basic capabilities. These layers do not imply obstructing of access to lower-level functionalities, but simply denote the dependencies as noted in the previous section. In other words, "lower" layers are more essential than "higher" ones, and hence that "higher" ones need to get access in some way to the "lower" capabilities (cf. Figure 25).

With this distinction we can classify the lower layer as "essential" modules and the layer on top of it as formed by "dependent" and "floating" modules. Components on an even higher level can be regarded as "tools" that extend the OS capabilities with respect to the goals pursued by S(o)OS, but are not essential for the functioning of an OS themselves. It must be stressed here again that all modules layer can access any component directly non-regarding the layer they are in. However, the "base" layers can also be abstracted away, thus becoming invisible to any higher-level layers.

Effectively, applications can run on the operating system in principally two modes: integrated and on-top. Integrated applications directly invoke the operating system capabilities to control and steer the behaviour on low level. Accordingly, integrated applications can be extended OS modules, drivers and highly-efficient and adapted applications. Since integrated applications are closer to the hardware they need to be specialised and thus lose their portability. Also they are generally more difficult to develop as the OS support needs to be explicitly identified and incorporated into the code, and hence is generally more error-prone.

For the average user, on-top application development is therefore recommendable. In this case, the application is unaware of the OS support and does not explicitly invoke the according functionalities. In other words, functions such as memory management are not explicitly invoked in the code, but implicitly used according to code behaviour. This implies automatic monitoring, segmentation and distribution. Notably, due to the flat hierarchy of S(o)OS, the application can mix the two models using according programming annotations.

Note that we avoid the terminology "user space" and "kernel space" here on purpose, as the connotation of either principle implies aspects that are not in line with the principally flat modular structure of S(o)OS.

1.A) BASE OS CAPABILITY SETS

As can be seen from Figure 25, the level of essential modules is particularly formed by the general messaging related support of the operating system. As discussed, this is simply due to the fact that the OS is organised in a modular fashion that allows full distribution, but implicitly requires some means of communication across modules. Accordingly, communication modules are essential at any system boundary point.

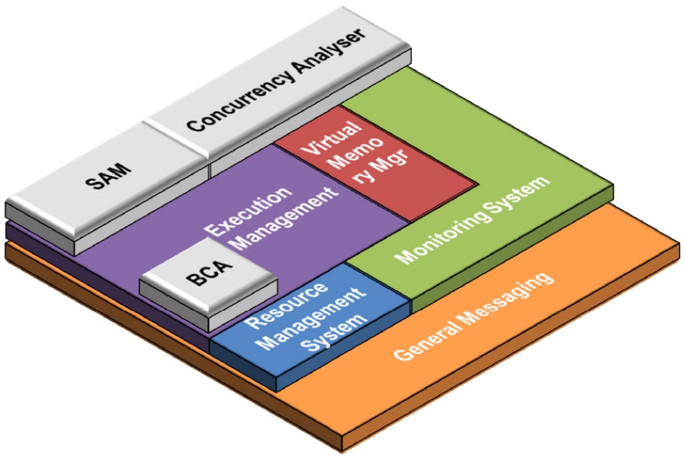


Figure 25: Basic dependency architecture model

GENERAL MESSAGING

The messaging related components comprise all capabilities necessary to establish communication interconnects across any two resources. Depending on the code / OS relationship, this can cover all levels from inter-core communication over inter-processor, across nodes, racks, the internet etc.

Implicitly, messaging must cater for different forms and protocols, depending on communication level and system architecture. This ranges from shared memory over DMA to any network protocol.

As has been discussed multiple times, communication can be either explicit, i.e. directly invoked by the application, or implicit, in which case the OS also needs to detect what kind of communication needs to be enacted, i.e. whether data needs to be retrieved, buffers need to be managed, streams handled, or even whole code blocks need to be distributed etc.

Components

Messaging support is comprised by the general *Connection Manager* and the supporting modules for the individual protocols, i.e. *In-Chip Messaging*, *Inter-Processor Messaging*, *Networking:*

- **Connection Manager**: establishes and maintains the connection
- **In-Chip Messaging**: enacts data exchange between units in a chip
- **Inter-Processor Messaging**: ensures messaging across processors
- **Networking**: can enact higher communication protocols

Deployment Indicators

As the connection manager identifies the right protocols and endpoints, it forms essential components in the deployment. Whenever distributed functionalities are involved, i.e. wherever the OS execution is not restricted to a single module in a single system, connection management is essential for the respective modules. In order to actually enact the relevant protocols, the OS configuration together with dependency data such as from concurrency analysis has to identify the right communication modules and deploy them *with* the connection manager.

OS MANAGEMENT

An additional functionality group, which does not appear in the figure though, relates to configuring and managing the OS instances themselves. This component takes a hierarchically higher stance than all other components and is relevant primarily to the main OS instance, thus not falling into any other category here.

Components

In order to manage the OS environment's setup and configuration, at least a management and a configuration instance are needed. Functionality-wise, they can be regarded as two different instances of the same component, i.e. global and the local controllers.

- OS Configuration Manager

Deployment Indicators

Configuration management takes two different positions: configuration of the OS as a whole and therefore management of the distribution and deployment of modules, as well as their local configuration. The local management position acts thereby mostly as an interface to requests from the global manager, but can take local decisions in a hierarchical fashion, too (cf. chapter IV.). Accordingly, at least one global management instance is required, that can take over full control over the infrastructure – however, in particular in hierarchical setups it is advisable to add further instances to reduce control overhead.

1.B) MAIN OS CAPABILITY SETS

The actual OS capabilities are formed by the groups that provide the essential functionalities to enable distributed execution across the infrastructure. Even though messaging effectively belongs to this category too, it is exploited equally by the functionality sets in order to give the appearance of a single environment to the application (see discussion at the beginning of this chapter).

To these sets belong the following functionality groups:

VIRTUAL MEMORY MANAGEMENT

Similar to messaging, virtual memory management takes a core position in OS execution, supporting all data access across the distributed infrastructure. It thereby builds completely upon the messaging capabilities, as was exemplified in section IV. 2.b) - at least, whenever memory or data across the infrastructure needs to be accessed. (Virtual) Memory thereby needs to cater for all different types of storage access, ranging from local cache management, over RAM and local hard-drive to internet streams etc. From usage point of view, storage effectively needs only to be distinguished in terms of size, speed and persistence.

Memory management therefore provides all necessary means to hide state and data management details from the application, so that the developer does not explicitly has to cater for location, consistency etc. unless explicitly requested.

Components

As virtual memory management integrates various types of storage, the set of components needs to provide all according means necessary for accessing the respective storage. Depending on the location of the storage, this may mean additional communication means, which need to be covered by the general messaging set.

- **Virtual Memory Manager**: acts as the configuration and management interface that provides process-specific virtual address spaces and intercepts / handles data access requests.
- **Cache Manager**: maintains the local cache – this includes principally strategies to keep the cache alive under certain circumstances, as well as providing consistency management etc.
- **Source Lookup**: maps the virtual address to the actual physical memory, respectively to the resource in the infrastructure
- **Local Memory Manager**: handles all local storage resources and their access information similar to the classical OS memory manager.
- **File System Manager**: manages the hard-drive organisation
- **File Access Manager**: exposes file access means, including streams and virtual storage.

Deployment Indicators

Virtual memory is essential for any application or process that handles data and does not want to run the risk of overwriting other process' memory space – this is particularly relevant for distributed execution cases. Memory management typically has a hierarchically higher task than the actual memory access and thus can be deployed in different fashions, according to the distribution of the environment in the first instance, i.e. hierarchical, central, distributed etc. As opposed to that, actual memory access and management components are required locally according to the type of memory access and infrastructure.

EXECUTION MANAGEMENT

The actual code layout and the implicit (and explicit) relationships in terms of state, data and communication are handled by execution management related functionalities. This set of components is responsible for ensuring that the distributed execution works correctly taking the additional information about code behaviour and restriction into consideration. This implies aspects such as scheduling, code migration etc. Execution management therefore interacts strongly with resource management and monitoring system in order to supervise the status and load balance execution. It can be extended with additional functionalities through the sets of “tools” which provide dependency information etc., respectively which support program profiling for better distribution and load balancing.

Components

Execution support forms the major capabilities of an operating system – since performant, large-scale execution is of main relevance for S(o)OS, this group circumscribes most of the relevant capabilities for S(o)OS. More specifically, this includes:

- **Dependency Manager**: maintains the dependencies between different code segments, i.e. in particular what data is accessed from where & when etc.
- **State Manager**: keeps state across segments in a consistent state.
- **Load Balancer**: is responsible for distributing the workload evenly across the infrastructure, so as to maximise resource utilisation for execution performance
- **Code Migrator**: can move code from one resource to another thereby calling support such as the code adapter to transform the code between different resource types
- **Code Adapter**: transforms the code between different resource (and potentially ISA) types. Supported by e.g. the binary code adapter
- **Transactional Memory**: supports consistency across resources, relates to state management
- **Scheduler**: manages the timing of the individual processes in the system.

Deployment Indicators

In order to be able to execute the code segments in the individual systems, at least part of execution support is required – in particular from the hierarchically higher management perspective that is responsible for the distribution and execution of the application *as a whole*, non-regarding its segmentation and distribution across the infrastructure. Higher level components, such as dependency management, load balancing and scheduling should generally be deployed at least once in the system at a higher hierarchical level than average code execution. Accordingly, Code Migration and Code Adaptation are generally only required once in the whole system, though, as opposed to dependency management and scheduling, they need not necessarily be located at a central instance, as long as they are reachable by the latter. Due to their nature, transactional memory and state manager should principally be located closer to the code.

RESOURCE MANAGEMENT SYSTEM

In order to enact and manage a distributed system, the individual available resources need to be known by the operating system. Again, this information can also be principally employed the application, too, if resource information is required, or specific resources are required. Here, resource management is passive and primarily serves the purpose of identifying resources – and principally configuring them. Information related to their current state can be accessed via resource management, but is actually provided by the monitoring subsystem.

Components

This set covers two major functionalities: resource management and information provisioning:

- **Resource Manager**: is responsible for handling resources for the distributed execution, this includes configuring it, getting endpoint information etc.;
- **Resource Discovery**: discovers specific resources according to a query (regarding type, amount, characteristics etc.);
- **Resource Information Provider**: hosts and provides the resource description in a fashion that specific queries can be run over it;
- **Resource Description**: is an object rather than a functionality as such, though the way of implementation grants extensive tests (see section V. 3.);
- **Resource Reservation**: ensures that the resources required are available for the processes upon need.

Deployment Indicators

Resource management in itself, i.e. reservation and configuration etc. belong to the general management activities of an operating system and are mandatory for handling scaled, heterogeneous environments. The according capabilities should thus be placed at points central enough to manage the environment. As opposed to that, resource discovery can be initiated from any point, whilst resource information provisioning should allow unique identification of the resource with queries from multiple different instances and should therefore ideally be co-located with the resource itself.

MONITORING SYSTEM

Accordingly, monitoring related components provide live status information about specific resources and are responsible for generating profiles out of this behaviour. In combination with the execution management system and additional “tools”, this profiling information can then be used to improve application performance. Typically, monitoring systems also serve as the quality of service enactment points and can take immediate actions (such as informing the execution management system) under specific conditions

Components

The monitoring related functionalities are comprised not only of the actual system monitor itself, but also the related analysis tools that help to evaluate and profile the performance, as well as to give execution indicators under specific circumstances (“predictions”):

- **Performance Predictor**: assesses the potential behaviour of a given resource under a specific algorithm type – this can include profile data, as well as abstracted information.
- **In-Depth Profiler**: profiles the resource / application at execution time. The depth and detail of profiling may vary with different use cases and applications.
- **QoS Controller**: evaluates the system behaviour and takes immediate actions – may act as the interface for load balancers to identify workload.
- **System Monitor**: monitors the actual resource behaviour, e.g. WMI or NAGIOS.

Deployment Indicators

Monitoring is closely related to the resources it is executed on and thus generally has to be deployed on the same system(s) – similarly, profiling and controlling should be co-located nearby to reduce messaging. However, similar concerns for hierarchy and detail of messaging applies than in most other capability sets, so that a more integrative view with less amount of information could be deployed further away.

1.c) OS Supporting Capability Sets

Finally, the highest-level of components or in this case “tools” relate to additional functionalities that extend the capabilities of S(o)OS without actually being required by the operating system as such. Generally, these tools support the analysis of an application with respect to its dependencies and hardware requirements and restrictions. In other words, capabilities that may either be directly provided in the source code itself, i.e. through additional programming efforts, or may be reused from previous execution runs (cf. Section II. 5.). While these capabilities extend usability and programmability, they are not essential for the scalability and adaptability of the OS.

CONCURRENCY ANALYSER

The concurrency analysis related functions (also referred to as “code analysis and segmentation”) exploit the performance behaviour information gathered at runtime, in addition to static analysis mechanisms. The according information is used to identify the state and data dependencies, as well as potential means to adapt the code to the hardware for better performance. Accordingly, the concurrency analyser builds strongly on the monitoring and resource identification capabilities.

Components

Even though concurrency analysis per se is essentially an integrated functionality set, it nonetheless consists of individual components, that could in principal be distributed across the infrastructure. More importantly, the differentiation allows for different levels on the hierarchical distribution:

- **Code Segmenter**: executes the actual segmentation of code according to the hints provided by the graph segmenter and on basis of the infrastructure information. This includes injecting commands for state maintenance, communication etc.
- **Graph Segmenter**: identifies segments of the dependency graph, so as to reduce the communication overhead and / or other criteria.
- **Graph Generator**: generates a dependency graph out of the online and offline code behaviour analysis.
- **Code Analyser**: analyses the dependencies in a given code in an online or offline fashion.

Deployment Indicators

In general, the concurrency analyser is a closed tool-set that acts directly on the (full) code, as the analysis and segmentation requires the full information set. However, as every code segment can be regarded and treated as a stand-alone process with dependencies to outside resources, the analysis can also be executed on individual segments. This is particularly sensible, if no detailed information is as yet available for a given code segment, but execution efficiency could already be improved by segmentation and distribution.

BINARY CODE ADAPTATION

Code adaptation related functionalities try to transfer the code form one destination platform to another. Whilst the developed modules particularly aim at binary transformation (i.e. from one ISA to another), the general functionality of this component set relates to different adaptation processes, depending on context (cf. section II. 5.).

Components

Similar to the concurrency analyser, the code adapter essentially acts as a full component. It can be differentiated into the following main components:

- **DISTAE Adaptation**: analyses the code for the right adaptation mechanisms
- **DISTAE Compiler**: (re)compiles the code for the new destination resource
- **DISTAE Execution**: executes the adaptation process
- **DISTAE Data Standardisation**: ensures data compatibility across platforms

Deployment Indicators

Just like the concurrency analyser, the code adaptation facility group is essentially a single tool set that acts on a single code segment to transfer it to another destination platform. However, it does not need to be centralised in the environment and can be added to the OS module deployment on time of need.

SPECULATIVE CODE EXECUTION

Speculative code execution capabilities actually relate to execution management, as they extend the parallel execution with means to ensure consistency and reliability, and trying to reduce the analysis overhead by allowing speculative parallelisation. Well-parallelised code however does not require these extensions any longer.

Components

Speculative execution is circumscribed by a single component that takes over all roles per execution segment – further subdivision is not necessary, as the rest of the capabilities are provided by execution and memory management:

- **SAM**: monitors a code execution segment / thread / process to detect (and roll back) data consistency violations

Deployment Indicators

SAM provides an extension to the local code execution and should therefore be co-located with the respective segment, otherwise risking additional communication overhead.

2. OS Components Relationships

Figure 25 already indicated how the individual set of components relate to each other in terms of functional dependencies, i.e. which other sets are required by the functionalities of the modules. It can be noted that with exception of the higher- and lower-level component sets, that all other sets in some way relate to each other. It should not be concluded from this, though, that all modules are "dependent" on each other, let alone that they need to be co-hosted in the same location / on the same resource. Instead, the invocation may be routed using the communication framework, depending on the dependency "strength".

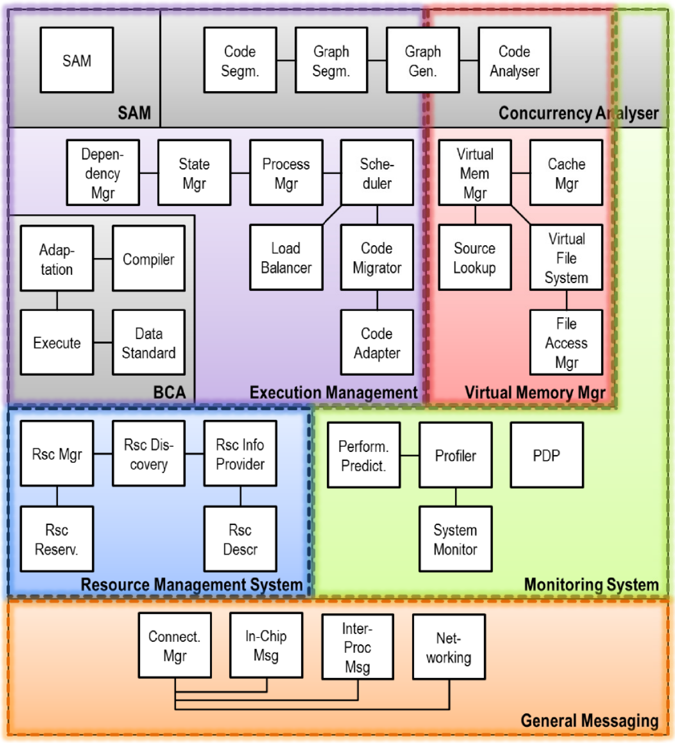


Figure 26: OS dependency architecture

Figure 26 resolves the architecture overview presented in Figure 25 by highlighting the individual functional components per set that contribute to providing the respective capabilities. The figure also depicts the relationships between the individual components in terms of invocations and dependencies, with no regards to the direction of dependency, i.e. which components invokes which. Note that the figure does not depict the exact relationships of components across these functionality sets, but instead only indicates them as the touching boundaries of these sets.

2.A) Summary of Dependencies

As the full dependency depiction would make the figure unreadable, we compile the relationships in form of a table (see Table 3). The table is constructed in terms of “row invokes column”, i.e. any module from the rows (e.g. “Resource Manager”) with an “X” at a specific column (such as “Load Balancer”) denotes that the former invokes the latter. An “(x)” in this context means that the specific relationship is still under discussion and may require further clarification in future iterations.

It can be noted from table that no component or set of functionalities stands isolated from the whole OS, in which case the respective functionality would not be integrated, or even not required.

		OS Level Services																											Higher Level Services									
		OS Mgt	Virtual Memory / Storage System						General Messaging & I/O				Resource Management System					Execution Management							Monitoring System				Concurrency Analyser				Binary Code Adaptation				Spec. Exe	
	(row calls column)	Configuration Manager	Virtual Memory Manager	Cache Manager	Source Lookup	Local Memory Manager	File System Manager	File Access Manager	Connectivity Manager	In-Chip Messaging	Inter-Processor Messaging	Networking	Resource Manager	Resource Discovery	Resource Information	Resource Description	Resource Reservation	Dependency Manager	State Manager	Load Balancer	Code Migrator	Code Adapter	Transactional Memory	Scheduler	Performance Predictor	In-Depth Profiler	QoS Controller	System Monitor	Code Segmenter	Graph Segmenter	Graph Generator	Code Analyser	DISTAE Adaptation	DISTAE Compiler	DISTAE Execution	DISTAE Data Standarization	SAM	
OS Mgt	Configuration Manager		X						X				X					X										X	X				X				X	
Virtual Memory / Storage System	Virtual Memory Manager			X	X	X	X		X	X	X	X																			X							
	Cache Manager								X	X	X	X																										
	Source Lookup								X	X	X	X																										
	Local Memory Manager								X	X	X	X																										
	File System Manager							X	X	X	X	X																										
	File Access Manager								X	X	X	X																										
General Messaging & I/O	Connectivity Manager		X	X	X	X	X	X		X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
	In-Chip Messaging		X	X	X	X	X	X					X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
	Inter-Processor Messaging		X	X	X	X	X	X					X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
	Networking		X	X	X	X	X	X					X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
Resource Management System	Resource Manager								X	X	X	X		X			X			X									X		(x)							
	Resource Discovery								X	X	X	X			X																							
	Resource Information Provider		X						X	X	X	X				X											X											
	Resource Description								X	X	X	X														(x)		(x)										
	Resource Reservation								X	X	X	X																										
Execution Management	Dependency Manager								X	X	X	X												X							X							
	State Manager		X						X	X	X	X						X								X						X						
	Load Balancer								X	X	X	X	X			(x)					X					X		(x)										
	Code Migrator								X	X	X	X	X									X											X					
	Code Adapter								X	X	X	X	X		(x)																							
	Transactional Memory								X	X	X	X	X											X													(x)	
	Scheduler								X	X	X	X	X					X		X	X		X		(x)			X							x?		X	
Monitoring System	Performance Predictor								X	X	X	X														X												
	In-Depth Profiler								X	X	X	X																X			X	X						
	QoS Controller								X	X	X	X	X	(x)						X	X					X		X	(x)									
	System Monitor								X	X	X	X																			(x)							
Concurrency Analyser	Code Segmenter								X	X	X	X	X									X	(x)							X								
	Graph Segmenter								X	X	X	X		X											X					X								
	Graph Generator								X	X	X	X						X								X												
	Code Analyser								X	X	X	X						X								X					X		X	X				
Binary Code Adaptation	DISTAE Adaptation								X	X	X	X																						X				
	DISTAE Compiler								X	X	X	X																										
	DISTAE Execution								X	X	X	X									X												X			X		
	DISTAE Data Standarization								X	X	X	X																										
Spec. Exe	SAM		X						X	X	X	X											X	X														
									X	X	X	X																										x

Table 3: Dependencies overview over the components

VII. Application Programming Interface

In this section, we summarise preliminary considerations and thoughts about issues around the interface between S(o)OS and the applications. The purpose of this chapter is not to provide an interface specification, but rather to describe the key principles that we identified at the basis a concrete API design. Specifically, we focus on how to properly support at the API level (or generally at the application interfacing/interaction level) the particular critical S(o)OS components and capabilities that have been analysed and described in the preceding chapters. Indeed, in order to improve and in particularly steer the behaviour of the S(o)OS components, the user must be enabled to provide control “annotations” to the code, that the compiler, or more correctly the execution framework can exploit. This also enables experienced developers to exploit the S(o)OS features for specific capabilities, such as branching code for specific hardware types, or pre-segmenting the code for specific scaling behaviour.

Within this chapter we discuss the general means to expose application behaviour control mechanisms to the developer in the context of specific, S(o)OS-related aspects, namely parallelism, real-time behaviour and transactional memory.

1. Parallelism and Concurrency

As has been thoroughly discussed in the context of S(o)OS deliverable D5.1 “State of the Art” [33], there is currently no appropriate programming model that allows for specifying concurrency in the program easily. There exist attempts such as StarSS that effectively tries to specify function interdependencies in the form of annotations that declare the input and output relationship with other functions. Though this is a promising means to identify and handle concurrency on this level, it does not help in actually developing highly concurrent and thus parallelisable code, but only provides an additional means of conveying the developers' intentions with this respect.

Fundamental changes, such as moving to the functional paradigm, have proven even less successful over the years, non-regarding their potential for concurrency analysis (see also [33]). However, over recent years principles of the functional programming paradigm were (partially) introduced into standard object-oriented programming languages (such as C#) to enrich their capabilities. As part of this movement it could be noticed that simple extensions to the programming model, such as replacing the `for int` loop with the `foreach`, which implicitly conveys aspects of concurrency in a form that can be easily taken up and understood by the developers. On the other hand, such extensions are only useful, if the compiler (or execution environment) can make appropriate use of them.

In the context of S(o)OS it is of primary interest which extensions are needed in order to supplement the relevant information to steer the execution framework to take correct decisions upon data dependencies, and implicitly upon concurrency and parallelism. For the first cycle, this relates in particular to the code analysis and segmentation set of components, which are trying to identify the dependencies according to the code behaviour during execution:

1.a) Interpreting Data Specific Behaviour

As has been noted multiple times, the major concern for appropriate analysis consists thereby in the data(in)dependent part of the code behaviour, which the analyser cannot assess appropriately. This leads to potential misinterpretations in terms of strengths of dependencies, and even to missed dependencies in the case of context-specific code branches. To improve the analyser's knowledge about data-specific behaviour, we can identify – amongst others – the following additional information that help in interpreting the scope of impact:

- **Ranges and sizes**: variables typically are only used within certain perimeters given a context – however, these perimeters cannot be known by the compiler or the execution environment, as they may vary depending on use. Even though most programming languages offer a wide variety of different types to allow more fine-grained distinctions, these are hardly ever used to this purpose due to the potential impact of wrong assignments. However, the developer can typically specify what perimeters variables, arrays and memory blocks *typically* take, thus allowing the environment to assess the general behaviour e.g. of loops, to identify the memory size to be generally reserved etc.
- **Constant and dynamic property scope**: Closely related to ranges and sizes, the scope of validity of a variable is difficult to assess – even though the analysis will identify that the respective memory space is not overwritten during execution, this does not necessary imply that a code branch would not lead to this behaviour. Knowing the scope, the analyser can rearrange local variables to replace them with constants within the respective segment, or vice versa, identify the domains in which variables should not be treated as constants, non-regarding their behaviour during first iterations (e.g. initialisation runs).
- **Complex prefetching and synchronisation**: As a consequence of scope and perimeter of parameters arises the issue of when and how to communicate data. The point of the analysis consists in identifying the hard communication points ("deadlines") and thus implicitly the latest point of communication. Through overlaps, the earliest point can be restricted, but not clearly identified. Along with the information on perimeters, clustering communication can be attempted to reduce communication overhead. However, in particular in loops and with dynamic ranges it becomes difficult to identify communication points and scope. Through additional annotations, such as "prefetch vars x,y,z from here", the analysis can be improved considerably.

1.B) IDENTIFYING CODE-DATA-RELATIONSHIPS

As the analyser's primary goal nonetheless consists in identifying the dependencies, we must at least consider how to improve and steer this behaviour – in particular where the developer for practical reasons wants the analyser to behave optimistic or pessimistic. In other words, the analyser may take indications for dependencies as a reason to trigger segmentation even after only few occurrences (optimistic), or delay it due to incomplete information (pessimistic).

In addition, whilst the few programming models that allow for concurrency exploitation focus on providing means for the developer to specify dependencies, it is as important for the compiler / analysis to identify clear NON-dependencies, which may however not always be obvious. Therefore:

- **Interpretation controllers**: The analyser's and segmenter's behaviour are steered by the purpose of reducing the communication overhead and need for cache swaps. To this end, they watch the code behaviour at runtime ("online analysis") or statically analysis the code itself ("offline analysis") - both information must be incomplete with respect to the data specific behaviour issues (see above). Performance can be improved by identifying e.g. when no segmentation should take place (manipulating the dependency graph weights), or to hint out what type of dependencies to look for / prefer (manipulating the interpretation of the dependency graph weights). For both purposes, the dependency graph should be exposed to the user, too.
- **Pointing out non-dependencies**: The level of indirection possible with modern languages makes it almost impossible for an analyser to fully identify relationships – specifically "pointers" either need to be fully simulated for offline analysis or be explicitly exempted from data specific

behaviour to be meaningful in the online analysis. By restricting the dependencies, but in particular the non-dependencies (effectively things such as the perimeter of pointers, the depth of arrays etc.), the analysis can be sped up. This is related to specifying the “analysis depth” for dependency identification and segmentation runs.

- **Explicit segmenting**: Related to the scope and non-dependencies, the developer may also explicitly point out which part of the code belongs intrinsically together, therefore avoiding that the segmenter tries to break it apart. This is particularly sensible in the context of resource restrictions as a type of scope delimiter (see below).

1.c) Resource Type Helpers

One of the most problematic tasks for analysis, segmentation and in particular adaptation consists in identifying the most appropriate resource type for hosting the respective code segment. Again, aspects of data specific behaviour play an important role in this context, but also the user can (and may want) to restrict or indicate the best types of resource properties for a specific code. This way, the resource queries during identification can be directly related to the identified and specified properties, reducing the risk of false identifications.

- **Resource properties specification**: As discussed in detail in the sections on resource descriptions and resource identification, S(o)OS can trigger queries for explicit resource properties, such as cache lines. Implicitly, we can also create queries for properties that are inherent to specific resource types, such as vector support. These queries should also be exposed to the user to allow the developer to specify resource type hints given the specific type of code, e.g. if he explicitly wrote vectorizable code, or requires streaming properties. In combination with scope information (i.e. which segment should adhere to the resource properties), this can also be exploited for layouting and segmentation purposes.

Note that S(o)OS will take user directives generally as “hints” and “indications”, rather than as strict constraints, unless explicitly specified. For example, the analyser may segment code, even if the developer did not intend it, if all indicators point to performance benefits. This way, operation of the code is also granted if the indications are wrong (see e.g. section V. 9.).

2. Real-time Programming

Traditional API for real-time programming on embedded platforms are centred around the fundamental concept that the parallelism structure of the application is known to the developer and enforced in the code. Therefore, these APIs allow applications to communicate to the OS what are the timing constraints of the various threads of execution (e.g., priorities, local individual deadlines, end-to-end deadlines, inter-arrival times, periodicity, etc.).

For example, the APIs designed in the context of the EU projects FIRST[7] [34], OCERA[8] [35][36] and FRESCOR[9] [37] address a wide range of both hard and soft real-time applications in the context of embedded (but unfortunately mainly single-processor) systems, and sometimes they address distributed systems as well.

However, when dealing with real-time programming on massively parallel hardware, a number of challenging issues show up. The first one is how to identify a clear deployment layout for the application over the heterogeneous and massively parallel computing units, that may be appropriate for respecting the in-place timing constraints. In such a context, the number of threads to be created may depend on whether or not the underlying computing unit can perform vector operations and on

[7] Flexible Integrated Real-time Scheduling Technologies. EU Grant IST-2001-34140.

[8] Open Components for Embedded Real-time Applications. EU Grant IST-2001-35102. More information at: http://www.ocera.org/.

[9] Framework for Real-time Embedded Systems based on ContRacts. EU Grant FP6/2005/IST/5-034026. More information at: http://www.frescor.org.

the size of the handled vector types, on whether or not there is availability of GPU and GP-GPU hardware, etc.

A clear answer to this question calls in the loop various requirements which need to be reflected in the communication model and API between the application and the OS/kernel:

- it should be possible to profile the execution of code segments, in such a way that the collected profiling data can be exploited to predict the performance of the various deployment options (a similar functionality has been designed as the QoS DB component of the FRESCOR QoS Management infrastructure [38], but with a task-level granularity, rather than a code-segment level one, and without considering heterogeneous multi-core platforms);
- whenever possible, the OS should try to use simple common models for predicting the performance of a parallelisable code segment when deployed on a variable number of computing units; such a model may either be explicitly provided by the developer, or sometimes can be inferred by the gathered profiling data;
- it should be possible to reserve at run-time some of the available physical resources to one or more applications, so as to limit undesired interferences among independent and unrelated applications;
- it should be possible to trigger corrective actions at run-time, so that the deployment logic can dynamically adapt its decisions based on the feedback provided on-line by the application.

Concerning the last point, the adaptive scheduling and adaptive reservations [39] which have been largely investigated in the real-time scheduling domain addressed mainly the adaptation at the level of the scheduling parameters for a single task for uni-processor systems, but this needs to be extended to consider the parallelism degree on multi-processor and many-core systems.

Also, the dynamic scale-out capability of typical cloud computing infrastructures [40] nowadays resembles this kind of adaptation, in that the number of resources occupied by the application is dynamically increased based on the observed response-times obtained for the users requests. However, such mechanisms are designed in the realm of loosely coupled components that can merely be replicated as much as needed, as opposed to the needs of a massively parallel real-time application that needs to provide as a whole a predictable performance (e.g., in terms of end-to-end response-time).

These issues will be investigated further in the second phase of S(o)OS.

3. TRANSACTIONAL MEMORY INTERFACE

Transactions are present in most existing OSes mostly for the sake of modularity and recovery. Such OSes have even introduced new syntactic sugar to address consistency needs [5,6,7]. The Google File System supports atomic `append` operation [5], Windows adopted support for transactions in NTFS and the Windows registry [6], and Apple supports atomic blocks in Grand Central Dispatch for Mac OS X 10.6 and iOS 4 [7].

In fact, transactional language constructs have to be exposed to permit the modular design of future OSes in sub-components. Recall that this modularity is a key requirement mentioned in Section FIXME.

Transactional memory can be used either in traditional development languages, by using particular libraries, or with some support by the programming language and compiler.

3.A) TM COMPREHENSIVE INTERFACE

A transaction is a delimited block of code that is usually represented as a syntactically balanced open-close block.

Several compilers support transaction language constructs. For instance, there exist at least three compilers that compiles transactions in C at the time of writing: the prototype DTMC[10], the TM branch of gcc[11], and the Intel C++ STM compiler (icc)[12].

For the sake of using transactions, the interface requires two transaction delimiters and two memory accesses:

1. **`tx_start()`** is used to delimit the starting delimiter of the transaction. It can be alternatively replaced by the opening of a compound statement.
2. **`tx_commit()`** indicates the end of the transaction in the code. It might also be replaced by a closing brackets indicating the end of the transaction.
3. All read accesses to shared memory locations within a transaction have to be replaced with **`tx_read(addr)`** calls.
4. All write accesses to shared memory locations have to be accessed with **`tx_write(addr, val)`** calls.

This basic interface has been extended in various ways. One of the major extension, called transaction polymorphism, relies on parameterizing the **`tx_start(p)`** with some application semantic hints **`p`** which allows to exploit the inherent concurrency of the application. This concurrency leads to **scalability** on highly contended workloads. Another extension targets **robustness** by supporting coordinated exception handling through the use of explicit **`or`** exploiting failure-atomicity of transactions and redirecting the control flow if an inconsistent state is about to be reached within a transaction.

This instrumentation must be performed from previous step for all functions that are invoked from transactional blocks or other functions invoked from transactional blocks (recursively). Below we describe how the instrumentation is done automatically by compilers.

3.B) TM REDUCED INTERFACE

Various languages already support transactions meaning that TM accesses can be automatically instrumented by the compiler.

In addition to the GNU Compiler Collection and the Intel STM C/C++ production compilers, Glasgow Haskell Compiler provides support for transactions in Haskell. Multiverse offers support for TM in Java and has been used in Scala. Fortress supports transactional language constructs and Pugs compiles and interpret transactions in Perl 6. The .NET Framework 4 supports software transactional memory. Multiple third-party programs allow also to support transactions in other languages, like STMlib for OCaml and Durus for Python.

In C, a transaction is simply delimited using the block **`__tm_atomic{ ... }`** while in
C++ a transaction is delimited by a **`__transaction{ ... }`** block where **`__transaction{}`** (or equivalently **`__transaction[[atomic]]{}`**) indicates the point in the code where the corresponding transaction **`tx_start`** should be called. The closing bracket **`}`** indicates the point in the code where the corresponding transaction **`commit`** should be called. Within this block, memory accesses are automatically instrumented by the compiler to call the transactional **`tx_read`** and **`tx_write`** wrappers.

More precisely, the binary files produced by the compiler call a dedicated TM runtime library through an appropriate application binary interface (ABI) specified. This ABI is used for both C and C++ and has been optimised for the Linux OS and x86 architectures to reduce the overhead of the

[10] The Dresden TM Compiler: http://www.velox-project.eu/software/dtmc
[11] The TM branch of the gcc compiler. http://www.velox-project.eu/software/gcc-tm
[12] The Intel STM C/C++ Compiler. http://software.intel.com/en-us/articles/intel-c-stm-compiler-prototype-edition/

TM calls and to allow fast accesses to thread-specific metadata shared by existing TMs. We have evaluated the overhead induced by overinstrumentation of compilers in [11].

3.c) Contention Manager Interface

Concurrent executions are inherently unpredictable, as mentioned above, and when mixing the transaction paradigm to real-time guarantee, one of the problems is that the developer should have some knowledge about how often a transactional block may need to be re-executed due to conflicts and failure of the commit, so as to properly consider that in the estimate of the execution time of a task.

With locks, the problem lies in estimating the blocking time before being granted the access protected by the lock. However as soon as all necessary are granted, the lock-based execution becomes more predictable and its duration can be evaluated, as no interference can delay the execution further. With transactions, the problem is the unpredictable number of restarts but the contention manager is a useful component to bound the number restarts for allowing a higher priority or more urgent task to commit before others, i.e., an interaction between the scheduler and the TM runtime.

The interface requirements are as follows:

1. **`set_priority.`** This can be set up using a parametrised transaction delimiter or can be done within the body of a transaction to provide dynamic priority.
2. **`set_policy.`** Similar to setting the priority, the policy is necessary to differentiate between the policy of tasks that might predominate over their priorities. It can be set up statically at the transaction starting delimiter or dynamically in the course of the transaction.

VIII. CONCLUSIONS AND FUTURE WORK

In this document, the initial architecture was sketched out for novel class of operating systems suitable for next-generation massively parallel and distributed platforms. The focus was on the main OS components that are being investigated in S(o)OS, and which constitute the critical OS core that is responsible for ensuring the fulfilment of the main requirements by the OS, as elaborated in Chapter III. .

1. GENERAL FUTURE WORK

The architecture description provided in this document is still preliminary and will be refined further in the upcoming deliverables, till the final OS architecture to be provided in the final deliverable D5.4. Specifically, it will be noted that the current document only touches upon the issues of deployment, as well as dynamic component selection for the purpose of readjusting to the destination platform and the infrastructure constraints. As elaborated in section IV. 2. , the service-oriented nature of the operating system contradict with a strict deployment architecture – however, the constraints indicated by the relationship types between modules imply that the modules need to adhere to specific deployment rules which implicitly impact on the scaling behaviour of the operating system. Similarly, the essential processes for steering deployment and component adjustment impact on the principle distribution and thus scalability and performance.

This behaviour also impacts on the programming extensions exposed to the developer / user, which have been of secondary concern in this first iteration, as the individual components have only now been fully specified. The combination of the components as elaborated during this first cycle will provide information about how to expose such programming extensions, whilst the components themselves providing the manageable details – this however will have to be assessed and elaborated through evaluation first in order to specify the application scope and feasibility.

The second cycle will therefore not only reassess the architectural layout, but in particular investigate further into control of the OS and exploitation of the capabilities from the user side.

2. COMPONENTS RELATED FUTURE WORK

The first cycle of integration and component assessment in terms of architecture has furthermore led to the identification of component-specific work that will partially be addressed by the respective technical work packages (WP2-4) and by WP5 in particular where it relates to aspects of integration, OS design and programming extensions.

The following summarises the open tasks identified by WP5 for these individual components.

SPECULATIVE EXECUTION

The first prototype has been developed using POSIX threads, and this provides a cache coherent with shared memory space. In the next steps it will be studied the implementation of speculative execution in non-shared memory address spaces. This will allow to run speculative programs with no cache coherence or even in different nodes. For this implementation it will also be studied the use of Transactional Memory.

BINARY CODE ADAPTATION ("DISTAE")

Currently the design is almost finished, and the development of the first prototype has to continue, having a first functional versions in a few weeks. After it, the number or targeted architectures on which the framework can run will be expanded, although it will depend on a python interpreter and gcc compiler. The second step will be to increase the data type support, because of the transformations and translations between different architectures at the current moment programs can be only be implemented with integers and floats. In the third step we will switch to transfer the executable package with RTL code instead of source code, to accelerate the binary adaptation without losing performance; RTL code is very low level, making it almost impossible to be human-readable. On the last step we will try to expand the programming language support (by now it is just C) by offering the means that will simplify the inclusion of DISTAE support into other languages.

CODE ANALYSIS AND SEGMENTATION

The Code analysis related modules as fleshed out during the first iteration focused in particular on the concepts behind the means for extended profiling, so as to gather dependency and behavioural information. This data will then be used for segmentation of the code according to the environmental conditions.

Whilst the first iteration already indicated the means to execute runtime profiling (or behaviour data gathering), it did not elaborate on this point in detail. Along the same line, the current architecture has a centralist flavour, where the code is analysed and adapted from a single point which executes the full code. Whilst a hierarchical deployment of this model could support this task, it will remain insufficient with respect to distributed maintenance of the dependency information and conflict resolution. Accordingly, the current architecture tends towards supporting offline analysis rather than online analysis, or at least not with the degree of performance and scalability intended.

Next to algorithm verification and development, the second architecture phase will therefore focus on a distributed, run-time deployment version of the with Code Analysis and Segmentation related modules. The according architectural choices will furthermore be investigated with respect to their impact and integration with the other modules.

These tasks conform in particular with the requirements to improve performance in large-scale heterogeneous infrastructures, as in particular the data specific dependency / behaviour information can only be gathered efficiently at run-time. Execution should thereby not be restricted to a single processing instance, as this would seriously affect efficiency during analysis time, which, depending on code and data may last a long time. As a side-effect, information gathered at run-time can also be exploited for run-time adaptation (such as redistribution of code) – this supports not only dynamic environments better, but also allows iterative adjustment of the code execution to specific (new and unknown) platforms.

RESOURCE DISCOVERY

So far we have designed and implemented partial resource discovery and matching in a COTSon simulated cluster which supports scalability for multi-core/CPU systems. In the next step we will deploy RD out of the simulator in a real many-core environment which includes physical and virtual nodes in different scales (scale up and scale out, many CPUs – many cores). After that, we will use COTSon and benchmarking tools for the purpose of experimental results evaluation. In the current architecture, RD uses a topology-aware approach in which each node has information about other nodes in vicinity. And also Anycast messaging based on hierarchical CDHT is used to explore network with homogeneous nodes. In future steps RD will be extended to support dynamic environment with frequent nodes join, leave and failure, therefore to achieve consistency and avoid discovering invalidated services, RD must be fault tolerant. To fulfil the main requirements of RD, in the next phase, we will enhance RD to support heterogeneity of resources, load balancing and quality of services.

To enable efficient resource discovery in dynamic distributed environment, Any cast mechanisms must be able to scatter discovery load to avoid overloading group members in the case of high demand for particular resource services in a certain part of the network.

Currently we have implemented the following functionality:

Requirements	Functionalities
Scalability, Efficiency	Support partial matching Any cast based messaging and querying mechanisms for homogeneous resources
Scalability, Adaptiveness	Support hierarchical resource description by using hierarchical chord distributed hash tables
Adaptiveness, Scalability	Support adaptation with multi core/cpus environments
Consistency	Topology-aware system which can support for static environment

In the next phase we will extend resource discovery to fulfill the functionality and requirements shown in the table below:

Requirements	Functionalities
Scalability	Support both of exact/partial matching for querying in dynamic environment including frequent node join, leave and failure
Scalability, Adaptiveness,	Support scalability and adaptation for many cores/cpus systems
Heterogenity	Support heterogeneity of resources
Efficiency	Support load balancing for servers that are offering same resources

RESOURCE DESCRIPTION

There are several OS modules that depend on the *resource description* module, including modules related to *resource discovery* and *code segmentation and analysis*. As (the ideas for) these modules were fleshed out independently there might be a mismatch between the information offered by the resource description module and the information required by the modules that depend on the resource description information. We should thus, in the next iteration, analyse the exact requirements posed on the resource description component by other modules, and determine how and if we can either adapt the *resource description* module, or the modules that depend on it. Additionally, we will investigate how to expose the information captured by the *resource description* module to other OS modules and/or arbitrary applications through an appropriate API.

SCHEDULING

In this document, we summarised the basic principles behind scheduling of entities in S(o)OS. These have been derived from the general high-level requirements posed on the OS, as well as concrete experiments performed on real prototype hardware (the Intel SCC), in which the major challenges to be addressed in tile-based architectures have been experimentally highlighted. Particularly challenging is the task of keeping under control the execution time of code segments, as due to the fact that interferences at the interconnect-level among tasks will potentially cause great variations in the experienced execution times. Prediction of the execution time of threads is critical not only for RT applications, but also for HPC workloads. Indeed, it is at the basis of the decision about whether or not to migrate code segments, along with the prediction of migration overheads.

In the second phase of S(o)OS, we plan to develop concrete models of the execution time variability in the context of tile-based architectures. These will be leveraged for designing concrete scheduling strategies, that need to be evaluated on the basis of either experiments made on the Intel SCC again, or, on a wider scale, on the MCoreSim latency simulator we developed during the first phase [53]. This simulator needs still to be further expanded in functionality, in order to allow for performing such evaluations.

TRANSACTIONAL MEMORY

The algorithm delivered in D4.2 of Distributed (message-passing) Transactional Memory for non-cache coherent architecture is now up and running on the Single-chip Cloud Computer architecture and is currently being tested on micro-benchmarks, under the name *Single-chip Cloud Transactional Memory (SC-TM)*. The current algorithm is already deadlock-free but in some cases transactions may restart an unbounded number of time due to the contention induced by concurrency. Our first next step is to extend the SC-TM to integrate a distributed scheduler in the form of a distributed contention manager that schedule transactions so that all transactions eventually commit, thus ensuring starvation-freedom. The goal is to monitor at runtime the dependencies that could not be derived prior to execution and to reschedule transactions to minimize such dependencies. We plan to validate experimentally starvation-freedom (and the absence of livelocks) on a bank application.

We also have implemented a Shared Memory Transactional Memory library implementation and we has started combining it with the Linux scheduler calls to give a higher priority to pending transactions than new ones. The next step is to integrate a dedicated scheduler able to cleverly schedule transactions. In addition, we plan to investigate the integration of TM in a real-time strategy game application to get a deeper understanding of the performance of TM in a many-core environment when running a real concurrent application.

REFERENCES

[1] M. K. McKusick, W. N. Joy, S. J. Leffler and R. S. Fabry. A Fast File System for UNIX. ACM Transactions on Computer Systems 2(3):181–197. doi:10.1145/989.990.

[2] D.E. Porter, O.S. Hofmann, C.J. Rossbach, A. Benn, E. Witchel Operating System Transactions. SOSP 2009.

[3] C.J. Rossbach, O.S. Hofmann, D.E. Porter, H.E. Ramadan, A. Bhandari, E. Witchel. TxLinux: Using and Managing Hardware Tansactional Memory in an Operating System. SOSP 2007.

[4] S. C. Tweedie. Journaling the Linux ext2fs filesystem. Proceedings of the 4th Annual LinuxExpo. 1998.

[5] S. Ghemawat, H. Gobioff, S.-T. Leung. The Google File System. SOSP 2003.

[6] M. Russinovich. D. Solomon. Windows Internals. Microsoft Press 2009.

[7] GCD libdispatch. http://libdispatch.macosforge.org/

[8] P. Mc Dougall. Microsoft pulls buggy Windows Vista SP1 Files. Information Week 2008.

[9] A. Chou., J. Yang, B. Chelf, S. Hallem, D. Engler. An empirical Study of Operating System Errors. SOSP 2001.

[10] D. Enger, K. Ashcraft. Race-X: Effective, static detection of race conditions and Livelocks. SOSP 2003.

[11] D. Hitz, J. Lau, M. Malcolm. File System Design for an NFS File Server Appliance. Technical report 3002, Network Appliance.

[12] Ganger and Patt. Soft Updates: A Solution to the Metadata Update Problem in File Systems. Technical report CSE-TR-254-95, Computer Science and Engineering Division, University of Michigan, 1995.

[13] A. Dragojevic, P. Felber, V. Gramoli, R. Guerraoui. Why STM can be more than a Research Toy. CACM. 54 (4) p.70-77. 2011.

[14] Rosenblum and Ousterhout. The design and implementation of a log-structured file system. SOSP 1991.

[15] T. Cucinotta, G. Lipari, R. Aguiar, J. P. Barraca, B. Santos, J. Kuper, C. Baaij, L. Schubert, H. Kreuz. S(o)OS Project Deliverable D5.2 – Definition of Future Requirements , July 2010.

[16] V. Gramoli, R. Guerraoui. Brief Announcement: Transaction Polymorphism. Proceedings of the 23rd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA), June 2011.

[17] D. Harmanci, V. Gramoli, P. Felber. Atomic Boxes: Coordinated Exception Handling with Transactional Memory. Proceedings of the 25th European Conference on Object-Oriented Programming (ECOOP), July 2011.

[18] K. Flautner, R. Uhlig, S. Reinhardt, T. Mudge. Thread level parallelism of desktop applications. 1999.

[19] G. Lakshminarayana, A. Raghunathan, N. Jha. Incorporating Speculative Execution into Scheduling of Control-Flow-Intensive Designs. IEEE 2000.

[20] S. Wang, X. Dai, K. S. Yellajyosula, A. Zhai, P. C. Yew. Loop Selection for Thread-Level Speculation. University of Minnesota 2005.

[21] L. Gao, L. Li and J. Xue - University of New South Wales, Australia. T. F. Ngai - Microprocessor Technology Lab, Intel. Loop Recreation for Thread-Level Speculation. 2008.

[22] Oracle – Solaris White Paper. Developing Parallel Programs – A Discussion of Popular Models. 2010.

[23] J. Oplinger, D. Heine, S. W. Liao, B. A. Nayfeh, M. S. Lam, K. Olukotun. Software and Hardware for Exploiting Speculative Parallelism with a Multiprocessor. Computer Systems Laboratory of Stanford University. 1997.

[24] The AMD Fusion Family of APUs, http://sites.amd.com/us/fusion/apu/Pages/fusion.aspx

[25] GPU-based heterogeneous Computing, http://www.sintef.no/Projectweb/Heterogeneous-Computing

[26] DistCC, a fast distributed C/C++ compiler, http://code.google.com/p/distcc/

[27] CCache, a C/C++ compiler cache, http://ccache.samba.org/

[28] GNU prof, the standard Unix profiler, http://www.cs.utah.edu/dept/old/texinfo/as/gprof.html

[29] A. Schüpbach, S. Peter, A.Baumann, T. Roscoe, P. Barham, T. Harris, and R. Isaacs. Embracing diversity in the Barrelfish manycore operating system. In Proceedings of the Workshop on Managed Many-Core Systems, Boston, MA, USA, June 2008.

[30] B. Santos, J. Zarrin, J. P. Barraca, R. Aguiar, T. Cucinotta, D. Faggioli, G. Lipari, J. Kuper, C. Baaij, L. Schubert, D. Rubio, V. Gramoli, A. Dragojevic, V. Trigonakis. "S(o)OS Project Deliverable D3.2 – First Implementation Set: Protocols", April 2011.

[31] C. Baaij, J. Kuper, J.Zarrin, L. Schubert, V. Gramoli, T. Cucinotta, D. R. Bonilla. S(o)OS Project Deliverable D2.2 - First Implementation Set: "Hardware". April 2011.

[32] C. Lameter, "Extreme High Performance Computing or Why Microkernels Suck," In: Proceedings of the Linux Symposium, 2007.

[33] T. Cucinotta, G. Lipari, D. Faggioli, F. Checconi, S. Kumar, R. Aguiar, J. P. Barraca, B. Santos, J. Zarrin, J. Kuper, C. Baaij, L. Schubert, H.-M. Kreuz, V.Gramoli. S(o)OS Deliverable D5.1 – State of the Art. July 2010.

[34] G. Fohler, T. Lenvall, R. Dobrin, A. Burns, G. Bernat, I. Broster, M. G. Harbour, J. J. Gutierrez Garcia, J. L. Medina Pasaje, G. Lipari, P. Gai. FIRST EU Project Deliverable D-SI.1v3 : Scheduler integration definition report. April 2005.

[35] G. Lipari, L. Palopoli, L. Marzario, T. Cucinotta. OCERA EU Project Deliverable D4.4: WP4 – Resource Management Components – Resource Management Components V2. February 2004. Available at: **http://www.ocera.org**.

[36] I. Ripoll, A. Crespo, P. Balbastre, J. Vidal, M. Masmano, and A. Lucero. OCERA EU Project Deliverable D5.4: Scheduling Components V2. February 2004. Available at: **http://www.ocera.org**.

[37] M. G. Harbour and M. Telleira de Esteban. FRESCOR EU Project Deliverable D-AC2v2: Architecture and contract model for integrated Resources II. January 2008. Available on-line at: **http://www.frescor.org**.

[38] G. Lipari, T. Cucinotta, F. Checconi, D. Faggioli, L. Abeni, L. Palopoli, P. Valente. FRESCOR EU Project Deliverable D-AQ3v2 : Design and implementation of a middleware for QoS management – II. March 2009. See **http://www.frescor.org**.

[39] G. C. Buttazzo, G. Lipari, L. Abeni, M. Caccamo. Soft real-time systems: predictability vs. efficiency. Springer, 2005. ISBN 978-0-387-23701-5.

[40] G. Katsaros and T. Cucinotta. Programming Interfaces for Realtime and Cloud-based Computing. In Achieving Real-Time in Distributed Computing: From Grids to Clouds. IGI Global, July 2011. DOI: 10.4018/978-1-60960-827-9.

[41] A. Depoutovitch, M. Stumm. "Otherworld" - Giving Applications a Chance to Survive OS Kernel Crashes. EuroSys 2010, Paris, France. April 2010.

[42] F. Mulas, D. Atienza, A. Acquaviva, S. Carta, L. Benini, G. De Micheli. Thermal Balancing Policy for Multiprocessor Stream Computing Platforms". IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (T-CAD), IEEE Press, ISSN: 0278-0070, Vol.28, Nr. 12, pp. 1870-1882, Dec 2009.

[43] F. Zanini, D. Atienza, G. De Micheli. A Control Theory Approach for Thermal Balancing of MPSoC", Proc. of 14th Asia and South Pacific Design Automation Conference (ASP-DAC 2009), Yokohama, Japan, ISBN: 978-1-4244-2749-9/09, pp. 37-42, January 2009.

[44] A. K. Coskun, J. L. Ayala, D. Atienza, T. Simunic, Y. Leblebici, Dynamic Thermal Management in 3D Multicore Architectures, Proc. of Design, Automation and Test in Europe (DATE '09), Nice, France, ACM and IEEE Press, ISSN: 1530-1591/05, ISBN: 978-3-9810801-5-5, pp. 1410 – 1415, April 2009.

[45] G.M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," 1967.

[46] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter et al. (2009). The multikernel: a new OS architecture for scalable multicore systems. In SOSP '09: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles, pp. 29-44.

[47] 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, eds., Springer, 2009, pp. 56-69.

[48] G. Jost, H. Jin, D. Mey, F.F. Hatay. "Comparing the OpenMP, MPI, and Hybrid Programming Paradigm on an SMP Cluster," EWOMP, Aachen, Germany, Sep 2003.

[49] J. Haller, L. Schubert, S. Wesner. "Private Business Infrastructures in a VO Environment," *Exploiting the Knowledge Economy: Issues, Applications, Case Studies*, P. Cunningham and M. Cunningham, eds., 2006, pp. 1064-1071.

[50] J. Lelli, G. Lipari, D. Faggioli, T. Cucinotta, "An efficient and scalable implementation of global EDF in Linux," Proceedings of the 7th International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT 2011), Porto, Portugal, July 2011.

[51] T. Cucinotta, L. Palopoli, L. Abeni, D. Faggioli, G. Lipari, "On the integration of application level and resource level QoS control for real-time applications," IEEE Transactions on Industrial Informatics, Vol. 6, No. 4, November 2010.

[52] L. Schubert, D. R. Bonilla, R. Aguiar, B. Santos, J. Zarrin, T. Cucinotta, J. Kuper, C. Baaij, V. Gramoli, A. Dragojevic. S(o)OS Project Deliverable D6.1 – Evaluation Criteria . February 2011.

[53] S. Kumar, T. Cucinotta, G. Lipari. "A Latency Simulator for Many-core Systems," in Proceedings of the 44th Annual Simulation Symposium (ANSS 2011), part of the Spring Simulation Multiconference (SpringSim'11).