Evaluation Results #1

Project Deliverable D6.2

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

Due date: 31/10/2011
Delivery date:31/10/2011

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	15/08/11	First ToC	Lutz Schubert
0.2	30/08/11	Initial content and bullet points	All
0.3	06/09/11	First evaluation results	All
0.4	19/09/11	Usability discussion basis	Lutz Schubert
0.5	22/09/11	Usage scope discussion	All
0.6	05/10/11	Extended evaluation results	All
0.7	11/10/11	Integration discussion basis	Lutz Schubert
0.8	14/10/11	Integration discussion extension	All
0.9	18/10/11	Future work & next steps	Lutz Schubert, all
0.10	19/10/11	Internal review feedback	SSSA
0.11	28/10/11	Final updates	All
0.12	31/10/11	Formatting and minor corrections	Lutz Schubert

EXECUTIVE SUMMARY

S(o)OS investigates alternative OS architectures to overcome the problems inherent to the growing scale and heterogeneity of future computing systems. To this end, the S(o)OS project identified the specific requirements put forward by future computing architectures and their usage (D5.2). On basis of this analysis, a set of essential functionalities could be identified that future operating systems need to fulfill (D6.1).

During the first project cycle, S(o)OS established a set of OS components related to these specific requirements (D2.2, D3.2, D4.2) and an according architecture outline on how these components can fit and interact together to support future use cases and provide the ability to deal with the scale and heterogeneity of future system architectures (D5.3).

In order to identify whether and in how far these components actually contribute to fulfilling these objectives, they need to be assessed against the requirements identified in D5.2 and D6.1. This document summarizes the results from this evaluation effort and analyses whether the chosen approaches are sensible in light of these expectations, respectively where they clash and why. This document does not conclude the evaluation or implementation effort, but provides feedback to future iterations of development and research in these domains. More specifically, the results will be used to shape the work in the second project cycle, namely to select the promising components and develop them further, as well as to identify alternatives. The final evaluation will thereby focus more than this one on the whole integration of the selected components.

As can be seen in the text below, the current S(o)OS components show a high degree of scalability for static use cases, i.e. where the infrastructure is not expected to change at run-time. Such dynamicity would imply additional aspects of reliability and state maintenance that have not been considered in the first project iteration and will be a more relevant issue in the second cycle. The basic architectural approach, namely to handle the OS components in a cloud-like work-balanced fashion, allows for higher dynamicity in handling the infrastructure, yet at the same time requires an efficient management system for maintaining the instances and their replication / relocation – these aspects however are of primary relevance only in dynamic cases, i.e. where the distribution of components may have to change in accordance with the applications' runtime requirements and changes in the infrastructure setup. With the first cycle we assume that the relationship between code segments, OS components and infrastructure are fixed prior to execution as part of the segmentation process.

Due to the principle flexibility of the current architecture, we can denote a wide usage scope of the individual components, without yet being able to provide concrete comparison values: each component shows clear behavioral strengths and weakness which restrict (or support) the application scope. However, since some components serve explicitly the support of specific use cases (such as supporting different types of scalability), they can principally be unplugged or replaced thanks to the component-based approach of S(o)OS in other cases. In other words, the components themselves can be adjusted comparatively easy to the use cases. However, in order to achieve the full potential, the interoperation between these components needs to be well adjusted and elaborated in more detail in the second phase.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABBREVIATIONS

Abbreviation	Description
E2E	End-to-End
HPC	High Performance Computing
MMO	Massive Multiplayer Online
NoC	Network-on-Chip
OS	Operating System
P2P	Peer-to-Peer
QoS	Quality of Service
S(o)OS	Service-oriented Operating System(s)
SOA	Service-Oriented Architecture
VO	Virtual Organisation

I. INTRODUCTION

The results of the first project cycle are evaluated against the criteria and requirements identified so as to guide the research and development of the second cycle.

The main objective of the S(o)OS project consists in enabling efficient program execution on future system architectures that are of large scale and integrate various types of resources (heterogeneity). Current operating system architectures thereby all exhibit a tendency to reducing the execution performance due to lack of adaptability to different infrastructures and of scalability (see [32]). As has been elaborated in the Project Deliverable D5.2 – Definition of Future Requirements [33], the general approach of S(o)OS to overcome this problem consists in a new, modular, or rather service-oriented approach to structuring operating systems in order to exploit concurrency inherent to code and OS execution at runtime.

To this end, the project consortium has identified a set of relevant components (OS modules or services) that are essential for providing the scalable, distributed and heterogeneous program execution in an efficient manner. Each of these components has been developed with a particular focus on efficiency and scalability, with the modular overarching principle catering for heterogeneity (see [36][37][38]). Further to this, an architectural model for the integration of these components into a single, yet service-oriented system has been sketched out in the context of the S(o)OS Project Deliverable D5.3 – First Set of OS Architecture Models [34]. Through the principles elaborated in the respective document, it is theoretically possible to spread out the whole application and operating system across the infrastructure, thereby increasing performance and scale.

In order to validate these capabilities, it is however essential to test (a) whether the individual components fulfill the respective requirements they are designed for, and (b) whether the combination of components does serve the use case scenario's needs and that none of the components obstruct each other, thereby nullifying their purpose. In other words, the system needs to be examined on two levels:

1. The individual components / modules and how they address the requirements
2. The architecture as a whole and whether this maintains the components' capabilities

As has been elaborated in the deliverable on Evaluation Criteria [35], there are different means to evaluate these capabilities, but for pragmatic reasons not all of these approaches are sensible, or even feasible. With the S(o)OS objectives being highly speculative and far-reaching, no concrete implementation, let alone full integration can be reached within the project lifetime without affecting the overall goals, namely to assess different approaches and their influence on one another. Further to this, since S(o)OS cannot address a whole OS setup itself, it will have to build up and exploit existing parts, which in turn cannot be integrated without high overhead, due to the current highly monolithic structure of operating systems.

We focus therefore specifically on means that allow theoretical evaluation and assessment given the behaviour of related components and the specific of the module in question. We thus restrict the means of evaluation to ones that meet this basic need. Along the same line, the full application and environment scope of a system such as pursued by S(o)OS exceeds the scope of a single project, and would make evaluation in light of all according parameters infeasible. As indicated in the preceding

evaluation related project deliverable [35], we restrict the S(o)OS application scope to the criteria that form the primary obstacles in building up large scale operating systems, i.e. in dealing with the future scale and heterogeneity of infrastructures.

In the following sections we will quickly recapture the objectives, criteria and means as detailed in [35] and elaborate their relevance for the evaluation as summarised in this document.

1. REQUIREMENTS

In order to deal with future large scale heterogeneous systems, a set of specific capabilities needs to be fulfilled

Future environments will be large scale and strongly heterogeneous – not only at the level of high performance computing machines that integrate more and more nodes of various types into large systems, but more importantly on the level of individual desktop machines that incorporate multiple cores of various builds into a single processor, and on the level of internet connectivity where an almost endless number of devices connect with each other and exchange information and computing power (see Grid, Clouds etc.).

It can be generally argued that these major domains, i.e. Grids, Clouds, but also High Performance Computing are essentially identical with the major difference being the capabilities of the underlying interconnect, i.e. bandwidth and latency. Accordingly, one would assume that a general purpose execution environment (operating system) respecting these differences is capable of handling essentially all use cases across these domains.

Table 1: the main application domains and their high-level characteristics

	Compute-intensive	Communication-intensive	Data-intensive	Simultaneity	Collaborative	Scalable	Interactive	Streaming	Real-time	Interoperable	Energy efficient
High Performance Computing	x	x	x								
Cloud Computing		x	x	x			(x)				
Mobile Computing							x			x	x
Office / Home				x	x		x			x	
eBusiness, social networking etc.			x	x	x	x	x		x	x	
Gaming (incl. MMO)	x			x	x	x	x		x		
Multimedia			x					x	x		
Embedded systems									x		

As elaborated in the "Evaluation Criteria" document [35], the potential use cases exhibit a wide range of characteristics, and implicitly of requirements.

Table 1 lists the high level characteristics of some of the major current use case areas. It will be noted that indeed interconnect details play a major role in most of these characteristics, however, that further capabilities would have to be examined to make sure that the use cases are addressed sufficiently. As S(o)OS aims particularly at improving the way of dealing with large-scale, heterogeneous infrastructures, it was decided to focus the effort in particular on according domains, to achieve the best results.

More specifically, we can list the following issues as the primary objectives pursued in the project:

GO#1 the capability to manage a large scale of heterogeneous resources in a single system
GO#2 to support runtime execution over large scale heterogeneous infrastructures
GO#3 to enable effective programming of large scale heterogeneous infrastructures
GO#4 to handle the growing data size effectively
GO#5 the capability to deal with multiple application areas

It was therefore decided to focus the evaluation specifically on domains related to High Performance Computing and Real-Time Systems. Whilst high performance computing is obvious from the perspective of scale, heterogeneity (GO#1, GO#2), efficiency (GO#3) and amount of data (GO#4), real-time systems seem to be less obvious. However, timing constraints need to be met in all efficient environments, to increase synchronicity and reduce communication / system overloading – in particular in heterogeneous environments, timing behaviour is however mostly unpredictable currently. Finally, real-time systems are also a major enabler for various other application areas (GO#5).

The "Evaluation Criteria" document [35] furthermore elaborates the 4 major research lines which represent S(o)OS' technical approach. These 4 lines can therefore be regarded as the technological representation of the general objectives according to S(o)OS' solution approach, which are however of secondary importance for evaluation purposes, besides for their base fulfillment, as they provide no measurable criteria:

- infrastructure resource modeling (PO#1)
- (runtime) code behavior analysis for segmentation & distribution (PO#2)
- operating systems architectures (PO#3)
- programming extensions (PO#4)

2. EVALUATION CRITERIA

The requirements can be broken down into a set of concrete technical criteria that the OS and the components have to meet

It was already noted in the preceding section that each application domain can be represented by a set of characteristics that specify the main requirements of this domain towards the application, middleware and infrastructure. In the "D6.1 Evaluation Criteria" document, we identify in particular the following high level criteria that impact on the application domain [35]:

- Resource Sharing
- Latency
- Bandwidth

- Timing constraints
- Scalability
- Heterogeneity
- Mult-Tasking support
- Efficiency / performance

With respect to the general objectives listed above, we must furthermore add the following criteria

- Ease-of-Use
- Applicability (application scope)
- Programmability

For example, massive multiplayer online games need to be highly scalable, exhibit little latency, maintain real-time constraints and be able to share essential resources, in particular in the form of data. With the specific focus on the two main application domains in S(o)OS, namely High Performance Computing and Real-Time Systems, the main criteria for evaluation obviously become:

C#1 Scalability
C#2 Heterogeneity
C#3 Efficiency / performance
C#4 Timing constraints

It must be noted though that in particular efficiency (C#3) implies other criteria related to connectivity, i.e. latency (C#5) and bandwidth (C#6), which should thus be considered secondary evaluation criteria. Within the first cycle of the project we consider ease-of-use (C#7) and programmability (C#8) also secondary criteria, as the main focus rests clearly on the technological basis before concentrating on the means to expose them to the user.

In the related document (see [35]) we also identified the concrete relationships between the overarching criteria and the phases of the program lifecycle, respectively the tasks that an operating system has to fulfill. This relates in particular to aspects such as when concurrency measurements actually improve scale or how resource descriptions relate to performance – as these criteria will be elaborated in more detail in the evaluation sections below, this will not be repeated here.

Rather than reaching explicit values, which can be considered improvements only for a relative time, however, it is one of S(o)OS' major goals to investigate means to enhance the base capabilities drastically by reducing the overall inhibiting factors. For example, it is the main goal of concurrency exploitation to reduce the classical impact of Amdahl with respect to "just" exploiting algorithm parallelism.

3. EVALUATION MEANS

There are multiple means to evaluate software and algorithms, but not all apply to the concepts of S(o)OS

More importantly, we have to identify the means to evaluate the S(o)OS' components: since S(o)OS is a highly research-driven and experimental project, full development and integration is not pursued in the project, so as to keep the main focus on technological and architectural concepts and to cover a wide range of approaches – implementation and integration would thereby restrict this

flexibility and would lead to a non-sustainable, short-term solution, rather than an alternative long-term approach, as pursued by S(o)OS [35].

Generally, and as elaborated in the "D6.1 Evaluation Criteria" document [35], we can distinguish between three major evaluation principles: (1) benchmarks and unit tests, (2) formal and theoretical methods, (3) experimental means. With S(o)OS focusing mostly on theoretical solutions, rather than on implementation and deployment, it is obvious that formal and theoretical based evaluation approaches suit the project objectives best. Accordingly, actual execution and / or test runs are secondary to these approaches and will only be applied to prototypes or partial implementations / integrations.

Concretely, we identified four major means for evaluating S(o)OS, which are summarized in the following (for more details see [35]):

1. Architectural Evaluation
   examines the general design of the software system to identify bottlenecks, scalability issues etc. This approach is particularly interesting for evaluation of the interaction between components, and thus for the evaluation of the OS as a whole. Due to the complexity, this approach will be supported by Abstract Simulation environments (see below).
2. Formal Evaluation
   uses mathematical and formal methods to evaluate complexity, behavior etc. of a specific component. Given the objectives of S(o)OS this is the most common evaluation means. The purpose in formal evaluation consists in identifying the general strengths and weaknesses and comparing them with existing approaches.
3. Abstract Simulation
   expands the capabilities of both the architectural and formal evaluation by providing simulation frameworks in which the general behavior of the components can be tested against each other. This allows in particular examining the same components easily under different circumstances, such as given different application characteristics. Due to the focus on individual components in the first iteration, this approach will be pursued more in the second evaluation cycle.
4. Informal, Non-Functional Evaluation
   Quality assessment bases to a good deal on characteristics that are not formally measurable (e.g. "ease of use"), or where the boundary conditions are not entirely clear (e.g. for new hardware). In these cases, an experience based evaluation is most often the only feasible approach – typically this is executed through in-depth testing sessions with multiple users, but due to S(o)OS constraints, this will mostly be restricted to the expertise within the community.

Not all means apply to all criteria and components – the individual descriptions below will elaborate in more detail which approach was chosen and why, along with the more detailed description in [35].

II. COMPONENT-BASED EVALUATION

The components developed in the first cycle are evaluated individually against the criteria identified in the project. Not all components address all criteria though, as they represent functional aspects of the whole OS.

It has been noted multiple times before that the behaviour of the overall operating system is defined by the capabilities and restrictions of the individual components forming the system. For example a system that addresses scalability issues and contains one component that does not scale, may actually have problems during execution, depending on the relevance of this one component. In addition to this, the capabilities offered by the components form also the capability scope of the integrated system, so that an increased efficiency of one single component can already improve the overall efficiency, again depending on the role of that component in the full system. For example, modern operating systems' often fail to create and destroy threads efficiently, so that an accordingly efficient OS component would improve execution performance already drastically[1].

In order to identify the potential capabilities, strengths and weakness of the different integration choices and options, the individual components have to be evaluated with respect the general S(o)OS objectives and criteria:

1. CODE ANALYSIS AND SEGMENTATION

The code analysis and segmentation related components provide the relevant means to assess the detailed relationship ("dependency") information within a set of operations, i.e. code. This equally addresses code, as well as data flow. With this kind of information, a dependency graph across different parts of the application (or source code) can be generated that implicitly reflects the potential degree of concurrency in the code. In order to exploit the concurrency, the graph can be segmented according to the degree ("strengths") of the dependencies, thus allowing parallel execution of the respective segments.

Dependency identification is difficult for the average user, and even more difficult to be employed in an efficient manner, i.e. without actually affecting the performance in a negative fashion, as the average programmer tends to either over- or underspecify the relationships. What is more, the best specification (and hence distribution) may depend heavily on the executing platform, so that developers may have to rethink their annotations depending on the destination infrastructure.

To support the developer in this task, the code analysis and segmentation tools provide a set of functionalities that user *and* operating system can exploit to help identify code dependencies and segments, as well as their adaptation to specific infrastructure types. The tools should thus enable different levels of annotations, ranging from highly abstract relationship indicators to concrete platform-specific distribution annotations. Abstract relationship indicators can help the tools in forming a more efficient decision in particular where the data types impact on behavior. More concrete annotations can be used to supersede the tool decision when the respective conditions apply (e.g. availability of the specified resource type etc.)

[1] Note in this context that S(o)OS does not even introduce threads, as it executes

Further to this, such extensions can be used for more efficient source code compilation, and additional specifications such as data usage range may allow further low level optimization strategies – this would however exceed the scope of S(o)OS and will be addressed in a fellow project[2].

The results of the first phase allow for offline analysis of the compiled code in a close-to-execution fashion, i.e. system- and data-behaviour can be respected to a certain degree. Due to the offline nature of the analysis, performance plays a secondary role in the current toolset. The analysis generates a fine-grained dependency graph that reflects data- and work-flow relationships. This graph can then be segmented in different fashions, following generally a mixed bottom-up top-down strategy, where (strongly) related code parts are merged and (weakly or) unrelated ones separated. See [36] for details.

OBJECTIVES ADDRESSED

Speedup through parallelization is nearing a standstill, very much in line with Amdahl's prediction, but even further impacted by hardware constraints and algorithmic restrictions. Parallel programming thus becomes increasingly important, as many-core processors takeover the market. Automated parallelization is very restricted and will pose increasing portability and application issues. Further to this, distribution and instantiation of threads (respectively processes) need to be aligned and related closely with the operating system to reduce the overhead for instantiation, handling, communication etc.

With current programming and execution models, the developer is generally forced to identify the bottlenecks and the points for parallelization himself. What is more, he has hardly any means to exploit concurrency, unless writing explicit threads for this. The analysis and segmentation toolset focuses in particular on supporting this part of the development and automating it on the operating system level. Therefore the main objectives pursued by this toolset relates in particular to increasing the degree of scalability of programs (GO#1, C#1) in a simple fashion yet controllable fashion for the developer (GO#3). The toolset thereby provides extensions (annotations) to the program that allows the execution framework to improve the scalable execution (related to GO#2) and to identify resource preferences for efficient execution (related to GO#1 and GO#2, C#2).

The main goal is thereby to exploit concurrency inherent to the code, so that the total code can be executed in less total time (C#3) with better resource usage (contributing to C#2, C#3). The principles are thereby applicable to any domain (GO#5), even though supporting applicability across usage areas is not an explicit goal of this toolset.

a) CURRENT CAPABILITIES

Related to the general objectives of enabling code to deal with large scale and heterogeneous infrastructures, we can identify various approaches, as listed in detail in [32] and [33]. The problem per se is not new, even though scale and scope have increased drastically and are expected to increase even further. Classically it was expected that the compiler deals with the according conversions, but it became more and more obvious that additional information (context) was required from the user. However with the need for higher portability as the destination platform

[2] ECOUSS – efficient and open compiler environment for semantically annotated simulations (http://ecouss.dfki.de/)

types start to deviate from one another, a higher degree of flexibility is required that led to just-in-time compilers and more experimental.

In addition, performance analysis related tools have been developed which allow in particular the experienced developer to identify additional parallelization potentials, as well as execution bottlenecks in the (running) code. These tools do not affect the code and thus do not actually exploit the information gained themselves – it is up to the developer to take actions accordingly.

In general, current technologies aim particularly at (semi-automated) parallelization for a specific destination platform. We must thereby distinguish between two major tracks: (1) addressing a broad user scope and simplifying development to a maximum extent and (2) aiming at high performance and exploiting the resources to their maximum. These two tracks generally oppose each other in so far as ease-of-use implies higher degree of flexibility, but needs to fall back on predefined patterns which are generally less efficient than code that has been explicitly developed to exploit the specifics of the destination platform, but thus is implicitly less portable.

The approach by S(o)OS builds a bridge between these two strands in so far, as the same principles can be applied by both the common and the expert developer. Depending on the degree of information provided, the user can influence the detailed behavior and thus steer whether general actions, or user specified modifications should take place.

With respect to this dichotomy, we must note the following main points about the current development status, which will serve as a reference for evaluating the toolset:

High Level Approaches
Just-in-time compilers and higher-level programming languages all aim at relieving the developer from the problems of having to adapt the code to the specific platform by providing a very abstract view on the hardware. In order to function, they require multiple intermediary translation steps and stacks that are adapted to the actual execution platform. This has two major consequences: first of all, the multiple conversions reduce the efficiency of the code execution accordingly, and secondly that the developer can typically not control the translation middleware and thus influence how code is treated in specific platforms. As this approach requires that each potential destination platform hosts an according middleware, adaptation to new resources and in particular mixed infrastructures can become tedious, as the .NET example showed with MONO.

Even though efficiency of these stacks has improved drastically over time, at least an initial compilation / adaptation delay is still notable. Further to this, the efficiency still falls way behind what an experienced developer can achieve for a given platform. Similarly, the degree of scale achieved is comparatively low, as most of these middlewares aim for comparatively small scale systems as yet. Due to the pattern based approach, the scale out can easily lead to decreased efficiency, when e.g. a loop is unrolled that carries too little workload to compensate the thread instantiation and data handling overhead. In fact, in order to make best use of the automated approaches, it requires a similar level of expertise than for the low-level case.

Low Level Solutions
The classical approaches rely on the developer actually taking the necessary adaptation steps or at least providing the relevant information for the compiler to take over this part. Current compilers are highly efficient in converting a given code to a specific platform, yet they offer little portability as

their main task consists in exploiting the destination resource's capabilities. Heterogeneous platforms are therefore particularly difficult to address using standard compiler techniques.

Most compilers have to make some dependency analysis in order to resolve local and global variables. More advanced compilers even try to rearrange the data allocation in order to make best use of frequently used vs. rarely used memory pages. However, all compilers base strictly on off-line analysis of dependencies and memory usage and therefore struggle with dynamic arrays etc. As the code is generally not self-dynamic and thus cannot adjust itself to cater for such changes, the compiler may lead not only to non-portable, but also inefficient code.

Analysis tools such as Valgrind[3] provide a lot of information about the on-line behavior of the code, i.e. including memory access and data usage. However this information is not employed by the compiler or any other execution management framework. Instead, the user will have to analyse the impact on the code manually and restructure it accordingly. What is more, the tools are not able to distinguish between data "fixed" points and behavior that changes when the data changes, so that the user has to manually analyse multiple runs to identify the appropriate sources for performance reduction.

Hardly any tool makes use of code *concurrency* – only parallelism is in the current focus of development. The only major exception to this rule build the StarSS programming extension set by Barcelona Super Computer [47] which allows the user to annotate dependencies between functions which are then exploited by the execution engine to execute functions in parallel which are independent of each other, i.e. exploiting their concurrency. StarSS however requires a lot of user input without being able to validate it, and only works on predefined functions, rather than on any code segment. Furthermore, the engine does not consider resource utilization aspects, i.e. the execution of one segment may stall other threads because the resources are blocked.

EXPECTED IMPROVEMENTS

The S(o)OS approach combines many of these aspects into an integrated adaptation cycle that allows improvement of code execution behavior over runtime in a fashion that normally would have to be done manually by the user. By analyzing the behavior similarly to Valgrind, identifying the data-dependent and –independent behavior and feeding it back to both compiler and execution management, the S(o)OS approach closes the circle that currently consists of developer, compiler, execution analyser and user.

More concretely, given the current status of the tools as described above, we would particularly expect future systems (such as S(o)OS) to extract platform-independent information about the code behavior that can be used to automatically execute performance improvements that currently have to be done manually, or to adapt the code so as to exploit specific resource capabilities without further complicated interactions by the user.

As noted, usability typically comes at the cost of performance and vice versa. All current approaches focus on either of the two, rather than trying to find the common denominator, as the S(o)OS approach tries to. However, we cannot expect the system itself to automatically create highly optimized code. What is more, compiler-techniques etc. are well advanced in terms of performance

[3] http://valgrind.org/

focused adaptation for a specific platform – any new approach would implicitly have to compete with these capabilities (or at least show the potential to compete with it).

The specific approach pursued by S(o)OS is not so much a *replacement* of current technologies, but an extension that enhances current capabilities, rather than trying to supersede them. Similarly, user input will still be required, but rather than looking for different types of information for experienced versus common users, we give them the potential to steer the behavior of the framework according to their own level of expertise. Depending on the user expertise, the underlying toolset will have to take more or less actions, so that we generally expect (a) that the performance gain through the tool is higher for non-experienced than for experienced users and (b) that the impact of portability on performance is less for non-experienced than for experienced users.

As the current realization provides only the base capabilities in this direction, we cannot expect full impact though. Given these circumstances, we expect particular improvement along the following lines:

a) That the resources are put to better use thus improving the general execution performance – this includes in particular that more resources are utilized and that existing resources are utilized better (see b)
b) That resource specific capability indicators can be principally identified that would improve code performance compared to a general x86 system

b) Evaluation Criteria & Means

The key criteria for evaluating the capabilities of the code analysis and segmentation toolset consist therefore in

1. The degree of parallelism (respectively concurrency)
2. The principle adaptation to specific resource types

Both aspects can also be subsumed as the degree of (potential) resource utilisation after analysis. It has also been discussed that the degree achieved cannot be justly compared to the degree achieved by modern compilers as they build on different approaches and, what is more important, the two approach extend, rather than oppose each other. In order to achieve an appropriate comparison, we therefore would need to incorporate the toolset into different existing compiler and thus lose effort and time in the according adaptation whilst we are still trying to assess the general algorithm principles and their impact. Given the current development status and the scope of the project, we therefore focus primarily on the principle improvements achieved in non-adapted or only sequentially compiled code for generic (single-core) x86 machines.

EVALUATION PRINCIPLES

The principle of the toolset builds upon exploiting implicit concurrency in the code rather than parallelising selected functions. It therefore extends the parallelisation scope beyond the classical interpretation of Amdahl, where the “degree of parallelism” represents the part of the algorithm that can be executed on multiple processors at once. As opposed to this, the S(o)OS toolkit does investigate parallelism on the level of the program as a whole, rather than low-level parallelism – though the information gathered may be used to improve the latter, too.

To achieve this, S(o)OS must incorporate additional communication points and means that implicitly create a delay which may in turn slow down the execution speed. For the purpose of evaluation we must therefore identify

a) The degree of concurrency principally achieved
b) The timing dependencies and impact, i.e. the general execution delay due to dependencies and
c) The time required for executing the communication

Obviously, the degree of improvement thus achieved depends on the type of code and its implementation. The results would thus differ exceedingly depending on the selected code type. To compensate this, we propose a theoretical approach that identifies the potential speedup rather than the concrete gain for a specific code set. The results must therefore represent a value scope or range, i.e. the scope from least possible speedup to maximum performance increment.

The algorithm can principally lead to slowdown of the program, if applied too strictly, i.e. if the segmentation is enforced beyond a threshold, such as segmenting too strong relationships or generating too small segments. The cutoff point is thereby determined in the same fashion as this evaluation proceeds, i.e. by calculating the potential gain scope which should implicitly be bigger than 1 as the lowest denominator.

It should be noted in this context that an additional constraint needs to be considered prior to actual enactment at run-time, and that is the additional delay created by relocation (instantiation etc.) of the segment. This however does not affect the principle performance gain that can be achieved by the methods proposed, but must be considered as a decision criteria at runtime.

As for identification of hardware indicators, the dependency graph must incorporate enough information details that can be mapped or related to hardware specifics. This relates strongly to the problem of resource capability description: it is difficult to represent functionalities of a given hardware architecture in a fashion that relates to specific code behaviour and the optimisation steps taken by experts generally base more on experience than specific deductions from the microarchitecture. Nonetheless, from this experience some general indicators can be derived which the hardware specification tries to capture (cf. section II.3). Such indicators include aspects of vectorisation, precision, cache requirements etc. As we shall see, the code analysis methodology has the potential to highlight multiple of these aspects in the dependency graph generated, even including aspects that are generally problematic for compilers, such as data usage scope for efficient rearrangement.

Again, there can be no general means to evaluate this aspect independent of code intention, chosen algorithm or implementation. As opposed to performance evaluation, however, it is not possible to generate an overarching value range for this case – instead, the main point of this function is exactly to identify the resource capabilities of specific code parts. Rather than trying to show the capability on basis of selected code types though, we focus primarily on showing that the principle of the code analysis and segmentation toolkit generates an information set that is very similar to the information required for resource capability description.

EVALUATION MEANS

As noted, evaluation of the code analysis and segmentation toolkit is primarily formal in nature, i.e. basing on mathematical assessment. However in particular in the context of hardware adaptability and support for programmability, expert assessment and therefore informal evaluation are necessary to interpret the information sensibly. Given the results of the current iteration, the algorithm will be improved and integrated with the other tools and components (in particular code adaptation) and then evaluated again on top of specific platforms and given specific codes as a validation for the formal assessment provided here.

EVALUATION CRITERIA

Given the objectives pursued by the code analysis and segmentation toolset (cf. above), the focus of the evaluation must rest on the following main evaluation criteria:

1. Scalability (C#1), in the sense of that the scale (concurrent usage of resources) of the analysed application increases through the means of the code analysis and segmentation toolkit. This is closely related to resource utilization and performance.
   Concrete criteria to be assessed:
   a. Amount of resources potentially used at the same time
   b. Potential maximum / minimum scale
2. Resource Utilisation (C#2, C#3), does not only cover the aspect of using more resources at the same time, but also to employ the resources more efficiently, in the sense of adapting the code better to the specific resource type. This is obviously closely related to scalability and performance. Concrete criteria include:
   a. Relationship between maximally to minimally used amount of resources
   b. Resource usage – in this iteration in terms of identification of resource capabilities
3. Performance (C#3) reflects the overall execution time of the code after being analysed and segmented. Performance is improved through two major means: increasing the concurrency (scale) and adapting the code better to the resource infrastructure specifics (resource utilization). Concrete criteria in this case is:
   a. Reduction of total execution time – here in the sense of potential reduction against the purely sequential execution

   Notably, additional performance criteria related to code analysis, segmentation and distribution, such as thread instantiation time,
4. Ease of Use (C#7, C#8) can effectively only be measured through informal assessment of the simplifications offered by the analysis and segmentation toolset. However, ease of use in the case here is improved primarily by reducing the programming overhead for reaching scalability and performance – in other words, ease of use here is defined through the effects with respect to criteria C#1-3. Accordingly the criteria can be formulates
   a. Amount of additional input required by developer to improve C#1, C#2, C#3

c) EVALUATION

In order to evaluate the code analysis toolset in the form discussed above, it is necessary to first discuss the underlying mathematics and principles:

CONCURRENCY EVALUATION

As discussed in the preceding section, the evaluation of the efficiency gain through the concurrency mechanisms is executed through a mathematical assessment of the code analysis tool's behavior. In the following we will conduct an iterative evaluation of the essential behavioural characteristics to derive the main capabilities (and limitations).

1) Simple Dependency Between Two Segments

Obviously, if we can identify a segmentation without any dependencies, i.e. perfect parallelism, the total execution time is identical to the maximum time of any thread generated by the segments. Due to the nature of the min-cut algorithm, such segmentation would be found during the first iterations of the algorithm. However, this would imply that effectively, the program consists of a set of independent processes rather than full application. In the simplest case there would be at least one communication dependency between all segments, namely to invoke them and to pass the initial value(s). Assuming we have two threads with a single dependency, we can therefore generate a parallel execution model where the second thread is executed with a delay that coincides with the value creation and its communication (see Figure 1)

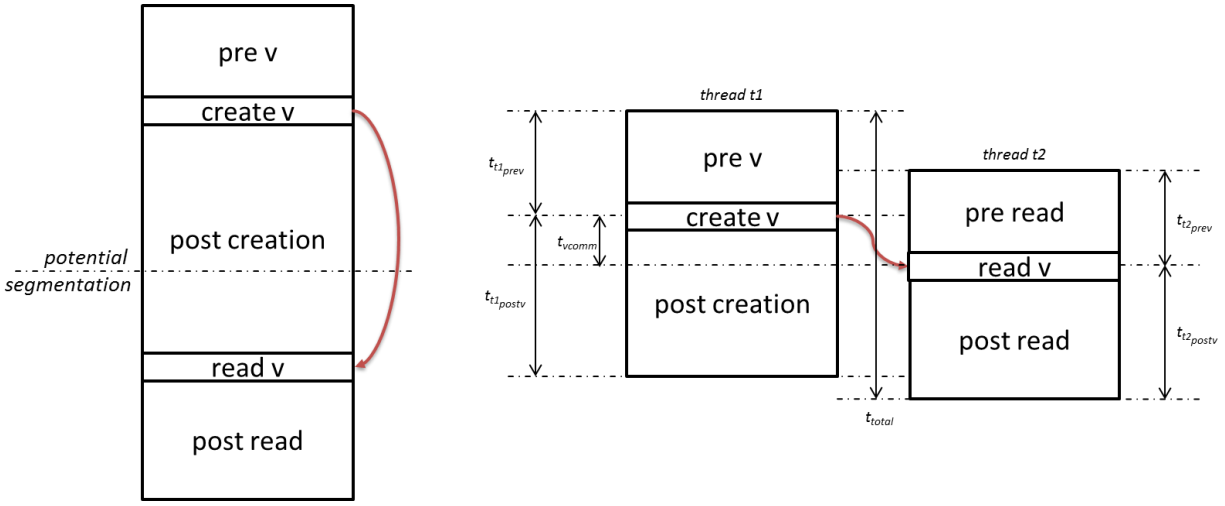

Figure 1: impact of a potential segmentation of a single dependency code part (left: prior to segmentation; right: distributed execution)

In this case, we can define the execution time as

① $$g_1(v) = t_{t[1]_{pre}}(v) + t_{t[1]_{post}}(v)$$

② $$g_2(v) = \max\left(t_{t[2]_{pre}}(v), t_{t[1]_{pre}}(v) + t_{comm}(v)\right) + t_{t[2]_{postv}}(v)$$

$$t_{total} = \max(g_1(v), g_2(v))$$

assuming that both threads are started at the same time (as opposed to Figure 1). It is thereby obvious that the speedup is zero, if the second thread cannot start executing at all prior to receiving the value v and the first thread ends with generation of this value. This is reflected in the algorithm as an extension to the weight between vertices, where distance (and hence time) is considered as a weight-reduction factor (cf. evaluation discussion below).

Obviously, the optimal segmentation is found, when

$$g_1 = g_2 \ \wedge \ t_{t2_{pre}} = t_{t1_{pre}} + t_{comm}$$

$$\leftrightarrow \ \cancel{t_{t1_{pre}}} + t_{t1_{post}} = \cancel{t_{t1_{pre}}} + t_{comm} + t_{t2_{post}}$$

i.e. when the segmentation across the post creation code part compensates for the overhead caused by the communication overhead.

Application Scope

Though the scenario (2 threads, 1 dependency) may seem unrealistic at first, the basis is quite common for function calls in a sequential program or, even more typical, event-based functions in average desktop applications. In all these cases, the total application contains (at least) one function that is invoked sometime in the main application block to execute a specific (recurring) task – frequently enough this is e.g. updating the data, the display etc. without having to provide return values to the invoking thread. It can be shown that the algorithm identifies these segments due to the nature of the invocation and the according representation in the graph.

Actual Behaviour

As discussed in more detail in the project document D4.2 "First Implementation Set: Execution Management" [36], the segmentation point in the post creation part of the code is not entirely clear and is influenced by the merge behavior of the pre-segmentation algorithm (i.e. the bottom-up part) – segmenting an individual node in the middle is not foreseen as such. Examining the current form of the algorithm, it will be noted that the segmentation will either take place right after post creation or right before read v, as the merge algorithm tries to keep consecutive operations together.

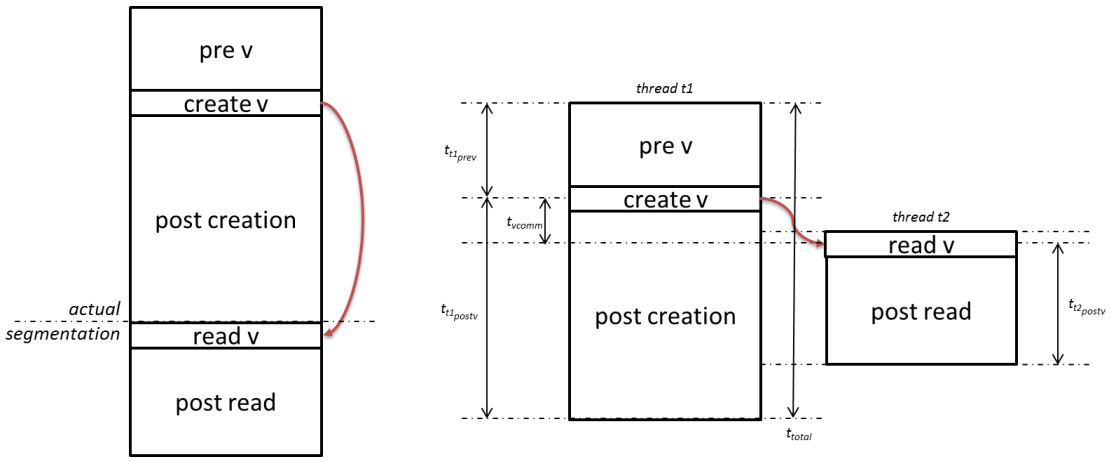

Figure 2: the actual segmentation creates a bigger t_{total} due to the first merge step of the algorithm

Accordingly, the actual execution time according to the current algorithm is

$$t_{total} = t_{t1_{prev}} + \max(t_{vcomm} + t_{t2_{postv}}, t_{t1_{finish}})$$

As opposed to the optimal case, the second thread's time prior to read v consists of pure wait time, i.e. underemployed resource. With respect to the optimal case we can note that

$$t_{t1_{finish}} = t_{t1_{postv}} + t_{t2_{prev}}$$

Thus, the algorithm's efficiency against the optimal case can be calculated as

$$eff_{loss} = \frac{t_{evaluated}}{t_{optimal}} = \frac{t_{t1_{prev}} + \max(t_{vcomm} + t_{t2_{postv}}, t_{t1_{postv}} + t_{t2_{prev}})}{t_{t1_{prev}} + t_{t1_{postv}}}$$

We can thus distinguish two cases

(1) $eff_{loss} = \frac{t_{t1_{prev}} + t_{vcomm} + t_{t2_{postv}}}{t_{t1_{prev}} + t_{t1_{postv}}}$

(2) $eff_{loss} = \frac{t_{t1_{prev}} + t_{t1_{postv}} + t_{t2_{prev}}}{t_{t1_{prev}} + t_{t1_{postv}}} = 1 + \frac{t_{t2_{prev}}}{t_{t1_{prev}} + t_{t1_{postv}}}$

In other words, the current algorithm provides good results with smaller post creation segments. As discussed below, segmentation should only take place when $t_{vcomm} >> t_{postcreation}$.

Conclusions

The segmentation algorithm, and in particular the merger part of the algorithm needs to be improved to respect the potential timing overlap between threads, i.e. segmentation of consecutive operations to compensate communication delays and increase resource utilization. As discussed above, the segmentation should try to reduce the result of formula ③ towards 1, which can serve as a cut-off criteria for the segmentation algorithm.

Notably however, the parameters of the segmentation depend on the layout of the destination infrastructure – in other words t_{vcomm} is influenced by the bandwidth and latency of the communication link between the possible thread hosts. That means that

$$t_{vcomm} = f(bw_{conn}, lat_{conn})$$

,where bw_{conn}, lat_{conn} are the bandwidth and latency of the underlying connection, respectively.

In addition to this, current operating systems create major delays when instantiating a new thread – even though S(o)OS tries to capture this by creating processing segments on a lower level than threads, the according cost for packaging, distributing and hosting the according code must principally be considered as an initial delay. This however affects only dynamic use cases, i.e. where the segments are distributed and initialized at run time and we will hence delay this discussion for now.

As we are currently only referring to a single connection across two threads, we need to assess the impact of having multiple threads with multiple different types of connections.

2) (Multiple) Bidirectional Dependency Between Two Segments

Typical function invocations are bi-directional, i.e. the calling instance requires a return value from the function called – often enough the code therefore behaves sequentially and is just written in an object-oriented structure. Typical examples of such invocations are

```
int a = f(x, y);
```

where the invocation could either point to an external library or a frequently invoked object's capabilities (cf. Figure 3.a). When analyzing the code behavior it will be noticed, however, that neither value is used directly at the entry points of either thread, so that the *actual* dependency looks more like depicted in Figure 3.b. It will be noted that the function calls involve a lot of indirect

dependencies due to the structure of invocations in particular in object oriented languages. In the compiled code this is generally noticeable as passing pointers to memory, rather than the values themselves, so resolution needs to distinguish between direct and indirect memory access.

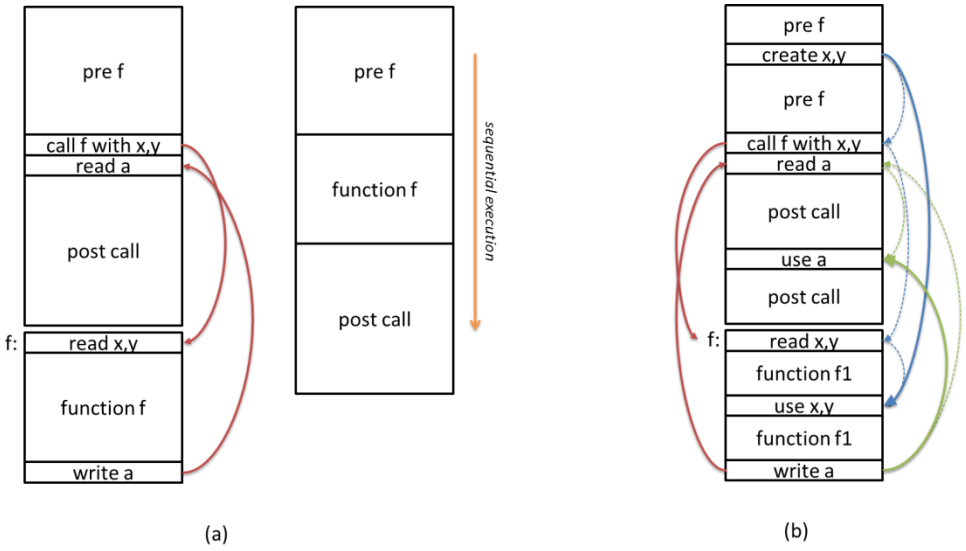

Figure 3: relationships between a main thread and function calls
(a: seeming dependencies from direct analysis; b: actual dependencies by deeper analysis - the solid lines represent the actual dependencies and the dotted line the indirect one, as interpreted by the analyser)

It is obvious, that bidirectional communication poses additional constraints on the timing behavior of their parallelized execution. This is effectively identical to having two consecutive cases like described in the preceding section. In Figure 4 we depicted the case of the bidirectional dependency under the assumption that both threads are started at the same time, and that communication delay is equal in both directions.

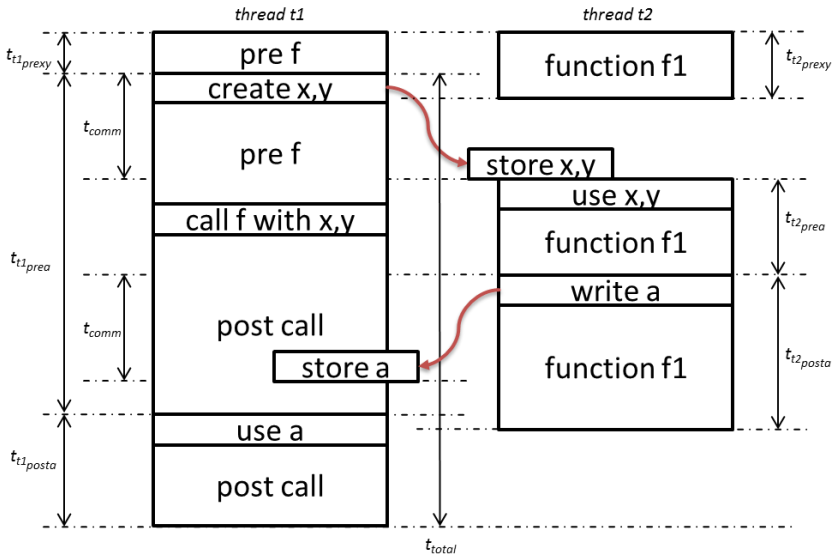

Figure 4: parallelising bidirectional dependencies

Notably, the communication overhead can cause delays on either side, as discussed for the single dependency case. This means however, that the calculation of the overall time is principally identical

to the one in the preceding section, even though we use a slightly different annotation here to reflect the full interaction

③

$$h_1(xy, a) = \max\left(t_{t1_{pre}}(xy) + t_{t1_{pre}}(a), t_{t2_{pre}}(a) + \max\left(t_{t1_{pre}}(xy) + t_{comm}, t_{t2_{pre}}(xy)\right) + t_{comm}\right) + t_{t1_{post}}(a)$$

④ $$h_2(xy, a) = t_{t2_{pre}}(a) + \max\left(t_{t1_{pre}}(xy) + t_{comm}, t_{t2_{pre}}(xy)\right) + t_{t2_{post}}(a)$$

and

$$t_{total} = \max(h_1(xy, a), h_2(xy, a))$$

It will be noted that this is a recursive extension of ① and ②, due to the symmetry of the two calls (respectively call and response). Even though it could be expected that the recursion holds along the line of

⑤ $$h(\vec{\imath}) = \max\left(\sum_{n=1}^{|\vec{\imath}|} t_{t[1]_{pre}}(i_n), l_{|\vec{\imath}|}(\vec{\imath})\right)$$

$$l_j(\vec{\imath}) = t_{t[2]_{pre}}(i_j) + \max(l_{j-1}(\vec{\imath}), t_{t[1]_{pre}}(i_j) + t_{comm})$$

$$l_1(\vec{\imath}) = t_{t[2]_{pre}}(i_1)$$

This holds also true for multiple encapsulated threads with according communication dependencies.

Application Scope

Calling an application and awaiting its response is a typical scenario in particular in object oriented programming models. Often enough the calls are executed synchronously, though the usage is effectively asynchronous (e.g. reading a data file where first actions can be executed with partial data, or calculating multiple vectors in a loop etc.). In particular mathematical applications with many function calls (towards a library) tend to have an tends towards having an apparent tighter dependency than actually the case when looking at the actual dependencies (cf. above) – the problem however in such cases consists in the identification of the actual dependencies from the ones visible in the dependency graph.

Actual Behaviour

Notably, the current algorithm exhibits exactly the interpretation of indirection problems discussed in the preceding section. Since the analyser generates the dependency graph on basis of variable and memory access, but discards them prior to the merger phase, the latter will merge along the wrong edges due to their typically shorter path-length. Shifting the resolution to the dependency graph generation however leads to leaving the loading operation unresolved which can lead to unwanted behavior when trying to resolve it. More importantly however, indirection is more typically expressing using indirect memory access, rather than directly sharing a register, which is currently not properly addressed by the analyser, as it takes the negative assumption that the memory could be altered by other segments, when in reality the memory address could be shifted to avoid overlaps.

In addition to this and as noted in the context of two segments with a single dependency, the segmentation algorithm (or rather the merging part of the analysis algorithm) tends towards keeping sequential operations together, thus reducing the probability of creating optimal segmentation between two communication edges. As depicted in Figure 5 the algorithm misses the opportunity to cut the calling segment at a lower point to make better use of the timing overlap (b). If the indirect dependencies are not resolved (a) the algorithm will tend towards suggesting a cut close to the call & response operation, thus leading to an almost sequential execution.

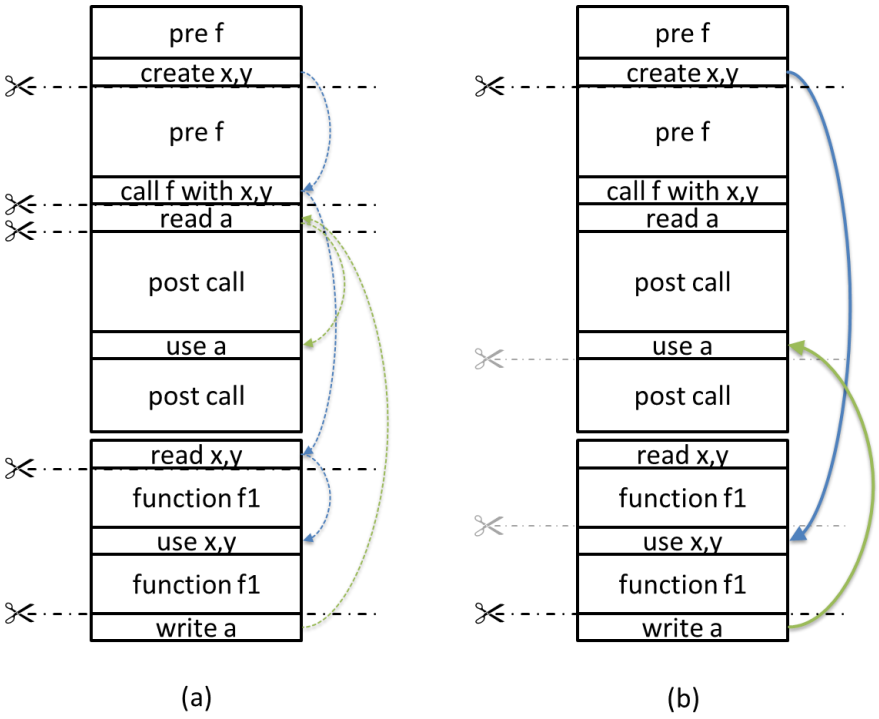

Figure 5: current segmentation results without (left) and with (right) resolved indirect dependencies

Notably, the algorithm identifies a relatively good segmentation in the resolved case (Figure 5.b) with cutting the code right under `use a;` However, the segmentation suggestion will prefer other segmentations to reduce the cut-cost (min-cut principle) – this is due to the fact that the timing distance for communication is not encoded properly (see also above). In order to assess timing better, it is therefore necessary to incorporate base principles from the real-time domain.

3) A Note on Multiple Segments, Multiple Dependencies

It can be derived from the preceding discussion that the timing function for multiple segments and multiple dependencies is recursively defined over the numbers of cuts (cf. ⑤). With increasing number of segments, however, the typical critical cutting segment $t_{t[n]_{postx/prey}}$ decreases in size, therefore reducing the impact from wrong segmentation as described in section 1). On the other hand, the wrong assessment of timing overlaps and the potential for wrong resolution of indirections again may lower the performance by increasing the risk for effectively sequential cuts.

The efficiency loss is obviously statistical and depends on the implementation choices, rather than on the type of application and shall therefore not be elaborated here – in particular since the base algorithm will be improved in the next iteration:

d) CONCLUSIONS

As discussed in the preceding sections, the analyzing and segmentation algorithm works quite well in identifying the relationships, in particular with growing number of segments (resources) and connections. As has been shown by Stoer and Wagner, runtime complexity of the segmentation algorithm is maximally $\mathcal{O}(|V||E| + |V|^2 log|V|)$, where V is the set of vertices and E the set of edges. With the segmentation being executed at maximum the number of times that resources are available we achieve a complexity of $\mathcal{O}(n|V||E| + n|V|^2 log|V|)$, where n is the number of resources available. Since it does not make sense to reduce the size of the segments beyond a certain point, the number of concurrent segment is also restricted by $|E|/c$ where c is a cost-constant for executing small threads – this is obviously identical to saying n = 1/c

The algorithm therefore is able to principally identify the right concurrent segments, yet tends to be too pessimistic in resolving indirections between dependencies, thus leading to a potentially too fine-grained dependency graph. The merging algorithm should therefore be extended to respect indirections occurring from indirect memory access – notably, it will have to be evaluated in the next iteration in how far the additional overhead for resolution of indirection impacts on the analysis performance. It is obvious, however, that the approach already shows a good segmentation performance for asynchronous calls, where the indirection is generally (more) properly resolved.

On the other hand, the merger tends to provide too tightly coupled segments that the segmentation algorithm will not attend to segment further, so that the timing overlap is misinterpreted (see above). The segmentation algorithm therefore needs to be enhanced to distinguish sizes of vertices and their impact on timing behavior, and assess the cost for concurrent execution more correctly.

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.

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 continuing 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 many-core processors; implementing many heterogeneous cores in the same silicon die. This implies that the same operating systems cannot run on different 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 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.

Those are the reasons that motivate us to design a set of mechanisms that allows the OS and the programs to distribute to different heterogeneous systems and adapt to the different architectures with only a moderate performance penalty compared to native binaries. As result of this the programmability is improved and code maintenance and adaptability to new and/or future systems and architectures is greatly simplified.

In summary, DISTae will allow to create a universal executable that is basically a package file that contains information about how to distribute it and facilitate the adaptation to the different architectures. The code analysis module will help to determine the partitioning of the code with the suggestions of the programmer making use of programming extensions. While preparing the execution the code has to be distributed and adapted to each computing units based on the decisions made by the resource manager and the scheduler. After it the program is ready to be executed, DISTae will take care of the communications and to translate the data types between the different representations of the various architectures involved in the execution.

With all these functionalities DISTae contributes in addressing some of the primary goals objectives of S(o)OS. GO#1 and GO#2 are addressed by the adaptation of code among the different architectures of an heterogeneous system and the automated distribution of the code segments respectively. DISTae can be automatically used by the code analyser module or by the programmer, suggesting which parts of the code should be distributed, addressing GO#3. One of the main DISTae

is the portability and adaptability of software, being able to use and combine simultaneously computations in small power embedded processors to HPC clearly contributing with achieving GO#5. Also some of the secondary objectives are addressed; although DISTae relies on the code analyser and the OS scheduler to know which segments to distribute where, it handles the protocol that transmits the executable and the communication during execution, using the adequate protocols contributing to PO#2. It also provides programming extensions by the use of pragmas that allows the programmer to suggest to the systems which code segments to distribute, addressing PO#4.

a) CURRENT CAPABILITIES

The needs for such a mechanism are mainly two; we want to migrate our software stack to a new architecture or we want to be able to run the same software in different architectures. In the history of informatics there have been a few attempts to implement a system that will enable to execute binary code for one machine to another kind of processor. There are a few techniques to achieve this propose:

EMULATORS

Emulators are computer programs that "duplicate" the functions of another computer, so that the behavior of the second system closely resembles the behavior of the first system. It focuses on the reproduction of the external behavior in contrast with other kinds of simulations:

- Apple Mac 68K emulator
  It is considered the first time that such a dual hardware architecture operating system had ever been successfully implemented. Basically it consists of a software emulator built into all versions of the Mac OS for PowerPC. This emulator permitted the running of applications and system code that were originally written for the Motorola 68k processor based Apple Macintosh computers.
  This emulator simulated the user ring of the Motorola 68EC040 instruction set with a 68020/68030 exception stack frame. Decoding each instruction and immediately carrying out a series of equivalent PowerPC instructions. Dynamic recompilation works by translating sections of the original code into faster PowerPC-native. This emulator was theoretically capable of emulating 680x0 code faster than any real 680x0 was capable of running it.

BINARY TRANSLATORS

Is the emulation of the instruction set (ISA) of a computer in another that has a different one. Sequences of instructions of the source binary are translated into the target platform.

- Rosetta
  Apple Rosetta is a lightweight dynamic translator that was part of early versions of Mac OS X for Intel operating systems that translated G3, G4 PowerPC and AltiVec vectorial instructions to x86 code.
  According to Apple, applications with heavy user interaction but low computational needs (such as word processors) are well suited to translation via Rosetta, while applications with high computational needs (such as raytracers or Adobe Photoshop) are not[4].

[4] Apple Inc. - Universal Binary Programming Guidelines, Second Edition, September 2009

- qemu
  It is a processor emulator that relies on dynamic binary translation. It is commonly used for testing operating systems and applications on different CPU architectures.
  It can run in many architectures and emulate many others. It simulates a virtual cpu and translate the binary instructions into C. Then the C code is compiled into native code and executed.

NATIVE

- fat binary (or multiarchitecture binary)
  This is a kind of executable file that includes the binary codification for various different architectures. On the execution of the file, the operating system loads into memory the correct binary code for the native architecture among the ones supplied in the executable file. The main disadvantage of this system is that the file is bigger and that it is not prepared for new architectures that were not taking in consideration in the first moment.
  Two Implementations:
  - Apple "Universal binary": This executables run natively on either PowerPC or Intel x86 based Macintosh computers.
  - FatELF: It is an extension of the ELF binary format used by most Unix-like systems. Additionally to the CPU architecture abstraction there is the advantage of binaries with support for multiple kernel ABIs and versions.
- JITC - Just in Time Compilation
  Usually computer programs have two modes of runtime operation: interpreted or static (ahead-of-time) compilation. Interpreted code is translated from high level language to machine code during every execution, while statically compiled code is translated just once. An hybrid combination of both is compiled in the local machine just before the execution. With this we obtain native performance with some advantages over statically compiled code, like handling of late-bound data types and the ability to enforce newer security guarantees. Microsoft's .NET Framework and most implementations of Java, rely on JIT compilation for high-speed code execution.

VIRTUAL MACHINE

It is a software implementation of a computer that executes programs like a physical machine. A complete implementation of a virtual machine allows the execution of an operating system. The resources and abstractions provided are limited and programs cannot "break out" of the virtual world. VM's allow the execution of multiple OS (and/or multiple instances of the same OS) but isolated from each other.
If the virtual machine uses the same ISA as the host machine, then the instructions can be directly executed in the CPU.

INTERPRETED EXECUTION

And interpreted program is a computer program that is not compiled ahead. It can be executed by:

1. executing the source code directly
2. translates source code into some efficient intermediary representation and then executes it
3. compiles it on run-time (JITC) to local machine code

The main disadvantage of interpreters is that when a program is interpreted, it typically runs more slowly than if it had been compiled. The difference in speeds could be tiny or great; often an order of magnitude and sometimes more. It generally takes longer to run a program under an interpreter than to run the compiled code but it can take less time to interpret it than the total time required to compile and run it.
Another key disadvantage of this solution is that an interpreter must be present on the user's machine, (as additional software) to run the program. But distribution of a program is easier as there is no need to adapt the code once the interpreter has been ported.

MULTI-ARCHITECTURE PROCESSORS

Another way is to create processors that can execute multiple ISA. This can be done in various ways:

- Incorporating two different processors in the same system, where each gets activated depending on the binary code to execute (First PlayStation 3 included the "Emotion Engine" of the PlayStation 2 to offer backwards compatibility).
- Translating the instructions by a microcode. Some processors can execute other ISAs by decoding the binary instructions and translating them to their native ISA. Most processors must actually decode their "native" ISA to internal micro-operations. Two examples of this are the Intel Itanium (it is an IA64 processor that can execute x86 instructions at a considerable performance penalty) and the Transmeta processors (they decode the x86 instructions into their internal VLIW operations).
- Some ISAs can be considered extensions (as most transitions from 32 to 64bit systems). Even if other changes are implemented, added to the extension in the size of the pointers and words, these usually constitute a mayor set of the original architecture (i.e. increasing the number of registers). This way the extensions, register size etc. only need to be deactivated for executing the older binaries. This is the case of the transition from IA32 (x86) to AMD64 (x86-64), or from PowerPC to PowerPC64.

But these mechanisms, even when they mitigate some of the problems, are not good enough for the current situation. The approaches based on emulation or interpreting the code have good portability but with a high performance penalty (sometimes a few orders of magnitude), while approaches that produce native code have good performance, but lack of adaptability for other architectures that were not considered in compilation time. DISTae is more near to JITC, it could be said that is a multiarchitectural distributed JITC as it distributes the code among the different computing units, even in heterogeneous systems that have multiple architectures, and handles the communication between the code segments.

b) EVALUATION CRITERIA & MEANS

In the next table a description of the different criteria applied for evaluating DISTae can be found. These criteria were selected because they allow us to evaluate the different some particularities described in the different execution phases described in the "evaluation criteria" project deliverable [35].

Table 2: DISTae evaluation criteria overview

	Name	Description
C1	Detect the code of the requested functions	Detect the DISTae pragma and the area of influence in the source code (the pragmas are inserted by the programmer or the code analyser). This evaluation criterion contributes to GO#3, PO#2 and PO#4.
C2	Extract the code of the requested function	After being detected a pragma the influence area has to be cut from the file and copied to a new one. This evaluation criterion contributes to GO#3, and PO#4.
C3	Generate the extra layer of code for the requested function	The extracted segment of code has to be wrapped with a software layer that handles the communications and translated the data types into standard architecture independent. This evaluation criterion contributes to GO#3 and PO#4.
C4	Detect the calls for the requested functions	The calls to the segmented code have to be detected correctly along the code. This evaluation criterion contributes to GO#3, PO#2 and PO#4.
C5	Create the necessary for the distributed call	The detected calls in C4 have to be wrapped into a special code that handles the communication with the distributed segments. This evaluation criterion contributes to GO#3 and PO#4.
C6	Send the distributed code to the correct distributed machines	Using the information of the scheduler and the resource manager, and making use of the adequate communication protocols, the code segments are distributed to the designed computing units. This evaluation criterion contributes to GO#1, GO#2, GO#3 and PO#4.
C7	Adapt correctly the distributed code in the distributed machines	Adapt to the native architecture the segment of code that has been distributed to the current machine, including the software wrapper that handles communication and data standardization. This evaluation criterion contributes to GO#1, GO#2 and GO#5.
C8	Generate correct calls at run-time	During the execution of the local executor, the calls to the segmented and distributed code must be detected and settle communication. This evaluation criterion contributes to GO#3 and PO#4.
C9	Adapt correctly the data types to	During the call to the distributed code the data types must correctly be translated into a standard representation. This evaluation criterion contributes to GO#1, GO#2 and GO#5.
C10	Adapt correctly the data types from	During the call to the distributed code the data types must correctly be translated from a standard representation to the native architecture. This evaluation criterion contributes to GO#1, GO#2 and GO#5.

C11	Many base systems	Test that the system can be set on top of different base system. This evaluation criterion contributes to GO#1, GO#2 and GO#5.
C12	Many architectures	Test that the system can distribute code to different architecture simultaneously. This evaluation criterion contributes to GO#1, GO#2 and GO#5.
C13	Correct final result	Test that the final result is correct.
C14	Theoretical performance gain calculation	Probe that there is a gain in performance by distributing and adapting the code in some cases. This evaluation criterion contributes to GO#1 and GO#2.

We expect the result of the evaluation of the previous criteria to probe the correctness of behaviour of DISTae and to contribute with some of the general criteria of S(o)OS for program.

c) EVALUATION

To test and evaluate the first implementation of DISTae framework a set of computers have been set with different architectures and different OS/kernels:

Table 3: overview over the test platforms for DISTae

Name	CPU	FREQ	ENDIAN	MEM	OS
distae-00	x86-64 (*2)	2,7 Ghz	LE	4GB	Ubuntu 10.10
distae-01	x86-64 (*1)	2,7 Ghz	LE	512MB	Ubuntu 11.04
distae-02	x86-64 (*1)	2,7 Ghz	LE	256MB	FreeBSD 8.2
distae-03	x86-32 (*1)	2,7 Ghz	LE	512MB	Ubuntu 11.04
distae-04	ARM11	533 Mhz	LE	256MB	Debian 6.0 “squeeze”

This is necessary to create different networks of heterogeneous computers to test the correct adaptation of code to different architectures, with the possibility of choosing any of them as “local” and the others are “remote”.

A set of tests, named Test-0 to Test-9, were implemented to test the evaluation criteria. These tests consist of a set of scripts and programs that check the different features of the DISTae framework (DISTae creator, DISTae remote, DISTae local) under different conditions. Criteria C-14 is formally evaluated as current implementation of the DISTae framework is a probe of concept, designed to test the viability and without performance in mind. In future documents real life performance will be evaluated.

Table 4: overview over the executed tests for DISTae

Name	Component	Criteria	Explanation
Test-0	Creator	C1, C2	DISTaeCreator parses original code and detect the programming pragmas and its area of influence and extract it.
Test-1	Creator	C3	DISTaeCreator extracts the segment of code that have to be distributed and is wrapped with a software layer that handles the communications and translated the data types into standard architecture independent.
Test-2	Creator	C4, C5	The calls to the segmented code are detected by DISTaeCreator inside the distributed and not distributed code and have to be wrapped into a special code that handles the communication with the distributed segment.
Test-3	Local, Remote	C6	DISTaeLocal distributes the code segments to the designed computing units which have installed DISTaeRemote.
Test-4	Remote	C7	DISTaeRemote adapts to the native architecture the segment of code that has been distributed to the current machine, injecting the software wrapper that handles communication and data standardization.
Test-5	Local, Remote	C6, C7, C8, C9, C10, C13	Test the full execution of a DISTae executable package and check that the result is correct.
Test-6	Local, Remote	C6, C7, C8, C9, C10, C11, C13	Test the full execution of a DISTae executable package and check that the result is correct in an environment with the same architecture but different base system.
Test-7	Local, Remote	C6, C7, C8, C9, C10, C12, C13	Test the full execution of a DISTae executable package and check that the result is correct in an environment with different architecture but similar base system.
Test-8	Local, Remote	C6, C7, C8, C9, C10, C11, C12, C13	Test the full execution of a DISTae executable package and check that the result is correct in an environment with different architecture and different base system.
Test-9	none	C14	Make a formal evaluation of the performance

EVALUATION RESULTS

TEST-0:

Status: Passed

Description: DISTaeCreator correctly parses original code and detect the programming pragmas and its area of influence (C1) and extracts it (C2).

TEST-1:

Status: Passed

Description: DISTaeCreator correctly extracts the segment of code that has to be distributed and wrapped it (C3) with the software layer that handles the communications and translates the data types into standard architecture independent format.

TEST-2:

Status: Passed

Description: The calls to the segmented code are correctly detected by DISTaeCreator inside the distributed and not distributed code (C4) and are wrapped into a special code that handles the communication with the distributed segment (C5).

TEST-3:

Status: Passed

Description: DISTaeLocal correctly distributes the code segments to the designed computing units which have installed DISTaeRemote (C6). For this DISTaeLocal must locate the computing units that are DISTaeRemote enabled, connect to them, and transmit the designated segments of code.

TEST-4:

Status: Passed

Description: DISTaeRemote correctly adapts to the native architecture the segment of code that has been distributed to the current machine (C7), injecting the software wrapper that handles communication and data standardization.

TEST-5:

Status: Passed

Description: Test the full execution of a DISTae executable package, DISTaeLocal connects to various DISTaeRemote enabled computing units, the code is distributed and adapted, then during the execution the calls to the distributed code are realized (C8) and the data must be translated to and from a standard architecture independent format (C9 and C10). The result is correct (C13).

TEST-6:

Status: Passed

Description: Test the full execution of a DISTae executable package. This in an extension to Test-5 in which many computing units with same architecture but different base system is used (C11). The result is correct (C13).

TEST-7:
Status: Passed

Description: Test the full execution of a DISTae executable package. This is an extension to Test-5 in which there are many computing units with different architectures (C12) but the same base system. The result is correct (C13).

TEST-8:
Status: Passed

Description: Test the full execution of a DISTae executable package. This in an extension to Test-5 in which many computing units with different architectures (C12) and different base system is used (C11). The result is correct (C13).

TEST-9:
Status: When available a distributed system, if correctly used and depending on the application, it can lead to great performance advantages.

Definitions:

- Program: P, composed of SS' to SSn and SP' to SPm
- Sequential Segment: SS
- Parallel Segment: SP, composed of SP1 to SPk
- Distributed Environment: D, composed of L and R
- Execution Units: E, composed of L and R
- Number of processors: N(E)
- Time to Create Thread: TC(E, S)
- Time to Join Thread: TJ(E, S)
- Time Execute: TE(E, S)

Description:

The total execution time T of a program P will be:

- Sequential program implementation:

$$T(P)=\sum_{i=1}^{n} N(SS^{(i)})\,TE(L,SS^{(i)})+\sum_{i=1}^{m}(\sum_{j=1}^{k_i} N(SP^{(i)})\,TE(L,SP_j^{(i)}))$$

  The total execution time is the sum of all the segments of the program multiplied by the number of times that it is executed.

- Parallel program implementation:

$$T(P)=\sum_{i=1}^{n} N(SS^{(i)})TE(L,SS^{(i)})+$$
$$\sum_{i=1}^{m} N(SP^{(i)})((\sum_{j=1}^{k_i}(TC(L,SP_j^{(i)})+TE(L,SP_j^{(i)})+TJ(L,SP_j^{(i)})))/N(L))$$

  The total execution time is the sum of the sequential parts plus the maximum execution time of each segment that can be run in parallel. The best execution time is defined by

Amdahl's law. A precise formula should contemplate the creation and destruction of parallel computing threads, memory locks etc..

- DISTae Program:
  An accurate formula for the execution time of a distributed application is too complex and difficult to calculate due to the large amount of variable and their complex inter relationship. There are many of these widely proposed in different research papers. Taken the formula that adapts better to the system, and modifying it to integrate the DISTae overhead (code distribution, code adaptation and data variable standardization) would be enough to calculate the total execution time.
  Intuitively it can easily be seen that maximum theoretical performance gain can be calculated (as in a traditional parallel/distributed system) with Amdahl's law. Compared to conventional parallel systems, DISTae executables have access to more computational resources thanks to the code adaptability (at the cost of some overhead), increasing the level of parallelism. It should be taken in consideration that even if a segment has a shorter execution time than the adaptation time, as the adaptation has to be realized just once, if the segment of code is executed many times it might worth to adapt it.

d) CONCLUSIONS

In the last pages we have tried to argument why we need a mechanism for dealing with multi-architecture systems that are more common each day. This is a very important element inside the OS architecture as it handles the divergence in the heterogeneity of computing architectures. Even if DISTae is in an early stage of development it has started to show that it is not a unrealistic approach. The first implementation preview shows that it is correctly functional and in future we will work to overcome the current limitations of this complex multi-architecture executable system.

3. RESOURCE DESCRIPTION

There is a need to decouple low-level heterogeneous resources from higher levels, and also to aid the development of programming models that hide the complexity of future architectures from developers. To achieve these goals the execution environment of applications has to be made resource aware. To make the environment resource aware, we need a way to describe the hardware components available in the system. The resource descriptions need to be able to capture both static (e.g. NoC topology) and dynamic (e.g. congestion of a NoC router) information of hardware components. The descriptions should also capture both the functional aspects (e.g. the number of pipe-line stages) and non-functional aspects (e.g. the delay of an operation) of a component.

We introduced the CλaSH hardware description environment in [41]; it focuses on the functional aspects of hardware architectures. It lifts the design process of hardware to a higher level of abstraction than offered by traditional tools and design environments. It allows us to capture and describe the complexity of contemporary and future architectures (far) more easily than possible in current design environments. The CλaSH can translate functional descriptions to synthesizable VHDL code through the use of a term rewrite system. This rewrite system exhaustively applies a set of meaning-preserving transformations on the functional specifications until it has reached a normal form from which translation to VHDL is trivial.

a) EVALUATION CRITERIA & MEANS

Given the complexity of today's hardware, we need a way of describing hardware components in a clear way, without sacrificing on accuracy. We therefore need a language and methodology that is abstract and comprehensive, so that we can capture the essence of the hardware in a clear way; but it should also be powerful, so that we can add detail when needed:

Abstract. The design and simulation framework should be abstract in the sense that one can define structure in itself, i.e., without the necessity to specify the details of concrete data elements or functionality of components. Features that the framework should allow for are, for example, polymorphism and higher order constructs.

Comprehensive. The framework should offer a specification language that is well readable and which has a clear semantics. Structural language constructs should be readily available such that the designer can quickly and effectively express the intention of the design in such a way that it is well recognizable for others as well.

Powerful. The framework should be powerful in the sense that the design can be dealt with on various levels, including the hardware architecture itself, the instruction set level, and the programming language level.

Usability for identification and adaption of code. The framework should be useful for the automation of code adaptation. It should be easy to extract information relevant for code optimization.

The evaluation of the resource description component will concentrate on various aspects of the functional hardware description language CλaSH. These aspects include: expressiveness of the language, readability/understandability and conciseness of a specification, support for designer productivity. The means of evaluation will typically consist of expert judgments and experience of users. We evaluate these

aspects in comparison to traditional HDLs. Also, properties of the specified hardware, such as longest path, clock frequency, energy usage, etc., will be compared for the different approaches towards specification of the same hardware.

From the above it follows that the means of evaluation will be mainly informal and conceptual. Testing and benchmarking will not be useful as means of evaluation, instead informal methods will be chosen by comparing concrete specifications and systems expressed in the specification methodology to be defined within S(o)OS with the same systems expressed in standard approaches. Thus, we will not evaluate actual hardware, but the effectiveness of hardware descriptions and of hardware description languages.

b) CURRENT CAPABILITIES

A Hardware Description Language (HDL) is any language from a class of computer languages and/or programming languages for formal descriptions of digital logic and electronic circuits. Well-known HDLs are VHSIC HDL (VHDL)[11] and Verilog [12].

A reason why these languages are not suitable for S(o)OS is that while these languages are very good at describing detailed hardware properties such as timing behaviour, they fail at generally expressing higher-level structural properties, such a parameterization and regularity. They may support some parameterization, but they are not general enough to cope with rapidly changing demands of new hardware designs.

Another problem with VHDL and Verilog is that the semantics of the language, and the correspondence between the description and eventually synthesized hardware, are unclear (especially to new users). In VHDL, some conditional signal assignments lead to multiplexers, others to registers (D-flipflop), and again others to asynchronous latches. So even though they are syntactically equivalent, the resulting hardware certainly is not. In Verilog, one can either define a signal as either a ‘wire’ or a ‘reg’, where the first implies a simple wire and the second a buffered (it can retain state) value. Although a wire always results in an actual wire in the input, certain ‘reg’ signals are not always translated to registers.

VHDL and Verilog specifications are executed within a simulator, this means that testing routines for hardware designs will usually have to be written in the same language as well, even though the languages are not ideally suited for this task. To alleviate this “burden”, most simulators offer a “foreign function” interface, to communicate with programming languages (usually C) that are better suited at specifying testing routines. This does however mean that a designer has to use two languages for an entire design cycle, which often have different syntax and semantics. Given the experimental nature of S(o)OS, we would want a single language that works well for both specifying the hardware itself and the testing routines.

We can conclude from the above that current industry standard languages do not meet the criteria of abstraction, comprehensiveness, and power that we would want in a HDL. It is for these reasons that we proposed a new language, CλaSH, in an attempt to make a more abstract, comprehensive, and powerful HDL.

Aside from general-purpose HDLs like VHDL and Verilog, there are also more specific Processor/Architecture Description Languages (ADLs) such as nML [8], and LISA [9]. These languages are targeted at specifying instruction-set machines, and allow a designer to define the instruction

set, the behaviour of the instruction set, instruction pipelines and memory architectures in a syntax-directed way. These ADLs seem highly usable for adaptation of code, as demonstrated by the ability to adapt a special retargetable C-compiler that yields optimally scheduled code given an nML processor description [10]. A downside of these ADLs is of course that they are not general enough to describe hardware components that are not instruction-set processors.

c) EVALUATION

The CλaSH language can directly capture the static properties of a hardware component in its description, these properties can be analysed using standard static analysis techniques. Because a hardware description in CλaSH is also a valid Haskell [7] program, they can capture the dynamic properties of a component through simulation/execution of the description using a standard Haskell compiler or interpreter. This also means that we can use the full-fledged programming language Haskell to describe our testing routines, meaning a designer does not have to switch between a different syntax and semantics every time he wants to test his design.

Although the functional aspects of a hardware component are directly captured by the structural nature of the language, non-functional properties are currently not (yet) included. The reason is that many non-functional properties, such as propagation delay and energy usage, are dependent on the manufacturing technology of a hardware component.

We compared our CλaSH language with the more traditional hardware description language VHDL. We documented this evaluation in two separate papers: [1][2], which describe the design and implementation of a dataflow processor. Here follow some important quotes from the conclusions in those papers:

Generally it can be said that with Haskell and CλaSH is it possible to describe complex designs with less lines of code (≈ 300 in CλaSH v.s. ≈ 1500 in VHDL), which simplifies both the actual design process but also verification and debugging. The CλaSH generated VHDL code is fully synthesisable and resembles the intended functionality, which was described in the Haskell code. Furthermore, the synthesis results showed that in terms of area and performance it could be, when taking clock gating not into account, even better than hand-written VHDL code.

Due to full support of Haskell's features like control structures and polymorphism in CλaSH, the processor could be described in a very compact way. In Haskell, different levels of abstraction can be used; in general the level of abstraction level is very high and close to the mathematical description of a design. Internal implementation details like timing are completely hidden from the designer. Summarising, we can say that it is possible to design a complex architecture using CλaSH with relatively little effort.

*We can derive from the quotes above that C*λaSH is meeting the requirements that we set out. It is abstract: in that we don't have to specify all the details. It is comprehensive: the design process is simple, and the resulting hardware matches the description. It is also powerful, as complex designs can be created.

Although the *C*λaSH prototype is already (partially) meeting the requirements we set out, we are currently working on an improvement in that makes mixing combinational and stateful component descriptions more comprehensive. At present, we use the Automata Arrow abstraction[4][5] to lift functions to components; these components are potentially stateful [3]. At present, the prototype

still uses the arrow notation as introduced for the Haskell programming language in [6], and not the notation we proposed in [3]. The new notation will allow us to mix stateful and combinational parts of a circuit in a comprehensive and uniform way, where the currently used notation forces us to move combinational parts to the component domain first. Once implemented, *C*λaSH descriptions will be even more comprehensive than they currently already are.

In order to improve the comprehensiveness of the functional descriptions even further, we intend to formalize the correspondence of the descriptions and the resulting hardware. Although extensive testing of the *C*λaSH prototype compiler has given us much confidence in its functional correctness, we are aware of certain corner cases where the resulting hardware does not match the descriptions. As the transformation system used internally by *C*λaSH was developed in an ad-hoc manner, we are unable to explain fully as to how these corner cases are currently giving incorrect results. We thus intend to make a formal proof of the transformation system, and potentially adapt the transformations, so that the resulting hardware matches the given description.

*C*λaSH is a general-purpose HDL; this means that it can be used to describe many types of hardware components, not just instruction-set machines. This generality hinders the ability to have a syntax-directed ability to identify structures and properties of typical components in an instruction-set machine: e.g. instruction set, pipeline depth, number of registers, etc. So for the purpose of identification for the adaption of code, the current *C*λaSH prototype framework is not meeting the requirements in a sufficient manner. However, even though a syntax-directed identification of structures relevant for code adaptation seams infeasible, we think name-directed identification methods offer a solution. We intend to provide description templates that adhere to a given naming convention, which an identification tool can process to extract the relevant information. This approach can also be extended to non-standard architectures, such as dataflow machines; machines for which languages as nML and LISA were never designed.

d) CONCLUSIONS

The evaluation results of the *C*λaSH prototype are promising, in that many of the evaluation criteria are (partially) met. For those aspects where the prototype is working subpar, potential improvements are suggested. We will consequently try to explore the given suggestion in the next design cycle (within WP2):

- As discussed in the previous section, although we designed a new notation that can uniformly combine combinational and stateful components, we still need to make this addition to the *C*λaSH prototype.
- To make the descriptions more comprehensive, in that the resulting hardware matches the description, we will attempt to make a formal proof of the transformation systems. And if shown that the current set of transformation is insufficient, we will add new transformations (and formulate respective proofs) to the *C*λaSH prototype.
- We will investigate the creation of design templates for instruction-set machines that are susceptible for name-driven structural/static analysis.

4. RESOURCE DISCOVERY AND MATCHING

The resource discovery and matching component provides the information about the current set of available efficient resources which are ready and appropriate to migrate and run a remote execution in a fully distributed system considering adaptation for many-core hardware platforms. Each node or processor can employ this component to exploit this kind of information about the other alternative resources and decide to distribute and map the overload executions to the discovered resources. Migration of a process must be cost effective therefore resource discovery should be able to contemplate the communicational and computational expenses of resources such as discovery communication overhead, network latency, in-chip or off-chip bandwidth consumption, etc.

Functionalities:

- Finding resources according to the application requirements
- Evaluating resources and extract the list of most efficient available resources for each particular resource demand.
- Ability for range-based and exact matching querying.
- Querying in dynamic environment including frequent node join, leave and failure

Main objectives:

- Scalability: The discovery communication overhead must be reasonable and system-size independent; it can lead us to achieve scalability for very large distributed systems.
- Addictiveness: there are various service discovery protocols which have been designed for a specific environment such as SDPs which is adapted for wireless ad-hoc network or SDPs for internet and web services. We are proposing a service/ resource discovery protocol especially for an environment with hierarchical level of resources. (A network with many-chips and many-core nodes).
- Heterogeneity of resources
- Efficiency: resource discovery should be able to provide a facility for the system to utilize all the potential resources in the system. It means that RD protocol must perform an accurate discovering process in order to match a resource demand with most appropriate set of resources.
- Reliability: RD System must be fault tolerant.

a) CURRENT CAPABILITIES

In a large scale distributed system which is composed of a diverse multitude of networked many-chips and many-core components resource discovery is a real challenge. It consists in discovering the processing resources that are required to execute distributed or parallel applications in such systems. There are many resource discovery models that have been developed in the area of large scale distributed computing with different advantages and disadvantages. These RD models include of information service components for GRID, centralized discovery [13] such as Condor, LDAP and DNS for structured overlays, Distributed discovery, Multi-cast Discovery, Federated UDDI [14], flooding-based discovery such as Gnutella [15], P2P discovery, pure DHT-based resource discovery [16] such as Chord, Pastry and Tapestry, Hierarchical discovery models like Glubus and MDS.

Several RD models have been proposed for producing a Grid information service which can categorize in two main approaches: centralized and distributed approaches. The easiest approach to create an information service is to employ a centralized server. As examples for this kind of RD

models we can mention about Condor [17] and BOINC [18]. The good advantage of these solutions is the simplicity to find all the resource information on the central server which makes the resource discovery latency reasonable. However these approaches are not scalable and fault tolerant.

Condors Matchmaker established a centralized architecture, where the descriptions of resources and resource discovery queries are sent to a predefined central matching server which is responsible for the resource matching job [17]. This centralized approach is only efficient for local area deployments for which Condor was initially designed. But for large-scale decentralized systems there is a central point of failure and a scalability bottleneck. To handle the fault tolerance issue, other proposals like PUNCH [19] propose to use replicated servers to maintain several replicas of the resource descriptions. However, the cost of replication will be significant if the scale of the Grid increases.

Another approach for doing the discovery in grid is using hierarchical servers. Globus [20] is a hierarchical architecture which offers a scalable information service called Monitoring and Discovery System (MDS). In MDS-2[8], a Grid is included of several resource description providers that register to index servers using a registration protocol. Resource requesters use a request protocol to query directory nodes to discover resource index servers and to obtain more detailed resource information from their resource description providers. The index servers form a hierarchical architecture. The top index server answers requests that the resources registered with its child index servers. This approach limits scalability, as requests trickle through the root server, which can easily become a bottleneck and consequently suffer from fault tolerance issues. Indeed, the loss of a node in the higher level of the architecture causes the loss of an entire sub tree.

MDS -3 is another version of MDS which uses tree-style hierarchy within a virtual organization (VO) to dictate node resource description dissemination in a Grid [20][21] . It is also employed in the Globus Toolkit for resource discovery and monitoring. The operation mechanisms of MDS3 consist in providing resource information by low level nodes and report it to the higher level nodes in the tree structure. The low level node runs an instance of a Grid Resource Information Service (GRIS), which gathers the relevant local resource description and reports it in its upper level to the parent nodes, which run Grid Index Information Service (GIIS) instances.

This process causes a single point of failure as well as possible network traffic near the root node as the size of VO increases. There are several other DHT-based implementations which have been pursued as alternatives to MDS for the storage and retrieval of resource description in computational systems. NodeWiz [22], Mercury [23] and SWORD [24], in particular have investigated about the issue of supporting multi-dimensional resource attributes and range-based querying.

Distributed RD approaches have been specially designed to provide a high level of scalability and fault tolerance required in large scale grids. The research work proposed by Iamnitchi [25] proposes a solution for using the benefit of the P2P resource discovery in Grids. The combination of P2P and Grid RD models would be desirable to build fault tolerant and large scale distributed systems. There are two kind of approaches in this filed which are based on structured and unstructured overlays networks. The first model used unstructured overlay networks with flooding based query propagation such as Zorilla [26] and Vishwa [27]. One of the advantages of this approach is the ability to perform resource discovery with a high expressiveness. However, the discovery systems

are not exhaustive and efficient. The response time of the queries is high Because of using flooding and blind search and also it would be difficult to do the discovery of rare resources. In NODEWIZ which is based on structured overlay networks, discovery can be exhaustive, particularly by deploying DHTs, which is more appropriate to locate rare resources. But the drawback is that the structured overlays suffer from a lack of expressiveness for the queries. Indeed, multi-dimensional queries or queries with dynamic number of attributes and range-base queries cannot be easily performed. in all we can say that for Grid computing , the traditional resource discovery systems, such as DNS and NIS [28], has the drawbacks of low fault tolerance, lack of scalability and reduced performance. And on the other hand the modern RD models such as MDS-X which are employed in computational grids despite their advantage over pure hierarchal systems, still suffer from the same basic problems of congestion and scalability.

The other common model for resource discovery is the RD systems for Peer-To-Peer (P2P) networks which offers a significant advantage over their hierarchal counterparts by the way of resistance to failure and traffic congestion. In particular, structured P2P systems based on Distributed Hash Tables (DHTs) such as Pastry and Tapestry [29] are very popular for file-sharing applications but not for sharing the processors. Moreover, typical structured P2P systems such as Chord [29] and Pastry [ref] are very sensitive to the dynamics properties of the network. They suffer from churn and temporary unavailability of the resources. It occurs in the case of the frequent leaving and joining of the nodes into the system. The research work in [30] proved that hierarchical overlays have the advantages of less discovery latency, less discovery message overhead, and scalability.

For unstructured type of P2P discovery we can use informed search or blind search. Blind search uses flooding which burdens the communication network heavily and requires limited hop distance settings and special precautions for loops. The best known system with flooding is probably Gnutella, which is also used by CORBA traders. In Gnutella the discovery requests are routed to all neighbor nodes of a given node. This keeps happening until the queries expire or until the matched resources are retrieved. The flooding mechanism creates a large volume of traffic for networks with many nodes, connections and resources. Federated UDDI is a P2P based approach which can perform discovery of static resource descriptions in efficient manner, but it does not support querying for dynamic attributes which makes sense in the issues related to the quality of service.

None of the current RD approach is designed for many-core environment and distributed computing which is considered as services on processing resources. Also there are some theoretical concepts and solutions for service discovery which generally have been implemented by different protocols for different environments and applications. However these main concepts have many advantages and disadvantages which conformable to the target applications could make trade-offs. Therefore it is very important to design a service discovery protocol which concentrates on the target environment characteristics and specifications and keep in mind that scalability is still an open issue for all kind of resource discovery.

Discovery process causes network load which deducts the total performance of the system and inconsistency between discovered services and real services which is caused by event driven or periodical updating service information that is not fault tolerant and on the other hand dynamic service discovery imposes latency and flooding to the system. Thus in current results we can generally say that we need to improve the discovery latency, deduct the discovery overhead and

enhance the reliability of services in a scalable method and finally we could expect the improvements such as reduced discovery loads and bandwidth consumption, reduced discovery time, improved accuracy, improved rate of service discovery, reduced rate of false discovery, totally improved utilization of resources and easy querying (flexible for single, multi and all search). In order to evaluate our expected results we could measure the indicators such as deduction of the total amount of discovery messages which needs to be transacted for each particular query, number of nodes that would be explored for a query, comparison between querying in network with low workload and congested network, etc.

Resource discovery provides an information service about the processing resources in the network which is a very important component of a distributed OS or any other emerging Cloud computing infrastructures. For the operation of a distributed OS in an environment which consists of dozens of thousands of many-chips and many-core nodes we need a very powerful RD component in case of scalability in large scale, adaptation with the many core environments, supporting strong and flexible queries (range-based and key-based queries) and supporting efficiency particularly for discovery latency. The current state of the art does not show a comprehensive model which fulfills all the requirements for our application case. Therefore in this research work in the line of SO(o)S project we propose a new solution for resource discovery to bring all the requirements together in one model.

b) EVALUATION CRITERIA & MEANS

The main criteria in this iteration have been described in the following table:

Table 5: Resource discovery evaluation criteria overview

	Name	Description
C1	Scalability	The capability of the resource discovery protocol to maintain its performance regardless of the network size. We must clarify whether the RD procedure is system size independent in case of performance, throughput, and overload or not. Resource discovery must scale up and scale out efficiently across multiple / many cores, multiple / many processors
C2	Overhead	Resource discovery creates an overhead for transaction of the query and reply messages between nodes which must be minimum. we analyse the communication overhead by calculating the average number of transacted messages for each particular query for different scenarios.
C3	Discovery Latency	The time that it takes for RD to transmit a resource request message and get reply between a requester and resource information provider which has the information about the target resource. This includes the time spent processing in any number of intermediate nodes for a round trip. For this metric we calculate discovery time for a different system size.
C4	Performance	We use simulation to calculate the percentage of discovered resources respect to the total number of potential resources in the system that could qualify the conditions of the correspondence query. We also calculate the percentage of invalid services

c) EVALUATION

In this section we elaborate on the experimental results regarding the resource discovery evaluation in multi/many core networked environments. A key factor in our evaluation is showing the functionality and scalability of the RD system. Also we consider other requirements such as efficiency, performance and reliability. To achieve these results we have setup a virtual networked environment with a number of multi core / multi-processor nodes where all the processors are enabled to invoke resource discovery module in order to find other possible alternative resource for the transaction of the overloaded processes to remote processors. The simulation experiments have been conducted based on COTSon Simulator. We simulated a networked environment with 15 multi-core nodes (each node has 2, 4, 6 or 8 cores) including the total number of 56 resources (cores).

According to the list of objectives we have established a set of experiment scenarios that are as follows:

1 SCALABILITY TESTS

Scalability for RD means the ability of the discovery mechanism to support a larger number of resources or a greater number of interoperations between the nodes or both. We analyze the RD salacity by performing the following tests:

Test1: Measuring discovery latency versus number of nodes, where one, a constant fraction of the nodes or percentage of the total nodes concurrently and periodically (with different intervals) generate query and tiger resource discovery to get information about other nodes in a congested network. We analyze the impact of the network size on the discovery delay which is a response time to get information. To achieve scalability, discovery delay should not have a big variation when we change the size of network.

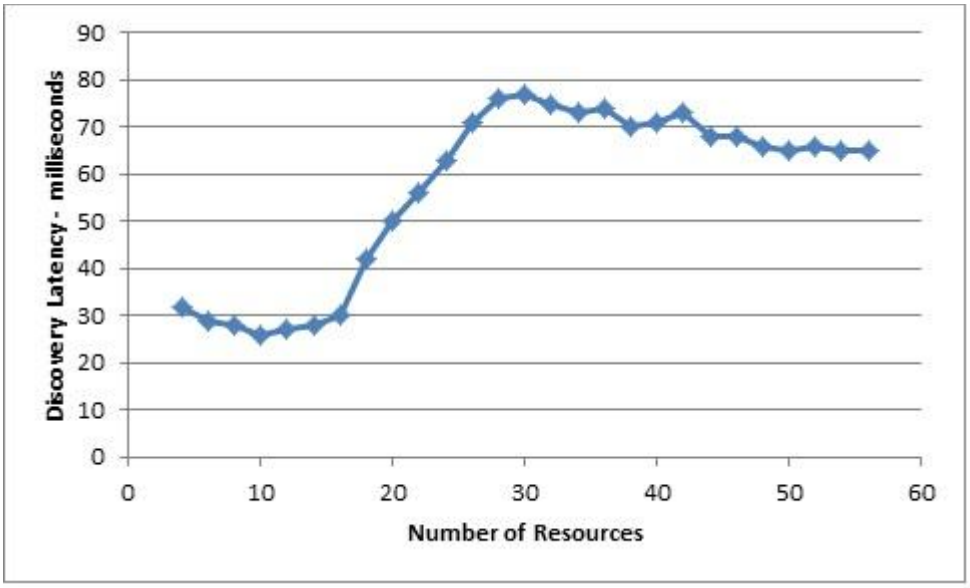

Figure 6: Average Discovery Latency for one requester with interval 30s

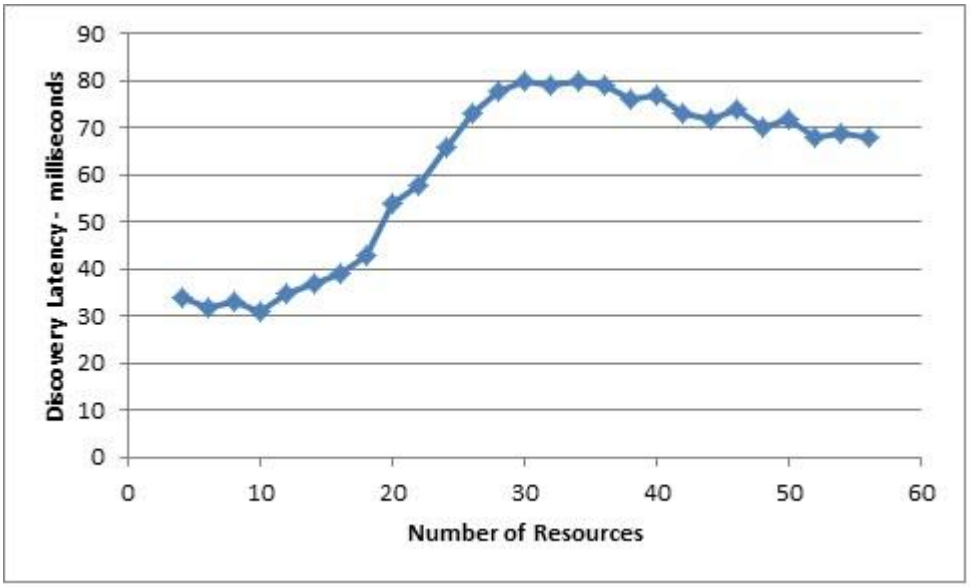

Figure 7: Average Discovery Latency for 25% of nodes as requesters with interval 30s

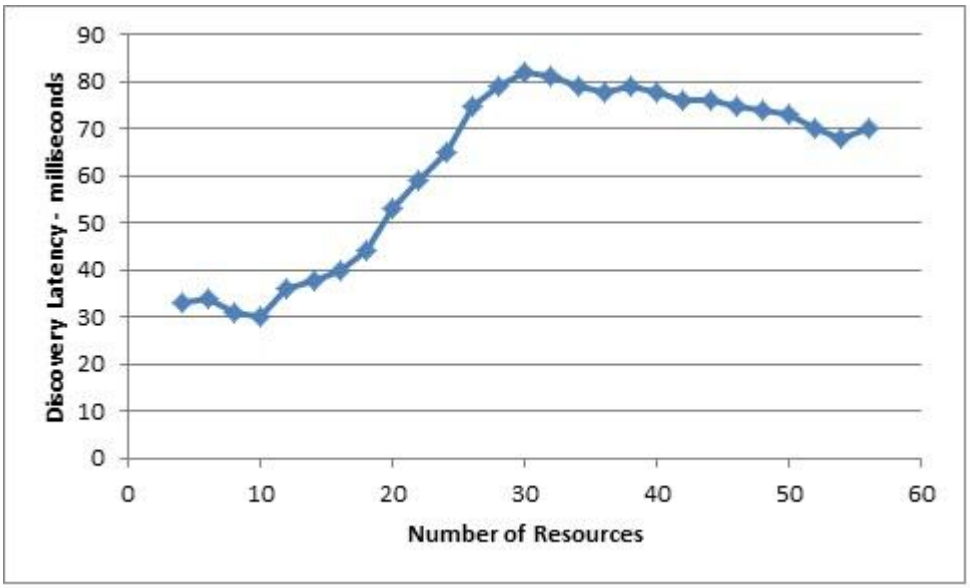

Figure 8: Average Discovery Latency for 40% of nodes as requesters with interval 30s

We evaluate the impact of the network size on the resource discovery latency for different number of requesters. Each of the requester submit a query to the system every 30s. We have run 100 RD queries for each given network size. The minimum query dimension is 4 and the maximum number of attributes is 20 we also vary the number of desired resources for each resource discovery query from 1 to 4. The properties of each query and also the network topology are random but we established a fixed number of resource information providers as remote directories for our simulated cluster. In Figure 1, 2 and 3 while we increase the number of resources the discovery response time starts to increase but for large number of resources it remains almost constant.

In all these experiments we almost see the same behavior. This results show that discovery latency is independent from network size which means that for the large scale network we could still have a reasonable response time for resource discovery queries.

Test2: Measuring the average of discovery overhead versus number of nodes: Analyzing the impact of network size on the average number of transacted messages for each discovery process.

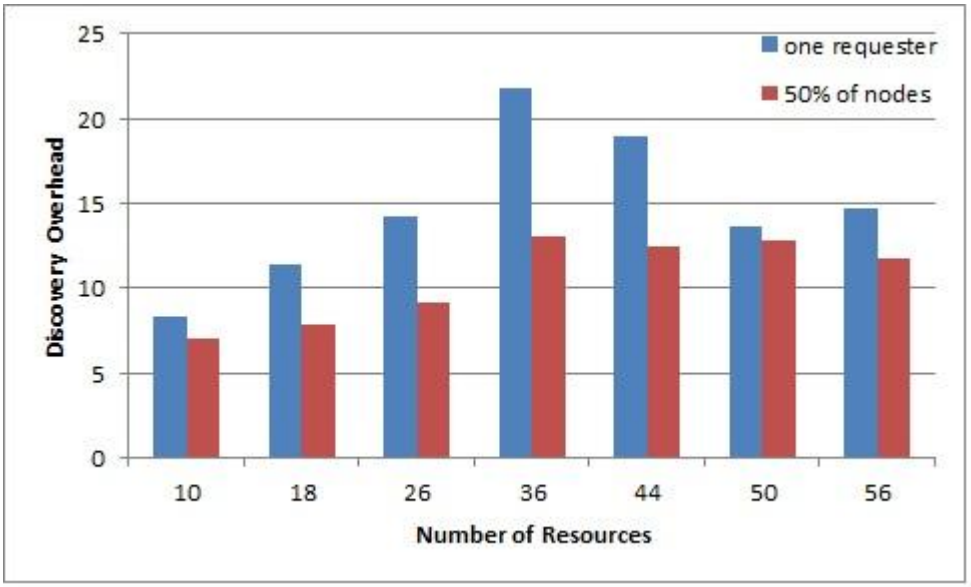

Figure 9: Average Discovery Overhead in different network size for 1 requester and 50% of nodes as requesters with Query Rate=.033/s

Figure 4 plots the discovery overhead of our system for different network sizes in all two configurations. in the first configuration only one of the nodes generates a random query every 30 seconds. by increasing the network size firstly , the overhead which is the average number of discovery messages per query increases until it gets to a threshold and then it decreases slightly and remains almost constant for large network size. By increasing the network size the RD system is required to forward the queries to other remote directories in the system to find any resource around the network. Therefore we could expect to see growing of the overhead along increasing the number of resources. The overhead after reaching the maximum starts to reduce which is due to the increasing of the replication of the resources and increasing the number of remote directories that provide the information about other alternative resources. Interestingly in the second configuration that we have conducted for a congested network with 50% of the nodes as concurrent requesters the results shows a significant decrease in the discovery overhead. The reason is that when we perform several queries concurrently by different distributed RD components, each intermediate remote directory stores the results of the successful discoveries for a specified period of time, so we could use the result of similar queries for other requesters that help to decrease the number of forwarding and query dissemination compared to the first configuration. In both configurations by large changing in the network size the overhead variations is not too much. It is small or constant, therefore the overhead should not be dependent on the network size which means that our RD solution is scalable.

2-EFFICIENCY TESTS

For the purpose of efficiency evaluation of our RD system, we will consider the RD properties such as Discovery Accuracy, Discovery Flexibility, Discovery Cost, Bandwidth Cost and Performance in the designed test scenarios. Discovery Accuracy measures the efficiency of the search mechanism. Since responses to a query may also include results that are not directly related to the query, it is important to distinguish them from exact matches. Therefore, assuming that there exists at least one

exact match for a query, search Accuracy represents the ratio of the number of matching results returned to the total number of query results . When an exact match is difficult to determine, Search Accuracy may also be evaluated based on the degree of relevance of the returned results to the query. Clearly, there exists a relationship between Search Accuracy and the number of nodes that are queried. As the number of queried nodes increases, the probability of a match being found also increases. In our experiment we will measure the rate of service discovery which is the ratio of the number of discovered resources to the total number of resources in the network that are able to fulfill the conditions of the correspondence query.

We provide flexibility for resource discovery by allowing the user components to do both range-based (partial matching) and key-based (exact matching) queries. There are some search techniques that only allow resources to be searched by their names or keys. The drawback of this is that users have to know the exact name of a resource in order to get a matching response. Richer search schemes, on the contrary, allow searching with keywords and thus provide far more flexibility to the user in specifying a query.

The performance of RD refers to the user-perceived latency of a search operation. In other words, it measures the time taken by a search operation to return a set of responses. While a centralized directory can pinpoint the exact location of a resource, a pure decentralized system may possibly have to initiate a system-wide search to determine the same search results. In addition to the time required for each node to process queries, time is also required to set up connections to the next node to which the query is to be forwarded. In such a case, the time required for a search operation will increase with the increase in the number of nodes in the system. In general, replication and intelligent routing mechanisms help increase the performance of search operations. It is important that we take care of the discovery cost for RD system. It means that in addition to decreasing the number of transmitted messages for a query, we need to reduce the number of nodes that actually process the query.

3 RELIABILITY TEST

RD system must be fault tolerant to work in dynamic environment with frequent node leaves, joins or failures. There is also a contention between concurrent requesters to grab resources which might be a reason for discovering invalidated resources. We could perform the below experiment to evaluate the reliability of the RD system.

Experiment: Measuring the rate of false service discovery which is the ratio of the average number of invalid discovered resources to the total number of discovered resources per query for different total number of resources

But since in this iteration we did not concentrate on RD for unstructured overlay and dynamic environment and we assumed that we don't have resource failures we will do this test in the next phase.

d) CONCLUSIONS

In a large scale system that we have a pool of variable processors, the enabling technology for enhancing the whole throughput of the system is resource sharing which means that for the overloaded processors we can migrate the overloaded processes to other potential processors in the network. But before resource sharing, resource allocation and execution migration, we need to find

resources and locate them. Resource discovery as a component of a distributed OS will be employed to discover an efficient set of available processing resources which are matched with the application requirements. The discovery latency has direct effect on the cost of migration and execution migration is not beneficial when resource discovery cannot provide information services in the reasonable time. Unlike resources discovery for various purpose and other domains, resource discovery for large many core environment is so sensitive to the discovery performance and it could be useless when it cannot satisfy the minimal parametric conditions of the system environment.

Another basic problem of resources discovery in many-core environments is scalability which is a generic challenge for all the research works in the area of resource discovery. The scalability problems refer to the methods for description of resources and the discovery procedure. These mechanisms must propose techniques and algorithms that efficiently be extendable for various number of resources. On the other hand the generated discovery overhead must be independent from network size. This is not fully attainable but we can have an effort to keep the discovery overhead almost constant with increasing the number of resources.

In this iteration we have presented a new resource discovery solution which is scalable and efficient for future many core systems with huge number of resources. We use Hierarchical Distributed Hash Tables (HDHT) to establish a distributed system with hierarchical resource description and we designed a probability-based Any cast mechanism that use DHT to locate resources in hierarchical levels core-chip-node-network. The results show the proposed RD supports efficiency and scalability for discovery in distributed systems.

5. CLUSTERED EDF SCHEDULING

As highlighted in the project deliverable “First Set of OS Architecture Models” [34] in S(o)OS we seek a novel architecture of the Operating System which is distributed across the many nodes that may compose a many-core system. Though, differently from [42], we also identified the possibility to keep an OS instance for managing multiple cores, for example in the case of a multi-core cache-coherent tile within a tile-based many-core chip. Or, we also identified the possible scenario in which multiple Operating System instances residing in multiple tiles might be managed in an SSI-like fashion, composing something resembling a single virtual OS spanning a wider set of computing resources. Differently from the SSI experience, in which the additional latencies for the communications among the nodes may have a greatly adverse impact on the performance of an application which was not designed for distributed computing, such an approach within the same chip may be far more viable thanks to the highly reduced overheads that are needed for in-chip communications (see also section II.6).

In this context, we are experimenting with kernel-level mechanisms increasing scalability of the scheduler with reference to the management of multiple (potentially tens) cores. Therefore, in the following we summarize our findings in the context of enhancing scalability of a global scheduler realizing an Eearliest Deadline First (EDF) policy in the Linux kernel. The complete discussion can be found in the paper recently appeared at OSPERT [31].

Linux supports global and partitioned scheduling through its real-time scheduling class, which provides

SCHED_FIFO and SCHED_RR fixed priority policies. Recently, the SCHED_DEADLINE policy was proposed that provides Earliest Deadline First scheduling with budget control. Here we give a brief overview of a new implementation for global EDF scheduling which uses a heap global data structure to speed-up scheduling decisions. We also compare the execution time of the main scheduling functions in the kernel for four different implementations of global scheduling, showing that the new implementation is scalable and efficient.

Both SCHED_FIFO and SCHED_DEADLINE scheduling classes already support global scheduling. However, to reduce memory contention and improve locality, the logical global queue is implemented using a set of distributed run-queues, one for each CPU. Tasks are migrated across CPUs using the so called push and pull operations. When a task is activated on CPU k, first the scheduler checks the local runqueue to see if the task has higher priority than the executing one. In this case, a preemption happens, and the preempted task is inserted at the head of the queue; otherwise the woken-up task is inserted in the proper runqueue, depending on the state of the system. In case the head of the queue is modified, a push operation is executed to see if some task can be moved to another queue. When a task suspends itself (due to blocking or sleeping) or lowers its priority on CPU k, the scheduler performs a pull operation: it looks at the other run-queues to see if some other higher priority tasks waiting on other runqueues can be migrated to the current CPU. Pushing or pulling a task entails modifying the state of the source and destination runqueues: the scheduler has to dequeue the task from the source runqueue and then enqueue it on the destination runqueue.

a) EVALUATION CRITERIA & MEANS

The evaluation criteria we use to evaluate scheduling policies for real-time workloads are as follows:

the deadline miss ratio, i.e., the percentage of jobs of a task that missed their deadline, over some observation horizon;

- the *laxity* of a real-time task, i.e., the difference between the absolute deadline of the tasks' jobs and their absolute finishing times; a positive laxity means a job finished ahead of time with respect to its deadline, whilst a negative value means the job missed the deadline; for soft real-time workloads, where deadline misses may be tolerated, the laxity adds the information about how late a task completed; this is usually compared with the task period, thus we normally use the normalised laxity, i.e., the laxity divided by the task period;
- overheads of the scheduler, i.e., the time wasted by the CPU(s) to select which task to schedule where and when; this may be measured in terms of microseconds or, while in the kernel, more conveniently (efficiently) in terms of CPU cycles;
- the number of context switches among tasks; this may be relevant when different scheduling policies impose tasks to preempt each other a significantly different number of times;
- percentage of cache misses: different scheduling policies may behave differently on the side of how many context switches are imposed over the tasks set (consider for example preemptive policies versus non-preemptive ones); also, when considering global or clustered scheduling, if a task is moved from one core to another by the scheduler, then it may incur additional cache misses for filling the cache(s) of the new core with its own data;
- cache-related overheads due to context switches, i.e., the additional time it takes for tasks to access their data as due to the additional cache misses as compared to when the same task was running in isolation.

In what follows, only a short summary of the main results about clustered and global real-time scheduling on many-cores is reported, while a more extensive evaluation may be found in [31]. Therefore, the metrics over which we focus below are the normalised laxity and the scheduling overheads.

SCHED_FIFO push and pull operations are quite efficient and scalable. In order to compete with them we modified the original SCHED_DEADLINE implementation (called original from here on) following an incremental approach. We focused on push operations, leaving pull operations as future work. The problem with original was a complete loop through all cores for every push decision, searching for a suitable CPU (the one which is running the task with the latest deadline among the m executing tasks on a m CPUs system) for a to-be-pushed task.

As a first step we argued that on a system with a large number of processors and relatively light load, many CPUs are idle most of the time. To improve the execution time of the push function we used a bitmask (fmask) that stores the idle CPUs with a bit equal to 1, trying to push a just woken-up task on an idle CPU first. However, when the system load is relatively high, idle CPUs tend to be scarce. Therefore, we introduced a new data structure to speed-up the search. We implemented, using a simple array, a max heap (heap) to keep track of deadlines of the earliest deadline tasks currently executing on each runqueue. Deadlines are used as keys and the heap-property is: if B is a child node of A, then deadline(A) ≥ deadline(B). Therefore, if there is no idle CPU where to push, the

node in the root points to the CPU currently running the task with the latest deadline, among all CPUs. Therefore, the deadline of the local just woken-up task (or the one being preempted by it) can be compared the directly with the deadline of the task running over the CPU at the root of the heap. If the former is more urgent than the latter, then the push operation actually occurs, and the task on the remote CPU is preempted and goes to sleep.

b) EVALUATION

We conducted our experiments on a Dell PowerEdge R815 server equipped with 64GB of RAM, and 4 AMDR OpteronTM 6168 12-core processors (running at 1.9 GHz), for a total of 48 cores.

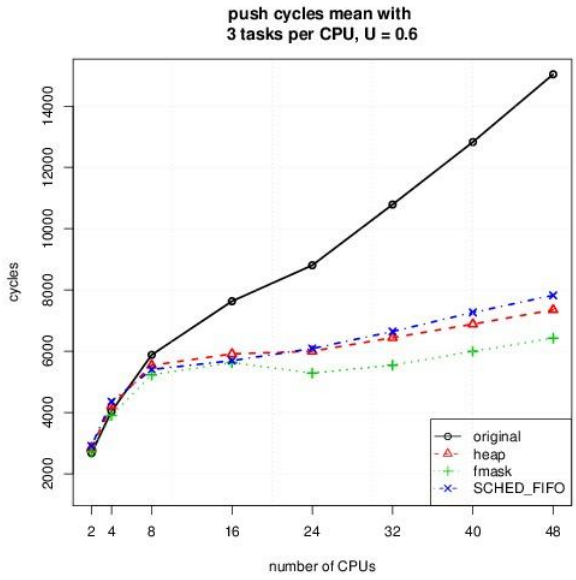

Figure 10: Number of push cycles vs. number of CPUs for average loads of 0.6.

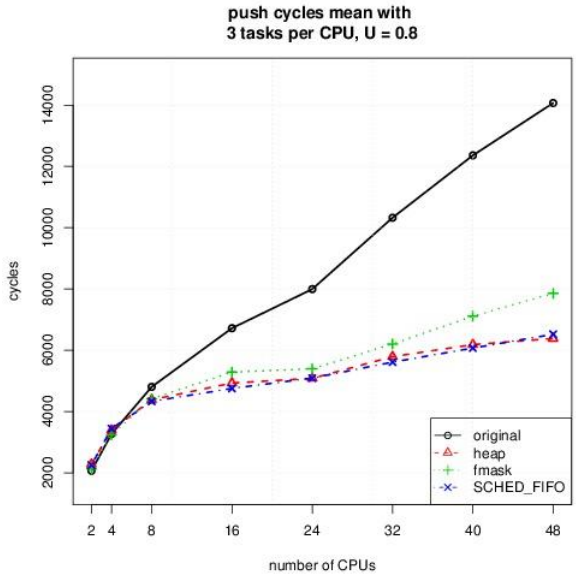

Figure 11: Number of push cycles vs. number of CPUs for average loads of 0.8.

From Figure 10 and Figure 11 it is clear that the overhead of the original implementation of SCHED_DEADLINE increases linearly with the number of processors, as expected, both for light load

and for heavier load. In fmask, we added the check for idle processors. Surprisingly, this simple modification substantially decreases the overhead for both types of loads, and it becomes almost constant in the number of processors. For a light workload, fmask is actually the one with lowest average number of cycles; this confirms our observation that for light loads the probability of finding an idle processor is high. For heavier workloads, the probability of finding an idle processor decreases, so the SCHED_FIFO and the heap implementations are now the ones with lowest average overhead. Notice also that the latter two show very similar performance. Similar performance figures were obtained varying the number of tasks in the system (Figure 12 and Figure 13).

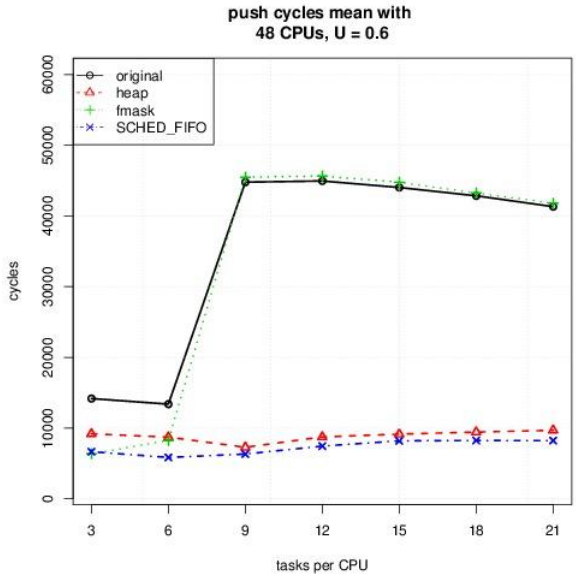

Figure 12: Number of push cycles vs. number of tasks per CPU for an average workload of 60% and 48 CPUs.

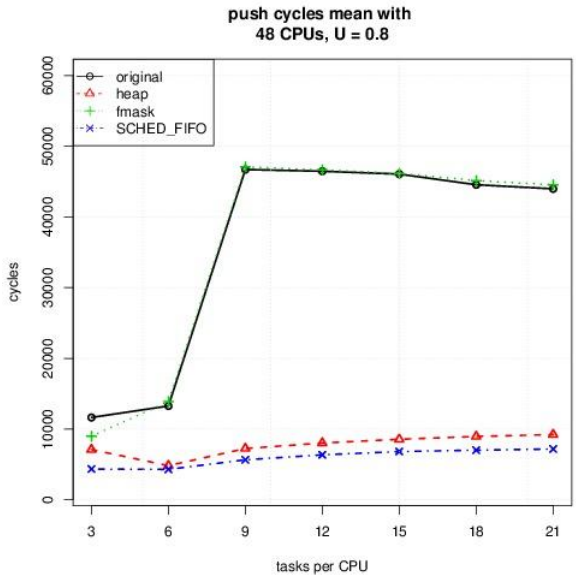

Figure 13: Number of push cycles vs. number of tasks per CPU for an average workload of 80% and 48 CPUs.

In order to justify the effort spent on finding right data structures to efficiently manage global scheduling on multi-core systems, we also performed an experimental comparison of various multi-processor scheduling algorithms by running synthetic workloads of real tasks on the same Linux

system. An excerpt of our outcomes is depicted in Figure 14. The metric considered is called normalized laxity and it is an estimation of how big the laxity (relative deadline minus response-time) of the various tasks is, compared to the task period. For each job of each task, this is given by the actual laxity divided by the task period. The average of all the per-job values gives the normalised laxity of the task, while the average among the laxity of all the tasks in a task-set provides the normalised laxity for the task set. It is evident that global and clustered strategies (both for EDF and RM, first and second couple of curves) differ in a negligible way from each other, and that they tend to lead to generally higher laxity values (better performance) than partitioned ones (cyan and yellow curves).

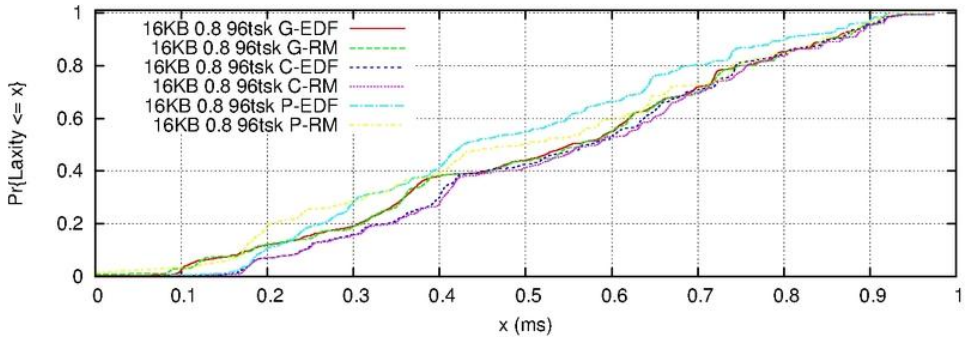

Figure 14: CDF of the normalised laxity for all the tasks as obtained under various scheduling policies, with U=0.8 and 96 tasks

c) CONCLUSIONS

As can be seen from the shown evaluation results, the overhead of implementing global EDF is comparable (and sometimes even lower) than implementing global Fixed Priority even within large multi-core systems. Moreover, our work was directed towards improving push operations only. However, migrations may happen also through pull operations. By using appropriate data structures for pulling out tasks from queues, we could speed up also this phase.

We were forced to omit here the whole set of experimental results for space reasons, but it can be stated that global and clustered algorithms are a viable solution for multi-core platforms with a high degree of cache sharing. Partitioned scheduling requires a potentially costly operation of allocation of tasks to cores, and it may not manage to make use of the whole computing power available on a system. On the other hand, global scheduling has the ability to perform automatic load balancing dynamically at run-time, achieving better flexibility and easiness of resources allocation and mapping. As already highlighted above, here global does not mean absolutely global across all the cores composing a massively parallel system, but it should be interpreted as globally among a set (or cluster) of cores over which there may potentially be some sort of cache coherency (e.g., the ones residing in the same multi-core tile). This is referred to as clustered scheduling as well.

6. COMMUNICATIONS IN TILE-BASED SYSTEMS

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 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 can be exploited to realize fast core to core communication primitives; these can be made available to applications directly through a user-space library; alternatively, a network driver may be realized that implements TCP/IP-based communications exploiting this local fast memory;
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.

In the latter case, 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.

The TCP/IP primitives realized on top of the internal in-tile fast memory surely provide a familiar environment for the programmer, who can use sockets, and the advantage of multiple applications being able to communicate across cores. However, the overheads added by the TCP/IP stack itself would be considerable, also observing that TCP/IP would wastefully replicate reliability features that are already available in hardware over for NoC-based communications. One possible direction of expansion of the evaluation of the pipelined application described below, is the one to replace the underlying use of RCCE (a message based communication library for the Intel SCC prototype) with this socket-based communications, in order to compare and estimate the extent of these overheads.

In order to compare the potential performance of the above communication means, we performed a few measurements on the Intel SCC prototype platform, which, even though not incorporating an exceptionally high number of cores (only 48), it exhibits the special capabilities mentioned in the above primitives, except the ones related to the first point, because the cache memories of the 2 cores within each tile cannot share a coherent view of the main memory.

a) EVALUATION CRITERIA & MEANS

As described in section 5.B of the Project Deliverable D5.3 "First Set of OS Architecture Models" [34], we set up a synthetic pipeline application scenario to study the effects of using various memory approaches and ordering. The primary metric that we used to compare was the end-to-end computation time that the pipeline took to process data – from the time the first stage starts on the first chunk of data to the time the last stage finishes processing the last chunk.

The factors whose effects we tried studying were:

- Inter-stage (or inter-core) communication overheads
- Type of memory used
- Overheads induced by absence of caching, and explicitly maintaining coherency in shared memory.

b) CURRENT CAPABILITIES

Note that the message-passing primitives as realized in the RCCE library by Intel are quite generic, and they allow to send and receive data only towards buffers in the main memory. Therefore, these primitives imply additional copies of the data to be exchanged as compared to what might be strictly necessary, even though the operations towards the shared memory are accelerated by the local caches. Further, these primitives are blocking, though there has been user-contributed non-blocking versions of these primitives that are available. In either case, they present the inability to run more than one application that uses these local-buffers.

As highlighted previously, the use of message-passing to move data leads to multiple copies and increased load on the NoC. An alternative is the use of shared memory, which significantly reduces this at the cost of having to manually maintain memory coherence across cores.

In our experiments we created a pipeline that could be configured to use either message-passing, uncached shared memory or cached shared memory for passing data between stages. In case of the shared memory models, the RCCE message passing capabilities are used to pass pointers containing the location of the data in shared memory regions. Table 6 below indicates the nature of these approaches in implementing the pipeline.

Table 6: approaches towards implementing the pipeline

Memory type	Communication characteristics	Caching characteristics
Private memory	High volume – entire chunks transferred	Cached in both L1 and L2
Shared uncached memory	Low volume – only tokens (i.e., pointers to start of data in shared memory)	Cached neither in L1 nor L2
Shared cached memory	Low volume – only tokens (i.e., pointers to start of data in shared memory)	Cached both in L1 and L2. Additional overhead in flushing due to no coherency mechanism.

In addition, the ordering of the cores could also be configured to be either a trivial ordering in terms of ascending physical core IDs or to reduce the routing distances between subsequent cores.

The pipeline processes the input data (residing into the main memory) in chunks of the same (configurable) size, which are subsequently sent to all the stages. Each stage is deployed on a different core. While a stage computes a chunk, the previous stage in the pipeline will be computing the next chunk, and the next stage will be working on the previous chunk. All communications from a stage to the next one require the same amount of data to be transmitted from one stage to the next one, which is equal to the chunk size, in case of the private memory model, or equal to 32 bytes, i.e., the size of the token which contains the pointer, in case of the shared memory models.

c) EVALUATION

Figure 16 and Figure 15 plot the variation of time against the chunk-sizes and input data-sizes respectively. It follows from these that the end-to-end time increases as the data-size increases, and further that the increase in chunk-sizes also increases the end-to-end time (as this adds overheads in the message-passing of the data from stage to stage).

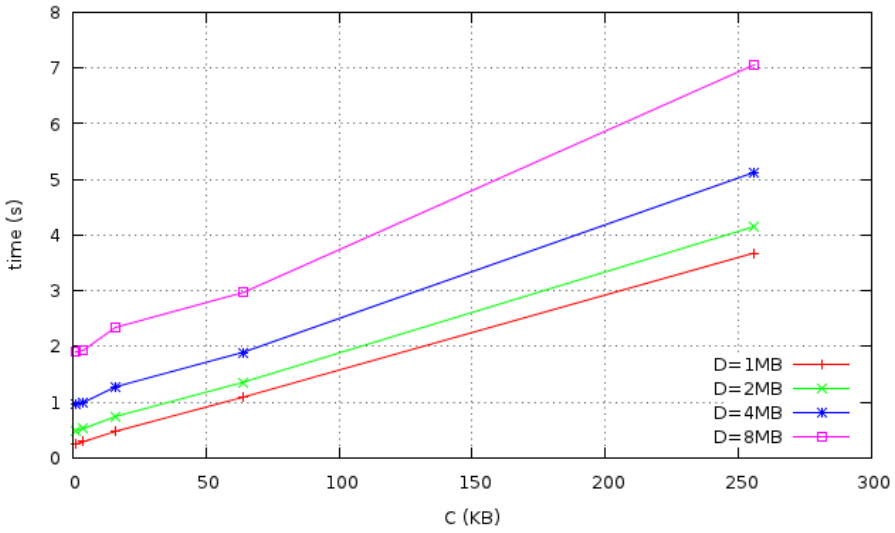

Figure 15: Chunk size against time for private memory implementation

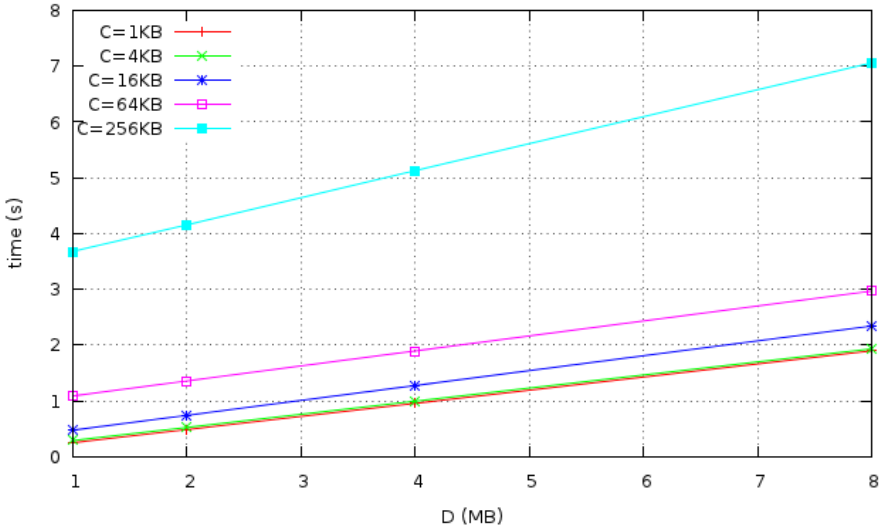

Figure 16: Input data size against time for private memory implementation

Figure 17 and Figure 18 similarly compare the chunk-sizes and data-sizes against overall end-to-end processing times for the shared uncached memory implementation. The lack of caching is apparent from the constant time it takes to process a given input size independent of the chunk size, and the doubling of time as input size doubles – since every access goes to the off-chip memory banks.

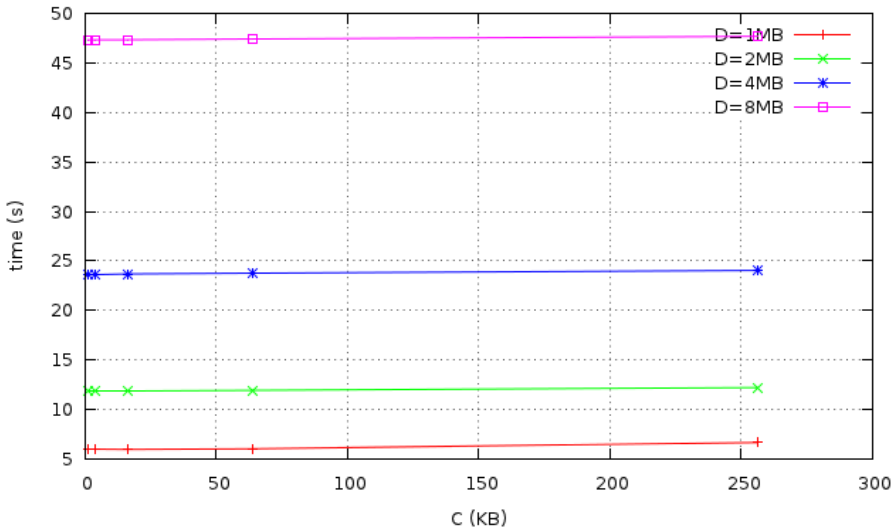

Figure 17: Chunk size against total time for uncached shared memory implementation

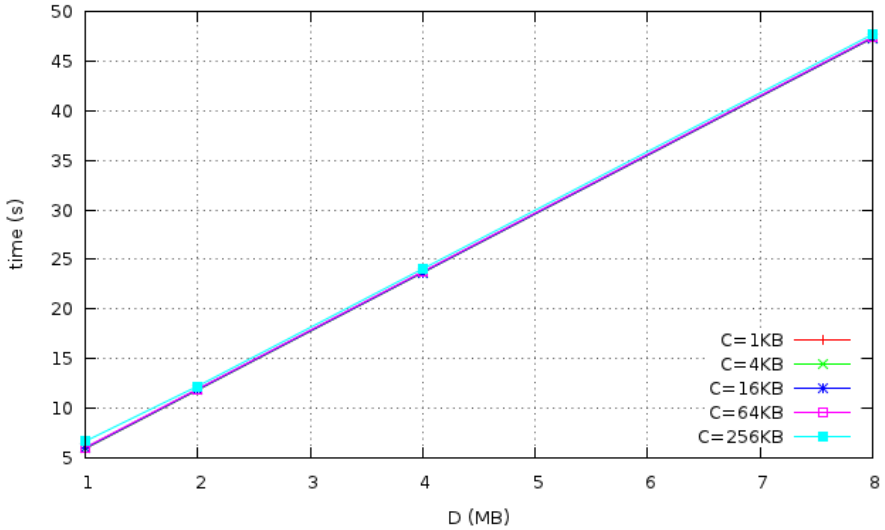

Figure 18: Input data size against total time for uncached shared memory implementation

Figure 19 and Figure 20 compare the chunk-sizes and data-sizes against times for the shared memory implementation with caching enabled. The behavior of these plots is due to the explicit flushing needed to maintain cache-coherency. The number of cache flushes required increases as the number of chunks increase adding additional overhead – at smaller chunk-sizes most of the time is spent in flushing caches. As the chunk-sizes increases, this overhead reduces. This is also the reason for the slopes of the plot for smaller chunk-sizes being steeper than that of larger chunk-sizes in Figure 20.

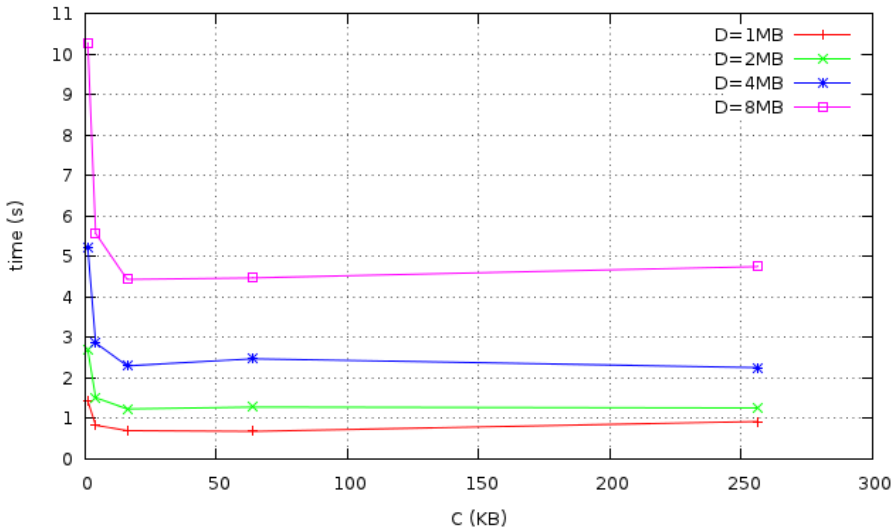

Figure 19: Chunk size against total time for cached shared memory implementation

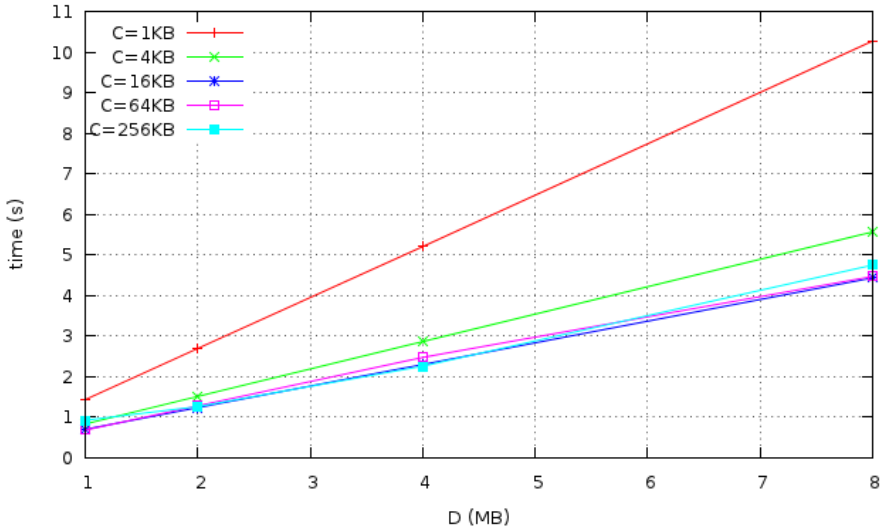

Figure 20: Input data size against total time for cached shared memory implementation

Figure 21 compares the performance of three implementations of a pipeline consisting of 48 stages in a linear workflow, using private memory and message-passing (tagged as “private”), cached shared memory and uncached shared memory.

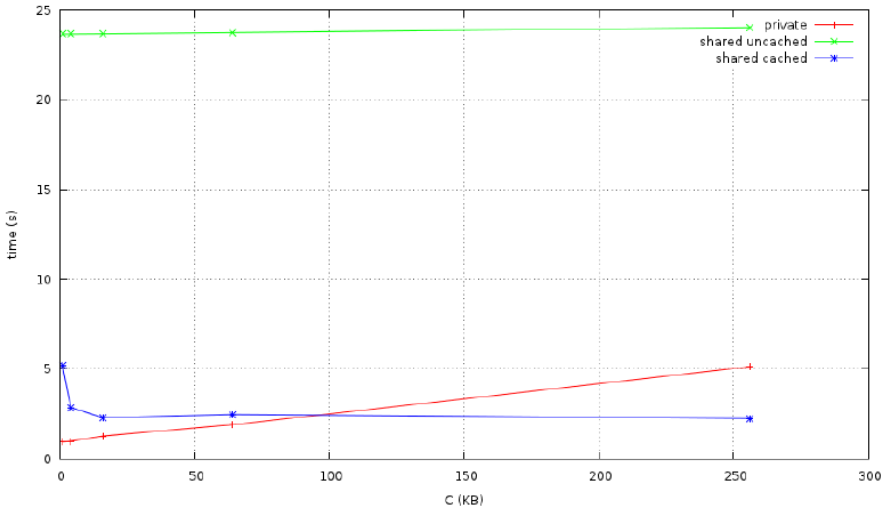

Figure 21: Comparison of the pipeline completion times as a function of the chunk size C.

The cases of overall data size D = 4MB and chunk-sizes C going from 1KB to 256KB have been considered. In the case of using uncached shared memory, the performance is far worse than the other two cases, due to the high latencies experienced as a consequence of having disabled the cache. Concerning the other 2 cases, at lower chunk sizes the overhead of flushing the cache in case of the cached shared memory far outweighs any advantage that comes from avoiding the extra copying of data that is instead necessary with the RCCE message-passing primitives. However, as chunk sizes increase, the number of chunks decrease and the overhead due to cache-flushing reduces, while the overhead of sending over RCCE continues to increase linearly as expected.

d) Conclusions

As it can be observed from the reported evaluation results, and particularly Figure 21, in a many-core chip like the Intel SCC, always relying on message-passing for any communication among cores may not be the most efficient choice all the times. Indeed, if the data to transfer from one core to another is sufficiently big, and globally shared (but incoherent) memory is available on the platform, then data can be made available from a core to another core simply by passing properly the pointer to that data, and ensuring that the sender flushes properly any pending write and the receiver invalidates any stale data in the cache, before accessing the data. We expect the threshold in data size beyond which one technique wins over the other to be highly variable depending on the platform characteristics, as well as the efficiency by which one can manage to realize both communication types. This has some impact on the programming model for core to core communications in many-cores. Indeed, a nice programming model should allow developers to not care about what communications means are being used, but let the Operating System automatically select the most efficient mechanism, depending on the knowledge it has about the platform and the data to be transmitted.

7. ALLOCATION AND MAPPING

This section elaborates further and expands the preliminary results about "Deployment Options" reported in the S(o)OS Project Deliverable D5.3 – First Set of OS Architecture Models [34].

As highlighted in the section about "In-chip Communications", tile-based systems exhibit different communication latencies depending on the routing distance between communicating cores. In the synthetic pipelined application described there, we played with a few deployment options, in order to evaluate the impact of the deployment logic on the performance of the application.

a) EVALUATION CRITERIA & MEANS

One of our primary interests is the end-to-end (e2e) time a pipelined or a parallel application takes to complete. Previously, we considered the effect of the underlying memory model used on the e2e time. The deployment of stages on cores depends not only on the memory characteristics but on the nature of the communications between stages.

As a first step we experimented with trying to reduce e2e time by means of reducing routing distances between cores. This helped us to study and model the overheads incurred for pushing data through the stages in the pipeline. We restricted the investigation to the case when the RCCE send-receive mechanism was used as the means of core to core communications.

b) CURRENT CAPABILITIES

The use of the network driver built on top of the message-passing buffers adds the additional overheads of the TCP/IP stack due to flow-control, marshalling/unmarshalling, header encoding/decoding etc. which might potentially overshadow the effects of the actual movement of data in the message-passing buffers (which forms the transfer mechanism). We restricted our communication means to the optimized RCCE send-receive interfaces for this very reason. Also, it allows the use of our models built for the message-buffer and memory accesses (see Figures 20, 21, 22 in [34]) and understand the effect of the NoC on scheduling and mapping strategies we will develop.

Our synthetic pipeline had very simple and predictable communication pattern – namely, receive the data (or pointer to data, in case of shared memory) and after processing, send the data (or the pointer to data) to the next stage. Though, this does not reflect what a real-world application might do, it provides a means to study in isolation the effects and behavior of the NoC.

c) EVALUATION

The initial experiments with mapping stages to cores with a reordering of cores to reducing inter-core routing distances showed us evidence that this does contribute some improvement the e2e times. The table of comparisons of the two cases with varying overall data size processed in 256KB chunks over 48 stages (one stage to a core) is reproduced below for reference.

Table 7: mapping speed for different data sizes and memory architectures

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

As 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%.

d) CONCLUSIONS

Simply changing the ordering of the cores to reduce routing distances (the overall memory access patterns over the lifetime of the pipeline was left unchanged), itself shows the effects on the e2e time which points that scheduling and deployment of applications is highly dependent on the characteristics of the platform.

Experimental results presented in the previous deliverable regarding the variation of latencies with regard to memory accesses, message-passing buffer accesses as a function of the routing distances and the load on the NoC indicate the role that these factors would play in determining a viable deployment of stages. Note that, since the memory is also accessed over the same NoC, the pattern of memory access would also drive the deployment.

While in our experiments we just had a single application pipeline running, in practice there would be multiple applications and pipelines running. In light of this, scheduling and mapping the stages to cores while meeting the deadline requirements is a challenge. Pushing this responsibility on the application would mean extra effort on the part of the programmer to build mechanisms into the application. This, in turn, leads to code that is customized for a certain platform (as these characteristics that we have been studying are very closely tied to the hardware architecture itself), and certainly hard to port to other platforms. Further, this presents problems in terms of scalability as well. In our case, the SCC has only 48 cores, but as the number of cores (and possibly the NoC) becomes larger, it would not be easy to determine manually a deployment and scheduling strategy.

A method of specifying in the application code the pattern and characteristics that segments of the program have with respect to communication and memory usage would simplify this task of deployment. These indications can be understood by the operating system which can then determine an optimal deployment strategy. Also, it removes the effort on part of the programmer to worry about the underlying hardware making the task of writing portable code easy.

8. COMPONENT-BASED ASYNCHRONOUS KERNEL

The component-based asynchronous kernel is a foundation focused on kernel communication in a network environment. Component allows us to abstract the implementation from the user, thus a single interface can be used wherever the underlying object is a proxy that uses message passing or it's the actual implementation. With this approach we can also migrate component instances, thus threads can be moved between non-coherent cores or even an entire process can be migrated from one machine to another.

In a network environment communications are intrinsically asynchronous, this can make a huge impact in the performance between blocking and non-blocking threads due to communication. It is of uttermost importance that an operating system targeted at many-core and cloud computing is designed with asynchronous communication in mind. The key issue is overhead imposed by a thread waiting for an acknowledge after sending a message. This overhead can be contribution of the following factors:

- **Wait queue overhead**
  This implies inserting the thread into the wait list, retrieve the next thread for execution in run queue and switch to it. Then when the thread is awakened from the wait, it must be removed from the wait list and inserted onto the run queue. The wait list and run queue must be protected with most low-level synchronization mechanism, typically a spinlock and/or disabling interrupts/preemption.
- **CPU context switch**
  It requires saving the CPU context of the blocking thread and then loading the context of next thread for execution. The overhead depends on the architecture but can be lightweight if the architecture supports lazy save/restore. Most architecture support lazy save/restore of FPU context since this unit usually features very wide registers, which makes it very costly to save/restore.
- **Address space switch**
  Address space switch must occur if the next thread for execution belongs to different process than the current one. Address space switch is very expensive since it requires a complete invalidation of the TLB that caches the page table, thus forcing the CPU a much more expensive access to main memory to translate a virtual address to a physical address.
- **Cache trashing**
  Each thread, even if it's a user mode thread, must have a kernel mode stack. Kernel mode stacks are critical in system performance since interrupts and system calls are serviced using the current kernel mode stack. As the number of context switch increases, so does switching of kernel mode stack, this can cause cache trashing where the CPU will most likely not be able to maintain the caching for the stack and the TLB entries. With an asynchronous kernel several threads can reuse a single kernel stack per core since no state is maintained on it. For threads that require real-time scheduling we can still use an dedicated stack in this case.

All of the previous factors can be mitigated with an asynchronous interface. When calling an asynchronous method execution continues on a completion handler passed to the asynchronous method. When an asynchronous method returns the stack is unwound and another thread gets its turn to use it. Context switching occurs when the kernel returns to user-mode or for threads with

real-time requirements. The overhead imposed by an asynchronous interface is the asynchronous method call and stack unwinding.

a) Evaluation Criteria & Means

The criteria we use to evaluate the component-based asynchronous kernel are as follows:

- Asynchronous call overhead: we compare the overhead of an asynchronous call that completes immediately with a synchronous call.
- Asynchronous wait overhead: we analyze the overhead of an asynchronous call that performs an asynchronous wait.
- Asynchronous completion: we analyze the overhead of completing an asynchronous call.

Since our objective is to have a fully asynchronous kernel to have with a unified component interface for local and network communication, it is important to analyse the overhead impact of asynchronous mechanism. This evaluation is done with our research kernel with the analysis of generated code and measurement of the overhead in CPU cycles.

The asynchronous kernel is a work in progress and it's not possible at this point to fully evaluate the component-based communication and process migration. We are finishing the implementation of a network interface driver that will allow us to evaluate communication and migration in a network environment. However, we did perform validation tests of the implementation of process migration and communication at a local level. A network environment without shared memory is thus critical to fully validate and attest the performance impact of process migration and component communication.

b) Current Capabilities

Operating systems have, for a long time, supported non-blocking I/O interfaces (files and sockets) for user land. The two major non-blocking interfaces used today are the reactor [43] and proactor [44] design patterns. These patterns have enabled the development of server applications that can scale to thousands of users, something that is not possible with synchronous interfaces. We acknowledge that non-blocking interfaces are critical for a networked operating system to scale to thousands of cores/machines communicating with each other.

The support for these interfaces in current mainstream operating systems is limited to certain operations (mostly read/write) and many do not enforce support for this, it depends mostly on the driver implementation. Windows supports asynchronous I/O in its driver model based on I/O Request Packets (IRPs) [45], and while the interface is tailored to asynchronously wait for IRPs it can be also be done synchronously. An inspection of the source code provide by the Windows Device Driver Kit, which features complete source code for some of the Windows filesystem drivers, shows that device drivers for Windows tend to avoid the asynchronous due to its complexity. In the case of Linux support for non-blocking mode is not mandatory, it left to the implementation of the component and/or driver in question. Linux also supports a POSIX Asynchronous IO, the implementation uses worker threads to emulate the asynchronous interface. However, using worker threads has scalability issues as outline in this chapter introduction.

While the reactor and proactor design patterns have been proven to scale, these interfaces are not suitable for usage on a kernel environment. These design patterns require a de-multiplexer, in the

reactor case it's used to known when a operation can be performed without blocking and the proactor case it's used o dequeue the completion events. The de-multiplexer is suitable for a process environment with is own set of threads, but this is not the case of a kernel-mode environment that must handle requests on behalf of user-mode threads belonging to different processes. Our approach [38] is reassembles some similarity with the proactor pattern using asynchronous methods with completion handlers but quite different, i.e., there is no de-multiplexer, objects are not associated with it and it also works with methods not associated with an object (free functions). The Barrelfish research OS also implements asynchronous communication. In Barrelfish a set of language extensions were introduced (Composable Asynchronous IO) for C/C++ [46] to support asynchronous operations. However, it can't be considered completely asynchronous because in the end, waiting for the completion of the asynchronous calls is done synchronously. It also requires a modified compiler to support the language extensions while our approach relies only on available C++11 language features (lambdas and variadic templates), thus it works on any standard compliant compiler.

c) EVALUATION

To evaluate the asynchronous interface overhead, we took as an example a write function call:

```
error_code write1(const void* buffer, size_t size, size_t& transfered)
{
    transfered = size;
    return error_code();
}
```

The asynchronous equivalent is:

```
void do_write2(const void* buffer, size_t size, tuple<error_code, size_t>& result)
{
   get<0>(result) = error_code();
   get<1>(result) = size;
}

template<class Completion>
void write2(const void* buffer, size_t size, Completion comp com)
{
   auto& as = ke::async::current(); //get current async primitive
   auto frame = as.prolog<Completion, error_code, size_t>(); //alloc async frame

  //element 0 of frame gives us a reference a tuple<error_code, size_t>
   do_write(buffer, size, get<0>(frame));
  //either call completion if completed or push it to the async stack otherwise
   as.postlog(comp, frame);
}
```

The above implementation of the asynchronous call adaptation is an optimized version of the one described in [38], thus slightly different. The major difference is that the asynchronous frame is allocated but the completion handler is only store if the call does not complete. If the call completes then the handler is called immediately and the allocated frame is released. The implementation of write1 and do_write2 merely set the transferred bytes to the same value and buffer size argument and set the error_code to success (value of zero). On completion we merely check the error code.

The tests were performed on a 16 multicore machine with AMD Opteron 6128 processors and 24GB of RAM. We use the GNU GCC 4.6 to compile the kernel with a –O2 optimization level. During the tests only a single CPU was used with step down locked frequency of 800 MHz, interrupts were disabled to avoid interference from the PIT timer. We used the timestamp counter to measure the number of CPU cycles and performed several measurements for each test. The first measurement is discarded as we consider it as warm up of the CPU caches. All the repeated measurements of each test were the same as would be expected since the time stamp counter incremented at each CPU cycle.

Table 8: Asynchronous call overhead

Asynchronous Call	Synchronous Call	Asynchronous Overhead
154 cycles	146 cycles	8 cycles

As we can see from Table 8 the overhead added by the asynchronous call is 8 cycles, it can be considered negligible. Looking at the generated assembly code we see a total of 3 memory accesses (2 reads and 1 write) to the async primitive were done at the prolog. The read accesses were to get the async primitive and the async stack position value and the write access was the new stack position value after allocating space for frame. This write access is the minimum, depending on the result types additional writes may be performed to initialize result types with constructors. At the prolog 2 memory accesses were done (1 read and 1 write), one read of the async primitive to check for the completion condition and, depending one the condition, the stack position is restored or the completion handler wrapper function pointer is stored. This is the minimum, since we have to consider copying captured data by the completion handler. The completion handler was inlined by the compiler, the generated code was equal to the synchronous case plus the additional postlog overhead mentioned above.

To test the asynchronous wait and wait completion we use a different do_write2 function to perform an asynchronous wait:

```
void do_write2(const void* buffer, size_t size, tuple<error_code, size_t>& result)
{
   auto& as = ke::async::current();

   as.wait([&result, size](error_code ec) {
      get<0>(result) = ec;
      get<1>(result) = size;
   }
}
```

We measure the number of cycles of the asynchronous call with the wait and the completion of the wait (completion of the wait causes the execution of the wait handler followed by the completion handler).

Table 9: Asynchronous wait and wait completion overhead

Test	Cycles
Asynchronous Wait	162
Asynchronous Wait Completion	198
Reference Call	138

From Table 9 we see that in the case of asynchronous call with an asynchronous wait introduces an additional 8 cycles of overhead. The asynchronous wait merely stores the wait handler in async primitive stack. In the case of wait completion, the wait handler and completion handler are executed in chain. The overhead is 60 cycles, it is much higher than the other cases because it involves two function pointer calls (one for the wait handler and another to the completion handler). Completing a wait involves calling the signal method of the async primitive, we use a reference call that is basically calling the signal method with an empty function body, which allows us to exclude the overhead associated with the timestamp counter.

Finally, we validated the migration process at a local level, moving the process to newly allocated components through serialization and de-serialization.

d) CONCLUSIONS

Asynchronous interfaces play a major role in the asynchronous component-based kernel. The performance impact of this mechanism must be negligee in comparison with cases where a synchronous function would not block (worst case scenario for an asynchronous kernel). From the results we get an overhead of 8 cycles which small enough for the worst-case scenario. The other results show an overwhelming advantage of the asynchronous kernel in blocking situations, i.e., a context switch would be much more expensive that the 60 cycles it took to execute the completion handlers in the tests.

9. TRANSACTIONAL MEMORY

The transaction paradigm is present in most operating systems to guarantee that a series of actions gets executed without interference from external executions. This is an important guarantee in a context where various services can execute concurrently. The transaction paradigm is known to be a key building block of existing operating systems, including Windows 7, Google OS, Mac OS X, iOS4. In future multi-core OSes, transactions promise to let a core execute a series of actions independently from other cores, thus serving as a synchronization facility between cores.

The Transactional Memory (TM) is a programming paradigm that provides transactions. It simplifies programming by adopting a divide-and-conquer approach: experts can develop a TM in the OS so that novices can develop applications using the transaction paradigm. Hence the novices do not have to care about robustness and concurrency, but simply needs to write a sequential code and delimit the regions that must appear as being executed atomically. Everything is handled transparently by the TM: if the system crashes within a transaction or if a transaction observes an inconsistent state, the system restarts the transaction until the transaction gets successfully committed---meaning that the transaction appears as being executed atomically.

By handling transparently concurrency and robustness, the transaction provides modularity to the OS. All transaction-based modules can be composed together in a modular fashion, as all modules are free of hardware constraints: for example, the TM can transparently handle both shared memory and message passing communication means.

a) CURRENT CAPABILITIES

TRANSPARENCY

Transactions preserve the sequential code in that their usage does not alter the sequential code besides segmenting it into several transactions. More precisely, the regions of sequential code that must remain atomic in a concurrent context are simply delimited, typically by a transaction{...} block.

Programming with transactions shifts the inherent complexity of concurrent programming to the implementation of the transaction abstraction which must be done once for all. Thanks to transactions, writing a concurrent application follows a divide-and-conquer strategy where experts have the complex task of writing a live and safe transaction system with an unsophisticated interface so that the novice has simply to write a transaction-based application, namely delimit regions of sequential code.

Traditional synchronization techniques require generally the programmer to first re-factorize the sequential code into a specific program that is dependent on underlying libraries or hardware primitives.

Lock-free synchronization. Using lock-free techniques, the program would typically use compare-and-swap or load-link and store-conditional depending on the considered architectures. Moreover, programmers would typically need to use subtle mechanisms, like logical deletion, to prevent inconsistent memory deallocations, yet the memory management would not even be guaranteed to be simple, and may require additional re-engineering.

Lock-based synchronization. Using lock-based techniques, the programmer must explicitly declare and initialize all locks before protecting memory accesses; the programmer may even need to use a logical deletion technique as well as an additional validation phase to guarantee consistency.

By contrast, the transaction abstraction hides both synchronization internals and metadata management. If locks are internally used, they are declared and initialized transparently by the transaction system. Moreover, as the transaction system wraps memory accesses, a simple reference counting can keep track of the status of transactions accessing a particular location, before freeing the memory.

MODULARITY

Transactions are also appealing for they allow concurrent programs to be reused in a modular fashion. More specifically, transactions allow Bob to compose existing transactional module developed by Alice into a composite one that preserves the safety and liveness of its components.

The Google FS. By contrast, alternative synchronization techniques do not facilitate composition. For example, consider a simple directory abstraction mapping a name to a file. With transactions, one can compose the removal of a name and the creation of a new name into a rename action. If a user renames a file from one directory *d1* to another *d2* while another renames a file from *d2* to *d1*, directories must be protected with care to avoid deadlocks. In the lock-based file system hierarchy of the Google File System, each directory at the same path depth has to be locked in a pre-determined ordering to prevent deadlock in such a scenario. In other words, Bob must first understand the locking strategy of Alice to ensure the liveness of his own operations.

The Linux Kernel. For the same reason, the header of the Linux kernel file mm/filemap.c comprises 50 lines of comments explaining the locking strategy.

Lock-free synchronization. Existing lock-free techniques are even more complex as they require a multi-word compare-and-swap to make the two renaming actions atomic while retaining concurrency. By contrast, a transaction system detects a conflict between the two renaming transactions and let only one of the two commit, the other one is restarted or resumed later.

Deciding upon the conflict resolution strategy is the task of a dedicated service, called a contention manager and various strategies have been proposed.

b) EVALUATION CRITERIA & MEANS

Table 10: overview over the evaluation criteria for transactional memory

	Name	Description
C1	Scalability	The ability for a property (like acceptable performance) to hold as the scale (like level of concurrency) grows.
C2	Robustness	The ability for a service to tolerate failures, inconsistencies.
C3	Modularity	The ability for a service to be devised into independent modules whose composition is the sum of these modules.
C4	Performance	The mean duration of an action, the number of actions executed per fixed period of time.

c) EVALUATION

In order to evaluate the scalability and performance of TM for research many-core operating systems, we have developed a prototype running on the so-called arbitrarily scalable *Intel(R) Single Chip Cloud* platform.

Please refer to S(o)OS Project Deliverable D4.2 - First Implementation Set: Execution Management (Section IV.1.A [36]) for the details of our DTM protocol, called *Single-Chip Cloud TM* (SC-TM), pseudocode.

In particular we have implemented three benchmarks to evaluate the performance and scalability of transactions: a linked list and a hash table benchmarks. The evaluation of Real-time TM has been postponed, the current status is the one described in [36] (Section IV.2.C).

LINKED-LIST AND HASHTABLE

Linked-list and Hashtable benchmarks belong to the micro-benchmark suite[5] and, as the names suggest, they are a transactional linked-list and hashtable implementations respectively. They support the following three operations:

- contains: checks if an item belongs to the list/hastable (read-only operation)
- add: adds an item to the list/hashtable (if it does not pre-exist) (update operation)
- remove: removes an item from the list/hashtable (if it exists) (update operation)

CONFIGURATION 1:

We compared the performance of the elastic transaction model [1] in our SC-TM implementation running with the FairCM contention manager, providing both fairness and starvation-freedom of concurrent transactions and with 20% (attempted) updates (i.e., 10% effective updates).

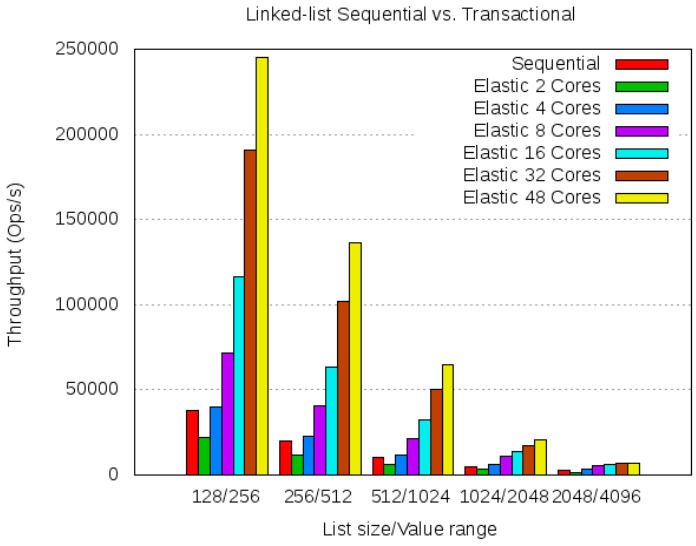

Figure 22: Throughput of sequential and transactional (elastic) linked list versions under different list sizes

This configuration reveals the performance improvement that can be achieved over the sequential code using SC-TM. Figure 22 illustrates the throughput of the system for different list sizes and

5 http://lpd.epfl.ch/gramoli/php/estm

Figure 23 the ratio of the transactional implementation compared to the sequential. SC-TM performs very well, since it provides better than sequential performance using only 4 cores (2 application cores). The best ratio is achieved for 256 elements list size and is close to 7 times faster. One can notice that the scalability drops when increasing the list sizes. This is solely due to the limited memory bandwidth, since the changes of the list size do not affect the number of messages sent by the elastic-read operations.

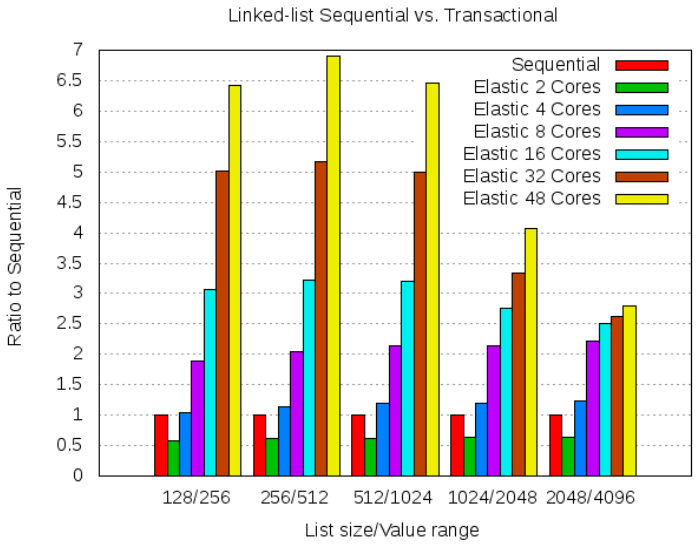

Figure 23: Ratio of transactional (elastic) throughput compared to the sequential under different list sizes

Hashtable has another important parameter, the *load factor,* which sets how many items each bucket will initially have. So, with 2048 cores and load factor 4, 512 buckets with 4 elements each will exist.

We tested Hashtable under two configurations.

CONFIGURATION 2:

We have tested the hashtable benchmark with 2048 elements, range value 4096 and 20% updates. We evaluate our SC-TM with our FairCM that provides starvation-freedom and fairness.

This configuration shows the performance improvement that can be achieved using SC-TM over the sequential code.

Figure 24 presents the throughput off all four versions for different load factor values on 48 cores (24 cores are dedicated to run the application itself, while 24 other cores are dedicated to run the TM service). The elastic-read version achieves up to 15.5 times performance improvement for load factor 1 (where hashtable is initially a plain array).

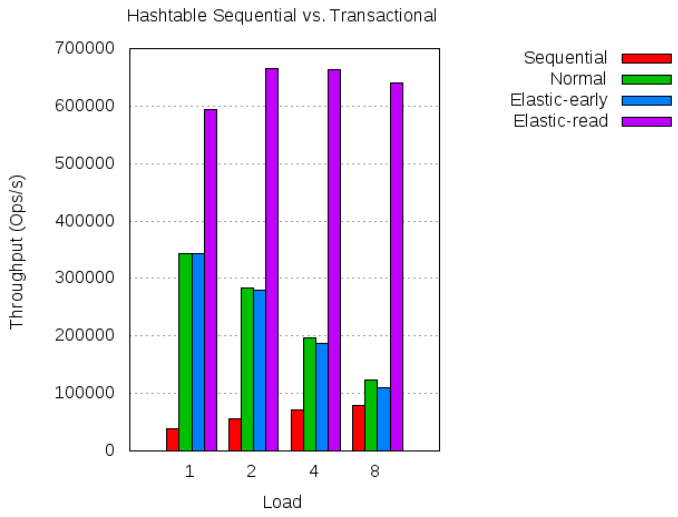

Figure 24: Throughput of Sequential and Transactional versions on the Hashtable benchmark under different load factor values

d) CONCLUSIONS

To start evaluating the transactional memory (TM) component for future OSes, we have proposed the first TM algorithm for manycores. Existing TM algorithms were only designed for two specific contexts: multicore systems and clusters. The former context requires cache-coherence to ensure that transactions terminate. The latter context assumes a fully decentralized systems whose cores communication has a low latency, rendering impossible to guarantee that transactions terminate.

In a manycore OS, one cannot rely on cache-coherence which is known an important bottleneck, preventing the system from scaling to large number of cores. In the meantime a manycore OS must guarantee a minimal quality of service, i.e., that every transaction eventually commits (as long as the programmer did not wrongly write an infinite loop within a transaction).

Our TM guarantees this quality of service and applies to manycore context. It does not require cache-coherence but ensures starvation-freedom. Experimental measurements on the Intel Single-chip Cloud Computer (SCC) research platform already show positive scalability and great performance. Specifically is improves bar sequential performance up to 20 times when running on only 24 application cores (cf. the hash table benchmark above with 48 cores in total). As for modularity, it is clear that the TM features a modular interface (presented in D4.2) fulfilling the standardization. It remains to evaluate the robustness of our solution: at the current stage we reasonably assume that cores do not get corrupted, but the inherent checkpoint of our transactions could be exploited to relax this assumption.

.

10. SPECULATIVE EXECUTION

Speculative Execution – also Speculative Access to Memory (SAM) - refers to the execution of parts of an application out of the logical sequential order, which 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 we will evaluate if the objectives of SAM are met and how it contributes to the main objectives of S(o)OS.

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 a thermal issues have made the approach of increasing clock frequency infeasible. Semiconductor manufacturers 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.

To this we have to add the problem of the access to memory latency. The developments in memory technology have not been able to improve consistently the memory to access. “First of all, as chip geometries shrink and clock frequencies rise, the transistor leakage current increases, leading to excess power consumption and heat... Secondly, the advantages of higher clock speeds are in part negated by memory latency, since memory access times have not been able to keep pace with increasing clock frequencies. Third, for certain applications, traditional serial architectures are becoming less efficient as processors get faster (due to the so-called Von Neumann bottleneck), further undercutting any gains that frequency increases might otherwise buy. In addition, partly due to limitations in the means of producing inductance within solid state devices, resistance-capacitance (RC) delays in signal transmission are growing as feature sizes shrink, imposing an additional bottleneck that frequency increases don't address.”[6]

Many applications exhibit irregular access to memory, a complex control flow and dynamic loop limits. 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 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.

[6] Intel Platform 2015 documentation, http://epic.hpi.uni-potsdam.de/pub/Home/TrendsAndConceptsII2010/HW_Trends_borkar_2015.pdf

In summary, the programmer, the code analyser and the compiler will segment the code and decide which parts of the code can be run speculatively. During the execution the SAM library has to take care of the memory accesses done by the speculative threads and the ones that retain the sequential correctness. Simultaneously it has to make execution checkpoints to rollback when a data dependency is violated.

These functionalities help to achieve some of the primary and secondary objectives of S(o)OS. SAM will increase the use of idling resources in a large system contributing to GO#1. It also allows to parallelise segments of code that statically is not possible to demonstrate that can be executed in parallel which is related to GO#2, although it can be done by the code analyser and the compiler, the programmer can decide which parts of the code to run speculatively by using a programming interface of the library and programming pragmas to help the automation, contributing to GO#3 and PO#4.

a) CURRENT CAPABILITIES

Most computer programs have always been written as a sequence of instructions that must be executed one after the other. But often some of these instructions or sets of instructions have no data or flow dependency between each other, so they can be executed in parallel, increasing the level the performance. The ILP (Instruction Level Parallelism) is the parameter that indicates how many instructions per cycle can be executed in a system.

The scalar processors represent the simplest class and can only process one data at a time, being classified as a SISD processor in the Flynn taxonomy. Different techniques have been developed along the last decades to detect and execute instructions in parallel (increase the ILP):

- Increase ILP per computing unit
    - Superscalar
      in the Flynn Taxonomy, a superscalar processor is classified as a MIMD processor (Multiple Instructions, Multiple Data). The hardware of this kind of processor dynamically checks for data dependencies between instructions at run time, being able to accept and issue multiple instructions per clock cycle. But it is limited by the degree of intrinsic parallelism of the instruction stream and the complexity of the dependency checking logic.
    - VLIW (very long instruction words)
      this processors are also MIMD that execute operations in parallel based on a fixed schedule determined by the compiler. The processors does not need to have all the complex hardware to detect de data dependencies between instructions as the compiler determines the order of execution, including which operations can execute simultaneously. As superscalar processors it is limited by the degree of intrinsic parallelism of the instruction stream.
    - Vector Processors
      is a CPU or processing unit inside another processor that implements an ISA containing instructions that operate on arrays of one dimension (vectors). In the Flynn Taxonomy these are SIMD processors. These kind of processors (or units) are very useful when there is the need to perform the same operation on a set of data.

- Out of Order Execution
  is a technique used mainly in superscalar processors to make use of instruction cycles that would be wasted by certain delays (multi-cycle operations, write to memory...). The processor executes instructions as their data is available and not in the original order of the program. In doing so, the processor can avoid being idle while data is retrieved for the next instruction in a program, processing instead the next instructions which is able to run immediately. This technique usually increases performance notably but it is also limited by the instruction stream data dependencies, by the size of the reorder buffers and the complexity of the hardware involved.
- Register Renaming
  is a method used in microprocessors (and/or compilers) to avoid unnecessary serialization of assembler instructions. In case of Write-after-read (WAR) dependencies, instead of delaying the write until all reads are completed, two copies of the location are maintained, with the old value and the new one. Reads will use the register with the old value while the operations using the data of the write operation, will access the register with the newer value. The false dependency is broken which increases the parallelisation of superscalar and/or Out-of-Order processors. Some modern ISAs (like IA-64) have enough visible registers that allow the compiler avoid WAR dependencies, at a cost of increasing the code size. But older ISAs usually have a limited set of registers, so modern processors that implement old architectures have a set of “hidden” registers that are used for register renaming.
- Multiple threads:
  - Symmetric multiprocessing (SMP): is a multiprocessor computer architecture in which two or more processors are connected to a shared main memory. It allows running two threads simultaneously but not all applications are programmed with multiple threads as it increases the programming complexity and not all applications are parallelisable. When more than one program executes at the same time it will have a better performance than a uni-processor because different programs can run on different CPUs simultaneously. But access to RAM is serialized (this is improved in NUMA systems), but cache coherency issues cause performance to lag, especially as the number of processors increase.
  - Symmetric multithreading: is a technique for improving efficiency of superscalar processors by running multiple independent threads to increase the utilization of hardware resources. As with SMP, for taking advantage of it we need to run various programs or a program with multiple threads. Because the technique is really an efficiency solution and there is inevitable increased conflict on shared resources, measuring or agreeing on the effectiveness of the solution can be difficult. In some cases it can even reduce the overall performance, mainly due to having to share the processor cache among the different threads.
- Speculative Execution: This is a performance optimization technique that consists of doing work of which the result may not be needed. The idea is to provide more concurrency if extra resources are available.
  - Instruction Level:
    - Memory prefetch: in current computing systems memory is a usual bottleneck and it creates the longest wait states in the processors.

Memory prefetch requests instructions or data from the main memory before it is actually needed, minimizing idle states, at the cost of increasing the idle state when fetching the wrong data or instructions.

- branch prediction: a branch predictor is a component in some processors that tries to guess which way a conditional branch will go before it is known for sure. The purpose is to avoid halting the execution of instructions while the condition is being calculated. The branch predictor takes one branch and starts executing it. If its decision was correct then the execution continues but if it was mistaken, then it has to restore the status of the pipeline and execute the other branch.
- load overtaking: as the memory access latency is the main bottleneck in current systems, an out-of-order execution processor will try to reorder the instructions so that the load operations are executed as soon as possible minimizing the idle state. In a conventional design, to avoid real data dependencies the load instructions cannot be re-ordered before a write instruction. Load overtaking is an improvement that allows load instructions to overtake the store operations and execute speculatively the assembler instructions. The processor hardware must take care of the memory addresses from which the load wrote, and in case the overtaken store wrote to that address restore the previous correct status of the processing with a great penalty price. These kinds of systems also heavily increase the transistor count.

- Thread Level: it is a dynamic parallelisation technique extends the main idea of Out-of-Order execution and branch prediction to thread level opposed to the instruction level. At compiling time not all data dependencies can be resolved, so the compiler or programmer will set into speculative threads those parts of the code that might or might not have data dependencies. The program will start those speculative threads and detect in runtime the dependencies. If there were no data dependencies, then the work of the speculative threads is accepted, if a data dependency is detected then the work of these threads is invalidated.

During the last decades lots of efforts have been realized to increase the ILP of processor cores in order to increase the performance of a single thread which resembles quite well with the logical sequential structure of most programs. But the current trend is to work in the parallelisation of programs due to the increasing number of computing units available in each system.

The current research in thread level speculation (TLS) is not yet very advanced and most approaches try to buffer all changes to individual memory elements. Our approach is similar to the current (TLS) but with important differences to reduce the high overhead at the cost of allowing to have some false positive when detecting the data dependencies violations. To greatly reduce the cost of tracking the memory accesses, instead of tracking individual memory elements as TLS, SAM tracks accesses to memory regions. It could be said that SAM expects a more positive data dependency behaviour. But we consider this approach as completely valid as only segments of code which most probably have no data dependency should be run speculatively, not those that in most cases will have.

b) EVALUATION CRITERIA & MEANS

In the next table a description of the different criteria applied for evaluating SAM can be found. These criteria were selected because they allow us to evaluate the different some particularities described in the different execution phases described in document [35]

Table 11: overview over the evaluation criteria for speculative execution

	Name	Description
C1	Correct Initialisation	Set up the library and initial structures correctly. This evaluation criterion contributes to GO#3 and PO#4.
C2	Creation of a speculative thread	Test that the creation and destruction of speculative threads in a SAM context is correct. This evaluation criterion contributes to GO#3 and PO#4.
C3	Creation of a protected area	Test that the creation of a memory protected area within speculative threads is correct. This evaluation criterion contributes to GO#3 and PO#4.
C4	Sending and retrieving information to/from worker threads	Test that the parameters are correctly sent to the speculative threads, and when it finished executing, extract correctly the result. This evaluation criterion contributes to GO#3 and PO#4.
C5	Roll-back of speculative threads	Correctness of the mechanisms for rolling back a thread with a RAW data dependency. This evaluation criterion contributes to GO#1.
C6	Speculative writing threads	The speculative threads can write in registered memory regions speculatively, and roll-back upon detection of a data dependency violation. This evaluation criterion contributes to GO#1 and GO#2.
C7	Execution of multiple memory protected regions	Creation of multiple protected memory regions within the same SAM context, creating a hierarchy of speculation between threads. This evaluation criterion contributes to GO#1 and GO#2.
C8	Correctness of execution	Test that the final result is consistent and correct.
C9	Performance gain	Analyse the performance gain/loss compared to non-speculative execution. This evaluation criterion contributes to GO#1 and GO#2.
C10	Scalability	Analyse the scalability of SAM with the increase of computing units and speculative threads. This evaluation criterion contributes to GO#1 and GO#2.
C11	Ease of Use	Analyse the added difficulty of using the interface of this library when implementing programs and compare it to other parallel implementation models. This evaluation criterion contributes to GO#3 and PO#4.

We expect the result of the evaluation of the previous criteria to probe the correctness of behaviour of SAM and to contribute with some of the general criteria of S(o)OS for program execution.

c) EVALUATION

To test and evaluate the first implementation of SAM library two machines have been used. Criteria C1 to C8 where mainly evaluated in a desktop machine with a dual core Intel E5400 2.70 Ghz cpu. This computer does not have a high degree of computing units (just 2), which is enough for developing and testing functionality, but not for evaluating the performance and scalability. For the criteria C9 and C10 a computer with 2x 8 core Opteron 6128 at 2 Ghz (16 cores in total). This machine has a high level of parallelism creating a situation in which many of the computing units would not be used by sequential or not high level of parallelism parts of a program, making it ideal for our needs.

A set of test programs named Test-0 to Test-6 have been implemented to test the evaluation criteria. These tests consist of a set of programs, in most of cases with a sequential and speculative implementation to be able to compare the final result (even if the execution of the program is speculative, the final result must be the same, as it has to maintain the sequential accesses to memory). The Criteria C-11 is a subjective characteristic, and there is no good formal way of evaluating it, later in this chapter a discussion and comparison with other possible implementations or parallel and speculative execution programming models will be given.

The aim of Test-0 to -5 is to test the correctness of the execution of the SAM library, to probe its consistency and that it deals correctly with detecting the data dependency violations and roll backs, with increasing complexity of the problem in each case. Test-6 also checks for correctness but it is mainly designed for testing performance difference between a full sequential implementation, a conventional parallel implementation and a speculative parallel execution of a program.

Table 12: overview over the evaluation tests for speculative execution

Name	Criteria	Explanation
Test-0	C1, C8	Basic program to test initialization and that the minimum system requirements are met
Test-1	C1, C2, C8	This test checks the creation and execution of speculative threads, also testing the OS ability to handle processes and threads
Test-2	C1, C2, C3, C4, C8	First test to sending data on creation and retrieving data on destruction of a speculative thread and handling of the registered memory regions
Test-3	C1, C2, C3, C4, C5, C8	This test consists of two programs that test different kinds of cases that create data dependencies violations and checks for the correct foll back of the speculative threads and program data consistency maintenance
Test-4	C1, C2, C3, C4, C5, C6, C8	This case generates data dependencies violations after the speculative threads have written to the registered memory region, the data consistency maintenance is checked
Test-5	C1, C2, C3, C4, C5, C6, C7, C8	In this test many protected memory regions are created in a hierarchical way, creating a priority tree between speculative threads
Test-6	C1, C2, C3, C4, C5, C6, C7, C8, C9, C10	Three versions of this programs are created; a sequential, a parallel, and a parallel with speculation. With them we will make an evaluation of performance and scalability

Name	Criteria	Explanation
Comment-7	C11	Discussion about Ease of Use.

EVALUATION RESULTS:

TEST-0:
Status: Passed

Description: This first test check for the correct initialization of the SAM library (C1). The computed result is correct (C8).

TEST-1:
Status: Passed

Description: In this test SAM correctly created and starts the execution of speculative threads (C2). The computed result is correct (C8).

TEST-2:
Status: Passed

Description: SAM library correctly sends data on the creation and retrieves it at the destruction of a speculative thread (C4). It also correctly handles the creation of registered memory regions (C3). The computed result is correct (C8).

TEST-3:
Status: Passed

Description: SAM correctly detects data dependency violations and roll back to a consistent state those speculative threads that were affected (C5). The computed result is correct (C8).

TEST-4:
Status: Passed

Description: WaW (Write after Write) data dependency violations – speculative writing – are correctly detected and resolved (C6). The computed result is correct (C8).

TEST-5:
Status: Passed

Description: a few speculative threads create different memory protected regions and inherit them hierarchically (C7). The data dependency violations and roll backs are handled correctly. The computed result is correct (C8).

TEST-6:
Status: Test realized with different results.

Description: Three versions of this programs are created; a sequential, a parallel, and a parallel with speculation. With them we will make an evaluation of performance and scalability. The computed result is correct (C8). The program is a synthetic benchmark that has a sequential part, a parallel

one, and a last one that might have data dependencies; the different parameters of the program allow controlling the length of each section and the degree of possible parallelism, and chances of data dependency violations. The result gives two parameters, the execution time and the effort (sum of processing cycles of all processing units involved in the calculations).

As expected the final result depends on the parameters used: the total effort is always higher in the version using SAM, this is totally expected for the overhead of using the library plus the not used calculations of the speculative threads that have been rolled back. The results show that in the worst case the overhead of SAM reaches a decrees of performance of a 6% while in the best case the execution time reduces by around a 30%, most cases show a performance gain of a 15 to 20%.

COMMENT-7:

The current proof-of-concept implementation of SAM resembles the POSIX implementation of pthread library with the addition of functions for handling the registered memory regions on which we are executing speculatively. Right now we have to say that the difficulty of programming is increased. It should be taken in consideration that we plan SAM to be automatically used by the code analysis and segmentation component without the need of the programmer to make explicit calls.

d) CONCLUSIONS

From a performance point of view, we can see that in some situations it can greatly improve the performance of a program. These situations always have two common requirements. First there must be available computing resources, we should not forget that the main aim of SAM is to increase effectiveness and scalability by making use of idle computing units. The second requirement is that the code has uncertainty about the data dependencies (if we know there is no data dependencies then we can do a conventional parallel computation which has less overhead, and if we probe that there are data dependencies then all computations will be not correct till data dependencies are resolved and there is no need to use SAM).

III. IMPACT FROM INTEGRATION ON EVALUATION

Monolithic operating systems are designed as integrated functionalities and capabilities that strongly depend on each other in order to realise the required functions – this way, the operating system can achieve maximum efficiency by aligning the individual tasks and their dependencies in such a way that they exploit each other best. Even though this means that the individual functions are highly efficient, their actual efficiency depends strongly on the usage context. In other words, functions that are triggered from outside the operating system, i.e. by the running application, can only be efficient if they are used in the fashion as intended by the OS designer and if that fashion is compliant with the intentions of the running application. The capabilities of the underlying frameworks define how the developer can express his intentions and convert them into the program – similar to the expressiveness an advanced speaker can make of a language, compared to a beginner. At the same time, the scope of the framework support may completely oppose the developer's intention and thus restrict him in efficiently realising the respective goals and reducing the potential application performance. For example, reading individual bytes from a harddrive is extremely complicated and inefficient, if the only way of accessing it consists in reading whole files into memory.

All middlewares – whether hardware, operating system or programming framework – are developed aiming at specific types of usage, such as office environment, (web) service provisioning, high performance computing etc. It is therefore no surprise that operating systems differ as strongly as they do, for even though there is a general tendency to cover a wide usage scope, their background in design, development and current usage has a clear impact on their actual usefulness in the sense of which type of user can get the most benefit out of it (and is therefore more likely to use it).

As discussed in more detail in the project deliverables D5.1 [32]and D5.2 [33], this tight integration of functionalities not only reduces the applicability scope, but also its adaptability to other domains, and in particular infrastructures and systems. Typically, even small changes in such tightly, integrated systems entail multiple adaptations of the whole middleware (or of various parts of it), due to the strong dependencies across functions. This makes adaptation of such operating systems to new and in particular heterogeneous infrastructure tedious and not cost-efficient. What is more, due to the nature of the adaptations, portability, i.e. the capability to run efficiently on multiple different systems, is heavily restricted – generally limiting the efficiency significantly.

In addition to this, the scale of such tightly integrated systems is limited significantly – mostly due to the simple fact that the operating systems becomes large and heavy-weight, i.e. every invocation entails so many other function calls that either major blocks have to be moved, thus creating page faults and memory swaps, or bottlenecks are created that produce unnecessary messaging overhead (see also [32] [33]).

1. COMPOSEABILITY

The approach pursued by S(o)OS consists in essentially reducing the dependencies and limiting the functionality scope to an essential minimum, thereby decoupling the individual functionalities from one another. The implicit reduction in capabilities does on the one hand restrict the efficiency for

specific usage scenarios, but on the other improves the applicability *across* use cases drastically. What is more, S(o)OS pursues a highly component-based approach which allows for easy extension with further, use case specific components, so that extended support for specific domains may easily be achieved through such extensions. The same principle applies to increase the portability and adaptability of the operating systems, as rewriting minimal components / functions is much simpler and more efficient than having to adapt the whole OS.

The S(o)OS Project Deliverable D5.3 – First Set of OS Architecture Models [34] discusses the specific architectural integration in more details (cf. Figure 25) – in the context of evaluation, it is of more interest what kind of impact such a component-based, as opposed to fully integrated operating system architecture may have on the overall performance of the system: It will be noted in particular that one of the major drivers for creating an integrated (monolithic) OS consists in increasing the efficiency of the individual functionalities, so that it could be expected that the dissection of these functions into separate components diminishes the positive performance effects. In addition to this, microkernel like approaches are generally criticised for the negative impact of message-passing based communication on performance aspects [TORWALDSREF].

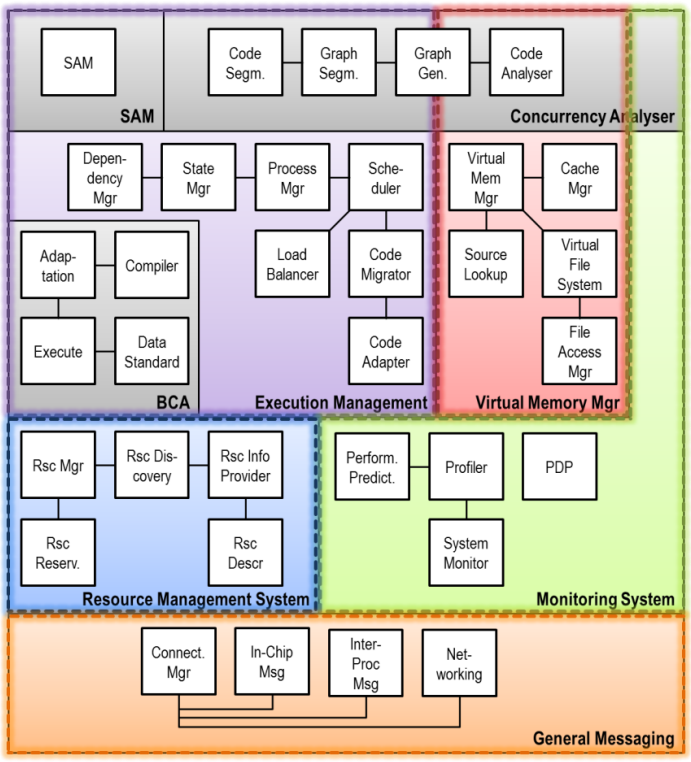

Figure 25: architecture schematic of S(o)OS (cf. [34])

a) Impact of Component-based Approach

First off, we have to assess the impact created by dissecting the individual functions into separate components. We can thereby denote in particular two potential sources of impact: (1) the misalignment of code and data in memory, creating additional page faults due to cross-component invocations and (2) the messaging overhead created by exactly this form of cross-component communication.

It is obvious that plain reorganisation of the OS functionalities into "logical groups" (i.e. components) is not sensible if not all according cross-component dependencies are also reduced to a minimum. It will thereby be noted that the average application makes use of most functionalities an OS provides – directly or indirectly. This is however mostly due to the programming approach in the first instance, as most modern programming models incorporate combinations of OS capabilities into their exposed operating set, rather than exposing the OS functionalities directly (cf. discussion on user target groups above). Further to this, most applications are very heterogeneous in their functionality scope themselves, i.e. execute various types of functionalities, such as reading / writing files, displaying graphics, communicating over the internet, reacting to hardware events (in particular keystrokes) etc. However, if the application is analysed in more detail, it will be noticed that it actually consists of various logical blocks that exhibit a comparatively uniform (or homogeneous) use of the operating system. As one of the major goals of S(o)OS consists in segmenting an application to exploit concurrency within the behaviour, S(o)OS' execution concept explicitly aims at identifying and exploiting such logical blocks. For example in typical desktop applications it is comparatively easy to identify relatively unrelated event-driven units that expose specific OS needs (updating the display etc.)

Accordingly, the general impact of the component-based approach depends very strongly on the use cases, or – more specifically – on the structure of application and infrastructure. In other words, if the application exhibits no concurrency and no scalability, and / or the environment consists of one single-core processor, then a component-based approach is not sensible in the first instance. On the other hand, if the application exhibits clearly separable blocks, and the infrastructure contains multiple processing units, a component-based approach can improve the performance, as only the essential capabilities and data are co-located with the application segment, thus reducing the effect of messaging, page faults and bottlenecks.

Given the explicit context of S(o)OS (executing applications on large scale and heterogeneous infrastructures), we generally assume that the applications in question principally exhibit concurrency, or are developed in a parallelisable fashion, and that the infrastructure incorporates a large amount of (different) processing units. Accordingly, the component-based approach should have a positive impact on general execution performance.

b) Impact of Microkernel-based Approach

Closely related to the problem of messaging overhead (see above), message-based kernels (or in fact any message-based software architecture) are generally criticised for their delays caused by message handling, as opposed to in-memory invocations. This is generally true for such environments such as web services, where the message handling implies in particular complex conversion and interpretation of the messages, as well as enactment of high-level protocols, such as HTTP for communication purposes. However, in the case of low-level process execution, the communication across threads (or here OS instances) is not really as complicated as that of web service environments.

As opposed to this, the classical approach to cross-unit communication, respectively data exchange consists in exploiting shared memory capabilities. Modern shared memory systems build upon cache coherency protocols rather than physically shared memory, which – due to their nature – create delays for validating and updating the caches. Even though the individual cache validation is comparatively fast, concurrent access to cache can impact on the performance significantly, as it

overloads the data lines. With the growing number of cores and threads, the likelihood for such overloads increases constantly, thereby slowing down the effective communication speed. Message-passing protocols on the other hand generally queue requests, therefore creating an even delay, rather than a cache line overload. Furthermore, asynchronicity can be exploited to reduce process stalling.

Baumann, Barham et al. evaluated message-passing versus shared memory protocols in 2009, and could show that the latency created by modern shared memory protocols quickly exceeds the message-passing delay [42], therefore making message-passing a better choice in particular for large scale application and infrastructures where high access to shared data is expected (cf. Figure 26) - this obviously is equally valid for transactional memory methodologies (cf. section II.9). In the figure, the "server" line represents the latency created by the protocol for handling the messaging protocol.

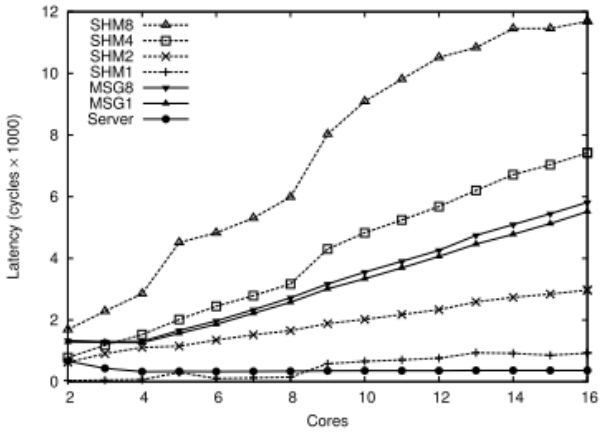

Figure 26: Latency created by message-passing (MSG) versus shared-memory (SHM) protocols (taken from [42])

We must note in this context that there is a general high expectation in the communities that cache coherency will disappear from future multi-core processors, due to exactly this scalability problem (cf. [33]). This only means, however, that the user (respectively the application / operating system) will have to cater for data and state maintenance across threads and processes on the hardware's behalf – in other words, not reducing the benefit of message-passing approaches.

c) Conclusions

The principles of the component-based approach do not conflict with the goals of an operating system per se, i.e. it does not generally follow that performance and capability scope is reduced through this approach – in fact, given all circumstances, it is more likely that the base structure itself boosts performance due to the nature of the destination environment, i.e. large scale and heterogeneous infrastructures. The component-based approach allows faster adaptation and better exploitation of the destination platform, therefore enabling operating systems to execute tasks with higher efficiency and exploiting the specifics of the processing units better.

It is however undeniable, that the components still exhibit dependencies and therefore must be carefully designed to keep the communication overhead at a minimum. Along that same line, the offered capability scope and the efficiency of such a distributed operating system depend highly on the right selection of core functionalities, and thus components that form the OS. We will discuss this in more detail in the next section.

2. ARCHITECTURAL CHOICES

S(o)OS components of the first iteration were carefully chosen to address the essential capabilities that a distributed, scalable operating system needs to provide to support efficient execution on future platforms. It was thereby a primary goal to keep these functions as isolated as possible to reduce the communication overhead – implicitly, the individual components grow slightly larger in size than they seem to be in monolithic systems, simply because they have to reproduce other functionalities that are normally provided as part of other functions of the (monolithic) OS, but thereby increase the cross-functional dependencies. This approach makes the S(o)OS components essentially stand-alone operating support systems – very much like a microkernel instance, but as we shall see, only designed to partake in the overall execution support, rather than to take it over themselves.

Implicitly, however, the range of support is highly restricted per functional component, thus targeting in specifically selected requirements by the application. As discussed above, this is not considered a problem per se, though, as it reflects the typical structure of most applications that it actually consists of separate logical units with comparatively restricted need for functional support. Nonetheless, the components still exhibit dependencies, due to two major reasons:

1. Indirect (application-based) dependencies:
   First of all, most applications expose very heterogeneous requirements on the overall level, even if the individual units have comparatively restricted requirements. As the application typically shares at least some form of state, system calls may become indirectly dependent upon each other – for example when loading and displaying a file.
2. Direct dependencies:
   Due to the strong reduction of OS capabilities to most essential functionalities, the individual components do not integrate all potential operations needed to cater for more complex invocations, such as streaming monitoring data to a file or similar.

To satisfy such dependencies, components are enabled to invoke each other and communicate data across. Even though a straight-forward approach consists in hosting all potentially needed system components local to the respective code segment, this is not necessarily the most efficient way to handle distributed execution, as it may increase the number of local page faults. In particular components that are hardly used and / or share data across multiple instances should be executed remotely. The performance can be further improved if the invocation is essentially asynchronous in nature, so that the local processing unit can continue its operation during the invocation.

This means however, that some form of communication across components and processing units needs to be provided. As was already detailed in the preceding section, such message-passing approaches for component communication do not imply performance reduction. However, the remote component may turn into a communication bottleneck, just like a monolithic operating system, when it has to serve too many remote components, or has to invest too much effort in maintaining state or data consistency. For example, components that have to share identical information, such as an infrastructure layout information system, have to communicate all updates in order to maintain consistency – accordingly, if they update frequently, basing on e.g. monitoring information, they should not be replicated too many times in order to reduce the potential amount of communication. On the other hand, if some information is required frequently from multiple

segments, but consistency across is secondary, due to few changes or high relaxation, than the respective components should be replicated multiple times to reduce the bottleneck effect.

The basic architectural choices therefore can have a huge impact on performance of the total system, e.g. if the design allows only for single instantiation, thus creating a bottleneck, or requires too much data exchange, thereby producing communication overhead. Obviously, this impact depends very much on the nature of the functionality provided by the component, i.e. how frequently it is used, how much effort it takes to realise the functionality, at what time(s) of the execution life cycle it is needed etc. The preceding evaluation sections and the S(o)OS architecture deliverable [34] provide an overview over the specific capabilities and requirements put forward to the individual components.

Depending on the requirements towards the functionality, the impact on depending components, and thus the performance impact on the overall operating system's performance may vary significantly. As noted we can thereby distinguish in particular two aspects that create an impact on the performance factors:

1. Invocation frequency
   The most important factor relates to how often the respective component is invoked and therefore whether it can principally turn into a messaging bottleneck.
2. Data (consistency) maintenance
   As components need to maintain (live) data across all (or at least multiple) of their instances, a consistency maintenance overhead is created, which reduces the efficiency of the respective functionality significantly – cf. section III.1.b)

The following table provides a rough overview over the components and their expected invocation frequency, respectively data dependency (see Table 13).

Table 13: the OS components and their potential impact on performance due to essential architecture characteristics

Component / Functionality	**Invocation Frequency**	**Data Consistency Requirement**	**Note**
Monitoring	Med-Hi	Low	Live monitoring information is of particular interest mostly to local resources and direct neighbors, not the whole infrastructure.
Scheduling	Hi	Med	The scheduler needs to maintain some information about all segments belonging to the executing application
Resource Discovery	Low	Lo-Hi	Information about the resources is particularly relevant at setup time and with any dynamic change in the environment
Transactional Memory	High	High	Whilst transactional memory itself does not carry data that needs to be maintained, its main purpose consists in maintaining data for the running application

Component / Functionality	Invocation Frequency	Data Consistency Requirement	Note
Code Analysis & Segmentation	Low	Low	Analysis and segmentation principally only takes place prior to code execution, even though resegmentation may be triggered when the environment changes
Code Adaptation (DISTAE)	Low	Low	Similar to analysis, code adaptation only is invoked when the code segment has to be converted for the destination platform
Speculative Execution (SAM)	Lo-Med	Med-Hi	Speculative execution is only used under specific conditions.
Component-based Kernel	n/a	n/a	This is the fundament of all executing application segments and therefore does mirror the segment's invocation and consistency needs

With respect to the invocation frequency, i.e. the amount of average calls towards the according component, it will be noted that a high frequency implies in particular that the respective component should not be hosted in a centralistic fashion, but should rather be co-located with the calling code segment. As discussed, major problems arise in particular once the component also has a high requirement for data consistency, since in these cases the component should be as centralized as possible to reduce consistency maintenance overhead. This affects in particular components that focus on supporting exactly this consistency across code segments, such as the transactional memory – as noted in the preceding section, cache coherency plays one of the biggest scalability challenges in modern infrastructures and the message-based approach offers a promising alternative to maintain a higher degree of scalability.

It will also be noted that data consistency is particularly demanding in cases where the data that has to be maintained changes often – comparatively static data produces too few communication overheads to be considered a performance threat. We can identify the following main sources for dynamic state data in the operating system itself: (1) hard- and software status information at runtime (monitoring); (2) application context information, in particular for multi-tasking environments; (3) dynamic environment changes. In this context we have to note that during the first iteration of the project, we focus primarily on the aspects of segmenting, distributing, adapting and executing the code, without considering the impact from reliability or security concerns, and without catering for multi-tasking aspects. In other words, the first cycle focuses in particular on the base concerns of operating systems. This way, many of the consistency issues mentioned above are simplified drastically.

Instead, during this first iteration we have to examine in particular how invocation frequency can be satisfied without overloading the local processing units, i.e. keeping the right balance between message passing overhead and page fault risk. We thereby lent from comparatively recent approach for dynamic load-balancing in large scale infrastructures: clouds.

a) Impact of Cloud-like Architecture

In the context of S(o)OS, applying cloud principles to the component architecture means that the individual components can be replicated and relocated on demand, i.e. dynamically. In other words, a component is instantiated as many times as suited best for the respective execution requirements and located so as to reduce average communication overhead and reduce page faults.

In classic cloud architectures, the replication and relocation behaviour is managed at run-time due to the access and usage behaviour of the respective services – for example, an auctioning house with variable numbers of bids and bidders will increase the numbers of servers hosting a specific bid according to the numbers of bidders interested in this specific item, and vice versa. This kind of dynamicity however requires (a) tight monitoring of the usage behaviour and (b) quick reaction to it – both are fairly easy to achieve in an environment, where monitoring and redistribution works in the same (or even faster) timewindows, such as in the internet of service domain. On the operating system level, however, this implies serious performance reductions that should only be risked for good reasons.

The major distinction between S(o)OS' cloud behaviour and the typical cloud scenario consists therefore in the control and management instance: in modern cloud systems, the service instances are typically self-controlled, where an increasing delay or a high latency automatically leads to scale-out and relocation behaviour. As opposed to that, the S(o)OS components are not self-controlled in the sense of controlling their own performance characteristics, as this would produce unnecessary overhead. We therefore envisage two triggers for cloud scaling behaviour: (1) triggered by the application according to extended annotations which define the OS' requirements and which in turn are interpreted as a scale-out / scale-in / relocation, and (2) triggered by dynamic reconfiguration of the environment. As noted, we ignore the dynamic aspects of the environment in this first project cycle.

Replication and relocation behaviour is controlled by the impact on performance, thereby considering the potential costs for this dynamic behaviour, and the consequences of this relocation. Cloud like behaviour can also be used in an exploratory training phase of the application, i.e. in an extended online analysis test – this is close to the dynamic case, where a component is triggered to relocate or scale out / in when the delays increase beyond a threshold, and where the gain seems to exceed the cost. Cost in this context is defined by the following aspects: (1) time for instantiation and relocation, (2) impact on communication overhead with the application, with other components and with other instances, and (3) impact on memory access, such as risk for page faults.

b) Conclusions

All components have been designed to allow principally for cloud like behaviour, i.e. they offer the capability to replicate and relocate their logic across the infrastructure. This allows the operating system to find a distribution of components according to the application requirements and the best mix of cost and benefit. The base decision for such behaviour is taken on grounds of the code dependency graph, respectively the according code annotations by the developer (or gathered through test runs).

The cost-benefit factor is thereby component-specific and varies according to the criteria of access rights and amount of synchronised data. It is a specific task of the second cycle to define these cost-benefit values and relate them in particular to the degree of dynamicity that such an OS can support.

IV. USABILITY DISCUSSION

Operating systems generally aim at supporting a wide range of use cases and applications – accordingly, evaluation of an operating system involves assessing its performance under various circumstances and with different types of applications. Given the scope of the S(o)OS project, this however is not feasible for multiple reasons:

- S(o)OS will not develop a fully functional, fully integrated operating system, but investigate into different structures, architectures and approaches to increase the scalability of operating systems
- Existing applications are all developed for a specific operating system, but S(o)OS does not aim at replacing an existing operating system, let alone being fully compatible with it.
- In order to investigate the full usage scope, application prototypes reflecting their typical behavior would be required – currently this exists only for selected use cases, such as the 13 dwarves for HPC [39]
- …

It must be noted thereby that the main challenges of S(o)OS consist in particular in addressing large scale and heterogeneous infrastructures, rather than trying to support the widest range of applications. It is therefore the primary goal to examine the capabilities of the OS architecture with respect to these major objectives (cf. also section I). At the same time, we must be careful to not focus too much on applications that adhere to these objectives and thereby neglect more regular application types, as this would imply that the operating system architecture is only valid for specific, isolated use cases.

In S(o)OS we follow therefore a bi-fold approach: on the one hand, we will focus on selected application types and their characteristics for actual evaluation, and on the other hand we will examine the components and their combinations with respect to usage restrictions. As has been noted in the preceding section, the first iteration of the project focused more on design and development of the individual functionalities and components, rather than on integration of the operating system as a whole. In the second cycle of the project, more evaluation mechanisms will be provided that allow for a more integrated evaluation of the OS architecture and the interactions of their components (cf. section V).

This first cycle therefore focused in particular on the assessment of the individual components and their respective usage scope, respectively restrictions.

1. COMPONENT APPLICATION SCOPE

Due to the nature of an operating system, the services developed in S(o)OS act on a very low level of the executing system, and thus have only indirect impact on the application behavior and therefore on the usage scope. Essential characteristics to look for are thereby related to principal scaling behavior, linkage and support to the executing code, impact on capabilities etc.

It must thereby be noted that provided functionality scope, i.e. application specific system calls are ignored. We assume that an application can generally be written just using the low level Turing operations, so that higher level complex operations are a generic support, rather than a requirement from the application side. This does not mean that future operating systems should not offer such support in order to support programming of typical, recurring tasks, but that an efficient operating system needs to focus primarily on the essential capabilities and functions (see also section III). In other words, the evaluation should not base on the functionality scope, but on behavioral restrictions, respectively support.

The following discusses in particular the principle characteristics of the components and their impact unto the application behavior

CODE ANALYSIS AND SEGMENTATION

Code analysis and segmentation can be considered a "passive" component, i.e. it does not impact on the application at run-time, unless runtime monitoring is activated in order to assess and restructure the program at runtime. Segmentation introduces communication and synchronisation operations into the program, which implicitly impact on the principle performance of the respective unit. Accordingly, the toolset works best on *applications that exhibit a high degree of concurrency*, e.g. by being event driven, or by executing multiple more or independent functions after another, such as multiple iterations over a matrix of grid cells.

CODE ADAPTATION

Similar to analysis and segmentation, adaptation takes a passive stance that does not directly interfere with the execution of the application. However, by converting the code for a specific destination platform, the performance of the respective code segment is implicitly affected. This is clearly due to the fact that the code was originally compiled and developed for a specific platform and that the conversion accordingly has to interpret code intention on a low level to identify the most equivalent interpretation on another platform. Code adaptation is not intended for conversion between completely different ISAs, but for minor adaptations to ensure compatibility between executing segments, such as processor precision. Accordingly, the component works best for conversions within a processor family and therefore for *applications that have clearly specified destination platforms*.

RESOURCE DESCRIPTION

The resource description is not a component as such, but a (more or less static) representation of the hardware specifics. The resource description itself does not impact on the application or restrict the usage scope. However, the resource description serves a usage-specific purpose, namely to identify the specific capabilities and requirements of a given platform regarding the application's behavior. The resource description therefore supports *in particular applications that have an unclear goal platform definition in their code,* i.e. that can principally be executed on multiple platforms.

RESOURCE DISCOVERY

The resource discovery system is essential for multiple tasks in a heterogeneous execution environment, in particular if dynamic aspects are considered during execution - with respect to the first phase, this is however of secondary importance. From the organization of code and protocol it becomes obvious that resource discovery works best for (a) recurring queries, (b) near neighbours and (c) hierarchically organized systems. In general, the application should not directly initiate resource discovery queries, yet it may cause the operating system to execute such queries, in

particular due to the results of analysis and segmentation (see above). However, the application structure can reflect the system structure to a certain degree and when the application does not prescribe the structure itself, the OS needs to map it to the system layout itself. Accordingly, the application should ideally follow a hierarchical structure itself, with little changes towards the resource requirements at run time – notably, this implies that the application actually has dynamic requirements. If the resource architecture is ideally hierarchical and supports recurring queries, this means that *the application itself tends towards a hierarchical organization with similar resource requirements (loops within loops)*.

SCHEDULING

Obviously, scheduling has a major impact on the application when it comes to aligning in particular parallel threads, but also comes to selecting and executing the right concurrent processes without causing delays in the overall execution. This is even more true if multiple tasks have to be scheduled at the same time, i.e. multi-tasking scheduling. In general, all problems boil down to the prediction of the execution times of the threads and their variability, as well as the interferences due to the OS and the execution times of OS-specific pieces of code (e.g., system calls, interrupts when present, etc.). In other words, the more precisely execution times statistics (such as the average, maximum or a percentile of its distribution) can be predicted, the more the schedulers will manage to satisfy their application-specific, as well as system-wide, goals (such as respecting the deadline constraints of applications). Therefore, it will be easier to keep under control various QoS metrics for the applications, for example their response-times or throughput. Note that, integrating knowledge and prediction about the execution times on the processors with the one about the data transmission times for communications, it is possible to predict the performance of parallel software components constituting a real end-to-end application distributed throughout the platform. This applies in particular to *applications with consistent behavior, non-regarding external dependencies, i.e. devices or data structure*.

TRANSACTIONAL MEMORY

As already discussed in section III.1.b), transactional memory is essential for maintaining data consistency across related threads in a simple way by employing message passing strategies. This allows higher scale and easier programmability, but cannot fully compensate the lack of scalability of consistency mechanisms in general. Implicitly, transactional memory is ideal for limited scope horizontal scale, such as between multiple subsets of threads. On the other hand, it is obviously unnecessary when the segments are essentially completely concurrent (embarrassingly parallel), so that it works *best for multiple groups of coupled threads executed independently of each other*, just as S(o)OS tries to achieve by identifying the concurrency in the application. In addition, transactional memory is particularly well suited for handling consistency over *non-deterministic data usage*, as is generally the case in strong horizontally scaled applications, such as clouds, social networks etc.

SPECULATIVE EXECUTION

Speculative execution tries to deal in particular with threads which have unclear data dependencies, i.e. that may be able to execute concurrently under specific conditions and where these conditions are not generally known. The main risk of speculative execution is obviously that the thread has to be rolled back and executed again, thereby reducing the performance over the purely sequential execution. Speculative execution should therefore only be employed if the *application has a general tendency towards high concurrency, unless specific conditions apply, such as external input or*

specific data types. The number of concurrent cases should thereby clearly outweigh the sequential cases, or otherwise sequential execution may lead to better performance figures – the exact cost-benefit ratio needs to be elaborated in the second cycle.

ASYNCHRONOUS COMPONENT-BASED KERNEL

The kernel forms the substantial basis for execution and as already noted thus also has the highest impact on the performance figures. At the same time, it exposes the least requirements towards the environment, its architecture or characteristics, as it builds just a local execution and communication environment. As it can cater for dynamic relocation and maintenance of the communication link, it supports specifically *applications with a strong I/O demand*, due to the asynchronous support for communication.

2. OVERALL APPLICABILITY SCOPE

It has already been discussed in section I.1 that for the specific objectives pursued by S(o)OS, the most relevant application domains consist in (1) High Performance Computing and (2) Real-Time Systems, as they both put forward requirements essential towards scalability, heterogeneity and efficiency. Both domains also are heavily impacted by the increasing heterogeneity of infrastructures and have to identify means to deal with them efficiently.

a) HIGH PERFORMANCE COMPUTING

Strong scaling HPC applications are denoted by the strong data dependency between the individual instances across the infrastructure. Typical HPC applications exhibit a homogeneous communication need (all nodes communicate at similar times with similar amount of data to same "organization" of neighbours) with high communication frequency and large amount of concurrent messages. Similar, most applications have been developed for homogeneous infrastructures, i.e. where not only the resources are identical to one another, but more importantly, where the mode of communication is identical between all code instances. More recent developments investigate heavily into exploiting the heterogeneity to reduce the unnecessary overhead and performance impact. HPC applications ideally strive to exploit a maximum number of processing units to improve the total performance.

As HPC applications have a strong mathematical focus, the segments and their individual behavior are clearly identifiable. However, as discussed in section II.1, the analysis and segmentation toolset will tend to interpret the synchronous library calls in a fashion that's mostly similar to their sequential execution. This has only secondary impact though as for a proper HPC application it is more important to identify the concurrency across the individual threads or segments (such as across loops), than within. As modern infrastructures tend towards more heterogeneity, adaptation of the code becomes of more interest – however, adaptation on code level should only be executed when recompilation is not possible: since processing units in an HPC domain tend not to be dynamic, adaptation on binary code level should only be executed when a large set of resources become available that match the specific behavioural structure of the according segments. The current resource description provides enough details for identifying essential characteristics that are of interest for execution and in particular distribution, but will require further extension to steer code adaptation, or potentially (re)compilation.

The restricted dynamicity and comparatively limited heterogeneity also implies that resource discovery (and description) do not have to be constantly updated, respectively have to exhibit high

efficiency – instead it is more important, that sufficient information about the infrastructure, its layout and capabilities is provided prior to execution, i.e. prior to alignment of the segments. As modern high performance computer show a strong hierarchical organization (cluster > rack > slate > node > …) which is ideal for the organization of the resource discovery protocol (see there). Due to the high conformity of parallel segments in terms of behavior, the impact of predictability is reduced for the scheduler and applies mostly to concurrent *sets* of execution segments.

Most HPC developers build up on some form of shared data architecture – be that physically (UMA), hardware supported (ccNUMA) or virtual (e.g. PGAS[7]) – as this is the easiest way to handle the data communication across parallel instances. Due to the differences in architecture, however, this can lead to significant performance issues. Both the transactional memory support, as well as the analysis and dependency resolution support the developer in even simpler data sharing development. By exploiting the resource specific characteristics and the layout as gathered from resource discovery, it is furthermore possible to choose the best communication modes within the asynchronous kernel. In other words, S(o)OS takes over the responsibility for modern hybrid programming models with little effort on behalf of the developer.

Overall, the decreased number of page faults for system calls and the increased resource utilization promises a considerable improvement in efficiency. As Planas, Badia et al. have already shown, the exploitation of concurrency is a generally feasible model for improved HPC performance [47], yet the programmability and the tendency for errors obliterates full exploitation of this concept, which S(o)OS overcomes. To further simplify usage, speculative execution can ensure correct behavior even given wrong annotations, but due the performance impact this should only be executed for significant data structure changes or lacking knowledge about the concurrency behavior of the code.

b) Real-Time Systems

Parallel real-time applications are characterized by timing constraints (e.g., deadline, response-time, and throughput) that may be so tight, relatively to the sequential execution of the involved algorithms, to require the scale out over multiple resources, in order to meet the constraints. This means that, on one side, we can find again the requirements towards the development and run-time environment typical of high-performance computing, but on the other side it is not necessary to push the level of parallelism beyond the (safe) thresholds that allow these applications to meet their timing constraints. Also, normally parallel real-time applications may be mostly embarrassingly parallel or loosely coupled, rather than requiring tight interactions among the parallel segments. This relaxes somewhat the requirements over the OS/run-time in terms of absolute efficiency and performance, in favour of maximum predictability of the timing behaviour at run-time.

For this type of workload, the operating system needs to make it easier for the programmer to develop the application over the foreseeable heterogeneous platforms, without the hand-tuning phase typical of the development cycle for real-time and embedded applications, in which the developer tries various critical code segments multiple times over the available computing unit types, in order to profile the achievable performance in each possible working condition. Instead, with S(o)OS, such profiling, when necessary, is done by the OS, and it is reduced to the bare minimum needed to output the needed estimations of performance for the various possible mappings of the application threads onto the physical resources for execution. During such an action, resource

[7] http://www.pgas.org/

discovery and matching play a critical role, in that they allows the OS to find appropriate resources that may manage to fulfil the timing requirements of the application. Similar to the HPC case, it depends on the degree of dynamicity in how far performance of the discovery impacts on the use case – more relevant in general is the accuracy of the description, and its mapping to the code specifics (respectively the correct interpretation of the characteristics). Obviously, the structure of the platform is not predefined in these cases (with respect to their hierarchy).

Rather than targeting hard real-time systems where deadlines cannot be violated, S(o)OS targets mostly soft real-time ones. This means that the primary mechanisms within the OS (e.g., for scheduling) may rely on rough estimates of the execution times. However, appropriate dynamism embedded within the scheduler, coupled with appropriate monitoring functionality, allows for dynamically reacting to sensed conditions that significantly vary from the (roughly) foreseen conditions. Situations that may require significant reconfiguration on this side may be the ones in which code re-adaptation is required, as the performance data relative to the resulting re-adapted/re-compiled code may not be available or accurate.

Other cases may include those in which the optimistic speculative parallel execution is activated. A parallel real-time application cannot expect the OS to take all decisions automatically, even though this may sometimes be possible to some extent. The best conditions are met when the application collaborates with the OS communicating explicitly its own dependency constraints, as well as its own requirements in terms of overall end-to-end latency/response-time or throughput. This is supported by the analysis and segmentation toolset, as well as through exposure of proper APIs and middle-ware layers tightly coupled with the OS scheduler at various levels of the hierarchical scheduling infrastructure (which copies the hierarchical arrangements of computing units, i.e., within a single system, node, processor or tile).

Also, considering the API level, Transactional Memory may constitute a viable communication and synchronization means in parallel real-time applications as well. In this case, it is crucial that the possible conflicts arising within each transaction be somewhat identifiable (either in a deterministic sense, or, more likely, in a probabilistic one). This allows building meaningful estimates of the overall execution time that it may be needed to commit a single transaction (including the re-executions due to aborts and restarts). Finally, in order to limit the occurrence of conflicts in code segments making use of (S)TM, particular scheduling policies may be embraced that fine-tune their behavior depending on the awareness that the various applications are traversing a transactional block.

3. CONCLUSIONS

The current scope of S(o)OS usability shows clearly that the components deal specifically with the aspects of large scale and heterogeneity. The few cases in which potential conflicts exist are typically contained by switching off one of the according features, such as transactional memory in embarrassingly parallel applications. This in turn is possible due to the component-based nature of S(o)OS. The next iteration of the S(o)OS project will investigate the interdependencies between component architectures with respect to the application scope of the integrated OS in more detail. To this end, the specific characteristics of high performance computing and real time applications (in terms of their behavior) will be elaborated further and encoded into the simulation frameworks (cf. section V).

V. CONCLUSIONS & FUTURE WORK

The first iteration of the project has focused on identifying and assessing the essential functionalities needed for composeable, scalable operating systems as envisaged by S(o)OS – this included:

- Functionalities that did not meet the scalability or heterogeneity requirements, such as
  - Scheduling
  - Monitoring
  - …
- Functionalities to support these requirements but did not exist in current OS, including
  - Code analysis & segmentation
  - Code adaptation
  - Resource discovery
  - …
- Functionalities that were subsumed in other OS capabilities and thus could not be replaced, adapted etc., such as
  - Messaging
  - Transactional memory
  - …

The main task thereby consisted in identifying the promising capabilities to meet the objectives and identify points of further work for the second iteration, respectively promote changes in the approaches chosen so far. The results thus have a strong impact on the second project cycle in terms of which work to (dis)continue and which aspects to respect when building up (and testing) the integrated architecture.

Within this chapter we will summarise the main steps to be approached within the next project cycle.

1. COMPONENT NEXT STEPS

This section provides a summary over the main conclusions derived from the evaluation results *per component.* It focuses mostly on what development and research work has to be done with respect to the existing component – therefore this section identifies which components should be continued, what kind of work remains to be done and whether alternatives should be investigated.

a) EXISTING COMPONENTS

CODE ANALYSIS AND SEGMENTATION

The analysis and segmentation approach seems to be principally promising, with main deficiencies arising in particular from the lack of intelligence for interpreting blocks and connections. In particular the merging algorithm tends to be overoptimistic – this needs to be carefully evaluated against too large graphs that would lead to too much segmentation processing overhead. Accordingly, even though the analysis and segmentation algorithm primarily serves off-line purposes, the performance of the algorithm itself needs to be evaluated in a bit more detail.

Since the approach itself seems feasible, the main focus of the second cycle will consist in particular in developing and assessing the means to supervise code behavior at run-time and derive the data fixed points from this. This will go hand in hand with an improved means to assess cost (derived from timing behavior) to evaluate the benefit from potential run-time re-segmentation and rearrangement. To this end, timing assessment methodologies from the real-time domain will be examined and integrated.

BINARY CODE ADAPTATION

For the second cycle of DISTae we plan to work on these areas to solve some of the current limitations:

- The first proof of concept presented in this document is limited to the adaptation of code that has only basic types for segment inter communication, this is quite limiting for programmers that is the reason we want to analyse how to handle complex and composed data structures (like C structs and unions), including memory regions (arrays and pointers). With this we will allow the possibility to distribute segments of real code as by now pure functions can be distributed, which is not enough in current programming models.
- The executable package generated by DISTae Creator contains the source code of the program to later adapt it to the native architecture. In the next iteration we will study the use of semi-compiled code for distributing the program segments. With this we expect to noticeably decrease the adaptation overhead when executing a program. It also has other advantages in situations when the code is not wanted to be read.
- Make it collaborate and coexist with other S(o)OS modules and components, like the code analysis and segmentation component, as it is the responsible of deciding in the segmentation.

RESOURCE DESCRIPTION

The evaluation results of the CλaSH prototype, discussed in chapter II.3, are promising, in that many of the evaluation criteria are (partially) met. For those aspects where the prototype is working subpar, potential improvements were suggested. We will consequently try to explore the given suggestion in the next design cycle (within Work Package 2):

- As discussed in section II.3.c), although we designed a new notation that can uniformly combine combinational and stateful components, we still need to make this addition to the CλaSH prototype.
- To make the descriptions more comprehensive, so that the resulting hardware matches always the description, we will attempt to make a formal proof of the transformation system. And if shown that the current set of transformation is insufficient, we will add new transformations (and formulate respective proofs) to the CλaSH prototype.
- We will investigate the creation of design templates for instruction-set machines that are susceptible for name-driven structural/static analysis.
- We will investigate on how to include non-functional properties (e.g. latency, energy usage, etc.) of a complete system in the hardware descriptions.

RESOURCE DISCOVERY AND MATCHING

In the results presented in section II.4 we had assumed that the list of nodes taking part in the system remains stable. This is of course unrealistic, and we should rather predict that the network will exhibit a large degree of dynamicity due to massive node joins, leaves and failures. That's the issue that we are planning to work on in the next phase of the So(o)S project . Currently 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 phase 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). 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, therefore to achieve consistency and avoid discovering invalidated services, RD must be fault tolerant. 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.

COMMUNICATIONS IN TILE-BASED SYSTEMS

We are also in the process of examining the effects of task division and topology knowledge on applications that are multi-staged. We consider where one or more stages maybe executed in parallel and how these sub-stages may be mapped to cores. The issues that we are interested in are the overheads added as a result of this parallelization, the relation of overheads to the degree of parallelization and the net effect of this on the end-to-end processing time.

Further, we plan to explore the methods to reduce the communication overheads and overall memory access latency by designing a strategy to determine and map data-intensive stages on cores closer to memory, and compute intensive stages elsewhere. One simple approach that we have experimented with is to reduce inter-stage communication latency by mapping stages onto cores with the aim of achieving minimal inter-core routing distances (see section 5.c of [34]).

ALLOCATION AND MAPPING

In our experiments with pipelined applications we explored stages which are themselves executed sequentially on one core. As indicated in the future work with regards to communications in tile-based systems (see above), we will explore parallelization of a stage itself. This parallelization introduces challenges in mapping these sub-stages to cores, in addition to the overall application mapping. Further, changes in the availability of resources themselves might either necessitate changing the degree of parallelization and the mappings.

Further, specific characteristics of the stages or applications might demand strategies of mapping that have to take in account factors such as memory-intensiveness or inter-stage communication-intensiveness. Identifying these phases or segments and deploying them in a manner that best fits each of these requirements are also part of the experiments we will work on.

COMPONENT-BASED ASYNCHRONOUS KERNEL [IT]

In a first evaluation of our work we evaluated the performance of the asynchronous support in the kernel and validated it at a local level. Our next step is to enable migration across machines in network environment and evaluate communication strategies, protocols and the different approaches to achieve full or partial migration of a process.

Further, we will also tackle communication at higher levels and how to handle location independent addressing due to process migration in a network environment. This will require developing protocols and new design approaches to network stacks to handle the location dependence characteristic of high-level communication protocols such as TCP/IP.

TRANSACTIONAL MEMORY

The evaluation of the Transactional Memory component indicates that is reaches the expected goal of performance and scalability on a small set of workloads.

- We are currently extending our set of workloads to a more realistic application that uses a MapReduce to analyze text file. These kinds of applications are more specific to the HPC use-case than the benchmarks tested so far.
- We also envision to compare the performance of our Distributed Transactional Memory protocol on a multi-core platform to confirm that it adequately benefit from the scalability offered by many-core systems. In particular, we expect to obtain from our many-core protocol the same overall performance as the one obtained on multi-core but with simpler cores.
- We also plan to adapt the protocol for exploiting locality further. The current design adopts a radical direction in avoiding the use of any cache-coherence protocol to respect the message-passing principle. One could adhere to a less radical approach by implementing a lightweight cache-coherence in software, by letting a core piggybacking the grant access and the read value in a single message. Such an approach would allow the implementation of a partitioned global address space (PGAS) programming model, where the responsible core could cache the value it is responsible for. It remains unclear what performance and scalability benefits such an approach could provide though.
- Last but not least, we are willing to evaluate the transactional memory component in a different context: to implement real-time strategy game application. The idea is to exploit the transactional memory to parallelize the strategy engine of an open-source game application: Globulation2.

SPECULATIVE EXECUTION

For the second cycle we plan to study speculative execution using threads with different memory address space which is the situation that we face in distributed environments and more precisely in HPC. Simultaneously we will study how to help the code analysis and segmentation component to detect the areas where SAM could be used. This includes the development of analysis and performance tools to determine the effectiveness of SAM in real applications.

b) OTHER / ADDITIONAL COMPONENTS

In addition to the assessment of the existing components, the evaluation, in combination with the architectural work executed in work package 5 (see [34]) also led to identification of additional capabilities needed to reach the full functionality scope. For example, it can be seen from the architectural evaluation and the cloud component approach that additional cloud management capabilities are needed that are tightly integrated with the OS monitoring functionalities.

These functionalities will be investigated in more detail in the second iteration of the technical work packages WP2 (hardware), WP3 (protocols) and WP4 (execution management) and the according

results will be published in the next work package reports. More specifically, we can identify the following issues:

- Messaging:
  Transparent invocation of components involves efficient messaging support that can deal with a range of different infrastructures. Though the asynchronous kernel component can deal with routing support, additional functionalities are needed in particular to ensure that code invocations and memory accesses are coordinated accordingly – which may also involve serialization and conversion aspects.
  With the cloud architecture support, the location and amount of instances can principally vary dynamically, thus affecting routes, protocols, and potentially also conversion aspects. In order to maintain transparency, i.e. in this case that the code does not have to be rewritten with every rearrangement of instances, code invocations need to be redirected to according management instances (cf. code adaptation below)
- General code adaptation:
  In addition to the binary adaptation between platforms, some means for general adaptation of code to maintain communication paths, state and data consistency etc. This relates to the code analysis and segmentation functionalities, where dependencies are identified and have to be resolved with segmentation. Different means for this resolution have to be investigated, along the line of runtime adaptation, page fault interception etc. (see also D42)
- Cloud management:
  As noted, the architecture follows the principles of cloud mechanics to ensure availability and best distribution across the infrastructure according to communication and interaction requirements. However, as opposed to the cloud domain, where sufficient time for adaptation and relocation exists, the operating system has to react more quickly and ideally find an efficient arrangement prior to execution basing on analysis and segmentation. Implicitly, the means for efficient distribution and management need to be examined further in the second iteration. This also implies support for
  - Migration of code segments and OS services
  - Replication of instances
  - State maintenance across different replicas
- Cost assessment
  All management choices, including segmentation and distribution of the code segments, must base on an assessment of the effect, respectively impact of such a choice. In the context of S(o)OS the main effect to achieve consists in improving the efficiency of application execution. With respect to dynamic tasks, we talk about the “cost” of a management choice as the impact it has (negatively) on the execution efficiency – in other words, the primary task of S(o)OS consists in reducing the cost for rearrangement, relocation etc. The second iteration must therefore identify the means to assess costs and relate them to each other.
  This obviously affects multiple components.

2. EVALUATION NEXT STEPS

It was noted at multiple points in the evaluation that the components' capabilities and characteristics in the scope of the OS depend on the interactions with other components and their according functionalities. Even though the cloud based approach allows for a certain flexibility regarding the instance and relationship management with the infrastructure (cf. section III.2), the actual behavioural details will have to be assessed separately to provide concrete performance results, and implicitly to assess the applicability scope.

With the base components and algorithms having advanced to a point where they exhibit promising results in multiple areas, the evaluation of the interactions and the impact of components upon each other can provide more sensible results than it would have done in this iteration (cf. section II). As the evaluation identified which of the selected approaches and behaviours are principally sensible for defining the overarching OS architecture, the next cycle can couple the selected behaviours and assess their interdependencies. This is obviously a complex task, in particular since the functionalities are neither defined nor realized on the same level, i.e. they cannot be integrated with classical software engineering approaches. Instead, it will be necessary to assess their principle behavior and influence upon each other in a way that still provides enough information for evaluation. Obviously, a formal approach would principally be possible, yet would generate a complexity that would unnecessarily stall all other development in the project. As has been discussed to greater detail in chapter I, we instead try to model the behavior and interactions in a semi-formal fashion using abstract simulations. This allows us to quickly assess the model in the context of different boundary conditions, such as infrastructure characteristics, application behavior tendencies etc.

As part of the second iteration, the work package will therefore develop simulation means that allow for evaluation of the interactions and their impact on the application performance. In this context, the specific application characteristics will have to be defined more formally to assess the relation between application and infrastructure (OS) behavior. The simulation should thereby help in identifying, amongst others, the following characteristics: (1) cost for dynamic behavior, (2) performance impact from distributed execution, (3) scalability improvements, (4) impact from architectural choices etc.

REFERENCES

[1] Niedermeier, A. and Wester, R. and Rovers, K.C. and Baaij, C.P.R. and Kuper, J. and Smit, G.J.M. (2010) Designing a dataflow processor using CλaSH. In: 28th Norchip Conference, NORCHIP 2010, 15-16 November 2010, Tampere, Finland. 69. IEEE Circuits and Systems Society. ISBN 978-1-4244-8971-8

[2] Niedermeier, A. and Wester, R. and Baaij, C.P.R. and Kuper, J. and Smit, G.J.M. (2010) Comparing CλaSH and VHDL by implementing a dataflow processor. In: Proceedings of the Workshop on PROGram for Research on Embedded Systems and Software (PROGRESS 2010), 18-19 Nov 2010, Veldhoven, The Netherlands. pp. 216-221. STW Technology Foundation. ISBN 978-90-73461-67-3

[3] Gerards, M. and Baaij, C.P.R. and Kuper, J. and Kooijman, M. (2011) Higher-Order Abstraction in Hardware Descriptions with CλaSH. In: Proceedings of the 14th EUROMICRO Conference on Digital System Design, Architectures, Methods, and Tools, DSD2011, 31 August – 2 September, Oulu, Finland. pp. 495 – 502. IEEE Computer Society. ISBN 978-0-7695-4494-6

[4] Paterson, R. Arrows and computation. The fun of Programming. pp. 201 – 222, 2003.

[5] Hughes, J. Programming with arrows. Advance Functional programming: 5th international school, AFP 2004, Tartu, Estonia, August 14-21, 2004: revised lectures. pp. 73 – 129, 2005.

[6] Paterson, R. A new notation for arrows. In: Proceedings of the sixth ACM SIGPLAN international conference on Functional programming – ICFP '01, 2001. pp. 229 – 240.

[7] Jones, S.P., Editor. Haskell 98 language and libraries, ser. Journal of Functional Programming, 2003, vol. 13, no. 1.

[8] Fauth, A., J. Van Praet and M. Freericks, "Describing Instruction Set Processors Using nML," Proceedings of the European Design and Test Conference (ED&TC), Paris, March 1995, pp.503-507.

[9] Zivojnovic, V., S. Pees, C. Schlager and H. Meyr, "LISA - Machine Description Language and Generic Machine Model," ICSPAT, Boston, 1997.

[10] Goossens, G., Lanneer, D., Geurts, W., Van Praet, J. , "Design of ASIPs in multi-processor SoCs using the Chess/Checkers retargetable tool suite," System-on-Chip, 2006. International Symposium on , vol., no., pp.1-4, 13-16 Nov. 2006

[11] VHDL Language Reference Manual, IEEE Std. 1076-2008, 2008.

[12] Verilog Hardware Description Languages, IEEE Std. 1365-2005, 2005.

[13] R. Raman, M. Livny, and M. Solomon. Matchmaking: distributed resource management for high throughput computing. In IEEE HPDC, 1998.

[14] P. Rompothong and T. Senivongse. A query federation of UDDI registries. In Proc. of the 1st International Symposium on Information and Communication Technologies. Trinity College Dublin, 2003.

[15] Meng, S.C., et al., A statistical study of today's Gnutella. Frontiers of Www Research and Development - Apweb 2006, Proceedings, 2006. 3841: p. 189-200.

[16] Carlini, E., et al., Reducing traffic in DHT-based discovery protocols for dynamic resources. Grids, P2p and Services Computing, 2010: p. 73-87.

[17] Tannenbaum, T. and M. Litzkow, The Condor Distributed-Processing System. Dr Dobbs Journal, 1995. 20(2): p. 40-&.

[18] Anderson, D.P., BOINC: A system for public-resource computing and storage. Fifth Ieee/Acm International Workshop on Grid Computing, Proceedings, 2004: p. 4-10.

[19] Punch: An architecture for web-enabled wide-area network-computing. Cluster Computing, 2(2):153–164, 1999.

[20] Zhang, X.H. and J.M. Schopf, Performance analysis of the globus toolkit monitoring and discovery service, MDS2. Conference Proceedings of the 2004 Ieee International Performance, Computing, and Communications Conference, 2004: p. 843-849.

[21] S. Zanikolas, and R. Sakellariou, "A Taxonomy of Grid Monitoring Systems," Future Generation Computer Systems, 21(1):163–188, 2005.

[22] Basu, S., et al., NodeWiz: Peer-to-peer resource discovery for Grids. 2005 IEEE International Symposium on Cluster Computing and the Grid, Vols 1 and 2, 2005: p. 213-220.

[23] Devarakonda, R., et al., Mercury: reusable metadata management, data discovery and access system. Earth Science Informatics, 2010. 3(1-2): p. 87-94.
[24] Albrecht, J., et al., Design and Implementation Trade-Offs for Wide-Area Resource Discovery. Acm Transactions on Internet Technology, 2008. 8(4).
[25] A. Iamnitchi and I. T. Foster. On fully decentralized resource discovery in grid environments. In Proceedings of the Second International Workshop on Grid Computing (Grid'01), pages 51–62, London, UK, 2001. Springer-Verlag.
[26] Drost, N., et al., Zorilla: a peer-to-peer middleware for real-world distributed systems. Concurrency and Computation-Practice & Experience, 2011. 23(13): p. 1506-1521.
[27] Reddy, M.V., et al., Vishwa: A reconfigurable P2P middleware for grid computations. 2006 International Conference on Parallel Processing, Proceedings, 2006: p. 381-388.
[28] Craig Hunt, Chapter 8, TCP/IP Network Administration, O'Reilly Publishers, Jan. 1998, pp 48-78.
[29] Hung, N.C., N.H. Giang, and T.P. Yew, Performance evaluation of distributed hash table (DHT) chord algorithm. Proceedings of the ISSAT International Conference on Modeling of Complex Systems and Environments, Proceedings, 2007: p. 16-20.
[30] Mastroianni C., Talia D. and Verta O. Evaluating Resource Discovery Protocols for Hierarchical and Super-Peer Grid Information Systems. 19th Euromicro Intern. Conf. on Parallel, Distributed and Network-Based Processing (PDP'07).
[31] Juri Lelli, Giuseppe Lipari, Dario Faggioli, Tommaso 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.
[32] 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. Kreuz. V. Gramoli. S(o)OS Project Deliverable D5.1 – State of the Art, July 2010.
[33] 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.
[34] T. Cucinotta, G. Lipari, R. Aguiar, J. P. Barraca, B. Santos, J. Zarrin, J. Kuper, C. Baaij, L. Schubert, D. Rubio Bonilla. S(o)OS Project Deliverable D5.3 – First Set of OS Architecture Models, August 2011.
[35] L. Schubert, D. Rubio Bonilla, R. Aguiar, B. Santos, J. Zarrin, T. Cucinotta, K. Kuper, C. Baaij, V. Gramoli, A. Draojevic. S(o)OS Project Deliverable D6.1 – Evaluation Criteria, February 2011
[36] V. Gramoli, T. Cucinotta, L. Schubert, D. Rubio Bonilla, A. Dragojevic, J.P. Barraca, B. Santos, J.Kuper. S(o)OS Project Deliverable D4.2 - First Implementation Set: Execution Management, April 2011
[37] C. Baaij, J. Kuper, J. Zarrin, L. Schubert, V. Gramoly, T. Cucinotta, D. Rubio Bonilla. S(o)OS Project Deliverable D2.2 - First Implementation Set: Hardware, April 2011
[38] B. Santos, J. Zarrin, J.P. Barraca, R. Aguiar, C. Baaij, J. Kuper, D. Faggioli, G. Lipari, T. Cucinotta, A. Dragojevic, V. Trigonakis, V. Gramoli, D. Rubio Bonilla, L. Schubert. S(o)OS Project Deliverable D3.2 - First Implementation Set: Protocols, April 2011
[39] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, K. A. Yelick, "The Landscape of Parallel Computing Research: A View from Berkeley", EECS Department - University of California, Berkeley, Technical Report No. UCB/EECS-2006-183, 2006. Available at: http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.pdfdwarfs
[40] Mechthild Stoer, Frank Wagner "A Simple Min-Cut Algorithm", Journal of the ACM, vol. 44, No.4, July 1997, pp.585-591
[41] Baaij, C.P.R. and Kooijman, M. and Kuper, J. and Boeijink, W.A. and Gerards, M.E.T. (2010) CλaSH: Structural Descriptions of Synchronous Hardware using Haskell. In: Proceedings of the 13th EUROMICRO Conference on Digital System Design: Architectures, Methods and Tools, 1-3 Sep 2010, Lille, France. pp. 714-721. IEEE Computer Society. ISBN 978-0-7695-4171-6
[42] Baumann, A., Barham, P., Dagand, P.-E., Harris, T., Isaacs, R., Peter, S. 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, pages 29-44
[43] D. C. Schmidt, "Reactor: An Object Behavioral Pattern for Concurrent Event Demultiplexing and Event Handler Dispatching," in Pattern Lan- guages of Program Design (J. O. Coplien and D. C. Schmidt, eds.), Software Pattern Series, ch. 29, pp. 529–545, Reading, Mass.: Addison- Wesley, 1995.
[44] I. Pyarali, T. Harrison, D. C. Schmidt, and T. D. Jordan, "Proactor - An Architectural Pattern for Demultiplexing and Dispatching Handlers for Asynchronous Events," September 1997.

[45] “Handling IRPs: What Every Driver Writer Needs to Know,” August 2006. http://msdn.microsoft.com/en- us/windows/ hardware/gg487398

[46] T. Harris, M. Abadi, R. Isaacs, and R. Mcilroy, “AC : Composable Asynchronous IO for Native Languages,” in Proceedings of the 2011 Conference on Object-Oriented Programming Systems Languages and Applications OOPSLA 2011, 2011

[47] J. Planas, R.M. Badia, E. Ayguadé, and J. Labarta, "Hierarchical Task-Based Programming With StarSs," International Journal of High Performance Computing Applications, vol 23 (3), 2009, pp. 284-299.