S(o)OS

Service-oriented Operating Systems

Definition of Future Requirements

Project Deliverable D5.2

Tommaso Cucinotta, Giuseppe Lipari [SSSA];
Rui Aguiar, João Paulo Barraca, Bruno Santos [IT]; ;
Jan Kuper, Christian Baaij [UT];
Lutz Schubert, Hans-Martin Kreuz [USTUTT-HLRS]

Due date: 31/07/2010
Delivery date: 28/07/2010

SEVENTH FRAMEWORK PROGRAMME

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

Version History

Version	Date	Change	Author
1.0	28/07/10	Delivered to the reviewers	

Executive Summary

This document assesses the expected development in the area of (large scale) computing infrastructures expected in the future from different standpoints, with a particular focus on the main IT tiers, namely hardware (across all layers), communication (including protocols), middleware (with a main focus on Operating Systems), and applications (including their programmability). We shall thereby distinguish between their structure and their usage, i.e. between the goals pursued by the manufacturers and the requirements posed by the users. This document therefore investigates the problem from three major perspectives:

(1) the short-term developments to be expected in the major tiers. As it can be expected, these are based on current production attempts, official roadmaps etc.. In facts, most of the short-term developments are quite clearly laid out, though it should be noted that this does not imply that all these roadmaps will actually be pursued in full detail. In the corresponding sections (II. 1. , III. 1. , IV. 1. and V. 1.) a strong tendency towards increasing scale and parallelisation of systems will be noted. As is the general (unwritten) manufacturing rule and with manycore technologies etc. still being in a "prototyping" stage, progress is comparatively slow as most commercial organisations only make careful attempts first. This is mostly due to the fact that little expertise about multicore systems exist as yet and has to base mostly on knowledge gained from HPC so far.

(2) More interestingly, however, is the long term development to be expected in these domains. The nature of such an assessment must be highly speculative due to the fast changes this area is subject to and due to the lack of expertise as noted above. Accordingly, many statements here are essential assumptions basing on the knowledge and expertise of the participants in the project. In order to maximise the degree of certainty, we needed to extract relatively generic assessments that, whilst being generic, still provided enough details to be meaningful for estimating future requirements. In general, we can note that the long term trends in general computing systems will be oriented along the line of current high performance computing (HPC) architectures, whilst HPC itself will introduce additional levels of hierarchy and different means to communicate across and along these levels.

(3) A major influence in development trends comes from usage of the system and therefore from the demands of the end-user. Even though industry will try to lead such development in certain directions fitting their own manufacturing plans, they nonetheless are subject to sales figures. In the final section we accordingly examine the *usage* trends to be expected in the long term future and their impact on computing infrastructures in the sense of which systems best suit the respective needs. In this context, we must obviously distinguish between such contrasting domains such as high performance computing and desktop computing, even though a general trend towards convergence of the two domains can already be noted. The main problems thereby consist less in usage though (interactivity in HPC is a nice, but often unnecessary feature; similarly, fast complex calculations in spreadsheet manipulation is not a mandatory feature), but in particular in programming focus and implicit complexity – with one of the main issues being parallelisation.

It must be noted in this context that the document is far from being a *complete* assessment of *all* developments related to computing systems, application requirements etc. - this is not only due to a lack of time and available effort, but in particular due to the fact that providing more detail would only be of limited additional benefit to the project: as S(o)OS does not aim at addressing an individual aspect optimally, but tries to identify the common issues across all these aspects, it is of primary concern how they interrelate and thus which requirements need to be addressed on a higher level. Along the same line, S(o)OS is less affected by the degree of uncertainty involved in assessing the long term developments.

Table of Contents

1. List of Figures

2. List of Tables

3. Abbreviations

Abbreviation	Description
AVX	Advanced Vector Extension
BIOS	Basic Input/Output System
CISC	Complex Instruction Set Computer
CDMA	Code division multiple access
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
DSP	Digital Signal Processing
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
GFX	Graphics Accelerator
GPP	General-purpose Processor
GPGPU	General-purpose Graphical Processor Unit
GSM	Global System for Mobile Communications
HD	High Definition
HDTV	High Definition Television
HPC	High-performance Computing
HSPA	High Speed Packet Access
IB	Infiniband
ISA	Industry Standard Architecture
IT	Information Technology
MESIF	Modified (M), Exclusive (E), Shared (S), Invalid (I) and Forward (F)
MIMD	Multiple Instructions Multiple Data
MIMO	Multiple-Input and Multiple-Output
MMO	Massive Multiplayer Online
MTU	Maximum Transmission Unit
MPI	Message Passing Interface
NoC	Network-on-a-chip
NUMA	Non-Uniform Memory Access
OS	Operating System
p2p	Peer to peer
PDA	Personal Digital Assistant
PGAS	Partitioned Global Address Space
QPI	Intel Quick Path Interconnect
QoS	Quality of Service
RAM	Random Access Memory
RCU	Read-copy-update
RDMA	Remote Direct Memory Access
RISC	Reduced Instruction Set Computer
SIMD	Single Instruction Multiple Data
S(o)OS	Service-oriented Operating System(s)
SSL	Secure Sockets Layer
TDMA	Time division multiple access
UMA	Uniform Memory Access

I. Introduction

A quick overview over the document's background and structure

The IT sector is a very dynamic and versatile industrial domain in which no single specific line of development is pursued. Substantial changes occur every so often which have to be catered for by the respective environment. For example, new cache and interconnect models require new processor architectures; new processor architectures directly affect compilers, but also the Operating System; new Operating Systems may imply new programming models, etc.

The most *visible* change of this type occurred during recent years with the introduction of multi-core processors and now with the movement from cores to tiles. However, even here the actual implications on OS architectures, interconnect specifications, cache coherency etc. only slowly begin to emerge and become visible for developers and designers.

So far, it is not possible to identify the best approach for dealing with the future concurrent environments, where not only cores but also pipelines, cache and RAM modules, I/O etc. are accessed in a (massively) parallel way. Processor manufacturers such as Intel are still examining different architectures and protocols, whilst Microsoft and others examine new Operating System models for dealing with such environments. Industry is primarily driven by (short-term) goal-orientation, i.e., even though all manufacturers will try to look ahead beyond the next 10 years to elaborate appropriate strategies, they will have to execute steps limited by such factors as feasibility, evaluation results etc., always trying to remain cutting-edge. Accordingly, long-term development typically deviates strongly from the short-term based predictions.

S(o)OS, like other initiatives (see e.g. [1]), takes a disruptive approach towards such development, addressing the issue from the perspective of long-term requirements and environments first, rather than trying to advance existing distributed execution support in a step-by-step fashion. Only this way can the appropriate merge and alignment of the individual developments be ensured and the necessary performance and efficiency, and thus long-term validity be achieved.

Accordingly, a long-term prediction of IT development needs to form the basis of the necessary assessment and research work. As noted, such predictions bear the strong risk of maladjustment due to the high dynamics of IT development.

This document tries to lay the foundation for such work by estimating the IT environment in 10-15 years time from now. Given this time-scope, such an estimation can almost only be highly speculative in nature. To reduce the risk of wrong predictions, the project builds on three major information sources:

1. expertise in all related technological areas [redacted] involved in the related research and development;
2. development trends in the IT landscape, basing on current technologies and plans, as well as lessons learned from recent developments;
3. the fact that all development is essentially driven by usage needs and hence by the typical application landscape and their expected development.

This deliverable can therefore be regarded as a follow-up to the D5.1 State of the Art document [2]. It builds on current cutting-edge R&D and the short-term trends inherent to these developments. Basing on the short-term trends and the development over recent years, as well as the research agendas and industrial strategies, long-term trends can be identified that mostly reflect essential tendencies, such as exponential increment in core numbers. By nature, these trends are much more generic than short-term ones, simply because future development is more unpredictable.

Obviously, these trends have significant impact on future ways of dealing with the infrastructure, as e.g., just the number of cores expresses the degree of scalability that needs to be supported etc. As noted, an essential driver for all development is uptake, i.e., usage of the system – notably, even if different application areas need to be distinguished, producers in all areas have to weigh costs versus benefits, thus tending towards covering multiple areas at the same time.

As already discussed in the context of the state of the art document [2], current development is mostly dictated by the impact of multi-core processors on individual domains, but also strongly influenced by the developments with respect to the future Internet, i.e., towards a more heterogeneous, dynamic and widely distributed infrastructure.

It may indeed be the case that no single system will be able to fulfill all these requirements equally well and that we have to distinguish between general purpose systems that cover the majority of areas and specialized systems that support specific working areas. Bearing in mind that S(o)OS does not intend to realize a full-fledged, fully integrated Operating System, but in particular aims at a comparative approach to identify the best architectural basis for future systems and to make justified recommendations towards future developments in all related areas, one of the major questions to address is the *degree* of (architectural and algorithmic) overlap between the respective application areas. Building on fundamental modularity concepts, it will be possible that appropriate modules may provide the needed functionality for adaptation to specialized areas. However, this will be addressed in future deliverables.

The document is organized as follows: in the first part (Chapters II-V) we address short-term developments and long-term trends from the point of view of the areas of hardware architectures, communication and networking, programming model and operating systems, which we address in this project. These chapters go beyond the state of the art analysis in so far as they summarize current experimental investigations of companies and research bodies with respect to near future developments. In other words they relate to non-established approaches of which results and actual usage are still dubious.

In the second part (Chapters VI and VII), we analyze future applications requirements, investigating both the area of general purpose computing, high performance computing, mobile and distributed system, multimedia and gaming, embedded and real-time systems. Gathering applications requirements will help the consortium to lay down the long term goals of the project.

Since the information provided in this document is based on the expertise and involvement of consortium members in the respective areas, including in particular industrial development areas potentially under non-disclosure agreements, this document is treated as confidential to exploit this expertise to the maximum extent without compromising this work.

II. Hardware and Processors

Multi-core and large scale systems are still in an experimental stage – many companies and researchers try to identify best ways to improve performance and efficiency of these systems. Current research is therefore very experimental and divergent.

1. Short Term Developments

Almost all processor manufacturers follow the so-called Tic-Toc model in their development process – this means that they intermittently employ new manufacturing processes to decrease size (Tic) to then focus on development of a new microarchitecture (Toc) on basis of this new process. As opposed to the classical attempt to increase clock rate whilst decreasing the manufacturing process, the current goal clearly rests on decreasing size to allow for integrating more cores. Not only will clock rate no longer increase, but, as we will also show below (Section II. 2. a)), it will actually constantly decrease in order to counterbalance energy consumption and heat dissipation problems: even though there are 3 GHz processors available, the frequency of up-to-date high-end multi-core processors tends towards 2 GHz.

Accordingly, the primary development trend goes clearly towards integrating more cores in a single processor. It will even be noted that a distinction between multi- and many-core processors arose recently, which reflects the lessons learned from Amdahl's law [4]: whilst multi-core processors incorporate in particular so-called “fat” cores that have large, coherent cache and are specialised in executing complex serial tasks, many-core processors generally consist of thin (or small) cores that are specialised in parallel execution, i.e. have stronger interconnects than fat cores, but less cache and a simpler instruction set. Besides improving performance in accordance with Amdahl's law (and hence depending on the type of application, see below), the many-core approach also supports the manufacturing process with respect to fault “tolerance”:

Since the probability of manufacturing-induced faults increases proportionally to the number of cores on the chip (e.g. the Intel Core *Solo* means that (at least) one of the cores is disabled – mostly because tests showed that they are faulty), many-core processors are in the long run more economically feasible: due to the high number of (small) cores – more than one hundred per processor are to be expected[1] – failure of individual cores have only minimal impact due to redundancy, if the according interconnect loss can be compensated.

The type of architecture (many- vs. multi-core) hence impacts on the system's performance regarding serial or parallel applications – however, even parallelised processes are not purely parallel but contain serial elements. According to Amdahl [redacted] depends on the relationship of serial to parallel segments and the according capabilities of the processor:

$$Speedup = \frac{1}{r_s + \frac{r_p}{n}}$$

where r_s and r_p is the ratio of the sequential and parallel portion in one program, respectively – accordingly $r_s + r_p = 1$. In other words, the speed-up of a program is limited by its sequential portion.

In Figure 1 the solid line depicts the effective speed-up of an application that is 90% (left), respectively 70% (right) parallelisable. As one can see, the curve saturates quickly at around 32-64

[1] Nvidia has already 256 cores in a CUDA processor architecture, but they have a restricted interconnect model due to the specific GPU requirements

cores – obviously this cut-off point diminishes further as the sequential portion increases. This means however, that as opposed to the constant gain through increased frequency ("frequency race"), the realistic gain through increased scale will degrade with the number of cores integrated into the processor. Note that the calculation is purely academic as it disrespects communication overhead and the fact that all parallel processes typically contain a sequential part again, so that the actual gain is even below the one indicated here.

Once saturation is reached, speed-up depends solely on the execution speed of the sequential application part (cf. dashed and dotted lines in Figure 1). Thus, in order to reach higher performance many manufacturers investigate into speeding up sequential and parallel application portions individually by employing a ***mixed (many-multi-core) architecture**.* This will integrate cores dedicated to the execution of the sequential portion of applications ("fat" cores) and to the parallel portion ("small" cores) – see also [2]. It should be noted in this context though that processing units are not "sequential" or "parallel" as such – in particular since all processors incorporate some means of parallelism (see e.g. [109]). The ideas to realise fat cores therefore include higher clock rates (maintaining an *overall* low energy profile), application specific accelerators etc. whereas small cores concentrate on simple execution pipelines with potentially lower clock rates and stronger interconnectivity etc.

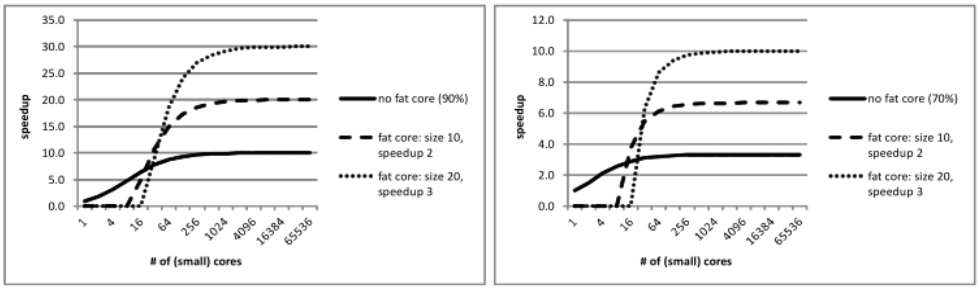

Figure 1: speedup according to Amdahl's law of a process that is 90% parallel (left) and one that is 30% parallel (right). The lines denote a configuration consisting of only small cores (solid-line), of small cores and one fat core size 10 and 2x the speed (dashed), of small cores and one fat core size 20 and 3x the speed (dotted)

Figure 1 illustrates the theoretical gain through mixing fat and small cores in a system: the effective speed-up through employing a fat core that would take up the space of 10 small cores but has 2 (dashed line), respectively 3 times (dotted line) the execution power of a small core would implicitly lead to a (saturated) execution speed of 2, respectively 3 times the one without such a fat core. Note that in the figure we start calculating the speed up with 16+ cores, as a fat core size 10 would accordingly reduce the possible maximum amount of units in the system. What can also be seen is that in the area before the sequential part of the process dominates the performance, a fat core actually decreases the performance.

Figure 2 illustrates this better by displaying the *relative* speed-up of the fat cores, where anything below 1.0 is actually a performance *loss.*

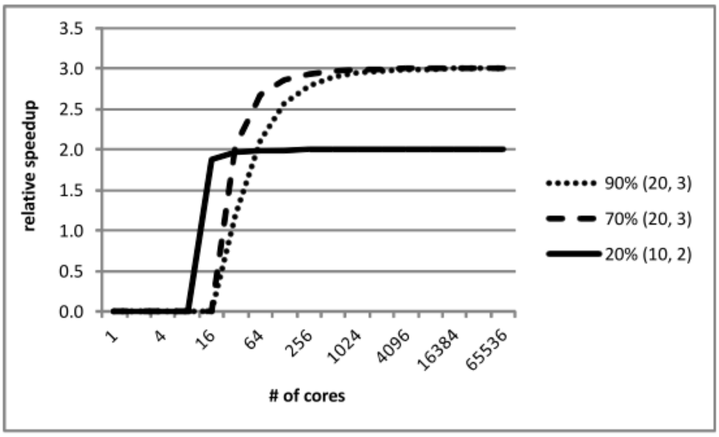

Figure 2: the relative *speedup from integrating a fat core size 20, 3 times speed (upper lines) and one size 10, 2 times speed (lower line). From left to right the application is 20%, 70% and 95% parallel*

As tempting as it may be, adding further fat cores does not further improve the performance as the primary assumption is that the sequential part can be subdivided no further.

The relationship is further complicated when it is observed that in desktop environments multiple applications with different degrees of parallelism are executed concurrently. Accordingly, multiple fat cores may take over individual applications, but their actual structure may change according to the respective application's requirements and capabilities. It is also not clear how much parallelism can be expected in future applications (cf. Chapter IV, Programming Models).

So far most manufacturers are building on a mostly fat core oriented multi-core architecture, where essentially all cores have the same computational “power”. It can furthermore be noted in this context that the number of cores in processors is not growing strongly exponentially, as was initially expected. Intel for example is building up from the Nehalem (4 cores) to Westmere (2010, 6 cores), Sandy Bridge (2011, 8 cores) and Ivy Bridge (2012, maybe 10-12 cores). At the same time, there is the more experimental 48 core system that is essentially a many-core architecture but not (yet) planned to be made available for the consumer market, though its application in HPC is to be expected soon. Similarly, AMD plans to release a 16 cores processor called Interlagos in 2011.

In order to further improve performance and to meet specific usage areas' requirements, a growing tendency to incorporate *specialised cores* can be noted: as opposed to general purpose cores, these units are dedicated to a specific tasks and have an architecture that is almost solely capable of executing according commands (such as graphics accelerators, encryptors etc.) These cores will extend the command set of the respective chip, implicitly leading to portability issues that have to be addressed either on programming, compiler or OS level. It should be noted that in a first iteration development (and integration) will focus more on micro architectures already known (vector processing units, Gfx accelerators etc.) and which have proven useful, as knowledge about general application support in scalable environments is still mostly missing (see also section II. 2.)

Whilst in particular in the embedded domain, specialised cores for e.g. communication and encryption related tasks can already be found and will appear more and more in the near future, more general-purpose areas, including in this case desktop and HPC, will remain more conservative regarding this development. As can be seen below, a long term trend towards dedicated “application accelerators” can be expected, but it first requires more information about the type of scaling applications to be expected in the future (see also Chapter VI on Applications). The more short-term development instead focuses on areas where “specialisation” in the widest sense has already shown successful and relevant – this includes in particular graphics accelerators and vector processing units – visible for example in Intel's AVX (Advanced Vector Extension) development which is expected to be part of the Sandy Bridge architecture (see above) with increasing SIMD length over future development. Similarly, full monolithic media / Gfx (graphics acceleration) integration seems to be planned from 2011 onwards.

With the advent of plenty (fat, small and mixed) cores in processors, new issues arise: current multi-core processors are “fully” connected in the sense of each core being connected to all cores (via the cache). Some of these processors even share higher-level caches, respectively employ hardware based cache coherency protocols such as MESIF over QPI. However, with the increasing scale of multi-core processors, these approaches are no longer feasible, due to lack of space and the implicit overhead for cache coherency related broadcasts (see challenges, below). On the one hand, this will lead to ***network like interconnects*** between cores, i.e. meshes and tori in order to maintain low latency and a maximum degree of connectivity – nonetheless, not all cores and caches will be reachable directly, meaning that more effort has to be vested into mapping the code onto the system structure. On the other hand, ***incoherent cache*** will be employed, cache-coherency will no longer be supported by the hardware, meaning that the software will have to cater for it.

Similar issues apply to memory: as the consumption speed of the processors increase, external memory and data sources will not be able to provide data as fast as it is consumed, so that the processor hits the so-called ***“memory wall”***. Generally, 8 cores are considered the maximum that current memory architecture can serve for average usage [19] – however, the effect of lacking memory bandwidth is already noticeable at smaller levels, so that memory intensive processes already run into problems with more than 3 cores. Whilst this problem is related to the efficiency problem in HPC, namely to reduce access to external resources, it nonetheless will lead to new memory access methods and structures. A current research approach to solving this issue consists in stacking memory directly on to the CPU (see e.g. [111][112]), thus increasing access speed and multiplying connectivity – at least Intel and Sandia already have first conceptual products of this type available. However, this approach leads to multiple issues: not only does production become more problematic and with the increased source of potential problems significantly decreases the yield, but also heat dissipation become more problematic due to the additional layers on top of the processing unit. Furthermore, this approach implies serious scalability issues, as stacked memory cannot scale infinitely even theoretically – whilst cache per core can reach up to 2 GB already, 3D stacked memory will always require additional “external” memory to compensate lack of space (see also long term trends below).

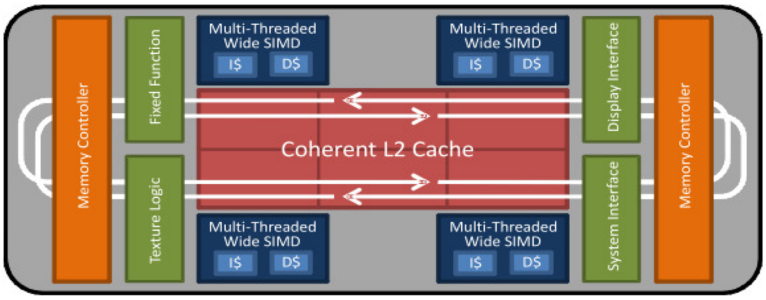

Figure 3: the Intel Larrabee architecture showing the L2 Cache ring across cores (adapted from [110]) – note that Intel does not officially refer to the architecture by name “Larrabee” anymore.

The shorter term approach to address this issue relies on network like interconnects on the cache hierarchy. For example, the architecture of Intel's Knight's Ferry builds on Larrabee's architecture and integrates a level 2 cache ring across cores. Individual cores can thus access neighbouring L2 cache with increasing latency over “distance” (see also below, section II. 2. a)). Access to main ("external") memory must thus not only be routed via local, but also via neighbouring L2 caches, thus leading to increasing delays to central data, thus requiring a higher degree of parallelism from the running processes in the first instance (cf. Figure 3). This implies that access to remote L3 cache through specialised interconnects may be faster than main RAM access for an individual core.

Summary:

- increasing scale of mostly fat cores
- specialised cores focusing on specific task types (SIMD, Gfx)
- network like interconnect between cores
- no hardware supported cache coherency
- lacking knowledge about future application structure and behaviour

2. Long Term Trends

Hardware and in particular processor development trends are primarily dictated by the market, as manufacturers have to ensure revenue for their production costs. With the change from single- to multi-core processors, it is however no longer clear what the market actually wants / needs. This is mostly due to the fact that no clear approach has been identified yet that tackles all the implications from parallel on chip processing . Accordingly, chip development is currently in a very experimental stage where manufacturers such as Intel still investigate different architectures and their benefits / drawbacks.

a) Challenges

The major challenge of hardware development, in particular on processor-level consists in the energy consumption and heat production of the chips – which was actually one of the reasons to initiate the change from single- to many-core architectures in the first instance. Power consumption grows increasingly with CPU clock speed (by the power of 2 or 3), and obviously also many-core systems are affected by this: in order to maintain power consumption at the same overall level, frequency would have to be reduced by 15% with every doubling of the amount of cores [89]. As such it is for example interesting that Intel's latest processor (Knight's Ferry) has a 2 GHz operating frequency.

Proportional to energy consumption, heat increases not only with the amount of transistors on the chip, but also with decreasing size in semiconductor technologies. In order to cope with the heat, new cooling methods are needed that again consume energy and, more importantly, space. Notably, decreasing the clock rate helps reducing this effect accordingly (cf. above).

Space is a critical aspect in semiconductor manufacturing, not only so as to merge more transistors, but in particular in order to maintain short(est) communication paths, and thus to reduce latency. As the number of cores (or tiles – see section II. 2. b)) on a single chip increases, the interconnect moves towards network like topologies (see also short term development above), the distance between cores increases accordingly, in the sense of increased latency to access remote cores. This is due to the fact that routers on the path take at least 1 cycle to react, thus increasing the amount of cycles required to connect over that path. It should be noted here that such network topologies on the chip (see also [2]) comes at the cost of space and power consumption. Shared memory design and implicitly tight coupling between caches is the general pursued programming model so far (even modern programming models such as PGAS rely on the principle of shared memory from the programming perspective, even though they act on distributed memory architectures – see Chapter IV. for more details) – this implies that cache coherency needs to be maintained across all cores. With the growing "distance" between cores, respectively between their caches, maintaining coherency leads to significant cycle loss (for example the MESIF protocol over Intel's QPI requires 3-4 hops for coherency check [90]). An even stronger performance impact, however, originates from the broadcasting like protocol involved in cache-coherency protocols: in order to reach all caches (for updates and queries) each cache has to be reached at least once – this impacts on the overall communication bandwidth with a factor of at least 2 (i.e. reducing it to less than 50%).

In summary, major challenges of future chip architectures are:

- Manufacturing cost
- Energy consumption and heat dissipation
- Communication latency & bandwidth
- Distance, number of hops
- Cache coherency

b) Development & Research Trends

Even though many-core development is still very experimental, there are nonetheless a number of trends notable which will most likely dictate the developments over the next 10-15 years.

There is a general, obvious trend towards constantly ***increasing the numbers of cores*** in future system architectures, as this allows for more manufacturing progress than trying to further improve the clock rate. It is furthermore to be expected that programming support (and future applications) will make better use of the parallel environment by exploiting threading and concurrency. As has been noted above, the number of cores is thereby limited by the memory wall related issues and by the type(s) of application(s) executed – following Amdahl's law leading to ***mixed many-multi-core processors*** (see also Section II. 1. above) take over the "sequential" (non-parallelisable) portion of an application and small cores focus in particular on parallel execution. With the architecture (and hence the relationship between parallel and sequential code segments) being as yet unknown, so is the architecture of the processor (and hence the number of fat and small cores to be expected).

Even though the future architecture is unclear as yet, it is obvious that the trend towards specialised cores will be pursued further, leading to very ***heterogeneous systems*** where specialised cores (or tiles) share the system space with general purpose ones. As already noted above most of these cores will support different computation approaches, such as with a vector architecture. Just like with the Gfx accelerators, however, it can be expected that even very application specific cores,or at least application specific instruction sets in hybrid-multicore systems, so-called ***application accelerators*** will arise which will be developed or adapted for the specific logical tasks of relevant applications (see e.g. [91][92])[2]. To this end, core requirements of these applications have to be identified first, which implies however that more information about the nature and structure of these applications is available first (cf. above), respectively that application behaviour is pushed into a specific direction from the hardware side, which requires tight collaboration between application- and hardware-developers.

Besides for very application- and usage-dedicated cores that focus just on a specific set of tasks, ***reconfigurable compute units*** will support more dynamism in task support, and thus enabling the program to dynamically select the capabilities that would benefit from according hardware support.

Through combination of different specialised units with clusters of general-purpose cores, one can effectively achieve a higher system parallelisation level, where sets of cores together form a tightly coupled compute unit on its own very much like a node in a current cluster machine. These units are referred to as "tiles" (***multi-tile architecture***) that again are connected with each other on a higher level, i.e. with potentially higher latency and in particular not using a full connection between all cores. Assuming that future applications will only make limited use of parallelism, given the natural

[2] Note that the "application accelerators" referred to here belong to the area of hardware accelerators (like Gfx, FPGA etc.). During the turn of the century, software "application accelerators" (e.g. compiler extensions) were not uncommon – similar to hardware accelerators these accelerators focused on application behaviour, yet typically were restricted to specific application implementations and not generally applicable

restrictions of the system and the algorithms, tiles can be adapted and exploited as a whole for specific applications, rather than to threads or process parts of one application. Similarly, they may process different scales (and in most case different computations) in simulations across different levels of scale. This can be exploited in particular for concurrent execution of coupled and uncoupled applications. The structure is essentially similar to a heterogeneous cluster with specialised connected units.

It will be noticed that introducing a multi-tile architecture has effective similarity to a ***hierarchical organisation of the system*** where individual units incorporate multiple subsystems – in these cases not all units are connected equally and do not fulfil the same purpose. What is more, individual units can effectively take control over or act on a higher level than other units, e.g. for scheduling and management purposes. Higher-level units can thus be specialised to the respective task, typically at the same time reducing its manufacturing costs – e.g. cache is already hierarchically organised (typically L0 to L3) where the lowest (closest) cache is the fastest (but typically smallest) and accessing higher levels implies according delays in communication.

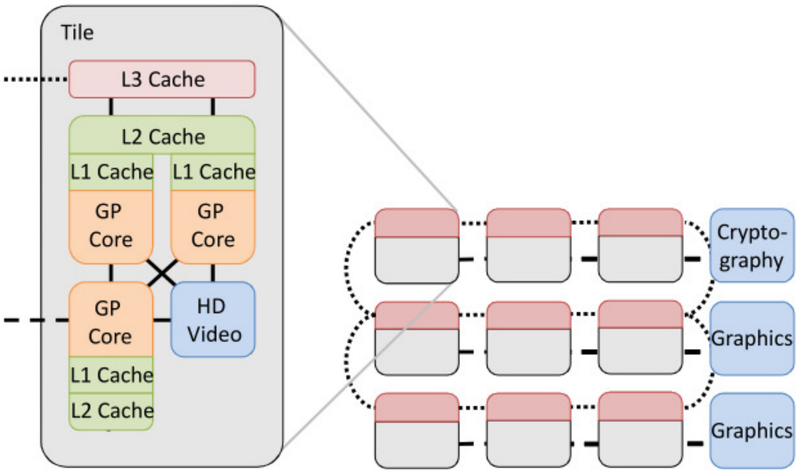

Figure 4: a potential heterogeneous mixed-core multi-tile architecture

Figure 4 depicts a potential structure of a multi-tile architecture with hierarchically organised cache and a networking interconnect on different layers, where an individual segment can take over control over the Gfx unit which is not exposed directly, i.e. always acts on a lower hierarchical level, and horizontal connected units can for example make use of HD Video processing powers. A potential use case for such a structure consists in a a set of video analysis tasks that use a small portion of physical simulation to assess e.g. the movement of an object in a video from different perspectives. Whilst this may seem very specific, the principle can be used in multiple related contexts.

As energy consumption becomes more important with the “green awareness”, but the amount of required energy increases to feed all cores, more ***dynamic control and management mechanisms*** will arise, such as dynamic clock rate and on / off switch-able cores. The Intel Atom PC already makes use of this principle to reduce power consumption, which is not simply applicable to large-scale mixed-core models, in particular when timed synchronisation is embedded in the program in some form. Due to this strong dependency with the actual program requirements, energy related control will have to be implemented on both soft- and hardware level.

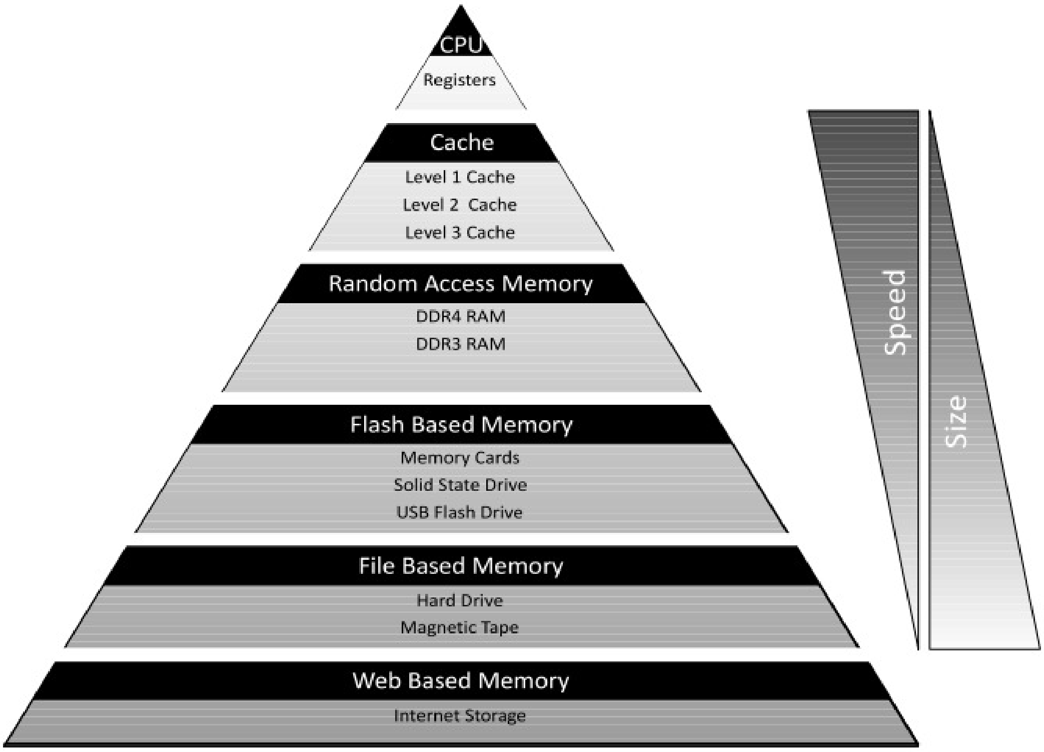

Figure 5: memory hierarchy

Similar development and structural approaches will be noted on the cache and memory level, too: not only will ***cache size increase***, but due to the increase of cost and the hierarchical structure of the computation units, ***cache hierarchy*** and even ***memory hierarchy*** will increase too (cf. Figure 5). This will imply new structures of memory architectures, such as 3d stacked memory on the CPU with new means for heat dissipation (cf. short-term developments).

Summary:

- increasing scale and heterogeneity, including specialised accelerators
- hierarchical organisation of the system
- architectural structure of the systems as yet unknown (lack of knowledge)

III. Communications

1. Short Term Developments

a) Network

Networking makes part of our daily life. Following this same trend of increasing importance of the communication flows, actual networks are expected to become richer and more powerful [42], while providing ubiquitous connectivity over heterogeneous networks [43]. We currently observe the trend where all devices are becoming network enabled, and wireless technologies become dominant in that aspect. All laptops are now Wifi enabled, most cell phones support multiple communication interfaces, and even more established equipments such as TVs and gaming consoles are being sold network interfaces, and some other devices are connected to large online communities [44]. With these devices, the network is now reaching us everywhere, and if we consider the example of IPTV and VoIP, we see that even traditional services are being migrated to the “new” network paradigms. Also in line with this convergence, large amounts of the wireless spectrums is being reused to digital communications and digital television [45][46]. Because in the future all devices are expected to be connected by some interconnect, this inter-host interconnection can be exploited in order to make devices cooperate and share more resources, providing an aggregated service, in line with the SOA paradigm. Network connectivity is becoming so natural to our life, and to our systems, that even execution environments are becoming network aware. The first form are Cloud infrastructures which can provide massive elastic execution environment for users and businesses to deploy their services [47]. Then, by incorporating networking concepts into the inner functional aspects of kernels, by the creation of forks which aim at distributed operating systems under a single system image [48][49][50].

In order to cope with different requirements, isolate complexity, and maximise component reuse, communication systems are organized in layers. This well tested paradigm allows for greater flexibility, as well as an increase in bandwidth. At the top of the network stack (see Figure 6) we have the Internet as a communication medium for Internet applications, and near the bottom we have physical interconnects such as Infiniband and Ethernet. As complexity increases, and systems become more distributed, concepts from packet based networks are being generalised to most interconnects. In this aspects, networks are coming closer to processors and cores, and we now find NoC providing interconnectivity between cores in multicore processors.

Transport and Wide area networks, while relevant in the area of communication networks, may have little impact in a distributed operating systems, as the[redacted] transfer of data. The utmost impact of these networks will be in making Internet system more tightly coupled.

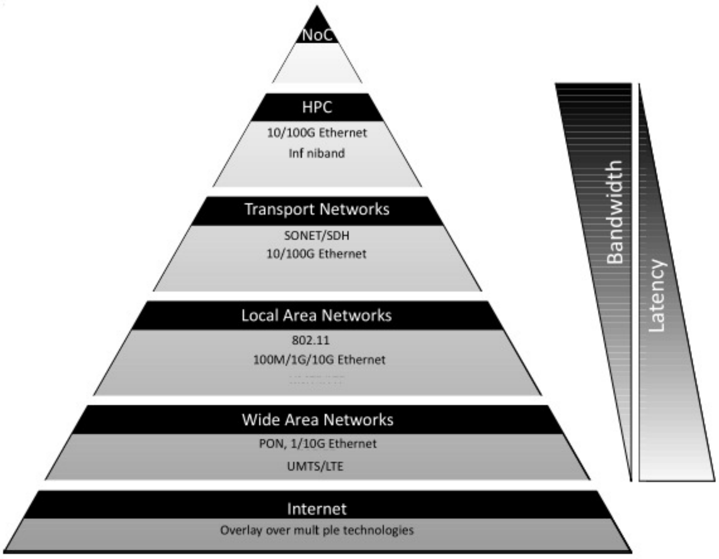

Figure 6: Hierarchy of common network types by latency and bandwidth

Internet

The Internet, while a combination of multiple technologies, provides a communication medium between applications sparsely located across the globe. Its transmission characteristics are highly dependent on the underlying technologies and present high variability in the time domain. Especially because end-to-end communications require crossing multiple administrative domains, as well as different technologies.

While we will see bandwidth increasing on the internet during the next years (see Figure 7), as well as all other networks, the same will not be true for latency, which is bounded by physical and processing limitations.

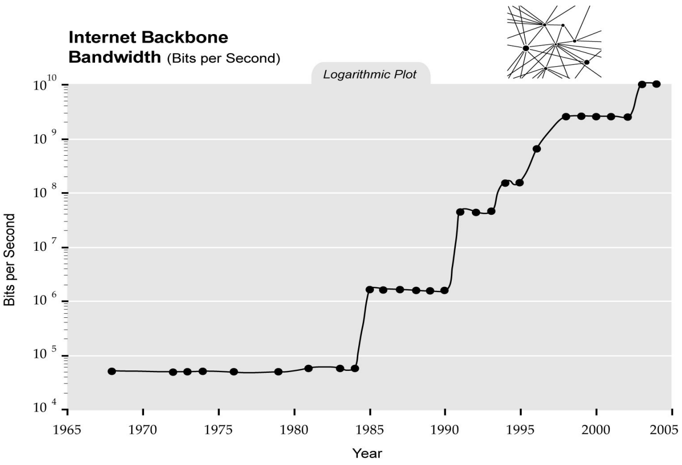

Figure 7: Evolution of the Internet Backbone Bandwidth over the last 40 years [41]

The most important limitations are the speed of light, and the processing capabilities of the routing devices. Speed of light imposes hard latency boundaries on pulse propagation across systems, which are much aggravated by the fact that light slowed down on fibre (about 2/3 of vacuum speed), imposing a 5ms delay per 1000km, which restricts optical transport networks. Other aspect is that while bandwidth is increasing, electronic equipments have a hard time dealing with the throughput required, which affects all types of communication networks. Processing latency will be increasingly higher in relation to the transmit latency, as it did in the past, which will surely pose a bottleneck if development on parallel systems does not accelerate. Even today it is not difficult to find that end-to-end delay between hosts exceeds 500ms. Unless reliability, bandwidth and latency improves, a distributed operating system over the Internet seems unfeasible in the following years. However, because internet is an important driver for innovation of systems, services, and underlying communication technologies, its development will have an impact in future operating systems.

Local Area Networks

Local area networking, as been since a long time dominated by Ethernet, and more recently by 802.11. Capacity of these networks has being increasing steadily, as both the underlying technologies also improved. Interconnects with 100Mbits are now well established and most equipments already support 1G Ethernet. It is expected for these networks to keep improving capabilities, as well as moving from wire based solutions to wireless solutions. The 802.11 family of protocols now reaches 300Mbits, supporting rich media contents as well as multimedia contents with QoS guarantees. In the short term, more equipments will start to be equipped with 10G Ethernet interfaces, and high speed LANs will be commonly available. This trend will make Local Area Networks more and more suited as a scenario for distributed operation.

HPC and Tightly Coupled Systems

Due to its nature, High Performance Computing requires fast interconnects in order to reduce execution delays (and hence performance reductions) through high latency. Typically, latency is the more critical factor than bandwidth and, as opposed to loosely coupled systems, bandwidth can rarely compensate low latency in these cases (see also sections IV. 1. b) , II. 2. b) and [2]). The most widely used technologies in this context are InfiniBand and Gigabit Ethernet, as can be seen from the Top 500 list[3]:

Gigabit Ethernet is the general term for all technologies that allow the transmission of a Gigabit of data per second. Current Gigabit Ethernet technologies can reach up to 100 Gigabit per second (100GbE). We can distinguish between (1) optical fibres (1000BASE-X), (2) twisted pair cables (1000BASE-T), and (3) balanced copper cables (1000BASE-CX) as physical medium for transportation. The choice impacts on the maximum length of the connection, as well as on its latency, with minimum latency being around 70 µs [113] and one can roughly estimate 1 µs added latency per 100m cable. Single mode optical fibre can thereby be extended to up to 70km – however, in HPC environments, the distance is typically much shorter (generally up to 10 meters). The main strength of Gigabit Ethernet consists on its ubiquity, it basing on IEEE standards, thus making it also comparatively cheap, yet slow to adapt to necessary changes. For example, the development of the 40GbE/100GbE took almost 4 years.

As opposed to this, the community behind **InfiniBand** (originating from Compaq, IBM, HP, Intel, Microsoft and Sun) is much more agile and thus allows quicker adaptation to new requirements and environments. The fastest Infiniband configuration can reach up to 300 Gigabit per second and has a latency of roughly 1-3 µs. InfiniBand even supports RDMA capabilities, in which case the latency is even less than 1 µs. These specifications make InfiniBand the preferable technology for tightly coupled parallel processes that have to communicate across nodes and potentially across connected units – however, the lack of a clear standard and according programming API makes InfiniBand slightly more complex to use. As opposed to Ethernet's hierachical switched network topology,

[3] http://www.top500.org/stats/list/35/conn

InfiniBand employs a switched fabric topology where nodes area typically connected in a Fat-Tree (CLOS), mesh or 3D-Torus topology via crossbar switches.

Networks-on-Chip

When dealing with multi-core (and more specifically with many-core) systems-on-a-chip, the interconnect fabric is increasingly viewed as the key limiter for effective system integration, and thus is becoming one of the major design challenges. Network-on-Chip (NoC) is an emerging paradigm for communications within large scale integrated systems: In a NoC system, modules such as processor cores, memories and specialized IP blocks exchange data using a network as a "public transportation" sub-system for the information traffic. An NoC is constructed from multiple point-to-point data links interconnected by switches (a.k.a. routers), such that messages can be relayed from any source module to any destination module over several links, by making routing decisions at the switches.

In Deliverable 5.1, Section II.3 [2], we did a recognition of current interconnect topologies for on-chip communication, most of them can be classified as Network-on-Chip. The communication protocols mimic similar techniques used in LANs, apart from some specific issues that need to be addressed. In particular, routers/switches are usually simpler than their LANs equivalent, as they only provide small buffers. Therefore, techniques derived from circuit switching (rather than from packet switching) networks are sometimes employed, like *Virtual Circuits/Channels*; in other cases, specific techniques to minimize buffering are employed, like *Wormhole switching*.

Network-on-Chip are still in an intensive research stage, as testified by a number of EU research projects. For example, the NaNoC[4] project addresses issues related to composability of IPs in a NoC architecture by providing a design platform constituting a toolkit for the construction of NoC-based multi-core systems in nanoscale technologies. The MOSART project[5] aims to design a multi-core architecture with distributed memory organisation, a Network-on-Chip (NoC) communication backbone and configurable processing cores that are scaled, optimised and customised together to achieve diverse energy, performance, cost and size requirements of different classes of applications. The GnoC[6] project explores various possible set of traffic patterns in different NoC architectures. Then a Gaussian-based NoC model is introduced to provide statistical guarantees on delays and throughputs.

It is still an open problem how an Operating System should deal with NoC architectures. An attempt to provide a standard API has been made by the Multicore Association[7]. The MCAPI is based on a classical message passing paradigm, but has been tailored for the specific issues of multi-core technology, addressing in particular the need for fast communication between heterogeneous cores on a chip and QoS support. However, such standard is not yet widely employed in products.

Quality of Service

Networks don't always transport data transparently. Packets may be reordered, delayed, and even filtered based on dynamic or previously defined rules. Particularly important to this project is the shaping applied to packet or flows, as a mean of differentiating end-to-end (E2E) Quality of Service (QoS). An operating system operating over a communication infrastructure, should take in consideration existing differentiation mechanisms, so that process execution priority is respected, or at least, system consistency is maintained (vital communications of the kernel will need to have priority over all other communications). Research in this topic is increasing as more distributed systems are developed.

[4] http://www.nanoc-project.eu
[5] http://www.mosart-project.eu
[6] http://erc.europa.eu/index.cfm?fuseaction=page.display&topicID=290
[7] http://www.multicore-association.org/home.php

Quality of Service mechanisms are not present in all technologies, nor at all layers of the communication stack, which makes the increased use of QoS as a short term trend upwards (in opposition to the long history of over-provision, wich is now being abandoned). Every communication above IP can be differentiated from the others using two well established mechanisms: Integrated Services(IntServ) and Differentiated Services (DiffServ). Current trends evolved the standard DiffServ model so that traffic can be intelligently identified [52] and then shaped accordingly. Short term developments will increase the accuracy of traffic identification methods. At the same time, we are observing the application of QoS to most technologies and its wide spread use in the control data plane [53].

A relevant amount of work has already being developed in differentiating communications by creating physical or virtual paths, and this has indeed became a standard procedure for high capacity networks supporting the GMPLS [54] family of protocols. Because GMPLS was initially designed for optical transport networks (where it proved to be much successful), the concept of Multi Protocol Label Switching is now generalized to other communication technologies such as Infiniband [55].

Infiniband is designed to have basic QoS mechanisms, built around the Flow Control mechanisms, which are also found in Ethernet (yet in a simpler form). Infiniband supports the concept of virtual lane (channel) which can be arbitrarily deployed, while Ethernet supports only on demand flow control (it is argued that VLANs can be used as Virtual Lanes). The use of Virtual Lanes in fact provides multiple channels between hosts, all with different buffering, and flow control resources. Also, lanes can be built in an hierarchical manner, thus allowing highly customised differentiation scenarios. Current trends propose datacenter consolidation using Infiniband, where virtualization techniques, allied to the congestion control and QoS capabilities of Infiniband are vital pieces [51], in order to provide isolation (and SLA conformance) between virtual network instances.

In Networks-on-Chip, there is also the need for QoS. Messages between cores are multiplexed into links and resources are shared. The result is high bandwidth at the cost of unpredictable delay, which may pose serious penalties if high priority messages are delayed. As well as with other technologies, research in NoC architectures tried to put hard boundaries in delay and throughput. Short term trends will focus in improving QoS support for NoCs so that the impact of distributed execution can be alleviated. One expected development is the support for explicit packet ordering, which would benefit the implementation of cache coherent systems [56]. Other is the creation of more QoS aware NoC architectures with different impact [57], thus allowing the NoC architecture to be deployed in low power devices, or with highly limited area.

Indirection Mechanisms

An important aspect being pursued is the creation of indirection mechanisms for communication between hosts and applications. This new layer sits bellow applications or just after IP and decouples location from endpoint identifiers Currently, users communicate directly, either personally or by sending information to others email addresses. Users are able to receive information independently of their location, because of the indirection service provided by the email system. In common networks that is not possible, and the research community already developed many solutions capable of making location transparent to actual communication. The result of this effort is in the form of the Mobile IP protocol [58] (and its many variants such as [59]), the i3 infrastructure [60] or the SIP [61] and HIP [62] protocols. All these solutions, which are all under active refinement, specify that either transport nodes or dedicated nodes can track the location of a given host, and redirect traffic to it in real time. Tunnels and additional overhead is implied, while providing almost free mobility over heterogeneous technologies (or even equipments). In the nearest term SIP will be the first protocol to actually be used due to the deployment of IMS platforms and the upcoming 3GPP LTE standard. Later, with the wide spread use of IPv6, Mobile IP, or one of its variants, is a serious candidate for the future indirection platform. A multi-core distributed system must tackles this aspect in two ways: first by supporting the indirection solutions being developed, and second by incorporating indirection mechanisms in its operation.

B) Wireless Transactions

Now that networking became part of our life, the next step is to improve ubiquity. Wireless technologies allow users to access a multitude of services with much freedom. Network providers believed that wireless technologies would provide a convergence of services between fixed and mobile (Fixed Mobile Convergence [63]). However, it was recently observed that wireless technologies are in fact replacing fixed technologies, and the actual trend is of Fixed Mobile Substitution [64]. Most technologies now are being replaced by their wireless counterpart such as ADSL [67] for LTE [68], WiMAX [69] and UMTS [70], Ethernet [71] for 802.11abg [72], or CableTV [73] for DVB-T/S/H [45] or IP based services. Simply put: for many services, and use cases, wireless technologies provide a service level similar to fixed technologies, with improved deployment flexibility and easy of use. This fact show the importance of wireless technologies, both in the near and long term. Surely communications are going wireless.

Wireless technologies can be divided in three areas of networking: Personal, Local Area, Wide Area. For the sake of brevity, we will ignore transport networks (typically using Ultra Wide Band), as well as satellite networks. The first comprises communications limited to a few meters around and individual. These networks adopt concepts where communication is centralized, or with great focus to a single, powerful device, connected to many, less power devices (Master/Slave). Bandwidth is low to moderate, with Bluetooth 3.0 HS [65] providing up to 24Mbits/s. Still, typical devices are currently much constrained, and the specification was only recently ratified, so most most devices are unable to sustain such transfer rate. In the short term we expect all devices to support this bandwidth as Bluetooth 3.0 HS becomes widespread. Also of interest are the protocols arising from 802.15.4 [66] which provide low range communication to sensors, home alarms, and other low power devices. 802.15.4 is one of the protocols leading the creation of the term “Internet of Things” as, due to its low complexity, allows very simple devices to be available to the Internet (frequently by means of a proxy). Since simple devices cannot carry on complex computation, in some case they can *off-load* part of their computation to other, , more powerful, devices. It should be noted, however, that the reduced bandwidth does not allow complex coordination schemes among remote devices, as it is needed, for intance, in distributed operating systems (see Chapter V.).

For more than two decades, Local Area Network was identified with Ethernet. Now a Local Area Network most certainly uses 802.11 at some point in its infrastructure. The exponential growth observed for this family of standards proved the value of mobility. 802.11 is now commonly available at many enterprises, university campus and homes, providing an attachment point to users. The standard evolved from its original bandwidth of 2Mbits/s to 300Mbits/s now provided by 802.11n [76]. At the core of such improvement are greater efficiency of the wireless medium and incorporation of concepts such as Multiple Input Multiple Output (MIMO) . 802.11n was ratified in October 2009, with pre-n devices available since over 2 years. In the near future we will observe the adoption of concepts such as Beam Forming [75], Sect[redacted] frequencies (3.650Ghz, 60Ghz) [77], with the respective increase in bandwidth, interoperability and reliability. From the point of view of an Distributed Operating System, the great bandwidth provided, as well as the active development around 802.11, makes it possible and very desirable to extend the distributed execution environment over devices communicating by means of the 802.11 family of protocols.

Providing connectivity over larger areas is not typically suited for 802.11, where range is limited to a few tens or hundreds of meters. In these scenarios, 3G as been the technology of choice to provide broadband connectivity to roaming users. This technology is the most responsible for the migration of users from DSL to wireless as it provides a reasonably compared service, but over a wide area.

Current deployments using the HSPA+ standard provide up to 56Mbits/s download bandwidth. Still, as with all cellular technologies, latency is an issue as it averages to a few hundreds of milliseconds [78], which may pose some constrains to distributed execution. In the near future, the 3GPP will release further improvements over the existing technologies, in the form of the Long-Term-Evolution standard [79]. This new standard will evolve cellular networks by providing more efficiency spectrum usage, as well as higher flexibility and support of packet based communications. Peak bandwidth is currently at more than 300Mbits/s and latency is much improved over existing UMTS networks, reaching values as low as 5ms [80]. Such high bandwidth and low latency (close to the latency of 802.11 and Ethernet) presents good expectations towards distributed operation over cellular networks.

Also relevant to the near future is WiMAX (802.16) [69]. In comparison with LTE it currently provides lower bandwidth, however there are many expectations on the upcoming 802.16m standard, which aims at 1Gbp/s over a wireless link. Still adoption of WiMAX has been rather slow and LTE, which is a natural evolution to existing network providers, may undermine its future use.

2. Long-Term Developments

On the long term, we will see an even greater increase of bandwidth as the efficiency reaches the maximum values allowed by the theoretical information limits. Surely, the world will be driven by networking technologies, pervasive communication will become a natural component of everyday life according to the *Internet of Things* paradigm. In particular, wireless technologies will dominate by connecting most devices to other devices connected to the high speed wired back-haul. Also, direct communication between devices, in a Peer-to-Peer fashion, may be an important technology. Not in the traditional sense of today P2P overlay networks over the Internet, but directly between devices in the neighbourhood. One example is the development around the UWB and 60Ghz frequency band to replace all wired interconnects over short distance, while providing over 1Gb/s.

Optical interconnects will play an important role in the long future. The potential for optical technology is great, and we are very far from its physical limits. Nowadays the standard choice for long distance fixed communications, it will also be used as an interconnect between cores in future systems. The impact will be lower total dissipated power and reduced latency. If optical memories prove to be possible and commercially viable, optical CPUs will be created, much boosting computation power.

For future operating systems, it seems natural to assume that interconnectivity between devices will become less than a bottleneck, allowing for more distributed operation models. Computation power will increase continuously due to the advances in optical technology. More importantly, most devices will have high capacity wireless links with other devices in the surrounding environment, making possible the creation of distributed, on-demand processing environments. Communication trends will evolve in the direction of indirect addressing, where data is exchanged between entities independently of their location. Other important aspect is that the volume of data exchanged between devices will also increase, continuing to put pressure the available bandwidth.

Summarising, we see the following challenges and opportunities:

- Increasing connectivity and increasing bandwidth
- Physical limits to the minimum achievable latency
- Wide diffusion of wireless communication
- Increased location independent distributed computation
- Use of networking techniques in system-level software
- Internet of things
- Optical interconnects also on chip

IV. Programming Models

Programming models for parallel environments develop slowly. Simplicity and efficiency are crucial for any changes in this domain

Programmability of distributed and in particular large scale environments is crucial for the success of any system. Unfortunately, there are enough high-tech machines out there that would provide powerful capabilities if developers would only be able to make efficient use of them – e.g. IBM's Cell processor is comparatively difficult to program given current programming models and compilers (for example the software must be designed to request a transfer between main memory and local memory [93]), and also with other new architectures the necessary hardware support (OS image, libraries etc.) is lacking in the beginning.

In the near (and longer term) future, scalability and heterogeneity of systems will increase rapidly (see also Chapter II on Hardware and Processors), so that multiple challenges arise on the programming level: not only should it be easy to program distributed, scalable environments, it should also be efficient and match the specific capabilities / restrictions of the system so as to offer maximum efficiency. With the increasing heterogeneity this means that the model needs to be highly portable and that the underlying compiler systems can adapt quickly.

In this section we will investigate some of the expected development in programming languages and models as indicated by current trends, as well as the expected long term changes along that line.

1. Short-Term Developments

As noted in D5.1 "State of the Art" [2], we need to distinguish between the aspects and developments in programming models on the one hand and the ones in the underlying compiler technologies on the other: whilst programming models evolve comparatively slow and try to maintain a maximum degree of downward compatibility, compilers have to evolve quickly to adjust to the changes in the hardware and thus maintain efficiency of the language on top. Still, programming languages evolve faster than compilers due to the simple fact that they do not have to accommodate for system changes so much.

a) *Compiler Development*

Most compiler suites consist of (at least) a front-end, dedicated to a specific source language (such as C extended with OpenMP) that is converted (in any number of steps) into an ideally platform-independent intermediary code, which is then compiled by the back-end into the actual hardware-specific optimised code. Accordingly, most adaptations hav[redacted] back-end in order to meet the specific hardware-capabilities and most language changes affect the front-end.

With the increasing diversity and heterogeneity in system architectures, the complexity of the back-end will increase accordingly. Since a wide range of quickly changing architectures are to be expected (cf. Chapter II on Hardware and Processors), this means that not so much time can be vested into fine-tuning the compiler any more, as was typically the case in particular for High Performance Computing (see D5.1 on HPC [2]). What is more, processor developers such as Intel plan to extend the instruction set of the processor to exploit the capabilities of the specialised cores, meaning that not only platform-independence will be more difficult to achieve, but also that the compilers will have to become even more processor-specific than before (at least in the wide consumer market).

At least in the **common end-user sector** compiler developers therefore will go a similar way than modern programming languages (cf. D5.1 [2] and below), namely to make use of parallelisable patterns to ensure scalability to the increasing amount of cores and to essentially leave all platform specific adaptations up to the compiler's decision. Whilst this simplifies development, at the same time it reduces efficiency and programmer control, as the developer will have no information available on why and which transformations have occurred.

A slightly different trend consists in extended "just-in-time" (JiT) compilation where the code is only pre-compiled into a platform-independent form that the JiT compiler will adapt to the respective environment at execution time. In principle, JiT compilers take up a lot of the performance, as they effectively are nothing else but virtual machines or code interpreters (cf. D5.1 [2]). In the area of **High Performance Computing**, however, where efficiency is more important than simplicity, it is unlikely to expect compilers to take over *all* code-adaptation and optimisation, even though the principle trend is clearly visible. Current compilers have a tendency to look for transformable patterns in the code, but do not respect any higher-order information about the code behaviour, so that e.g. nested loops and double-linked lists can cause optimization problems and performance loss [101]. Next generation compilers (and implicitly programming languages, cf. below) will therefore have to respect additional information provided by the developer about the code behaviour, its restrictions and limitations – StarSS (see programming models below) is a step in this direction. This means, however, that even in the HPC sector, compilers have to become more intelligent to actually interpret the code-behaviour information – in the long run, this will make code more hardware-independent, too (cf. long term trends below).

Summary

- in common mass market, maintain simplicity of usage for developer
- in HPC, higher-order code behaviour needs to be respected
- more intelligent compilers that actually perform the scale and hardware-adaptation
- multi-pass and JiT compilers to increase platform-independence

B) PROGRAMMING MODELS

Development in programming models is basically following the same tendencies as the ones of the compilers (cf. above) – we thereby need to distinguish in particular between common (end-user) programming models and high performance computing related ones, even though there is a wide range of end-user programming languages that partially overlap with HPC.

In the **common mass market sector,** programming languages in particular aim for *simplicity* of usage, that means that they generally want to provide a powerful high-level language that incorporates most of its functionalities in simple operations according to the typical requirements in the according usage domain. Scripting languages for example provide a very explicit set of commands that fulfil the specific purposes of controlling the respective application it belongs too. It should be noted that functional programming models found little uptake - non-regarding their benefits for parallelised developed (cf. [2]). Nonetheless, more and more modern programming models take over concepts from functional programming, such as the Lambda operator for improved function passing etc. to increase and even simplify the development tasks. The .NET framework was specifically devised to provide functionalities that encapsulate most of the low-level tasks that a developer typically has to perform, to e.g. open a communication channel across the web etc.

Next to increasing simplicity, this also makes the language more platform-independent and leaves more tasks to the compiler which can take hardware-specific optimisation decisions. Effectively, programming language development in all areas tends to add abstraction layers on top of existing languages, thus hiding complex algorithms in more simple operations and leaving it up to the compiler to transform the according pattern.

With the increasing necessity to achieve performance by exploiting the scale (and potentially the heterogeneity, cf. below) of future processors, programming models need to come up with solutions that allow for parallelising and scaling the code. Since automatic parallelisation is NP hard, some information needs to be conveyed by the developer: a particular starting point for serialisation thereby consists in particular in the so-called “unrollment” of loops, to which end some information about the dependency of data between loop iterations needs to be known, though. The current trend thereby goes towards extending the language with a set of additional commands that clearly indicate their parallelisability (such as “parallel_for”), leaving it up to the compiler to do the most efficient spread-out and manage the parallel threads. This is thereby generally restricted to a specific set of loops (such as database queries) and without additional information the scale-out of the compiler does not respect that overhead for communication may produce more delay than a sequential execution. The set of extensions and the type of information that needs to be provided with it will have to be elaborated further.

More responsibility will hence automatically be left to the developer, who is already encouraged to explicit develop threads in the form of lambda operators. Thread management by the runtime environment (compiler) will accordingly increase in a comparable fashion. It is therefore to be expected that in a first iteration, in particular the concepts from distributed (as opposed to parallel) computing will be elaborated in the direction of concurrent execution, such as e.g. decoupling of the application interface from the code, which is essentially already possible, but not respected in code distribution etc. As discussed in more detail below (on HPC programming languages), provisioning of meta-information for scale-out will remain difficult within the near future, so that semi-automatic scale out and hence parallelisation will focus on specific patterns, such as parallelisable loops.

At the same time, extended features of specialised cores will become visible in the programming language, but uptake will be restricted to specific use cases (such as mobile devices) in order to maintain a maximum degree of portability. It is to be expected that in the long run (cf. long term trend below), the knowledge acquired from next generation HPC programming models will influence the next developments in common programming languages by extrapolating patterns for dealing with heterogeneity and increasing scale.

Programming languages in the area of **High Performance Computing** develop more slowly than in the common end-user domain, as has already been noted. Rather than complete new programming models, it is rather to be expected that current programming models that are basically still under development will find more uptake, as their stability and performance in new environments increases. To these languages we can count in particular:

- the *Partitioned Global Address Space (PGAS)* model which exposes a virtually shared memory to the developer and takes care of messaging during synchronisation and memory locks itself, thus relieving the developer from having to think about the specific memory architecture (cf. D5.1 [2])
- As the long-term development will go into identification of more annotations (cf. below), intermediary steps will investigate into means to define dependencies between code segments and functions. To this exploration already count models such as the StarSS extension by Barcelona Super Computing [94].
- At the same time, with a clear trend for using GPGPUs as part of HPC applications, in particular for embarrassingly parallel tasks, more and more language models (and in particular compilers) will extend to integrate these types of platforms, thus elaborating heterogeneity of the languages more closely. NVidia is particularly interested in opening GPGPU platforms for HPC as can be seen by their development of “graphic card clusters” in recent years. Along that line, NVidia has published and does further develop the *CUDA*

toolkit[8], which provides the means to develop vector-like calculations to GPGPUs, typically using the C programming language. A similar line is visible by Apple's *OpenCL (Open Computing Language)*[9] which provides a similar environment to PGAS, but is devised to deal with heterogeneous platforms, and in particular with GPGPUs in these environments. CUDA already integrates OpenCL.

- As already noted in D5.1 [2], supercomputer clusters already incorporate multi-core processors that are (currently) effectively shared memory machines (UMA), whereas the nodes follow the NUMA concept, i.e. do not share memory, and there is generally no cache-coherency protocol in place. Accordingly the communication and synchronisation principle differs in clusters according to the hierarchical level, but there is no according programming model to reflect both NUMA *and* UMA architecture. An approach growing in popularity consists in a *hybrid programming model*, where OpenMP is employed on the processor level and MPI for any communication across processors (or nodes, depending on the system architecture). Obviously this implies more careful planning about code distribution. It is however not expected that this model as such will pertain long (i.e. become a long term trend), as cache-coherency in multi-core processors will probably disappear (see Chapter II), so that some form of messaging / synchronisation will have to be specified at all levels, though potentially in slightly varying forms.

It must be noted again here that even though most of these approaches already exist in the form of experimental and under development programming languages, we nonetheless regard them as short-term development trends, as uptake will only slowly increase and thus lead to better efficiency and stability of these models.

Summary:

- Simplicity is a primary concern over efficiency to deal with scale and heterogeneity
- Principle approach by adding further layers (annotations) on top of existing models
- Examine means to convey additional (meta) code information
- Tendency towards improving the compiler, rather than the language

2. Long Term Trends

Talking about programming models, long term trends are very similar to short-term development, simply due to the fact that the issues to be addressed within the near future will not be solved quickly enough. Similarly, the development of programming languages so far did not so much result in completely different concepts, i.e. declarative and imperative models did (and will) prevail. Even aspect-orientation did not so much introduce a new concept, but took the next step along the development of compiler directives.

In particular in the more **common end-user domain** programming models always aimed at ***simplicity of usage***, even at the cost of efficiency. This trend will continue but will have to cope with the additional constraints and requirements derived from the heterogeneity and scale of future architectures. This will particularly lead to ***more intelligent compilers*** that can identify a larger scope of parallelisable patterns and match those to the characteristics of the infrastructure. In the long run, this will also lead to changes in the language constructs that allow implicit parallelisation, such as “for_each” as opposed to “for (int x=...”, since the prior does not proscribe an order of execution, therefore stating the independence of the algorithm within the loop etc.

The common programming models will thereby lend concepts and lessons learned from the high performance computing domain with respect to patterns, dependency declaration etc. - however,

[8] http://developer.nvidia.com/object/gpucomputing.html

[9] http://www.khronos.org/opencl/

with a primary goal to automate and integrate these features into the compiler, as much as possible. As HPC and Desktop PC converge architecturally at least in principle, an important aspect for future programming models (and the according support through hardware, OS and compiler) consists in increased portability that retains a maximum degree of efficiency, which may require hardware specific annotations though (see below). There are already first research projects being initiated (see e.g. the call of the German Bundesministerium für Bildung und Forschung[10]) that aim in particular at increased portability between desktop PCs and supercomputer clusters to leverage the parallelism of future many-multi-core to test and pre-process also complex simulations before they are deployed in a large-scale environment. This means particularly that programming languages have to provide common features that can be equally exploited in different environments and hence that the compiler can deal with. As opposed to the common "mass-market" development area the goal consists not so much in *automating* parallelisation and adaptation, but to relieve the constraints on the developer, so that he / she can concentrate on the code instead. This can take different forms (which most likely will converge in the long run):

- Annotation of functions and / or code segments in a loop can provide information about the relationship and hence dependency of operations. This cannot only be used for scale-out (degree of information exchange), but also concurrency spread-out, i.e. which segments can be executed in parallel, even if they do not belong to the same loop. StarSS is already providing the basis for such behaviour [94].
- At the moment, the developer still has to respect the word length in vector processors for efficient loop unrolling – however, with the compiler reading that information at execution time and the developer providing dependency information on top of even nested loops, unrolling can be done more efficiently by the compiler.
- The developer also knows which data is most likely to be used frequently, so that he / she can give indications with respect to whether they should remain in cache if possible or not
- Hide the memory management details from the user regarding shared- or distributed memory, so that he / she can assign preferences to data distribution that the compiler will actually map to the hardware and converts synchronisation points into the necessary actions (PGAS model)
- etc.

In effect, we will notice the following main development steps:

Foremost, memory management of the code has to (and will) improve so as to allow for the differences in memory architecture in future processor and HPC architectures – this does not only involve the PGAS like "virtual shared memory" structure, but also annotations that provide information about the dependency and usage scope of data.

Due to the increasing heterogeneity and scale of the computing environments (desktop PCs as HPC clusters) future models will include higher levels of abstraction that do not only affect the memory architecture, but also specifics regarding communication, I/O etc. A great deal of system adaptation must be taken over by the compiler and / or the operating system, to which end new constructs need to be examined that give the compiler more options to optimise the code behaviour. At the same time, in particular develops of highly efficient code (in the HPC domain), will want to exercise enough control over the compiler so as to exploit specific hardware characteristics – ideally in an aspect-oriented like fashion that increases the portability of the source code and performs hardware specific decisions (branches) either at compile or run-time.

[10] http://www.bmbf.de/foerderungen/14191.php

It is as yet unknown to what degree programming models, compilers and / or hardware are affected by these developments, i.e. how much can (and should) be realised by e.g. the compiler, whether some kind of adaptations are needed on hardware level etc.

Summary:

- Compilers will become more intelligent to match hardware specifics with annotated code
- More code annotations to specify the overall behaviour of the code
- Typical development patterns will move from the actual language into compiler and annotation model
- Platform-independence will have to increase
- Full scope of impact from parallelism unknown as yet.

V. Operating Systems

The Operating System is the component that most of all is experiencing the pressure of change due to the paradigm shift from single-core to multi-core and many-core technology. On one hand, the OS has to provide appropriate hardware abstractions so that applications do not depend too much on the underlying hardware platform; on the other hand, it must manage hardware resources in an efficient way, to unleash the power of future parallel hardware architectures.

For this reasons, in the next future we will see Operating Systems go though a deep revision of their internal architecture and of the way they provide services to applications.

1. Short-Term developments

Monolithic Operating Systems (see [2]), which is now the architectural model of most commercial OS, are going through a profound transformation due to the need to adapt themselves to the upraise of multi-core technology.

Currently, monolithic OS are implemented by running identical copies of the same kernel image on every core, all of them share the same global address space and the same data structures. To prevent data inconsistency in kernel data structures, monolithic kernels make a large use of spin-locks, reader-writer synchronizations, read-copy update (RCUs), etc. Unfortunately, performance of kernel routines strongly depends on the lock implementation, which in turns may depend on the underlying hardware architecture. When moving to multi-core systems, most of the important shared data structures (for example the scheduling queues and process descriptors) become a bottleneck, due to the high bus contention and the overhead of cache coherency protocols. Therefore, the need to change the internal data structures and locking strategies from one architecture to the other. Such modification take a long time and huge efforts to kernel programmers, and cannot be considered definitive as we move toward different hardware architectures.

As stated in [1]:

> *Even so, operating systems are forced to adopt increasingly complex optimizations [...] in order to make efficient use of modern hardware. The recent scalability improvements to Windows7 to remove the dispatcher lock touched 6000 lines of code in 58 files and have been described as "heroic" [...]. The Linux readcopy update implementation required numerous iterations due to feature interaction [...]. Backporting receive-side-scaling support to Windows Server 2003 caused serious problems with multiple other network subsystems including firewalls, connection-sharing and even Exchange servers.* [redacted]

Most of the problems cited above are due to the lack of modularity of existing monolithic kernels, and the lack of clear, stable and well-defined interfaces between the different services. The lack of modularity is a consequence of the race of OS vendors to optimize the performance for a specific service by introducing short cuts to direct hardware access. However, this is also the reason for the commercial success of monolithic kernels. With the advent of multi-core technologies, the deep intertwining of the different kernel services and the complexity of the cross interaction between kernel functions has been exacerbated by concurrency problems and race conditions introduced by parallelism, making the design of next version of the kernel an overly complex problem.

It is clear that the situation has come to a point in which a complete rethinking of the kernel structure is needed. In facts, several researchers and practitioners reports that the scalability of current monolithic kernels is limited to 8-16 cores [1]. A different approach is urged as hardware moves toward an increasing number of cores.

We expect the current monolithic kernel structure will not survive the next advancements in hardware design. As discussed in D5.1 [2], there are different proposal to change the structure of the kernel: the multi-kernel approach [1], in which a different instance of the kernel runs on each core, each one with its own private data, and instances communicate by means of remote procedure calls; GenerOS and FOS [29][30] in a number of cores are reserved for the OS; Corey [31] in which the application has the responsibility to specify access modes and the level of contention of the resources.

It is important to notice the analogy of this proposal with the micro-kernel architecture (see [2]).

Micro-kernels are a promising technology for massively parallel architectures, because of their minimal footprint, their modularity and their strong focus on efficient message passing mechanisms. As multi-core and many-core systems start to resemble distributed systems on a chip, and as shared memory systems become less convenient due to the complexities of cache coherency protocols, we believe that the micro-kernel modularity will be a better tool to tackle complexity and address scalability.

However, recently researchers have focused on enhancing the security aspects of microkernels [23] [24], and for using the microkernel as a hypervisor [25][26]. As a matter of fact, security and fault tolerance are two distinctive features of microkernels that cannot be easily found in other systems. However, we believe that microkernels can play an important role in future many-core architectures. Unfortunately, only few researchers are investigating the use of classical microkernels for multicore systems [27][28].

Lutz et al. [81] recently proposed a Service-Oriented Operating System (S(o)OS) model constituting an enabling technology for future distributed collaboration scenarios called Future Workspaces. In this work, it is suggested that the Operating System should possess a distributed and heterogeneous nature. A "Main" OS instance is the one offering the most complex services to applications, comprising process management, virtual memory management, I/O, networking, and a graphical user interface, whilst other OS instances exhibit a limited set of functionality focused/specialized on the capabilities locally available on the nodes they are running on. Specifically, it is envisioned that an "Embedded Micro OS" instance should be used to directly control remote resources and devices, while a "Standalone Micro OS" instance should be used to expose access to virtualised resources made available through virtualisation technology by some further Operating System. This structure will enable an unforeseen enhanced level of experience for mobility, where the actual resources (computational power, storage, data) will be maintained remotely through dedicated corporate server farms, thus greatly reducing administration efforts.

One advantage of such a structure is that the Operating System may easily embed the additional features needed by GRID applications [82], which usually are made available today by means of specialized extensions to the OS. S(o)OS introduces a novel resource provisioning concept [83] that differs from existing approaches of Grids and Clouds, in that it aims for making applications and computers more independent of the underlying hardware and increase mobility and performance.

One of the ideas that is stressed in the S(o)OS concept, is that the programs do not necessarily need to be written in a parallel nor distributed way, like it happens in the MPI approach. Rather, it should be possible in principle to take a sequential application and automatically identify those portions of the program code that may be run remotely, then migrate them on a more powerful Embedded OS instance for a faster execution. The additional latency due to the distribution of the functionality would be largely compensated by the increased performance of the code when running remotely. For example, this would be easily the case for a laptop designed for mobility running the main OS instance, whose code is partially remotely executed on a high-performance remote machine. In order to achieve this, sophisticated monitoring mechanisms will need to be built into the OS, such as monitoring the memory access patterns, building statistics on the frequency of access to the various pages, tracking dependencies and interactions among various code segments.

Also, in a cloud environment, a vast amount of computational resources will be at reach of each process across the web or locally. Therefore, the S(o)OS concept calls for deep investigations into dynamic and intelligent processes (re)distribution policies according to resource availability and demand, and it proposes [84] a micro-kernel OS architecture model designed to compensate these deficits. All these elements are under investigation in the context of the S(o)oS project.

Summarising,

- OS are subject to pressure
- Current monolithic structure to be changed, some microkernel concept will be used
- Need to use smaller kernels for efficient distribution of code

2. Long-Term developments

The OS has to *mediate* the hardware aspects with the programming model: keywords will be modularity, scalability, distribution, dynamic configuration and reconfiguration. In order to efficiently support future parallel hardware architectures and programming models, the following areas and services will be revolutionized.

Synchronization

Parallel and distributed software need to be synchronized and coordinated in a more or less strict way, depending on the application coupling level. However, the underlying hardware platform could be highly parallel, and or distributed; the underlying interaction model could be based on shared memory in on-chip parallel architectures; or on message passing on distributed systems. The operating system will need to dynamically and adaptively support all possible models of interactions. Also, the OS must support different application scenarios, from remote execution on a cloud, to local execution of the same service with a reduced QoS. We envision a hierarchical approach, where an application is recursively split in subsystems with different coupling levels at each layer of the hierarchy; at the lowest level, there will be highly coupled parallel software that needs to execute on many-core systems on chip, probably with some shared memory; at higher layers, subsystems will interact less frequently and communicate via message passing. Therefore, the OS will need to support a mixture of shared memory (whenever possible) and remote procedure call, and automatically and transparently switch between the two models depending on the application requirements and the underlying platform support.

We also envision a tighter interaction between OS and hardware features: many of the low level mechanism that are today realized in software will be supported in hardware in future hardware architectures. One example is transactional memory [103], which has been proposed as a way to simplify the writing of lock-free data structures. Another example is support for message passing features in hardware: rather than simply sending an interprocessor interrupts to realize communication between cores, it may be possible to rely on NoC features to safely and efficiently implement message passing and remote procedure calls, with support for priority based message enqueueing and Quality of Service.

Concurrency

The dynamic and distributed nature of future hardware platforms (many-cores, but also clouds) will require an high degree of adaptation to the application and to the OS. In particular, the OS must allocate resources and distribute the load across a set of nodes that is potentially unknown (or scarcely known) during the design of the application and its implementation, and may even be scarcely known at execution time. In order to efficiently perform load balancing and optimization of computational resources (with a energy saving impact), it wil be necessary to efficiently migrate code across nodes, let them be cores in a multi-core architecture, or complex subsystems of a distributed environment. This will have an impact on processors ISAs and on compiler technology, but also on the Operating Systems. In fact, the OS will need to “split” and “merge” the application code depending on the specific underlying platform that is actually available. As a consequence, the concepts of *process* and *thread* as we know them may need to change in the future. One possibility is to use lightweight active objects with a reduced state/context, that may be used as simple functions in a sequential program (with a minimal overhead) or as threads in a parallel many-core environment, or as processes in a distributed environments. The OS will be in charge of split and merging those objects into threads or processes as needed.

In the same way, the kernel code needs to be designed and built in a component-based way, and the kernel will dynamically change its code and profile depending on the needs of the application. While such a trend is already present in modern OS (e.g. Linux kernel modules), in the future this approach needs to be taken to an extreme level.

Distributed Control

Future OS will need to cooperate in a completely distributed manner. It is simply impossible to think of a central Operating System that manages all local and remote resources and takes all decision in a optimal way. To enhance scalability, instead, every processing unit will have its own copy of the OS. By processing unit here we mean several different things, depending on the granularity at which we consider the system: it could be a multi-core chip, or a tile in a many-core architecture, or an entire subsystem of a distributed cluster. These OS will interact with each other in a distributed manner to carry on their job. Not all OS will be the same, instead each one could provide a different set of services. This means that an application that needs one specific service will ask the local OS to “discover” other OS nearby that may provide the service. Service-oriented Architecture (SoA) concepts and techniques will be essential to provide such mechanisms. Also, this will enable an unprecedent level of fault-tolerance, as one service may be provided by different peers, and the local requesting OS may request the service to more than one node in a replicated manner and then choose the most appropriate results.

Scheduling

One important problem that needs to be addressed is how to properly *schedule* the different activities in the system so that all timing constraints are respected. In order to keep performance under control, especially for real-time applications, it is important to precisely control how computation is distributed across nodes and when it is executed; and how and when messages are enqueued in the network, be it a on-chip network interconnect, or on a classical inter-node network.

Some researchers state that the number of cores in future processor architectures will be so large that there will be no need for multitasking on cores, thus making processor scheduling not relevant. For example, Wentzlaff and Agarwal argue [86] that, in the near future, “the reliance on temporal multiplexing of resources will be removed”.

It is our opinion that such a scenario will not actually happen, or at least not in all systems, and not so drastically, for a variety of reasons. First, the various cores, caches and memory elements will need to communicate and exchange messages, but they will hardly be fully and completely connected, thus various parts of the physical links and interconnects will be time-shared among multiple activities.

Second, while the tremendous take-off of parallelism at the hardware level could not be followed by software at the same rate, we expect in the future to see much more parallel applications. This means that applications will try to exploit to the maximum extent the underlying parallelism available in the hardware, creating a number of small parallel tasks that is unforeseen in the current programming culture. This will imply an increase in orders of magnitude in the number of parallel threads created by each single application.

Third, in order to achieve good resources saturation levels, there will be anyway the need for temporal scheduling of multiple tasks on the same core. In fact, while cores communicate, synchronize, exchange data, among each other or with external I/O devices, a context-switch to another ready task will promptly allow for a good exploitation of the core computational power.

Also, there will be anyway the need for inter-mixing, on the same system and core, application workloads of heterogeneous types and with different timeliness requirements, e.g., batch and interactive applications. A batch activity like a software compilation will likely try to occupy all the available cores, in order to exploit available computing power to the full extent. Yet, in response to a user action, some task will need to preempt the currently executing task on some core, in order to execute something else. Waiting for some core to finish its current task before activating the interactive task may simply lead to unacceptable response latencies for the application. On the other hand, a static partitioning of resources (e.g., cores) among applications may be detrimental in that some cores would likely remain greatly under-utilized.

Furthermore, considerations on efficient power management may shed a different light on the problem of scheduling and allocation of tasks on cores. For example, it may be important to compact all tasks on fewer processors and turn off the other cores, thus reducing energy consumption. Similarly, in order to keep under control heating and temperature in some parts of the chip, cores can be dynamically switched on and off according to some *scheduling policy*, and tasks be migrated between cores so to make power dissipation easier.

Hence, in our opinion, *temporal scheduling* will still be an important topic in future many-core systems.

Summarising:

- Synchronisation will remain a crucial aspect of kernel development, need to make it adaptive
- Some of the kernel low level operations implemented in hardware
- Efficient migration of code
- distributed control, no centralized kernel
- temporal scheduling and load balancing still an important research aspect

VI. Application Types and Areas

Development is primarily driven by *usage*. As areas converge over time, future systems will have to address a wide scope of application types.

Each company defines their next generation of products according to the needs of the respective usage domain – e.g. over recent years, vector processing has had a huge demand in high performance computing. This demand has led to many manufacturers providing vector based computing systems, such as the NEC SX 8 in 2003 [95]. At that time, the end-user market was interested in vector systems only in the sense of graphics cards, contributing to the success stories of companies such as NVidia and ATI. IBM has taken a more courageous (albeit co-developed with manufacturers for Gaming Consoles) approach back then by manufacturing the CELL processor which shows an attempt of serving both domains' needs [96][97].

In other words, the individual companies are concentrated on the applications of the respective user domains. With the advent of GPGPU to a broader audience, this focus and distribution across domains has slightly changed though and graphics cards manufacturers now offer vector based cluster systems for intermediate scoped high performance computing etc. But even in these cases, companies such as NVidia clearly aim at vector computation use cases and not at say audio processing. Intel on the other hand tries to serve different markets equally well and therefore investigates into general purpose processors which can execute computational intensive, vector-related, stream-centric etc. applications.

The corresponding application domains pose completely different demands on the system (the most obvious ones being e.g. vector versus scalar systems) and this does not only circumscribe HPC and desktop users, but also the full range from embedded systems, via mobile devices to distributed data centres. By examining these application areas, it is possible to identify joint (respectively divergent) requirements towards the underlying resources. On this basis, the degree of convergence through future systems can be analysed, and the usability across (and within) domains of specific solutions be assessed.

In the following, we give an assessment of some important application domains and their general requirements for parallel computation.

1. High Performance Computing

HPC applications have a high demand for computational performance. Latency is a critical factor

High Performance Computing arose from the need to perfor[redacted] and with higher accuracy than current computers could / can achieve. HPC systems do not necessarily consist of parallel cluster machines, but can in principle be realised using high-end processors, even though multi-processor architectures proved to be the most successful approach given the hardware limitations. Most modern clusters employ multiple specialised, high-end processors high-speed connections such as Infiniband (IB) in order to combine the benefits of fast processors with parallel execution.

Multi-core processors are obviously being employed in high performance computing, thus adding another hierarchical layer to distributed machines where nodes do not only contain multiple processors, but each processor also contains multiple computing units, so that we can identify the following layers in modern day distributed settings:

1. Inter-processor (core-2-core using e.g. QPI)
2. Inter-node (processor-2-processor using e.g. e.g. IB, IXS)
3. Inter-node (node-2-node using typically infiniband)
4. Beyond clusters (e.g. user-2-cluster e.g. via Ethernet)

Standard applications obviously do not benefit from such a distributed setup as they typically cannot use more than a single compute unit (i.e. core). Applications exploiting cluster machines need to explicitly be programmed to execute parts of the algorithm concurrently across multiple compute units (classically processors, now cores). Not all algorithms can actually be treated in this fashion and some do actually show worse behaviour in parallel environments – as we will show below, there are also different parallelisation strategies that will not only lead to different efficiency depending on the algorithm type, but also to different infrastructure requirements.

What is more, though, the process(es) not only need to be parallelised, but also adapted to the specific hardware: a great deal of performance of cluster machines is achieved through specialisation to computational task *types* and implicitly to specific algorithms. For example, vector processors are specialised to repeating the same mathematical / logical operation on multiple data sets successively (SIMD).

A) Specific Application Types

Though the scope of applications that can efficiently run on high performance machines is vast, we can actually make a few generic statements about the types of algorithms and their generic usage areas that benefit most from HPC environments. With the high relationship between HPC and multi-core systems, this distinction is of particular interest for exploiting the parallelism of multi-core systems. In accordance with Amdahl's law, the performance gain through parallel systems depends on the parallel versus serial portions of the application to run [4]. However, one must thereby respect the degree of connectivity between the parallel threads, as this defines the synchronisation and messaging overhead, which, in the worst case, can lead to performance lower than serial execution – these are generally examples for algorithms that can not be parallelised in the first instance.

General Classification

Degree of connectivity plays an important role in selection of the infrastructure, as this defines connection speed and type, ranging from shared memory (maximum connectivity) to p2p computing (completely loosely coupled). We can distinguish applications by the degree of scalability according to the connectivity between data / threads:

Unrelated Instances

The most extreme case of scaling applications consists in multiple instances of the same application / process being executed in parallel – neither result nor execution are related to each other and hence no synchronisation or message exchange between the instances needs to take place. Often enough the initial state and parameter set is identical for all instances so that no data coherency needs to be maintained either.

Such applications show no real scaling behaviour, i.e. the execution performance is not affected by the amount of compute units employed – neither in the sense of strong or weak scaling (cf. below). In fact, “scaling” here means “replication” rather than distribution. These types of applications can typically be found on server farms or data centres where the degree of replication defines the availability of the according data / service, and are the ideal candidates for cloud environments.

Implicit requirements:

- Extended status information about the hardware
- Remote deployment and execution (potentially "live")
- Dynamic scaling up and down as needed

Embarrassingly Parallel

Similar to unrelated instances, embarrassingly parallel applications can actually be executed in parallel without requiring synchronisation or high messaging between instances. As opposed to unrelated computation, the problem space and hence data sets and results do relate to each other, in other words, each instance either computes an (independent) subset of the full data, or uses different parameters on the same data set. The execution time per instance is independent of the amount of processors employed, even though calculation of the full problem space proportionally decreases with the number of instances. Examples of such algorithms are projections of 3d models into 2d space, as the projection space can be segmented into sections that each can be calculated independently and all p2p computing applications (such as Folding@HOME[11]) which effectively execute completely independently and results are collected and integrated at random.

Along the same line, we can thereby distinguish between *weak* and *strong scaling* behaviour: in *weak scaling*, the amount of required processors grows proportionally to the problem space (e.g. size of the data set) whilst execution time remains constant, or in other words the problem size per processor remains effectively constant. This typically applies to p2p computing like applications, where the calculation is effectively serial and the problem (either data or algorithm) cannot be segmented further. On the other hand, in *strong scaling,* the problem size is fixed and by increasing the number of compute units, execution time can be reduced – with the amount of compute units changes also the workload per processor. The 3d projection example shows strong scaling behaviour, as is clearly demonstrated by modern day graphics cards that improve performance time through employing multiple cores on small data sub sets.

Strong scaling applications typically show higher data dependencies, as the results need to be synchronised more often to improve throughput. Nonetheless, the communication and synchronisation need in embarrassingly parallel use cases is still much lower than the one in tightly coupled applications (see below). In either case, scalability is limited by the overall problem space: as soon as it cannot be segmented or distributed further, adding more instances does not make sense any more.

Implicit requirements:

- Extended status information about the hardware
- Remote deployment and execution (potentially "live")
- P2P like routing
- Fast & broad bandwidth
- Data & process structuring for workload balancing

Tightly Coupled

Most typical (and most demanding) HPC applications show strong coupling between the instances on the individual compute units – this is simply due to the fact that most algorithms simply are not embarrassingly parallel and results need to be constantly exchanged between calculations. For example, when simulating particles in plasma, even weak interactions between remote areas have to be considered for accuracy. Scaling can principally be achieved either by segmenting the problem space (the data set), or by distributing parts of the logic that is typically executed multiple times. In the first case, too small segments will lead to too high data dependencies along the boundaries and too large segments will hardly improve performance. Similar issues apply to distributing the logic,

[11] http://folding.stanford.edu/

where e.g. small loops lead to high data dependencies whilst too large loops will remain comparatively slow if not segmented further.

A huge amount of time is hence wasted on synchronisation and data exchange. As opposed to embarrassingly parallel applications, scalability of tightly coupled applications is more limited by the messaging overhead than by the data set (even though this plays a role, too). Due to the high information exchange, a primary requirement consists in fast and broad interconnects. Messaging and implicitly resource access are major obstacles to be considered during coding – ideally all relevant data and functionalities are fitted into the local cache, so as to minimise overhead. One criteria during data preparation hence consists in adjusting the segment size to available memory (cache) and structuring it so as to reduce dependencies between segments.

Typically, tightly coupled applications tend to fall into the category of *strong scaling*, as the general goal behind distributing such algorithms consist in speed up rather than higher data throughput, even though the long-term goal may consist in higher accuracy and hence larger data calculation.

Implicit requirements:

- standard communication and messaging
- typically some form of shared memory / cache-coherency
- large & fast cache
- extended resource status information
- code / process deployment (typically not "live")
- fast & broad interconnect
- synchronisation (of communication and execution)
- concurrency management
- maximise cache usage, minimise communication and memory swaps

Computation Patterns: The "13 Dwarfs"

Most applications on High Performance Computing systems fall into either of the preceding categories and pose the according requirements towards the system – as Colella notes, High Performance Computing applications typically adhere to a base set of algorithmic patterns [98]. Asanovic et al. elaborate on these base patterns to identify a total of 13 so-called "dwarfs", i.e. application kernels that are at the mathematical and functional heart of a specific set of applications [3]. Asanovic et al. try to deduce the general hardware requirements for these dwarfs and implicitly for the set of applications behind it – similar to the goal of this document's section. Instead of repeating the full paper, we will only provide the main highlights of the paper and recommend the interested reader to check out the full document at Berkeley[12]. It should be noted in this context though that while Asanovic et al. try to identify the base kernels that make up the core of a wide set of applications, we try to go the other way round, i.e. try to identify the core hardware requirements from a set of applications. Whilst the Berkeley approach is particularly good for benchmarking a system environment with respect to its capabilities regarding a specific application suite (cf. D5.1 on "benchmarking" [2]), the top-down approach seems more useful to predict further development and to generate a more holistic view on the requirements in general, as has been noted in D5.1 too.

[12] http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.pdf

	Embedded	General	Database	Gaming	Mach. Learning	CAD	Health	Image	Speech	Music	Internet	HPC
Finite State Machines	+	+	+	~	~						+	
Combinatorial	+		~		~						+	
Graph Traversal	+	~	~	~	+	+	+		+	~	~	
Structured Grids	+	+		~				+				+
Dense Linear Algebra	+	+	~	+	+	+		+	+	+		+
Sparse Linear Algebra	~	~	+	+	+	+	+			+		+
Spectral Methods (FFT)	~			~	~	~	+			+		+
Dynamic Programming	~		+		+	+			~		+	
Particle Methods		~		~								+
Branch & Bound			~		+	+				~		
Graphical Models			~		+					+		
Unstructured Grids				~	~	~	+	+		+		+
Map reduce		~	+		+		+					+

Table 1: the 13 dwarfs (vertical) and their relevance for a set of application areas (horizontal). Red ("+") denotes highly relevant; green ("~") slightly and an empty block denotes no relevance

As the application domain of HPC is currently well defined, we will use the findings from the Berkeley paper as a main reference in this area. Table 1 lists the 13 dwarfs and their relationship / relevance to a given set of usage domains – for a detailed description of the individual dwarfs, please refer to Berkeley's "Dwarf Mine"[13].

It can be noted from Table 1 that in particular grids, linear algebras, spectral methods, n-body calculations and map reduce form the main applications of the high performance computing domain – this implicitly involves applications such as particle flow, plasma calculations, molecular tests etc. This is not to say that the other application kernels are not parallelisable, but that they are (currently) atypical for parallel computation environments (i.e. HPC).

For reason of completeness, we list in the following the specific hardware requirements per dwarf though highlighting in this context again that the HPC related dwarfs are of major importance in this context:

1. Classic vector and matrix operations belong to the class of ***Dense Linear Algebra* –** they are naturally ideal for vector machines. The performance depends very much on the size of the matrix in relation to the word length of the vector unit, where a mismatch can lead to unnecessary memory swaps and hence communication overhead. Data hence needs to be carefully distributed in order to achieve load balancing and thus optimal performance. Cache hierarchies can lead to data feeding issues and hence to delays if the number of calculations exceeds the size of cache (i.e. if data has to be loaded from the RAM).

   Dense Linear Algebra calculations typically scale very well.

[13] http://view.eecs.berkeley.edu/wiki/Dwarfs

2. If the vectors / matrices contain a large number of zeros, we speak of ***Sparse Linear Algebra*** – in these cases, any zero calculation can be typically ignored, for which reason the matrix is typically compressed. This is e.g. achieved by maintaining the index of the non-zero values. This requires additional progressing steps for identifying and annotating the non-zero entries. Due to the unpredictable dependencies between data (according to the distribution of non-zero data), data distribution and hence efficient scalability is quite difficult. Generally, the same issues as for Dense Linear Algebra apply, with the addition of more complex workload and increased messaging overhead (due to non-optimal distribution).

3. ***Spectral Methods*** (operations in the spectral domain) generally consist of computation of complex numbers (i.e. 2 inputs and 2 outputs). As the same operations are executed on a large data set, the algorithm is typically highly vectorisable. Each core / node typically only needs to access local data reducing communication to transposing steps.

4. In ***N-Body Methods***, the calculations relate to a number of n bodies which *all* interact with each other in some form, so that in the simplest programming approach, complexity reaches O(N^2) and thus typically touches the memory wall problem (see D5.1 [2]). Divide & conquer approach accumulating the influence of remote particles increases performance – however, for parallelisation, load balancing is the major problem, in particular since the movement of bodies imply constant redistribution. Shared concurrent memory access (link to all bodies) causes communication overhead. Notably there is specialised hardware for astrophysics[14] and molecular dynamics[15] calculation.

5. Most simulations can be segmented into ***Structured Grids*** where each cell contains a subset of the environment of the same size. This is optimal for regular "stationary" data distribution (as opposed to moving particles), as e.g. in the case of heat distribution over an area. As basically the same operations are performed again and again, the code is highly vectorizable. For distribution, each core is typically assigned an individual cell. Since neighbouring cells influence each other at the boundaries, some data needs to be shared between cores – this does not necessitate shared memory though, as typically only a fraction of the data is needed and shared accordingly (so-called "halos"). In most cases this data needs to be exchanged after every calculation iteration across the whole grid.

   In some cases, only specific areas are of main interest, in which case locally refined grids are applied – these require a more adaptive and better aligned load and data distribution (see also Unstructured Grids).

6. Most simulation and modelling problems actually act on irregular shapes (such as the human body or a car) which cannot be segmented into regular cells (see Structured Grids), but are instead segmented into a set of cells of different size and shape, i.e. an ***Unstructured Grid***. They basically behave like structured grids and have similar requirements, with the difference that the size of the individual cells have to be adjusted for best load distribution (in the worst case dynamically).

7. Some calculations effectively consist of a series of mostly independent function execution, where all results are aggregated and processed in order to get the actual result of interest. To these count Monte-Carlo algorithms, or more generally ***MapReduce***. The function is therefore effectively embarrassingly parallel, with communication only being performed at the end of each individual function – which does not necessarily have to occur synchronised, since no further processing is blocked by the delay (besides for the central gathering). Load balancing, to reduce this delay, may potentially be difficult, as the

[14] http://grape.astron.s.u-tokyo.ac.jp/grape/

[15] http://www.research.ibm.com/grape/

functions may not always have the same execution time (e.g. with a precision breakout criteria).

8. Searches through graphs and trees, as e.g. in data mining base on the principle of ***Graph Traversal***, i.e. traversing through a number of (linked) objects and executing (the same) process on each object. If the function is essentially small, the main task consists in (unpredictable) traversal across memory. If the graph / tree is balanced, than the traversal can essentially be parallelised with good load balancing by segmenting the graph in sub-graphs. Each according traversal is effectively embarrassingly parallel but segmentation may be difficult due to the nature of the graph. In most cases the main overhead is caused by random memory access latency. Memory requirement per core is proportional to its search space.

9. Some problems can be solved by solving simpler subproblems and then comparing results, as is typically the case in optimisation tasks – the according technique is called **Dynamic Programming.** Whilst this increases the problem space, it decreases the computational overhead – also, in an ideal case, the sub-problems can be distributed across individual cores in an almost embarrassingly parallel fashion. More typically however, any degree of data needs to be exchanged between the sub-problem calculations in which case the performance increase is directly dependent on the relationship between communication overhead and computation (i.e. high degree of communication and only little computation leads to low efficiency).

10. Problems such as route finding are similar to graph traversal and dynamic programming, where the function traverses the search space according to some optimisation criteria and accordingly performs a ***Backtrack or Branch & Bound*** algorithm. Also problems such as Linear Programming and Boolean Satisfiability etc. fall into this category. The problem generally poses the same requirements as graph traversal, i.e. the graph can be segmented and traversed in parallel, with the same problem of load balancing. Since most problems in this area use some form of weighing for estimating the optimum, the according invariants ideally need to be communicated between all processes – however, this does not necessarily need to occur after every step, weighing the uncertainty against the communication overhead.

11. Knowledge maps, Bayesian networks, neural networks etc. fall under the category of ***Graphical Models***, where nodes represent variables and edges conditional probabilities. Typically a graphical model is traversed multiple times, e.g. for the purpose of sound analysis or for applying a neural network to a data set. A simple approach towards distribution consists in either segmenting the data space or the graph into subgraphs per core. If the data space can be segmented ideally (i.e. without temporal dependency), the problem becomes embarrassingly parallel – in all other cases, data needs to be transferred after each iteration step, whereas typically the size of data is small compared to the computation to be executed, making the problem scale very well.

12. Most programs can be represented in the form of ***Finite State Machine*** consisting of tuples of in- and output values, and a state S. Finite state machines (FSM) can accordingly be represented at graphs where nodes represent (active or passive) states and edges input, respectively output values. Whilst the principle of executing a FSM is similar to graph traversal, an FSM is signified by having an explicit state traverse the graph – even though the state may branch out at designated transitions (and can hence be parallelised), the graph can not be segmented and computed in parallel as in the graph traversal case. A FSM

therefore represents the sequential and parallel aspects of typical applications which means that there is no generic parallelisation approach to it[16].

13. Simple boolean operations on large data sets, such as computing checksums can typically be represented in **Combinational Logic.** Most operations are on a bit-level, making them difficult to develop due to lacking appropriate coding support. The algorithm can typically be broken down into data pipelines that can be executed in parallel – individual cores can take over steps in this pipeline and forward results to the next core(s). Quite generally, the overhead for data transportation has to be weighed against the computation effort to achieve a good workload. The memory wall (cf. Section II. 1. is thereby a critical factor (as for most stream processing).

Quite generally it can be stated that most HPC applications are defined by the relationship between computation and communication, where all communication (including data exchange as much as memory swaps) is essentially a performance loss that should be avoided. Accordingly, cache tends to be as large as possible, with hierarchical structures as a means to address manufacturing issues. However, with high data throughput in a process, memory access latency becomes the major efficiency throttle where performance is directly proportional to the slowest memory connection, as it has to forward all data to the fastest cache level. In such cases, direct access to the data source is preferable to any cache size and hierarchy, as these add fixed access delays that cannot be avoided.

2. Personal Embedded Devices

Smart phones and other personal embedded devices include an ever increasing amount of features, becoming more general-purpose, and require a constant connection to internet

Personal Embedded Devices are one of the faster growing markets today. The necessity of having a personal computation and communication device is a growing factor in modern day's lifestyle: future users will handle more and more of their private *and* business life on the move as devices get more and more powerful. Nowadays it is expected of most users to be online almost all the time. Related to that and as a consequence of the growing web2.0 uptake, a noticeable shift from personal to internet based social interactions has occurred and such electronic networks are an essential part of many peoples' social life. It is to be expected that this tendency will grow even further, as modern environments even support the elderly and disabled, and as the distance between friends and family grows – amongst other due to the increased mobility of users.

a) Historical evolution of personal communication

Increased mobility has been one of the trends driving progress for the last decades. In the start of the Internet revolution (mid 80s), the availability of communication platforms such as the BBS, and later email and the World Wide Web, triggered a rapid increase in the amount of information being exchanged. The internet made possible to exchange information at a speed which abolished frontiers and distance. It was now possible to place different parts of the same corporation in the most suitable place, while maintaining information flows. One restriction at that time was that connectivity was tied to copper (and later fibre) links. In order to support the increasing nomadic way of life, cell phones were introduced to the mass market. These devices, which required no physical wires, allowed individuals to move and still communicate with others. They existed since the 50s but its use was not really widespread due to coverage and cost issues. The creation of Global System for Mobile Communications (GSM), in 1991, and its dissemination into most of the world, allowed to migrate the traditional analog communication networks to more flexible scenarios where

[16] Note that it can be segmented though, having state traverse across different nodes or cores

information was already digital. GSM, already provided call continuity for users on the move by means of rapid handover between base station cells. Due to the simple technology used, handovers were quickly executed and mobile device operating systems were relatively simple.

Cell phones kept driving mobility but soon it was clear that ubiquitous connectivity was the next step. This required porting the internet concepts to the small handheld devices, which became extremely powerful and complex in relation to their ancestors. In particular, evolution made it clear that the traditional technology, still incorporating many concepts from circuit switched networks, needed to be upgraded to a full packet driven network. With IP protocol support on our cell phones, now commonly called Smart-phones or PDA, we are able to move almost freely in any developed country while maintaining our email, chat and even voice sessions active. The trend is to put more power into these little devices and make them a portal to the representation of our digital life, which will exist somewhere in the Internet cloud. 3GPP [68] is since a long time creating standards such as Internet Multimedia Subsystem and Long Term Evolution, which make available more and more bandwidth following the principle of more rich devices supporting more complex services, which are integrated into the network operator. Future trends point to intelligent devices which, being location aware (by means of GPS), and having access to our personal preferences and friends recommendations, can make automatic suggestions, help in some decision processes or provide contextualized services.

With the increase in capacity, the number of devices increased very dramatically [5] followed by the number of users. It is now common to see users with more than 1 device: one for personal use, one for professional use. Some devices already support multiple environments [8] where users are able to switch environments. According to [7] in 2008, the number of mobile subscribers reached 4100 million, which represents more than half of world population. In comparison, this number is much than the number of Internet users (1542 million), a technology which also radically changed our world.

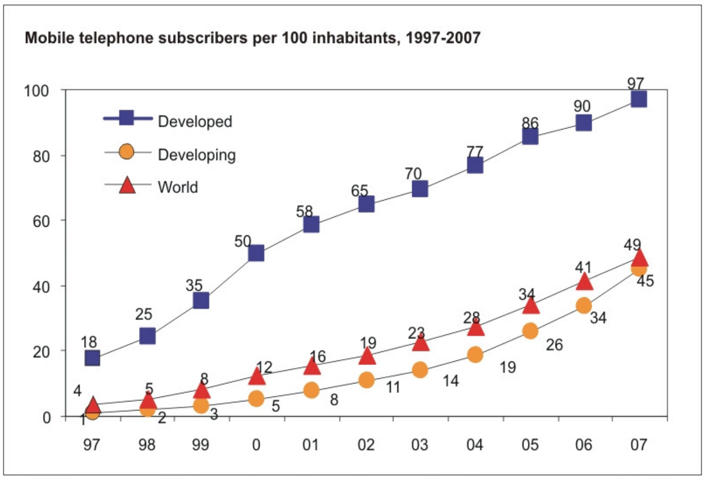

Figure 8: Evolution of number of cell phone users according to ITU ICT Eye

While the project does not directly address cell phones, it addresses embedded devices such as PDAs. These devices, while popular, have never been massively adopted mainly due to the lack of resources, and in particular due to the lack of communication technologies. Recently, it became common for PDAs to incorporate a cellular communication technology such as GSM, so that most PDAs currently are smart phones. The market is changing and the PDA is becoming our personal communication hub. This trend is gaining momentum as phones become more powerful, and even start to compete with small laptops, as the Apple iPad is an example.

b) Requirements of Embedded Personal Devices

The transition from voice call based devices (cell phones) to smart phones was long anticipated and seemed natural. These devices absorbed the market of PDAs and are becoming a single market. The objective is to provide ubiquitous connectivity to the Internet, which continuously provide more services, and with the advent of Social Networks, starts to replicate our social fabric. This trend imposed major challenges to operating system development, underlying hardware and protocols.

Moving to a digital packet based network much increased complexity of the underlying communication stack. As an example, in certain interfaces of the 3GPP UMTS specification, more than 10 logical layers are required before an application can add its payload. The number of systems required for a single handoff also increased, and a single location update can generate more than 20 messages and involve at least 4 different systems. Such numbers are normal for a traditional host connected to the Internet, but the specificities of the applications used on mobile scenarios impose heavy limitations.

Smart phones (as other embedded devices) are very constrained in terms of resources. A typical system has a ARM CPU with a clock ranging from 200Mhz to 700Mhz, runs on batteries with capacity for 2 or 3 days of operation (non-continuous), and has less than 128MB of RAM.

Operating systems designed for smart phones, and other embedded devices, usually have limited multitask capabilities and are single user [9]. They must employ intelligent garbage collection mechanism so that available memory is maximized, and reduce power consumption by keeping complexity at a minimum and by disabling any unused hardware. According to ETSI regulation, cell phones can transmit data up to 2W of EIRP (900Mhz band), which would deplete a 3.6V, 900mAh battery in about one and an half hours if in continuous operation. An emerging trend (that will not slow down) asks for bigger displays, higher CPU performan[redacted] which further imposes higher efficiency.

On this platform users expect to have their email, chat with friends, surf the web, listen to music, watch television (using 3G, 4G or DVB interfaces), watch movies, and more importantly make voice and video calls (both using cellular and VoIP technologies). Web surfing, chatting or email imposes that devices have multitasking capabilities, as well as enough CPU and memory for processing and storing information.

c) Requirements for mobility

One important requirement created by mobility is that hand-off between attachment points, event when considering intra-technology, requires many messages, and the entire process must be completed very quickly. This requirement and its challenges applies both to embedded devices such as smart phones or PDAs, and to laptops. GSM technology, being a TDMA technology, allows devices to use their idle time to hand-off between attachment points. However, the communication stack must be able to react with real time assurances, which imposes the need for hard scheduling guaranties. In packet based networks using CDMA, such as the widely used 802.11 family of standards, hand-off is more complicated. A typical hand-off, in optimized situations, can take around 50ms (see Figure 9) [6], and 4ms is the minimum achieved internally by our consortium. However, these measurements are obtained in situations where terminals are mostly idle and no other traffic exists.

Streaming video or audio requires hard scheduling deadlines, where packets must be sent at a monotonic rate (10ms when considering ITU G.729). At the same time, the stream must be displayed (or captured), and sent through the codec. With too many processes requiring hard deadlines, and competing with a complex communication stack which must handle several interrupts per handoff (at least one per packet), it may be difficult to always have hard guaranties. The result is that video will lag or handoff will take longer, introducing packet loss, and consequently degradation in the user perceived quality. Moreover, as multimedia applications become richer, the interval between packets will decrease, imposing higher requirements on QoS and scheduling. With the advent of multi core devices, its important for real time scheduling to be supported so that applications can benefit for heterogeneous and mobile environments.

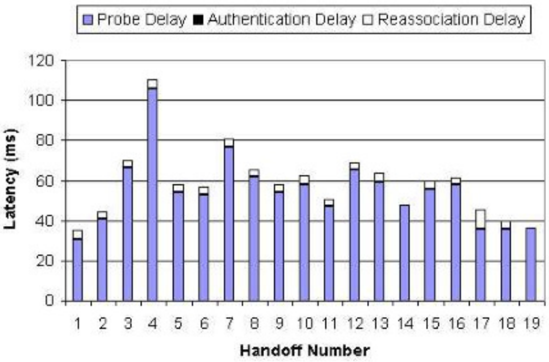

Figure 9: Handoff latencies of 802.11 with different phases identified [6]

D) Real-Time Requirements and Quality of Service

Personal embedded devices (and in general other embedded devices like TVs, set top boxes, etc,) exhibit some real-time requirements, although not critical as in the case of hard real-time systems (see Section VI. 7.). In particular, such systems must interact with the user providing a high level of Quality of Service (QoS). Quality of service includes functional requirements (as quality of video/audio, resolution, graphical effects, etc.) as well as non-functional requirements (as frame-rate, or sampling rate of audio streams, video or audio jitter, end-to-end delay). Of course, QoS must be as high as it is possible, to make the user experience in using the device more satisfactory. Given such constraints, these systems are usually referred as *soft real-time* systems, as computation and communication must typically be performed within certain time windows; nothing catastrophic happens if a time constraint is not respected, however missing deadlines can negatively impact on the QoS.

In addition, such systems are subject to many other constraints. One is power consumption and heating: these devices are powered by batteries that must last as long as it is possible. Also, they are subject to a high pressure on production costs, which are reflected in the final cost for the user and in the net income for the company. Therefore, as always, the objective is to achieve more with less.

In most cases, such systems are already multi-core and multiprocessor. The hardware architecture is heterogeneous, consisting of one general-purpose micro-controller with one or more specialised DSPs for specific tasks (like implementation of the RF communication protocol, or decoders for audio/video). Streaming applications, which present the hardest challenges in these systems, can be modeled by Data Flow diagrams, consisting of directed acyclic graphs of computations blocks, each performing one specific functionality. The graphs are then mapped on the heterogeneous hardware architecture, and an analysis is performed to identify bottlenecks, calculate throughput and end-to-end delay, dimensioning communication buffers, etc. Typically, the analysis is stochastic, as it is very difficult to estimate a-priori the execution time of such computational blocks, especially for compressed variable bit-rate streams.

Interaction with the user (human-machine interface) is executed as a normal application on the general-purpose processor which features a reduced version of a general-purpose OS (e.g. Win CE, Android), or an OS specifically developed for embedded devices (iPhone OS, Symbian, etc.). Usually, the DSPs run with a simple BIOS or a minimal OS that do not support multi-tasking and concurrency, and do not allow preemption.

One of the main problems in programming such systems is the lack of standard libraries and programming tools. Every vendor provides its own set of tools specifically targeted at their technology. Most of the programming is done ad-hoc, though low level C/C++ or even assembler code that must interface with the libraries provided by the vendor. Code is statically mapped on the cores. A more standard and coherent approach would greatly benefit the development of such application, greatly reducing development costs.

In the future, we expect a convergence between portable personal devices and regular PCs. On one end, such devices will become more and more general purpose, including a larger amount of services and applications. On the other end, the classical desktop PC has been substituted in most cases by laptops and netbooks, which will probably be substituted soon by tablet PCs, and other personal devices. We do not expect the "one-size-fit-all" paradigm: there will always be a large amount of different solutions to cover different markets. For example, in professional environments the classical PC with a keyboard and a screen will survive for a very long time. However, there is a clear trend for integrating applications that now are only available on PCs into "personal" mobile devices. Also, these devices will be "always connected" with local networks or Internet.

From the software point of view, we will see an increasing pressure for integrating different heterogeneous applications in portable devices without losing too much in performance.

One possibility will be to off-load part of the computation on remote servers, so to save on energy [104][105]. When in close proximity to a cloud, capacity can be borrowed from the surrounding environment and make embedded devices capable of more intensive tasks. Cloud computing could thus be an important opportunity to bring complex applications on these devices which will just support interaction with the user, demanding the bulk of the computation to the cloud. This is a scenario where S(o)OS can much improve over existing and near future use cases.

Also, interaction with the user will be much more sophisticated and "natural", featuring voice-recognition and synthesis, gesture recognition, and artificial intelligence to make the device more "human". In our opinion, sophisticated user interaction will still be performed on the device to avoid large response-time delays and unavailability in case of broken connection. Such services will require a medium-large computational power with low energy consumption, which can only be achieved by using multi-core technology. We still don't know if the most appropriate multi-core technology for these devices will be a standard general purpose array of processor or a heterogeneous network-on-a-chip architecture with dedicated processors for different tasks, although the second one seems to be more promising.

3. Social Network Applications

Social applications act in particular over largely distributed data.

Compared to other computer usage (such as High Performance Computing or Multimedia applications), social network applications (from the client perspective) have very little demands towards the processor power. Also, data throughput and storage is comparatively low. However, if we consider completely distributed social networking applications (without centralized systems), or we consider the back-end supporting a typical social application, many challenges arise. In particular with the growing interest social networks, the major obstacle is not so much the individual data, but the combination of all concurrent data requests and processes, as well as the need for fast petabyte scale storage systems – for example LinkedIn counts more than 60 million users already, and Twitter even more than 75 million, and the numbers are growing. According to [13], in Facebook, which already surpassed the 400 million active users, each person is connected to an average of 130 other users and 60 other pages and events (groups, fan pages). Also form the same sources, each month, the 400 millions users (100 million using mobile devices), exchange 25000 million objects, which gives an average of 62.5 million transactions per user per month. The result is an astounding 260000 million page views [13]. If we only consider photos, Facebook reports that its storage increases at a rate of 25TB per week. That is about 250MB per user per month for photos, numbers will be much higher if we consider the complete set of interactions. As an example, reports indicate that it generates 25TB of log data per day [10]!

In order to handle that size and amount of data, the principles of cloud computing have been employed by data centres for a long time now. This means in particular that data (and some functionality) moves and grows with the user: the higher data of a specific source is in demand, the more often it is replicated. For example the eBay platform recorded in 2003 to have 670 million page views, 120 million searches and 4 billion database calls a day [88]: if all auctions would get the same degree of availability through replication, high-demand auctions would either be hardly available or the full data size would go beyond the currently available data space of any data centre.

In general it can therefore be noted that both mobile and social networks put forward specific requirements towards data throughput and availability.

A) Requirements of Social Networks

Obviously, social network applications focus primarily on communication, and message / status, and object (photos) exchange between members of the respective community. However, this can take various forms, depending on demographics of the users primarily addressed by the networking platform. For example, professional networks, such as LinkedIn or Xing focus on business-oriented relationships and thus aim mostly at professionals aged between 30 and 50 – in 2008 they recorded 7.5M unique visitors in a month[17]. As opposed to that, social networks such as MySpace, Facebook and Twitter aim at a broad user scope and provide general communication, status and event services, such as picture sharing etc. Even though these networks typically attract a more younger user demographic, Facebook registered 484M unique visitors in 2010 in one month[18].

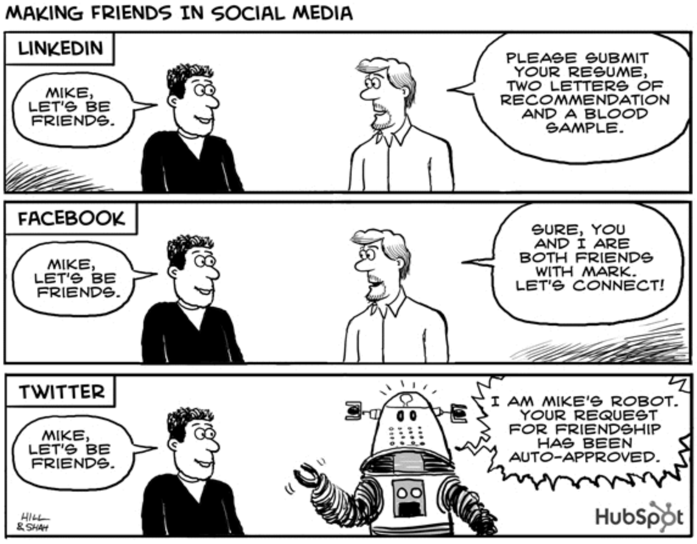
MAKING FRIENDS IN SOCIAL MEDIA
LINKEDIN
MIKE, LET'S BE FRIENDS.
PLEASE SUBMIT YOUR RESUME, TWO LETTERS OF RECOMMENDATION AND A BLOOD SAMPLE.
FACEBOOK
MIKE, LET'S BE FRIENDS.
SURE, YOU AND I ARE BOTH FRIENDS WITH MARK. LET'S CONNECT!
TWITTER
MIKE, LET'S BE FRIENDS.
I AM MIKE'S ROBOT. YOUR REQUEST FOR FRIENDSHIP HAS BEEN AUTO-APPROVED.
HILL & SHAH
HubSpot

Figure 10: community building policies in different social networks [87]

[17] According to http://www.web2innovations.com/most-popular-web-2.0-sites.php

[18] According to http://techcrunch.com/2010/04/21/facebook-500-million-visitors-comscore/

General purpose networks can take on different flavours and shapes, though they generally aim at combining fun and entertainment with the communication and messaging capabilities. Second Life takes a strong game-like approach towards such networking which includes a full 3d virtual environment, very closely related to massive multiplayer online (MMO) gaming. As opposed to such games, however, Second Life does not prescribe game missions and hence restricts itself to specific gamer types, but instead allows people to "build their own world". Surprisingly though, Second Life attracts less users than World of Warcraft (WoW), one of the most successful MMO games: whilst WoW attracted 9M users in one month, Second Life peaks at 1M. Even though 3d platforms are expected to become more and more relevant even in social networking, the underlying requirements and application types with respect to the virtual environment are effectively identical to gaming (see section VI. 6.).

What is more important for social networks is the sheer amount of data communication: as already mentioned in the introduction, the data per connection is comparatively small, as not even in MMO games the full environment is transferred, but primarily control data, or just communication and status update messages. The main stress on the system originates from the number of users which access (down- and upload) data concurrently: e.g. in 2008 there were already 2-3 TB of photos being uploaded to Facebook every day and traffic peaked at 300k images *per second*; Twitter served 3M messages per day and Last.FM 18M tracks per day[19].

Other type of applications which is around the corner are fully distributed social applications. These arose from the fact that traditional web applications such as Facebook, MySpace, LinkedIn and Twitter, create much concerns regarding data privacy. In these systems, users are frequently lost between tens of privacy settings, and companies can change their privacy policy at any time. Moreover, all information added to a system is seen as an asset of the company, in the same manner a car or a building are assets. When companies are sold, merge or declare bankruptcy these assets are made available to new persons. Eventually we will all loose track of who owns our information.

Following this trend, future environments (e.g. Get6D) promote close collaboration between users but in a decentralized manner. In this case, there are no privacy policy to agree upon and users are responsible for making available or delete all their personal information. While public pages, such as blogs, may be available, actual data is not centralized on ones system.

From a technological point of view, these environments rely in the concepts first developed for P2P file sharing networks, yet they extend the standard TCP/IP network stack. The result is a pervasive communication environment which must keep track of tens or hundreds of other systems all over the world, with the support for distributed object access and reliable transaction support.

SUMMARY

The major requirements put forward by social network origins from their high data driven application: this does not so much relate to latency and bandwidth per connection / communication, but in particular to the amount of concurrent data streams and messages that are forwarded to (and possibly broadcasted from) the individual user machine. On the server side, data management implies in particular (1) high availability given (2) large degrees of concurrent access, implying according means of (3) storage; finally (4) data consistency given the large degree of concurrency will play an increasing role, though not in the same scope as in Cloud applications or shared workspaces.

[19] All statistics have been taken from [14].

4. Office and Home Usage

The most common usage of computing systems consists in work desk and home applications which require comparatively high interactivity and response rate

Home and office usage still dominates computer usage and as such defines many of the requirements for next generation computer development. Nowadays, most desktop systems can be distinguished according to three categories (of use): gaming, multimedia and office – each of these systems put a different stress on its hardware configuration so as to meet the respective domains usage requirements best. Whilst a large degree of home usage is defined by gaming and multimedia applications, as well as by internet and social networks, a substantial part is defined by the more office-related applications, including document management but also personal calendars, contact management etc.

Since almost all enterprises and companies make extensive use computing systems as part of their daily work, such applications have a substantial share in the market and even though the average requirement towards such applications is low compared to e.g. gaming, users not only demand richer and easier features, but in particular high interactivity and reaction speed: a surprisingly high number of users complains about start-up time of both computer and application, even if that is only in the range of 1 or 2 minutes in total [99][100].

a) Specific Application Types

Work related usage of machines is not restricted to word and spreadsheet processing, but actually consists of a – typically strongly integrated – set of applications, which base on similar underlying technologies though. Investigating the whole range of applications, one may in particular note the following applications:

- word processing
- spreadsheets
- slide generation & presentation
- emailing
- task managers
- calendar
- contact manager / address book
- shared workspaces (document managers etc)
- database management
- etc.

It will be noted that common to all (home) office applications is the capability of (1) editing data through the user, as well as generally some means of (2) saving and retrieving data (database, file system). Further to this, some applications have specific functionalities in common which will be examined in more detail in the following:

3. data management

   this involves not only storing and retrieving data from a database, but also selection, querying and some (simple) combinatorial and logical functions on these. The demands of such calculations and combinatorial tasks are however way lower than the ones of typical HPC applications (see there) by the magnitude of a few orders.

4. calendar / timer

   a simple functionality to keep track of date and time in a way that allows association with events and initiate reminders.

5. communication (internet etc.)

   data (including files, databases, simple texts etc.) is more and more frequently communicated over the web – not only in the form of email, but also for the purpose of sharing documents and tasks in a workspace etc.

Emailing for example essentially consists of the capabilities to edit text (1) and communicate it over the network (5). Typically, one will also store the mails (2) and potentially execute searches over them (3), and finally maintain the information about time of arrival, response etc. (4). A spreadsheet consists of a mostly textual editor (1) that enables data management functions (2) on storable data (3). We will not elaborate this for all applications, though, as the principle is quite apparent.

From these categories it can be noted that the general requirements of these applications types are comparatively weak (regarding computational performance) as compared to the other application domains. Main computational load rests in particular on potential data queries and combinational tasks, as well as graphical displaying and editing, depending on the complexity of the graphics – note that explicit graphic manipulation capabilities are treated under the subsection of Multimedia and Gaming (see there).

Requirements

With the set of application kernels (cf. “13 dwarfs”, section on HPC), a set of functional capabilities can be derived that are needed in order to realise the application's demands. For most of these, the direct requirements are quite obvious:

1. editing data:

   To support data editing, some form of visualising data is needed, as well as some means of interactivity, such as keyboard input management. Whilst these requirements sound like they can be taken for granted, graphics demands even in common desktop applications increase constantly – be it for the (aesthetic) visualisation of (potentially dynamic) data or for preparation of slides with diagrams and charts. As compared to more demanding, specialised applications, the data set is typically much smaller, but similar visualisation techniques are employed (see Multimedia and Gaming; HPC).

   More critically though is the aspect of interactivity in this context: for direct interaction with a machine (i.e. typing etc.) to not become cumbersome, interactivity delay must be lower than 100 ms, and in order to not interrupt the user flow, displaying information should not be delayed more than 1s, though the user will not have the feeling of acting directly on the data any more [34]. Both these aspects imply that direct working on the internet without additional precautions is not acceptable for the (home) office application types. This also restricts the acceptable routing delay within a distributed system for acting on the data – in other words: fast processing is secondary, if the data is not displayed fast enough.

2. saving & retrieving data:

   Next to the availability of some storage element (typically some disk storage somewhere) that is persistent enough for the purposes of the application, a similar delay time as for data editing needs to be respected (less than 1 second to maintain user flow, less than 10 seconds to keep user's attention focused [34][35]). To this end, loading and saving of data is typically executed in the background of the application, which means that an image of the data at save time is needed to not create data conflicts or block the user interaction with the data.

3. data management

   In home office applications, data is not only created and stored, but typically processed too (e.g. spreadsheets). Though mostly the required combinatorial functionality scope is comparably low, most spreadsheet systems offer a mathematical capability scope that does not yet fulfil the complexity or depth of MatLab or similar programs, but is getting closer.

   Essentially, Excel formulas are small programs which can essentially be solved by a finite state machine and thus have similar requirements towards the system – in worst cases, linear algebra solvers etc. have to be executed though again typically on less data than in more demanding applications (see section on HPC). Accordingly, workload distribution will typically cause more delays due to segmentation and distribution than gained by the parallelised execution.

4. calendar / timer

   The capability to maintain date and time does not pose specific demands on the system, unless real-time requirements have to be respected. In case of the given applications, this is however not the case.

5. communication (in the sense of internet)

   Even though the size of data in these kinds of applications is small compared to the data in multimedia or HPC applications, data throughput of the network is a critical factor in these cases due to their *interactivity* demands. In particular with the growing integration of networked resources for application purposes (outsourced storage, collaborative workspaces, cloud environment etc.), it is important that this does not cause unnecessary delays or bad user experience. Unfortunately, not all tasks can be executed in the background of the application, so that time constraints, and more critically coherency and synchronisation constraints have to be respected across the systems involved. Reduced latency and increased bandwidth alone are insufficient to address this issue as data need will grow faster – instead new data distribution and data management approaches are needed, as well as means for handling updating and coherency maintenance in the background.

B) SUMMARY

The most critical aspect in (home) office applications consists in their demand for interactivity and the implicit real time constraints with time windows of less than 0.1 seconds for direct communication (mouse, keyboard) and less than 1 second for interactions that maintain usage flow. In particular in shared workspace environments, where changes on one end would affect the data on all other ends, time delays may lead to bad user experience.

On the other hand, most applications of these kind are strongly data *editing* (and hence interactivity) centric, but produce comparatively little processor work load, so that the actual computational requirements are mostly low and have only little parallelisation requirements.

5. Multimedia & Stream-Processing

Applications acting on (audio & video) streams require high data throughput.

Speaking of multimedia we think of digital audio, video and still images. Presentations including effects which require 3D support like the visual controlling enhancements when browsing through a collection of images or a document can also be added to this section.

Encoding and decoding high resolution video including one or more audio channels is a very processing intensive task when done by a multi-purpose processing unit without support of specific functions in hardware. Equipping devices with such powerful processors is both expensive and not very wise thinking of the power consumption and heat generation especially on mobile devices.

To enable HDTV decoding on netbooks NVIDIA recently came up with the ION chipset[20] which enables the Intel Atom, mostly built into small desktop systems and netbooks, to replay H.264 full HD video content.

The recording and encoding of digital video with high resolution is a much more computation intensive task than decoding and is therefore mostly done in hardware to meet the real-time requirement, for example in camcorders or video telephony and conferencing systems. When recording digital video, not only the processing power to compress the frames in real-time is a limiting factor but also the access to data storage for video and audio data which also implies the handling of large data sets. Most times this is a trade off between data size and processing power. Keeping the same image quality can either be achieved by less processing intensive codecs and a higher data bandwidth or more advanced computing intensive video compression at lower data rates.

3D accelerated handheld devices like the iPod, iPhone etc. also use specialized hardware to display 3D effects. Unlike the modern graphics adapters from ATI and NVIDIA these controllers are also optimized for low power consumption and support only a set of specific operations like resizing or rotating still images in real time[21].

Not only mobile equipment but also desktop machines can gain profit from application specific hardware audio and video accelerators. Hardware for a specific purpose can be optimized for low power consumption, handling high data rates from audio and video sensors and covering real time requirements regardless of the power of the main processor.

Most graphic adapters of consumer home computers contain hardware accelerators for video decoding of online content and DVD video to reduce the CPU processing load or even enable the system to decode the streams in real time. This has been the usual setup since the late 1990s when the MPEG-2 standard was used for digital video on the nowadays widely available DVDs. With the rising core frequency and with that the additional processing power decoding an MPEG-2 stream was no longer a killer task, but bandwidth to broadcast or stream video compressed in MPEG-2 was an issue. The follower of MPEG-2 was MPEG-4 Part 2 more commonly known by the names of codec implementations like Xvid or DivX which were capable of compressing video at much lower bitrates but keeping the same image quality. Many consumer standalone DVD players nowadays are able to playback not only MPEG-2 video stored on DVD discs but also MPEG-4 compressed material which usually fits on a single CD-R at the same image quality[22] .

In the last few years, when high resolution flat panels got more and more affordable by consumers, two optical media standards emerged: the HD DVD and the Blu-ray Disc. The two formats stood in competition to each other on the market which was won by the Blu-ray Disc when Toshiba dropped

[20] More information is available at: http://en.wikipedia.org/wiki/Nvidia_Ion.

[21] http://en.wikipedia.org/wiki/IPhone

[22] http://arstechnica.com/gadgets/guides/2009/12/from-cinepak-to-h265-a-survey-of-video-compression.ars/

the development of HD DVD standalone players and Sony's PlayStation 3 was a widely available game console capable of playing Blu-ray Discs[23].

The video compression used for high definition material on Blu-ray Discs and for digital television broadcasting over satellite or cable is MPEG-4 Part 10 or more commonly known as H.264 which again outperforms MPEG-4 Part 2 in times of video quality and bandwidth usage at the cost of being more processing intensive to decode. The follower of H.264, currently under development, will be H.265 introducing a loss in picture quality when looking at the single frames of a video sequence. But since video is about moving pictures the single frame quality of fast paced video sequences won't be noticeable by the human eye.

Besides the development in video compression technology content creators keep looking for new ways of improving their material by adding new visual experiences like 3D effects to movies like the 2007 production 'Beowulf' viewable in IMAX movie theatres and the 2009 production of 'Avatar' which was also available for regular movie theatres equipped with a technology to synchronize special shutter glasses worn by the audience.

Offering 3D material for the consumer market TV panel producers like Samsung, Panasonic and Sony recently started selling kits of shutter glasses to be used with their products for watching 3D enabled Blu-ray movies. Focusing the desktop PC market NVIDIA offers solutions to enable 3D experience for computer games as well as digital video on home computers equipped with their recent graphics adapters in combination with LC monitors capable of running at a display rate of 120 Hz and a pair of shutter glasses[24].

Specific Application Types

Large streaming applications include radar and radio astronomy. These application are now currently implemented as phased-array receivers that use multiple antenna elements to pick up radio waves. The antenna elements are positioned at a certain distance from each other, such that the wave front arrives at slightly different times at the various elements. By combining the signals from all elements, the original signal can be restored and the direction of the transmitter is obtained. This process is called Beamforming. Multiple transmitters can be traced at the same time by applying different delays to a copy of the input signals. State-of-the-art radar systems can for example form 10 beams, thus tracking 10 objects, at the same time.

A state-of-the-art radar has around 4096 antenna elements, that are sampled at 16 bits at 200 Mbps. The resulting stream is thus 13.1 Tbit/s at the input of the digital system. The required processing is equally large, with around H*410 GMAC/s for the antenna processing (H-order FIR-filter with H/2 non-zero elements), and around 8.2 complex TMAC/s for the beamforming (when beamforming 10 beams). This also results in a large amount of internal communications, where the communication between the antenna processing and the beamformer reaches 54.2 Tbit/s (assuming 32bit complex numbers) and 128 Gbit/s between the beamformer and the information processing part. These figures should give some indication about the large amount of processing that is required for these systems, and that even only buffering for a second already requires a large amounts of low-latency (and most likely high bandwidth) storage.

Synthetic Aperture Radar (SAR) Satellites also use beamforming techniques, and although they do not form as many beams as radar systems found on military ships, they are burdened with compressing the information stream to fit the limited bandwidth of the communication link between the satellite and the base-station. The resulting computational complexity (given by the number of (complex) operations) is in the same order as the computational complexity of the radars found on

[23] http://en.wikipedia.org/wiki/High_definition_optical_disc_format_war
[24] (http://www.nvidia.com/object/3D_Vision_Main.html)

the military ships. The power envelope of SAR-Satellites is however many orders smaller than the power envelope of ship-based radars (30 Watt vs many KiloWatt).

Summary

- Stream processing (encoding and decoding) often performed by dedicated processors in heterogeneous platforms
- Typically organized as Data-Flow graph applications
- Soft Real-Time requirements that impact on Quality of Service (QoS), require precise QoS control over computational and communication resources
- Higher definition and 3D will require more bandwidth and higher computational requirements
- In SAR, power requirements and large amount of data require

6. Gaming

3D worlds dominate the gaming industry which demands high interactivity and fast graphics processing.

Playing console or PC video games is listed amongst today's most common leisure activities in Europe, where 40% of the European population spends about 6-14 hours a week on playing video games [17],[18]. In the USA about 68% of the households play computer of video games, 42% of homes in the USA have a video game console, and 37% of heads of households play games on a wireless device such as a cell phone or PDA [15][16]. In Europe 44% of gamers also play on a wireless device [17]. In the USA (2008), game console software sales totalled $8.9 billion with 189.0 million units sold; computer games sales were $701.4 million with 29.1 million units sold; and, there is $2.1 billion in portable software sales with 79.5 million units sold. Although these figures do not include the revenues made from the advertisements usually displayed in the (browser-based) free-to-play games, it is clear that (portable) console games are dominant in the interactive entertainment industry.

Examining the specifics related to these console software sales we see that the most popular game genre is “Family Entertainment”, which accounted for 19.3% of all games sold in 2008. In addition, of the games sold in 2008, 57% were rated “Everyone (E)” or “Everyone 10+ (E10+)”. Also the top 5 console games includes 4 games for Nintendo's Wii console, which are all “Family Entertainment” games. Only 16% of the games sold holds the “Mature (M)” rating, which includes the 5th most sold console game for 2008, Grand Theft Auto IV for Xbox 360. Of the top 20 (portable) console games, there were 7 games for the Wii, 6 for the Microsoft's Xbox 360, 5 for Nintendo's portable DS console, and 2 for Sony's PS3.

Examining the specifics related to computer software sales we see that the most popular game genre is “Strategy”, which accounted for 34% of all games sold in 2008. Also Role-Playing is a popular genre among computer game players, accounting for 19.6% of all games sold. This last figure correspond with the top 5 computer games, which includes 3 games belonging to Activision-Blizzard's popular World of Warcraft series.

62% of European computer and video gamers also play these games online [17]. 42% of gamers between the age of 6 and 42 in the USA play games online for 1 or more hours a week [18]. Of these games, the games that are played the most belong to the type “Puzzle/Board Game/Show/Trivia/Card” which are usually (browser-based) free-to-play games. Followed by “Action/Sports/Stragegy/Role-Play” with 21%, and “Persistent Multi-Player Universe” on 3rd place with 16%.

A) Specific Application Types

Games will be subdivided into 3 main platforms: Non-portable consoles (such as Wii and PS3), PC and Mobile (which includes portable consoles such as the Nintendo DS). As all three platforms can connect to the internet, online games have been included in this section as well.

Console – Low-Performance

The main difference between console games and pc games, is that console games are usually the only running program on the platform. Console games are also expected to be playable within seconds the console is turned on. So this means that the demands on an operating system are few, just that it initialises the system fast, and relinquishes control to the gaming application as soon as possible.

PC – High-Performance

PC Games that are most demanding on their hardware, and are used to benchmark the latest GPUs and CPUs are those that fall within the "First-Person Shooter" (FPS) genre. Familiar titles include Epic Games' "Unreal" series and ID Software's "Quake" series. Crytek's "Crysis: Warhead" is considered todays most demanding (and perhaps least efficient) game in the FPS-genre used to benchmark the latest range of GPUs. Only the latest GPUs such as Nvidia's GeForce GTX480 (Fermi/GF100 architecture), in combination with high-end CPUs (Intel Core i7 920), can now can display the large open worlds with a minimum of 30 frames-per-second[25] (although the average case is higher), when the game is set to its highest visual fidelity settings on a FullHD (1920x1200) screen.

Although the introduction of video cards that support DirectX 11 might make one of the newer games, such as A4 Games' Metro 2033 (which uses specific DirectX 11 features) the "default" gaming benchmark. A synthetic benchmark used to test how well graphics cards perform for these high-performance games is the 3DMark benchmark suite, with 3DMark Vantage being the latest instalment.

The reason that the FPS-genre, and especially games like Crysis, are so demanding on the hardware is that they try to display interactive, realistic, open 3D worlds, with a lot of foliage, as a live stream. Having 3D worlds approach reality does not only mean the realistic rendering of all its objects, but also modelling the physical interactions between all its objects, such as trees moving in the wind and the destruction and collapse of buildings.

Mobile

With the porting of Snake from TI's graphical calculators to Nokia's mobile phones in 1997, the first game for a mobile phone was born. With the introduction of smart-phones, new games can be downloaded and installed on a phone, increasing the market for mobile game developers. Games cause a mobile phone to be in constant use, so although the battery-hungry antenna might not be used, the backlight of the screen as well as the CPU of the mobile device are put to constant use. This means that the OS should cater for several power-saving techniques that are "gaming-friendly", such as saving the state of the game and shutting it down after a certain time of inactivity.

B) Online Gaming

The nature of the Internet poses several challenges for the real-time requirement of online games. The Internet is a shared medium and packets can be lost or delivered out of order. Which means that, there are no guarantees of network bandwidth, latency or reliability at all.

Most online games run on a client-server architecture, which enables large scale Internet gaming. In this architecture the server is authoritative and, as such, it is responsible for all game play decisions

[25] 30 FPS is considered as the minimum frame-rate for high-pace games at which motion appears smooth

[20]. If the latency between the client and the server is large enough, the responsiveness of the game to a player action decreases, and the player performance is likely to degrade. However, not all games are equally tolerant to latency. In a “First Person Shooter” game, shooting at a moving opponent is greatly impacted by latency, while in a “Strategy” game, selecting a set of troops and moving them across a battlefield tends to be less sensitive to latency [22].

Game traffic behaves differently in different phases of online gaming. During a play session, the traffic rate of an action game depends mostly on the players actions, while in a MMO game the traffic rate is more consistent [36]. A player can join a game in the middle of a session and it may need to download assets from the server if these are not cached locally. Downloading imposes a high demand on bandwidth which as to be balanced with other clients already playing. But due to lack of QoS, game servers have to use a hard limit on the download rate, resulting on less efficient use of bandwidth.

Action games, like a “First Person Shooter”, are one the most demanding games for bandwidth, reliability and latency. If network bandwidth was unlimited, at each game state update[26] a server would send an exact game state update to all clients. The reality is that, the limited bandwidth and the low latency requirement makes this difficult[27], and while bandwidth has been increasing so has the complexity of action games (more complex game physics, wide open areas with player manned vehicles and on foot). To overcome this issue, a game server transmits a reasonable approximation of the game state to the client [20]. Client side prediction is used to compensate this approximation and also packet loss, however its accuracy depends on available bandwidth and rate of packet loss.

The TCP transport protocol is widely used for the majority of reliable Internet traffic. In the early days of online gaming, it was used by most game engines. TCP provides means for flow control, congestion control, ordered delivery and reliability. Thus it was far more easier to write a game engine using TCP, and at the time, very few had Internet access. However, for more demanding game genres TCP provides a much more poor experience, because reliability comes at the price of retransmissions that increases latency. While today most demanding games avoid reliable transport protocols, games that are less sensitive to latency have successfully use it, which is the case of games like World of Warcraft [20]. Nevertheless, TCP still has some undesirable features that impact latency. One of these is the Nagle algorithm, which is supposed to improve efficiency by waiting for small chunks of data to be gathered for sending within a threshold. This has the obvious side effect of increasing the latency, thus it is usually disabled by the option known as TCP_NODELAY[28].

To overcome the poor experience provided by TCP in more demanding games, many have implemented middleware solutions on top of UDP. UDP is much like the underlying IP protocol, in that there is no congestion control and mechanism for reliable or ordered delivery of packets. For games with high sensitive to latency, its critical that clients receive frequent updates of the game state, if a packet is lost there's no point in being retransmitted because only the most recent game state matters. While receiving the position of opponents can be unreliable other actions, such as firing a weapon, must be transmitted to the server reliably. Unfortunately, UDP does not provide any support for reliability, congestion control, flow control and connection management. The middleware solutions implemented by most games address some of these. But some issues, like congestion and flow control, are not trivial.

One protocol that provides many of the requirements for online games is SCTP. Originally develop for telephony, it provides reliable and unreliable packet based communication. However, because of

[26] A game state update is the state of the game simulation at each iteration.

[27] For low latency it is desirable to send a game update in one packet, thus avoiding the additional delay between packets for the same game state update, but this is limited by the MTU. To overcome this, games prioritize individual object state to be sent to clients based on their impact on game play [20].

[28] In Berkeley like socket interfaces, see: http://www.opengroup.org/onlinepubs/000095399/basedefs/netinet/tcp.h.html

its lack of support on major commercial operating systems, little is known on how it performs for online games in a real world scenario.

c) Summary

Figures [15][16][17][18] show that console gaming is the dominant form of gaming as far as revenues are concerned. Within this domain, by total units sold, the casual “Family Entertainment” genre prevails, and games for Nintendo's Wii hold the #1 to #4 spot in the top 5 console games. This shows that cheap/low-performance hardware, combined with software for 'casual' gaming, is the platform/game combination where most money is made. Games that are most demanding of their hardware include games that fall within the “First-Person Shooter” genre, such as Crytek's Crysis, and are usually rated “Mature (M)” which only account for 16% of the sales, for both video and computer games.

So from an economic point of view, it seems wisest to focus on those aspects of an operating system that are most beneficial for running 'casual' games on low cost / low performance hardware, and not the hardcore/mature games that run on high-cost / high-performance hardware.

There also seems to be a move to duplicate parts of the game to an online platform, for example Activision-Blizzard's World of Warcraft has an iPhone application with which you can access the in-game marketplace. In some sense, World of Warcraft's in-game market has actually become a service which you can not only access using the regular game-client, but also a mobile application or even a web application.

In spite the best effort nature of the Internet, online games have been successfully in providing a general good experience. Still, many improvements can be done as the complexity in games will continue to increase. Requirements on future networks to support bigger MTUs will help keep low latency, as more information can be transmitted on a single packet. More flexible transport protocols with reliable and unreliable communication, packet and stream based flows for the game communication needs. And QoS that can be adapted for different game traffic patterns during the various game play phases.

7. Hard Real-Time Control Applications

Hard real-time systems require predictability of response-time, and at the same time early performance estimation through accurate hardware models

This class of applications includes plant control, robotic and industrial automation systems, aerospace and automotive systems.

Typically, these applications have critical timing requirements, so they are called *hard real-time systems*. They are implemented by a set a periodic concurrent threads, that must complete before their deadlines, otherwise the system correctness could be compromised. Depending on the criticality of the applications, missing a deadline could have catastrophic consequences. An example is the software for controlling the brakes of a car: a bug can cause the malfunctioning of an essential component of the car safety, which in turn can cause injuries to human beings.

The design of these systems is quite complex due to the strong requirements on software testing and verification. The main programming language is plain C, mainly for efficiency reasons. However, development is supported by design and analysis tools that guide the developers through all phases of the life cycle:

- specification of the control algorithm in high level graphical block languages (e.g. Simulink and Stateflow from Mathworks, or UML Statecharts and MARTE UML);
- to code-generation;

- mapping or code on threads, and of threads on computational nodes;
- configuration of threads parameters (e.g. priorities, periods, etc.)
- static analysis of the code for estimating execution time;
- scheduling analysis to identify potential deadline misses or bottlenecks;
- and finally, generation of test-cases.

In certain contexts (e.g aerospace applications) reducing the cost is a secondary objective, safety being the first. In other areas, developers are subject to a high cost pressure: it is the case of automotive software, where electronic boards and software are produced in large amount of pieces, and reducing the cost of the hardware can bring significant cost reductions. Thus, from one side there is pressure on safety and validation; on the other side there is pressure on reducing cost by using the least powerful computational resources.

In the past, safety and fault tolerance were achieved by using a separate hardware board for each different feature. Considering again the case of automotive software, brake control was running on a physically separated board than e.g. power train and gear shift controllers. However, such approach does not scale well with the ever increasing number of required features. Hence, the need to integrate different features on the same physical board, possibly featuring a multi-core embedded microcontroller.

Many chip vendors are proposing their own multi-core solution tailored to embedded systems, with a significant reduction in power consumption, heating, etc. with respect to single core solution with similar computational power. Examples are the ARM 11 [32], the PowerQuiCC Series from Freescale [33].

Currently, penetration of such architectures is still slow in the domain because of the lack of proper design and tools for multi-core, for the increased difficulty of synchronization, and for the lack of an establish real-time scheduling methodology.

For the next future, we foresee a slow but steady increase in the adoption of multicore technologies for such systems. However, it is clear that the adoption of multi-core and many-core platforms will only be justified by a strong market case, as most of the developers are concerned with the increased difficulty in designing, programming and analysing parallel software rather then single-processor concurrent software. Therefore, as the availability of software design techniques and tools for multi-core processors will be more widespread, the market will move to multi-core solutions as well, most probably with a few years of delay with respect to non-real-time applications.

VII. Conclusions: Requirements & Relevance

Given the expected developments and the usage scope, an initial assessment of which features future systems need to address can be provided

1. A Glimpse of the Future

From the analysis on the expected future trends in common application domains made in the previous chapter, we can now try to forecast the emergent future applications, and the requirements they will pose on future generation computing systems. As already discussed, the user requirements are the main forces driving development in the computing domain, the ones that will select the most successful strategy out of the many tentative paths that will be tried by hardware and software designers.

However, it is important to underline that future computing architectures, and the expected increase in computing power, will enable new applications (or new services for existing applications) that are simply impossible right now due to their high computational requirements that cannot be fulfilled by current technologies. Therefore, it is not always clear who is the driver: is application requirements that impose the development of certain technologies, or new technological advances that make new applications possible?

This chicken-and-egg problem, however, should not refrain ourselves from reasoning about the future of computing technology and application usage: while it is impossible to forecast the future, the general directions that the computing technology will take are quite evident from the discussion made in the previous chapters. In the following section, we will pin-point general characteristics of current and future applications and the requirements they impose on the underlying technology.

2. Requirements for Realising the Future

In order to identify the requirements of future emerging application, we first classify applications according to their general *characteristics.* Then, we will derive requirements from these general characteristics. The following table provides a correspondence between applications described in the previous chapter, and characteristics.

		Collaborative	*Interactive*	*Real-Time*	*Streaming*	*Data-Intensive*	*Computation-Intensive*	*Bandwidth required (high = large)*	*Latency required (low = little latency)*	*Coupling level (degree of synchronisation)*	*Scalability*	*Off-loading*	*Mobile*	*Specialized HW*
HPC	*Embarrassingly parallel (P2P computing)*	No	Generally no	No	Rarely	Sometimes	Often (e.g. Folding@ Home)	Relatively high	Relatively high	low	Essentially horizontal	Yes	No	No
	Embarrassingly parallel (like visualisation)	Maybe	Maybe	Maybe	Maybe	Often	Rarely	average	Relatively low	Low (rendering)	Horizontal / Vertical	No	No	Yes (GPU)
	Tightly coupled (e.g. structured grids)	No	No	Rarely	Rarely	Often[29]	Often	Relatively low	Very low	High	Vertical	No (latency too high)	No	Often
	Cloud Computing (Grid and similar)	Often	Often	rarely	Rarely	Rarely	Rarely	high due to replication	Relatively high[30]	Low	Horizontal	Yes	Maybe	No
	Personal devices (client)	Maybe	Yes	Rarely	Maybe	No	No	Relatively low	Interactive (<0.1s)	LOW	-	YES	YES	yes
	Social Networks (server)	YES	Yes	-	Maybe	YES	-	High due to replication	Interactive (<0.1s)	LOW	horizontal	-	-	-
	Office/Home (server)	YES	Yes	-	-	YES (server)	-	High due to replication	Interactive (<0.1s)	LOW	horizontal	YES	-	-
	Office / Home (standalone PC)	Rarely (but increasing)	Yes	No	No	Maybe	Rarely	-	Interactive (<0.1s)	None	None	no (cf clouds)	Rarely	No
Gaming[31]	*standalone*	No	Yes	Soft	often	Yes (rendering)	Increasing (AI, PhysX)	n/a	Interactive	n/a	None[32]	No	some-times	Yes (consoles)
	MMO (client)	yes	Yes	soft	No	Yes	High (graphics)	average	interactive	low	none	Yes (see MMO server)	Sometimes	Yes (consoles)
	MMO (server)	Indirectly	indirectly	soft	No	No	Low	High (server)	interactive	average	Horizontal[33]	Yes (rendering)	No	No (GPU)
	Action (client)	Yes	Yes	Soft	Yes	Yes	High (rendering)	Average	Low	Low	Vertical	Maybe	-	Yes (GPU)

[29] simulations
[30] Depends on interactivity
[31] See HPC: visualisation for rendering related aspects (part of most games)
[32] Vertical for graphics cards
[33] Also vertical for graphics

		Collaborative	Interactive	Real-Time	Streaming	Data-Intensive	Computation-Intensive	Bandwidth required (high = large)	Latency required (low = little latency)	Coupling level (degree of synchronisation)	Scalability	Off-loading	Mobile	Specialized HW
	Action (server)	Indirectly	Indirectly	Soft	Yes	No	Medium-High (physics)	High	Low	High	Horizontal	Yes (rendering)	-	Maybe (physics)
	Remote Rendering (client)	Yes	Yes	Soft	Yes	Yes	Low	Medium	Low	Low	Vertical	Yes	-	Maybe (video decoder)
	Remote Rendering (server)	Indirectly	Indirectly	Soft	Yes	Yes	High	High	Low	Low	Horizontal and Vertical	No	-	Yes
Multimedia	*Offline Video & Audio*	-	-	Soft (frameskip)	Yes	Yes	rarely	n/a	n/a	Low	Vertical (Vector)	n/a	Maybe	often
	Online Video & Audio	-	-	Soft (frameskip)	Yes	Yes	rarely	high	low	low	Vertical & Horizontal[34]	Implicit	Maybe	often
	Video Chat	Yes	Yes	Soft (frameskip)	Yes	Yes	Rarely	High	Low	Medium	Vertical & horizontal	Partially	Maybe	often
HRT Control applications		-	-	YES	Maybe	-	-			HIGH	YES	-	-	-
Data retrieval (search engines)		-	-	-	-	YES	YES (indexing)			LOW	YES	-	-	Maybe
On-line trading		YES	YES	YES	-	YES	-			HIGH	YES	-	-	-

[34] Vertical for vector processing, horizontal for handling multiple clients

In the following, we will analyse and discuss each one of the characteristics listed above and derive requirements for the 4 research areas addresses in the project, namely hardware platforms, communication, operating systems, programming models.

A) Collaborative

An application is collaborative when it consists of multiple clients/peers, relative to different users, that can access and modify common shared data. All clients/peers share some state: however, the different views of the global state may or may not be always synchronised and consistent (see coupling level below). For example, in certain collaborative applications like document editing or chatting, at some instant one user can see an old view of the state while the others may see a more updated view. The different views of the state will eventually be consistent among each other. In other applications (like on-line trading, or on-line gaming), instead, it is important that all peers always have the same consistent view of the shared state.

Requirements:

- Collaborative applications are mostly distributed (client/server or p2p). The bandwidth and latency requirements depends on the specific application type. Some collaborative applications are interactive in nature (e.g. online chats).
- They are expected to deal with a large number of users, thus one important requirement is scalability with respect to number of users (horizontal).
- The memory model (shared or distributed, global consistency and coherency) depends on the type of application.
- Generally, consistency and coherency needs to be maintained.

B) Interactive

An application that interacts with one (or multiple) human user, which requires a response in a time relative to typical human time-constants, depending on the type of interactivity desired: for hard interactivity, less than 100ms reaction time are desired; for web like interactions (user flow), up to 1s delay is acceptable [108].

Requirements

- The typical requirement of interactive application is low-latency. To achieve low-latency it is necessary to operate at different levels of the system:
  - latency of computation (response time) requires proper scheduling/allocation of resources;
  - latency of communication (synchronisation) for distributed applications;
  - latency of I/O devices for data-intensive applications.
- Notice that we make a distinction between interactivity and from real-time requirement, as interactivity is a fuzzy concept that is mostly treated in a best-effort, heuristic way.

C) Real-Time

The application has real-time constraints, either hard real-time (i.e. missing a deadline may cause some incorrect computation) or soft real-time (i.e. late computation degrades the quality of service without causing incorrect behaviour). Again, real-time imposes a constraint on latency. As opposed to interactive applications, however, in real-time applications latency needs to be estimated a-priori, either at design time, or when launching the application. In both cases, an admission control test is done to estimate the expected performance, and if the performance is not satisfactory, an appropriate action is taken. This requires precise modelling and analysis of the system performance off-line; and on-line monitoring of resource usage at run-t[redacted] needs to have predictable behaviour and predictable response time, and this in turn may impose constraints on the hardware (i.e. cache may be locked or even disabled because it introduces unpredictability).

Requirements

- Predictable hardware; this implies precise control over (and knowledge about) hardware features like cache, processor interconnects, so to minimize unpredictable execution time.
- Predictable OS; operating system services must be designed and implemented to have a predictable execution time; heuristic algorithms will never be used in favour of predictable algorithms. For example, the scheduler must be based on priorities; memory allocation algorithms must have a predictable worst-case execution time; priority-based access to I/O devices; etc.
- Predictable, QoS-aware communication protocols: it must be possible to allocate a virtual channel with guaranteed minimum bandwidth and maximum latency over the network.
- On-line monitoring; both the hardware and the OS must allow for on-line monitoring of the performance of the real-time tasks.
- Memory model can be based on shared memory or distributed message passing, as both models allow for predictable algorithms as long as the previous requirements are met.
- Predictable application behaviour: in particular computation driven applications are difficult to assess with respect to their execution time.

d) Streaming

Applications that stream data from/to the user, in which data must be processed periodically at a certain rate (e.g. audio/video encoders and decoders). The main characteristic here is the constant (or periodic) arrival of data that can be of constant size (CBR) or variable size (VBR). In the case of MPEG2 coding, data belonging to different frames

Streaming applications essentially require that the same logic / operations are executed on a series (stream) of data and the behaviour is (only minimally) influenced by the data itself.

Requirements

- *Network on Chip architectures* may be used in this case, as streaming applications have a data-flow-graph structure, where each block in the graph can be implemented on a different processor, and data flowing between blocks can be supported by dedicated point-to-point processor interconnections.
- *Specialized hardware resources*; SIMD processors perform best in these cases (for example, vector processors or dedicated FPGAs for encoding and decoding).
- *Cache for data*: the typical data throughput is higher than the available cache size allows to store – accordingly, streaming applications quickly reach the point where data has to be gathered via external I/O (disk, network). Data throughput therefore primarily depends on how fast data can be forwarded to the compute units. A simple cache hierarchy can thereby lead to additional delays (as requests are routed to the higher levels) [107].
- Bandwidth may compensate latency by pre-fetching data – e.g. by employing stream load cache extensions [106].
- The problem can be solved by a parallel algorithm, which can thus employ multiple data / cache lines.
- The OS needs to support asynchronous I/O services, to pre-load successive data frames while processing current frames.
- The OS needs to support some real-time mechanism, to allow synchronization and periodic execution of the code. For example, audio and video must be synchronized to prevent bad

effects. However, in most cases streaming applications have soft real-time requirements, thus it is not required that the OS be fully real-time.

- The programming model should take into account the fact that most of this applications are modelled by DFG. MPI is already a good choice, and in general support for communication and buffering at the programming/library level helps the programmer to write clean and portable code. Streaming SIMD extensions may have to be explicitly invoked [106].

E) *Data-Intensive*

Applications that deal with large amount of data and comparatively little operations on this data, such as is the case for most SIMD ("single instruction multiple data") applications – this includes data mining, and graph traversals, as well as operations on (audio and video) streams (see also "streaming", above). It must be noted that not all of these applications can be equally well parallelised, as this depends in particular on their synchronisation and coherency requirements (audio streams are for example more coupled that data mining algorithms and hence more difficult to parallelise).

Requirements

- The general requirements are highly related to streaming applications.
- In most cases, bandwidth should be high, but throughput is automatically limited by the lowest bandwidth in the interconnect chain.
- Bandwidth can compensate latency.
- Cache should either be large enough for all data to fit in, or should address the streaming requirements above.
- Vector processors and similar SIMD units can support the repetitive actions on the data set.
- Parallelisability (in the sense of horizontal scale) depends on the specifics of the respective algorithm (see also section 13 dwarfs above).
- If the application is parallelisable, the same issues as for compute intensive algorithms apply (see below).

F) *Computation-intensive*

Applications that must perform a large amount of computation on comparatively low amount of data (e.g. many iterations in simulated annealing, magnetic field calculations etc.). Note that some simulations, in particular fluid dynamics applications etc. show behaviour similar to data-intensive computation, as the particles flow through the computation grid – this has specific implications on throughput which is generally higher and more dynamic than in e.g. simulated annealing cases. However, as the behaviour is data-dependent (i.e. the operations may differ, depending on data) and as each datum may have many computation iterations, we considered this a computation intensive application case, too.

Compute intensive algorithms can belong to either of the three types: (1) not parallelisable due to too many dependencies (simple example: many iterations on a single datum), (2) embarrassingly parallel, where e.g. the same computation and data set are subject to different parameters, and (3) tightly coupled parallel where the amount of compute resources define the overall execution time ("strong scaling").

Requirements

- In most computation-intensive algorithms, some form of data exchange is needed, so that synchronicity, coherency etc. should be supported, e.g.
  - Shared memory,
  - hardware- or software-cache-coherency (e.g. ccNoRMA),

    - UMA or NUMA architecture.
- The degree of communication, and hence the requirements on bandwidth and latency differ, depending on the parallelisation case:
    - not parallelisable: no communication needed (obviously),
    - embarrassingly parallel: higher need on bandwidth than on latency,
    - tightly coupled parallel: higher need on (low) latency than on bandwidth.
- The clock rate of the processor should be as high as possible (the higher, the faster the execution).
- Cache should be large enough to host the respective data segments and minimize cache misses.
- In parallel applications, there should be as little access to restricted resources (OS, I/O, disk etc.) as possible, as these resources may become bottlenecks.
- The threads / processes should take up as much of the processor's capacity as possible (exclusiveness, no time sharing).
- There should be no centralised control (such as OS), as this would lead to synchronisation and communication delays.
- Either data or work needs to be segmented – in either case, the programming language should allow for according segmentation and distribution, as well as implicit (and explicit) communication.

g) Coupling level

The amount of interaction between the different (parallel or concurrent) tasks of the application. For example, embarrassing parallel algorithms are loosely coupled (i.e. rarely exchange data), whereas fluid model simulation is typically tightly coupled (i.e. frequently exchange data) - see also the 13 dwarfs classification in the previous chapter for low, medium and high coupled algorithms. The coupling level partially depends on the presence of shared state in the algorithm. For embarrassingly parallel applications, it is possible to divide the (full) state space into separate and independent subspaces, so that every parallel task acts independently on the assigned subspace without influencing the others. In tightly coupled applications, instead, every modification to a state variable may influence all the state space.

Requirements for low-coupling

- Applications with a low coupling level may be easily implemented on modern distributed systems, like GRID systems, or cloud computing systems, because the low interactions between parallel applications makes it possible to distribute the load over a wide-area network of distributed systems, and the high latency in communication only minimally influences overall performance;
- There is no requirement on communication latency; the bandwidth depends on the specific application;
- GRID-aware distributed OS (like Xtreme OS) and Cloud Computing infrastructure support well.
- Message passing programming models like MPI are already adequate for this kind of applications.

Requirements for high-coupling

- Highly connected clusters or many-core systems with fast interconnection.
- On a single node they may require a large shared memory with cache coherency protocols, so to make it easy to represent the shared state, and maintain its consistency.
- Programming models that provide a shared memory abstraction (like PGAS) are best suited for this kind of applications.

h) Bandwidth

Related to the coupling level between instances / processes, the bandwidth requirements indicate whether the respective system has a high demand on data size, i.e. whether a large bandwidth is needed to transport the necessary information. In some cases, bandwidth can compensate lack of latency, but only if the data is not dependent on application behaviour (e.g. through interactivity) in which case latency would predominate the according requirements.

Most internet applications have a higher demand on bandwidth than on latency, as long as interactivity is still low in these cases.

Requirements

- Bandwidth is defined by the lowest common denominator, not only by the I/O – accordingly, data throughput, cache speed etc. play an important role
- In order to keep data available, cache size should be comparatively large
- I/O should allow for broad connections
- Bandwidth requirements can also arise from high level of concurrent access, in which case parallelism can be exploited (server case)
  - multiple I/O chipsets
  - multiple processors
- the OS (or execution environment) should try to merge multiple data sets / messages into larger packages, according to the bandwidth to exploit the connection maximally. Latency needs to be observed in this context
- the programming model should ideally make the developer aware of the latency / bandwidth capabilities and restrictions, respectively be able to restructure messaging accordingly (e.g. turn multiple small web requests in[redacted] by data dependency)

i) Latency

As opposed to bandwidth, latency is in particular critical for interactive and tightly coupled applications, i.e. where data needs to be available more or less immediately, as any delay leads to either frustration or essential performance loss. Notably in these cases bandwidth cannot compensate latency, as this would mean predicting user / data behaviour.

Latency is restricted by multiple factors, in particular by distance and type of connection – in general, the longer the distance between two endpoints and the more intermediary points exist on the route, the higher the latency.

Requirements

- Shortest and ideally direct connections
- The system (execution model, OS, compiler) must be able to collocate data & code closest as possible, according to their relationship / relevance
- The programming model should allow the user to specify segmentation boundaries, dependencies, connectivity etc. and thus define distribution restrictions
- The operating system must respect data access frequency / requirement etc. during data distribution

j) Scalability

Scalability of an application refers to the degree in which it can spread out (“scale up”) across multiple resources. We need to distinguish two types of scalability with respect to typical usage scenarios (see also [85]):

Horizontal scalability reflects the degree in which the application can utilise the infrastructure to increase its availability and thus potentially its performance. In the strong horizontal case, the individual instances are unrelated and do not share data. This is the typical case for e.g. Cloud Computing, where an increase in service access leads to its replication, and thus to its (horizontal) scale out.

Vertical scalability, on the other hand, relates to the case where a single application exploits multiple resources for directly increasing its own performance as a whole. In other words, where the application is split up into coupled processes or segments that can spread out across the infrastructure. Typical ("coupled") HPC jobs scale out vertically by employing multiple compute units for its calculation, e.g. by segmenting the data into a structured grid, and assigning each cell to an individual unit.

Note that accordingly, P2P computing takes an intermediary between horizontal and vertical scalability.

Requirements from horizontal scalability

- a number of mostly independent, complete compute units (PCs) that reflect the number of instances to be scaled out
- typically high bandwidth between these units during scale out for data and code replication
- the Operating System or execution environment must provide a "platform" in which to upload and execute the application instance (instantiation)
- typically a high degree of concurrent access must be handled
- coherency of data is generally secondary, but still needs to be achieved at least eventually – ideally it is maintained "on the spot"

Requirements from vertical scalability

- applications must be able to scale
- typically low latency
- similar requirements than for tightly coupled HPC
- coherency is essential
- distributed, synchronised execution
- on-the-fly instantiation and execution of code segments
- data distribution without delay
- programming model must allow easy segmentation
- operating system must reduce overhead

k) Off-loading

Whether the application can delegate some part of the computation to a remote server infrastructure (e.g. the cloud). This may require code mobility, as the application may not be present on the server – respectively at least a platform like environment in which to host the application's logic.

Also, the byte code from the source host may be incompatible with the remote host, and require run-time adaptation or the creation of a virtual environment. This may also impose additional overhead to clients as they have to provide information about the desired environment.

Current applications which exploit off-loading capabilities are designed to a particular system and little code mobility or adaptation is required. In practice, most off-load methods work by delegating part of the application tasks to a well known remote system. Applications interacting with Clouds are a good example, but there are others such as real-time gaming with remote rendering, or even mail clients. Because most Cloud systems are mostly closed or have incompatible interfaces, off-loading is at a very incipient age. Still, with the appearance of many small devices with low resources, proxy computation (an off-loading mechanism) becomes more interesting.

Considering systems with lesser complexity, off-loading as been used for a long time as part of the standard data centre model: application servers would off-load database queries to clustered systems, or SSL cyphering and deciphering to dedicated hosts (usually with dedicated acceleration hardware).

From the operating system perspective, the challenge resides in identifying off-loadable segments of code, identifying other systems capable of correctly executing the code, and split the execution environment between all participating systems.

Requirements

- the offloading process itself typically requires large bandwidth but has no latency requirements (this obviously changes with the execution of the remote instance)
- the hosting (source) environment needs to provide some form of sand boxing mechanism for hosting the code / logic / data
- in order to maintain availability and communication stability, messages must be dynamically routed without affecting the code (behaviour)
- the new host must meet the communication requirements (synchronisation, coherency, latency, bandwidth) of the application (or provide a means to compensate it accordingly)

L) MOBILITY

If the application is mobile, the location of some of the nodes involved in the computation can change during execution. The most classical example for such behaviour is obviously mobile phones.

Mobile applications are aware of the location of the hosts where they execute. Location can be expressed both in terms of physical location (typically using the GPS system), or topological location (using the IP address). Most client applications of web and cloud services running at the mobile nodes (such as PDAs and Notebooks) can be considered as mobile. Mobile applications can adapt their execution to the current location. Examples of this adaptation can result in changing the protocols used (e.g Skype and other VoIP applications), choosing peers located closer (e.g. cloud clients), or adapting the interface or other execution parameter (e.g. CoreLocation in OSX 10.6). Moreover, some even may exploit locality and coordinate execution with neighbour peers (e.g. BitTorrent clients). Some operating systems such as Apple OSX 10.6 have support for location adaptation by allowing dynamic configuration of internal parameters (proxy, ip address, security parameters, etc...) in a reactive manner.

Mobility can also be classified as slow or fast, depending on the time the transient mobility process lasts. VoIP and Real Time Interactive applications are very sensitive to delays and packet loss and require fast mobility. Other applications such as mail clients or IM clients can tolerate disconnection time of several seconds .

Requirements

- connectivity should at no time be interrupted for longer than the messaging requirements of the application specify (e.g. latency and real-time constraints)
    - this is particularly true, if the endpoint is part of a distributed & parallel process
- handover to the new location must be seamless without interrupting or affecting the process' behaviour
    - dynamic routing
    - maintain streams
    - maintain quality

- mobility of a client could imply that some information of the server “follows”, i.e. is passed on into the vicinity of the endpoint to maintain availability
- support for fast mobile aware network protocols (Mobile IP family of protocols)
- support for local discovery protocols such as UPNP, Bonjour or M-DNS.
- low latency of the networking stack (to allow fast mobility)
  - low latency process scheduling
  - event driven, asynchronous interface to applications.
- provide applications with low level information about current environment (e.g. packet loss, MAC access delay)

m) Specialized hardware

Some kind of applications work best if a specific type of hardware is employed (typically a type of “application accelerator”). Without restriction of generality, we assume here that any other resource than a general purpose processing unit (GPPU) must be considered a “specialised hardware” - even though graphics processing units (GPUs) have basically already reached “general purpose” status.

Requirements depend very much on the use case (application), as the point behind application accelerators (or specialised hardware) is exactly to support specific tasks. The best example consists in the FPGA chipset which can be dynamically reconfigured to meet the specific application's behavioural requirements.

3. Conclusions

In this deliverable, we discussed the future trends in computing systems in the four principal areas related to the SooS project: Hardware and Processors (in Chapter II.), Communication (in Chapter III.), Programming models (in Chapter IV.)and Operating Systems (in Chapter V.). We also discussed the requirements of future applications, with a specific focus on High Performance Computing (Section VI. 1.), Personal Embedded Devices (Section VI. 2.), Social Networks (Section VI. 3.), Office and Home usage (Section VI. 4.), Multimedia Stream-Processing (Section VI. 5.), and video games (Section VI. 6.).

The main goal of this activity consisted in assessing the development that the infrastructure is most likely to take within the next few years and which kind of support will be needed through a middleware / operating system. We thereby particularly examine the aspects related to a “holistic” environment that not only covers a wide range of application domains, but also addresses the different developments in different areas, stacks or tiers forming a computing system. It is as yet not clear, whether such full convergence can be achieved and is as yet one of the major problems of both operating system and system developers that leads not only to interoperability and portability issues but also to instability and failures, as well as performance loss. The relationship between application requirements and the supporting technology is indeed very complex, and this work can be seen as a tentative systematization of the large set of issues that are involved. Our work certainly does not pretend to be exhaustive: however, we believe it is important to propose a global perspective on the different issues involved and on their interworking. For this reason, in this deliverable we tried to widen the perspective as much as possible, rather than to focus on a small set of specific issues or problems. At the same time, we tried to highlight the important details, the ones that we believe will be the key issues in the development of next generation computing platforms.

Therefore, in this chapter we proposed a possible taxonomy of application requirements and discussed its impact on the technology development trends. The results of this work will thus not only serve as a basis for evaluating the project outcomes through defining a set of testsuites (WP6), but also as driving input for the technical work in Workpackage 2-4. It must be noted in this context that this work already incorporates input from these work packages as part of the state of the art assessments in the respective areas.

Finally, we believe that such an assessment is relevant for any attempt to support future computing systems, in particular due to the increasing amount of processing units in a single system.

References

[1] 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.

[2] Cucinotta, T. (ed.) (2010). Service-oriented Operating Systems: State of the Art. *In production.*

[3] Asanovic, K., Bodik, R., Demmel, J., Keaveny, T., Keutzer, K., Kubiatowicz, J., Morgan, N., Patterson, D., Sen, K., Wawrzynek, J., Wessel, D., Yelick, K. (2009). A View of the Parallel Computing Landscape, In *Communications of the ACM*, Vol. 52, No. 10, pages 56-67

[4] Amdahl, G. (1967). Validity of the single processor approach to achieving large scale computing capabilities, In *AFIPS spring joint computer conference*. Available at http://www-inst.eecs.berkeley.edu/~n252/paper/Amdahl.pdf - accessed March '10

[5] ITU ICT Statistics, http://www.itu.int/ITU-D/ict/statistics/, as in June 2010

[6] Mishra, A., Shin, M., and Arbaugh, W. 2003. An empirical analysis of the IEEE 802.11 MAC layer handoff process. SIGCOMM Comput. Commun. Rev. 33, 2 (Apr. 2003), 93-102.

[7] 3GPP, http://www.3gpp.org/, as in June 2010

[8] Nokia E72 Product page, http://europe.nokia.com/find-products/devices/nokia-e72, as in June 2010

[9] SYMBIAN EKA2: http://developer.symbian.org/main/documentation/reference/s%5E3/doc_source/guide/KernelandHardwareServices/kernelarch/index.html, as in June 2010

[10] Facebook Now Has 30,000 Servers http://www.datacenterknowledge.com/archives/2009/10/13/facebook-now-has-30000-servers/, as in June 2010

[11] Needle in a haystack: efficient storage of billions of photos, http://www.facebook.com/note.php?note_id=76191543919, as in June 2010

[12] Facebook Statistics, http://www.facebook.com/press/info.php?statistics, as in June 2010

[13] Facebook boasts 11 times more page views than MySpace, 59 times more than Twitter, http://royal.pingdom.com/2010/01/05/facebook-twitter-myspace-page-views/, as in June 2010

[14] http://socialmediastatistics.wikidot.com as of June 2010

[15] Entertainment Software Association (ESA) (2010). Essential facts about the computer and video game industry. http://www.theesa.com/facts/pdfs/ESA_EF_2009.pdf [redacted]

[16] Entertainment Software Association (ESA) (2010). Annual report, fiscal year 2009. http://www.theesa.com/about/ESA_2009_AR.pdf

[17] Nielsen Games (2009). Video Gamers in Europe – 2008. Prepared for the Interactive Software Federation of Europe (ISFE). http://www.isfe-eu.org/tzr/scripts/downloader2.php?filename=T003/F0013/8c/79/w7ol0v3qaghqd4ale6vlpnent&mime=application/pdf&originalname=ISFE_Consumer_Research_2008_Report_final.pdf

[18] Interactive Software Federation of Europe (ISFE) (2008). Key Facts – The Profile of the European Videogamer 2007. http://www.isfe-eu.org/tzr/scripts/downloader2.php?filename=T003/F0013/44/c5/w7ol0v3qaghqd4ecogdcb081y&mime=application/pdf&originalname=Key_Facts_English_final.pdf

[19] Samuel K. Moore, “Multicore is bad news for Super Computers”, IEEE Spectrum, November 2008, http://spectrum.ieee.org/computing/hardware/multicore-is-bad-news-for-supercomputers

[20] Sweeney, Tim. (1999) Unreal Networking Architecture. Epic Games, Inc. http://unreal.epicgames.com/Network.htm

[21] Claypool, Mark and Claypool, Kajal. (2008) On Latency and Player Actions in Online Games.

[22] Svoboda, P., Karner, W. and Rupp, M. (2007) Traffic Analysis and Modeling for World of Warcraft.

[23] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch and Simon Winwood, “seL4: Formal verification of an OS kernel”, in Proceedings of the 22nd ACM Symposium on Operating Systems Principles, Big Sky, MT, USA, October, 2009

[24] Gerwin Klein, “Correct OS kernel? Proof? Done!”, USENIX ;login:, 34(6), 28–34, (December, 2009)

[25] Gernot Heiser and Ben Leslie, “The OKL4 Microvisor: Convergence point of microkernels and hypervisors”, Proceedings of the 1st Asia-Pacific Workshop on Systems, New Delhi, India, August, 2010.

[26] Udo Steinberg, Bernhard Kauer, “NOVA: A Microhypervisor-Based Secure Virtualization Architecture”, Proceedings of EuroSys 2010, Paris, France, April 2010

[27] Udo Steinberg, Bernhard Kauer, “Towards a Scalable Multiprocessor User-level Environment”, IIDS'10: Workshop on Isolation and Integration for Dependable Systems (Eurosys 2010 affiliated workshop) , April 2010

[28] Andreas Merkel, Jan Stoess, and Frank Bellosa , “Resource-conscious Scheduling for Energy Efficiency on Multicore Processors”, Proceedings of the 5th ACM SIGOPS EuroSys Conference (EurosSys'10), Paris, France, April 2010

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

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

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

[32] ARM Ltd. “ARM11 Mpcore family”
http://www.arm.com/products/processors/classic/arm11/arm11-mpcore.php

[33] Freescale semiconductors, “P4 Series, P4080 multicore processor”,
http://cache.freescale.com/files/netcomm/doc/fact_sheet/QorIQ_P4080.pdf

[34] Miller, R. B. (1968). Response time in man-computer conversational transactions. Proc. AFIPS Fall Joint Computer Conference Vol. 33, 267-277.

[35] Dennis, A. R., and N.J. Taylor, "Information foraging on the web: The effects of 'acceptable' Internet delays on multi-page information search behavior.". Decision Support Systems 42 (2006) 810-824.

[36] Claypool, Mark, LaPoint, David and Winslow, Josh. (2003) Network Analysis of Counter-strike and Starcraft.

[37] Sweeney, Tim. (1999) Unreal Networking Architecture. Epic Games, Inc.
http://unreal.epicgames.com/Network.htm

[38] Ho, R.; Mai, K.W.; Horowitz, M.A.; , "The future of wires," Proceedings of the IEEE , vol.89, no.4, pp.490-504, Apr 2001

[39] Jayasimha, D. N. Zafar, Bilal. Hoskote, Yatin (2006). On-Chip Interconnection Networks: Why They are Different and How to Compare Them, Intel Corporation, Platform Architecture Research

[40] William J. Dally and Brian Towles, "Route Packets, Not Wires: On-Chip Interconnection Networks", in Proc. ACM IEEE Design Automation Conference (DAC), pp 684-689, 2001

[41] Ray Kurzweil, THE SINGULARITY IS NEAR: When Humans Transcend Biology, Viking Press, 2005

[42] 3GPP – IMS, http://www.3gpp.org/article/ims, as in June 2010

[43] Do van Thanh, Paal Engelstad, Tore Jonvik, Do van Thuan, Ivar Jorstad, "Towards a Uniform IMS Client on Heterogeneous Devices," wimob, pp.196-201, 2008 IEEE International Conference on Wireless & Mobile Computing, Networking & Communication, 2008

[44] Xbox Live Arcade http://www.xbox.com/en-US/, as in June 2010

[45] DVB - Digital Video Broadcasting - DVB-T, http://www.dvb.org/technology/dvbt2/, as in June 2010

[46] Frequenzversteigerung 2010, http://www2.bundesnetzagentur.de/frequenzversteigerung2010/, as in June 2010

[47] Amazon Elastic Compute Cloud (Amazon EC2) http://aws.amazon.com/ec2/, as in June 2010

[48] DragonFly BSD http://www.dragonflybsd.org/, as in June 2010

[49] OpenSSI, http://openssi.org/cgi-bin/view?page=openssi.html, as in June 2010

[50] LinuxPMI, http://linuxpmi.org, as in June 2010

[51] Ada Gavrilovska, Sanjay Kumar, Himanshu Raj and Karsten Schwan, "High-Performance Hypervisor Architectures: Virtualization in HPC Systems",1st Workshop on System-level Virtualization for High Performance Computing, Portugal, 2007

[52] Fang Yu, Yanlei Diao, Randy Katz, T. V. Lakshman. "Fast Packet Pattern Matching Algorithms", Algorithms for Next Generation Network Architecture, edited by Graham Cormode and Marina Thottan. Springer Computer Communications and Networks Series, October 2009.

[53] R. A. Gupta and M.-Y. Chow, "Networked Control Systems: Overview and Research Trends," IEEE Transactions on Industrial Electronics, March 2010

[54] Generalized Multi-Protocol Label Switching (GMPLS) Architecture. IETF RFC 3945

[55] Encapsulation Methods for Transport of InfiniBand over MPLS Networks, draft-puri-pwe3-ib-encap-01.txt, IETF Draft, March 2009

[56] Bolotin, E.; Guz, Z.; Cidon, I.; Ginosar, R.; Kolodny, A.; , "The Power of Priority: NoC Based Distributed Cache Coherency," Networks-on-Chip, 2007. NOCS 2007. First International Symposium on , vol., no., pp.117-126, 7-9 May 2007

[57] P. Bhojwani, R. Mahapatra, J. K. Eun, and T. Chen, "A heuristic for peak power constrained design of network- on-chip (NoC) based multimode systems", Proc. IEEE International Conference on VLSI Design, pp. 124-129, 2005.

[58] IP Mobility Support for IPv4, IETF RFC4721

[59] Proxy Mobile IPv6, IETF RFC5213 (Proposed Standard)

[60] Ion Stoica, Daniel Adkins, Shelley Zhuang, Scott Shenker, Sonesh Surana, "Internet Indirection Infrastructure," Proceedings of ACM SIGCOMM, August, 2002

[61] SIP: Session Initiation Protocol, IETF RFC3261

[62] Host Identity Protocol, IETF RFC 5201

[63] Renjish Kumar, K.R.; Rokkas, T.; Varoutas, D.; Kind, M.; Von Hugo, D.; Harno, J.; Smura, T.; Heikkinen, M.; , "Fixed-Mobile Convergence: An Integrated Operator Case Study," Telecommunication Techno-Economics, 2007. CTTE 2007. 6th Conference on , vol., no., pp.1-6, 14-15 June 2007

[64] Call My Cell: Wireless Substitution in the United States, http://www.nielsenmobile.com/documents/WirelessSubstitution.pdf, Nielsen, 2008

[65] Bluetooth Core Specification 3.0 + HS, http://bluetooth.com/SiteCollectionDocuments/HS_Doc_Web.pdf

[66] IEEE Standard for Information technology--Telecommunications and information exchange between systems--Local and metropolitan area networks-- Specific requirements Part 15.4: Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications for Low Rate Wireless Personal Area Networks (LR-WPANs)

[67] Asymmetric Digital Subscriber Line, ITU G.992.5

[68] 3GPP - LTE, http://www.3gpp.org/LTE, as in June 2010

[69] IEEE Standard 802.16-2004, http://standards.ieee.org/getieee802/download/802.16-2004.pdf

[70] 3GPP - UMTS, http://www.3gpp.org/article/umts, as in June 2010

[71] IEEE Standard 802.3, http://www.ieee802.org/3/, as in June 2010

[72] IEEE 802.11, The Working Group Setting the Standards for Wireless LANs, http://www.ieee802.org/11/, as in June 2010

[73] CATV Cyberlab, http://www.catv.org/frame/cmt_standards.html, as in June 2010

[74] Subramanian, A.P.; Lundgren, H.; Salonidis, T.; Towsley, D.; , "Topology control protocol using sectorized antennas in dense 802.11 wireless networks," Network Protocols, 2009. ICNP 2009. 17th IEEE International Conference on , vol., no., pp.1-10, 13-16 Oct. 2009

[75] Jain, V.; Gupta, A.; Agrawal, D.P.; , "On-Demand Medium Access in Multihop Wireless Networks with Multiple Beam Smart Antennas," Parallel and Distributed Systems, IEEE Transactions on , vol.19, no.4, pp.489-502, April 2008

[76] IEEE 802.11n-2009—Amendment 5: Enhancements for Higher Throughput. IEEE-SA. 29 October 2009

[77] IEEE 802.11 TG AD - Very High Throughput in 60Ghz, http://www.ieee802.org/11/Reports/tgad_update.htm, as in June 2010

[78] Gomez, C.; Catalan, M.; Viamonte, D.; Paradells, J.; Calveras, A.; , "Internet traffic analysis and optimization over a precommercial live UMTS network," Vehicular Technology Conference, 2005. VTC 2005-Spring. 2005 IEEE 61st , vol.5, no., pp. 2879- 2884 Vol. 5, 30 May-1 June 2005

[79] 3GPP - LTE, http://www.3gpp.org/article/lte, as in June 2010

[80] LTE Benefits V3.3, https://www.lte.vzw.com/Portals/95/docs/LTE%20Benefits%20Guide.pdf, LTE Product Design, May 14, 2009

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

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

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

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

[85] L. Schubert, K. Jeffery, B. Neidecker-Lutz, and others, "The Future of Cloud Computing," Cordis (Online), 2010.

[86] Wentzlaff, David and Agarwal, Anant, “Factored operating systems (fos): the case for a scalable operating system for multicores”, SIGOPS Oper. Syst. Rev., 43, 2, 2009, pp. 76-85

[87] http://blog.hubspot.com/blog/tabid/6307/bid/4514/Making-Friends-LinkedIn-vs-Facebook-vs-Twitter-cartoon.aspx

[88] Lynn Reedy, http://academicearth.org/lectures/ebay-volume-statistics

[89] B.L. Jones, "Do Newer Processors Equate to Slower Applications?," DevX online publication, 2010. Available at: http://www.devx.com/enterprise/Article/34588/1954

[90] Intel Corporation,"An Introduction to the Intel QuickPath Interconnect," Intel Whitepaper. Available at: http://www.intel.com/technology/quickpath/introduction.pdf

[91] Proceedings of the 2009 Symposium on Application Accelerators in High Performance Computing (SAAHPC'09), Urbana, USA, July 2009.

[92] Convey Computer Corporation, "Hybrid-Core Computing: Extending a Commodity Instruction Set with Application-Specific Logic, " In: *Proceedings of the 2010 Symposium on Application Accelerators in High Performance Computing (SAAHPC'10)*, Knoxville, USA, July 2010 *to be published*.

[93] W.P. Petersen, "Playing with parallelism on PS3s". Seminar for Applied Mathematics, 2007. Available at http://cluster.qubits.ch/files/u2/20070328_PS3_Talk_LBL_WPP.pdf

[94] 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. DOI= http://dx.doi.org/10.1177/1094342009106195

[95] S. Tagaya, M. Nishida, T. Hagiwara, T. Yanagawa, Y. Yokoya, H. Takahara, J. Stadler, M. Galle and W. Bez, "The NEC SX-8 Vector Supercomputer System," *High Performance Computing on Vector Systems*, Part 1, 2006, pp. 3-24

[96] K. Krewell, "Cell Moves Into the Limelight," *Microprocessor Report*, 2005

[97] M. Gschwind, H.P. Hofstee, B. Flachs, M. Hopkins, Y. Watanabe, T. Yamazaki, "Synergistic Processing in Cell's Multicore Architecture," *IEEE Micro*, Vol 26 (2), 2006, pp. 10-24. DOI= http://dx.doi.org/10.1109/MM.2006.41

[98] P. Colella, "Defining Software Requirements for Scientific Computing," DARPA HPCS presentation, 2004.

[99] S. Seow, "Designing and Engineering Time: The Psychology of Time Perception in Software," Addison-Wesley, Amsterdam: 2008

[100] P. Zimbardo, "The Secret Powers of Time," Presentation at the Royal Society for the encouragement of Arts, Manufactures and Commerce (RSA), 2010. Available at: http://www.thersa.org/events/vision/vision-videos/philip-zimbardo-the-secret-powers-of-time

[101] D.F. Bacon, S.L. Graham and O.J. Sharp, "Compiler transformations for high-performance computing," ACM Comput. Surv., Vol 26(4), 1994), pp. 345-420. DOI= http://doi.acm.org/10.1145/197405.197406

[102] S.P. Midkiff and D.A. Padua, "Issues in the Compile-Time Optimization of Parallel Programs". Online publication, 1990. Available at: http://www.csrd.uiuc.edu/reports/911.ps.gz

[103] Herlihy, M. and Moss, J. E. “Transactional memory: architectural support for lock-free data structures.” *SIGARCH Comput. Archit. News* 21, 2 (May. 1993), pp. 289-300

[104] Li, Z., Wang, C., and Xu, R. “Computation offloading to save energy on handheld devices: a partition scheme.” In Proceedings of the 2001 international Conference on Compilers, Architecture, and Synthesis For Embedded Systems (Atlanta, Georgia, USA, November 16 - 17, 2001). CASES '01. ACM, New York, NY, 238-246.

[105] Karthik Kumar, Yung-Hsiang Lu, “Cloud Computing for Mobile Users: Can Offloading Computation Save Energy?,” Computer, vol. 43, no. 4, pp. 51-56, Apr. 2010, doi:10.1109/MC.2010.98

[106] A. Jha and D. Yee, “Increasing Memory Throughput With Intel® Streaming SIMD Extensions 4 (Intel® SSE4) Streaming Load - Intel® Software Network,” Nov. 2008.

[107] Fritts and W. Wolf, “MultiLevel Cache Hierarchy Evaluation for Programmable Media,” IEEE Workshop on Signal Processing Systems, IEEE, 2000, pp. 228-237

[108] Miller, R. B. (1968). Response time in man-computer conversational transactions. Proc. AFIPS Fall Joint Computer Conference Vol. 33, 267-277. 100 ms

[109] L. Hennessy and D.A. Patterson, “Computer Architecture: A Quantitative Approach,” Morgan Kaufmann, 2006.

[110] A. Bader, J. Chhugani, P. Dubey, S. Junkins, S. Lyalin, T. Morrison, D. Ragozin, A. Ryzhova, S. Sidorov, A. Soupikov, and M. Smelyanskiy, “Game Physics Performance on the Larrabee Architecture,” Intel Corporation 2008.

[111] G.H. Loh, “3D-Stacked Memory Architectures for Multi-core Processors,” Proceedings of the 35th Annual International Symposium on Computer Architecture, IEEE Computer Society, 2008, pp. 453-464.

[112] D.H. Woo, N.H. Seong, D.L. Lewis, and H.S. Lee, “An Optimized 3D-Stacked Memory Architecture by Exploiting Excessive, High-Density TSV Bandwidth,” Proceedings of the 16th International Symposium on High-Performance Computer Architecture, Bangalore, India: 2010, pp. 429-440.

[113] J. Huang, “Understanding Gigabit Ethernet Performance on Sun Fire™ Systems,” Sun Blueprints, 2003. Available at: http://www.sun.com/blueprints/0203/817-1657.pdf