Scalable Computing: Practice and Experience
Volume 12, Number 2, pp. 179–191. http://www.scpe.org
ISSN 1895-1767

BEYOND CLOUDS – TOWARDS REAL UTILITY COMPUTING*

MATTHIAS ASSEL, LUTZ SCHUBERT, DANIEL RUBIO BONILLA AND STEFAN WESNER†

Abstract. With the growing amount of computational resources available, not only locally (e.g. multicore processors), but also across the Internet, utility computing (aka Clouds and Grids) becomes more and more interesting as a means to outsource applications and services, respectively. So far, these systems still act like external resources / devices that have to be explicitly selected, integrated, accessed and so forth. In this paper, we present our conceptual approaches of dealing with increased capacity, scale and heterogeneity of future systems by integrating and using remote resources and services through a kind of web-based "fabric".

Key words: distributed systems, resource fabric, utility computing, service-oriented operating system, distributed application execution, management of scale, multicore architectures

1. Introduction. Over the last few years distributed computing has gained in relevance, not only as a concept, but – more importantly – as a commercial reality: through "Clouds" and multicore processors which make concurrent compute units available for the average user [1, 2]. Notably, usage of these two environments differs significantly. Cloud systems, on the one hand, are essentially server-like machines available over the Internet and accessed accordingly (via e.g. remote desktop or SSH). On the other hand, the capabilities of multicore machines are basically locally available – implicitly there are no specific means needed to access and use such resources.

What is more, cloud systems typically deal with scaling a specific service or application multiple times according to access and availability needs, i.e. perform scale by replication. As opposed to this, multicore systems, just like high performance computers deal with scaling a single application "vertically" across the resources in the form of instantiating multiple processes that together form the application logic, as opposed to individually [3]. It should be noted in this context though that desktop systems and high performance computing (HPC) systems differ in this respect: whilst the latter typically only deal with execution of one single process or thread per core at a time, desktop usage typically implies concurrent execution of multiple services and applications in a time-sharing manner.

In both the cloud and the multicore case we talk of "distributed systems" in the sense that the system on which services, applications or even single processes are being executed consists of multiple resources (i.e., compute node vs. compute cores) connected via a communication and / or messaging link (i.e., interconnect vs. buses). From this perspective, we can specifically note that clouds and multicore systems provide (very) similar capabilities on different environments. Section 2 will elaborate how a common "denominator" of such a distributed system could look like.

By exploiting the commonalities, rather than focusing on the differences, it would in particular be possible to exploit remote resources as if local, thus truly realising the Grid's original concept [4, 5]. For example, an application could be replicated beyond the restriction of the local system, and new applications executed remotely without interfering with any local executions, thus competing over resources. In addition, even simple laptop systems would be enabled to execute demanding applications in the same fashion as on a home or office PC [6]. Whilst this is not a new idea as such (see [4]), its realisation has many impacts not only on a middleware, but more importantly on operating system and programming model. Sections 3 and 4 of this paper will highlight how such a "resource fabric" [20, 21] could be realised and to which specific technological issues it could be applied.

Merging the different paradigms of "distributed systems" would enable new types of systems and implicitly new types of applications that allow following the worldwide trend of internet integration, dynamic outsourcing etc. in short, the Future Internet. Section 5 will describe two types of application areas in more detail. As will be shown, it is not yet possible to overcome all technical issues easily, in particular since the "natural" technological development (i.e., the industrial / commercial provisioning) follows the laws of stepwise development, rather than disruptive (r)evolution. We will conclude this paper with an analysis of the main outstanding issues in section 6.

*This work is partially supported by the S(o)OS project under FP7 grant agreement no. 248465 (http://www.soos-project.eu).
†HLRS – University of Stuttgart, Dpt. of Intelligent Service Infrastructures and Dpt. of Applications & Visualisation, Nobelstr. 19, D-70569 Stuttgart, Germany({assel,schubert,rubio,wesner}@hlrs.de).

2. A "New" Von Neumann Architecture. "Distributed systems" in their widest sense , i.e. including multicore processors, clusters, clouds and grids all have in common that they integrate compute units over a communication link. The main difference thereby being the specifics of the linkage: whilst communication between cores in a multicore system has a very low latency, cloud and grid systems generally use the intra- / internet for communication, meaning high latency but considerably large bandwidth. Finally, HPC systems integrate different levels of linkage, ranging from multicore interconnects to fast, broadband networks (100GB Ethernet, Infiniband).

In principal, latency can be compensated by bandwidth, i.e. if the delay is l and the bandwidth b, the effective data d_e communicated in a set of messages over a time-frame t is

$$d_e(t) = t * \frac{b}{l}. \tag{2.1}$$

In other words, latency is anti proportional to bandwidth: if latency is high, bandwidth should be large too, and vice versa. Low latency can compensate a small bandwidth, in order to reach the same effective data throughput. However, an important factor has been ignored in this calculation, namely the numbers of communications within that time-frame. Obviously latency impacts only per individual invocation, respectively message exchange (actually leading to half the throughput per communication side), meaning that the full throughput depends on the number of invocations, which is accordingly high if the latency is small, and low with a large latency.

$$Invocations = \frac{t}{l}. \tag{2.2}$$

For non interactive systems this does not play a major role, the delays in communication are not noticeable as such – however, in applications that directly interact with the user (such as a word processor), any delay occurring between user input and system reaction beyond 0.1 seconds leads to the impression of a "non-reactive" system, and beyond 1 second, it will even disrupt the user's flow of thought [7]. Browsers take a position somewhere between interactive and non-interactive and users show a slightly higher tolerance towards waiting time than in local applications (4 seconds for maintaining the flow of thoughts according to Young and Smith [8]). Implicitly, offering applications via the web typically leaves an impression of them being slow and unresponsive.

Latency is always a source for problems, when task execution is synchronously dependent on communication, i.e. when the querying task is blocked as long as it is waiting for a response. This affects in particular parallelised applications where the individual processes need to synchronise data. The average developer however is not aware of the connection details – not only are they difficult to acquire, but even more difficult to represent and cater for in the program in some form. The general tendency is therefore to make use of asynchronous messaging in the web domain (clouds, grids) where latency and bandwidth is high, and synchronous messaging inside compute clusters, trying to minimise latency. However, applications with little communication needs may nonetheless occupy multiple resource instances for their execution - in particular embarrassingly parallel applications fit and scale well in either environment. By removing the interactive part from the processing tasks, web-based systems can even be exploited for demanding user applications.

To achieve this, we do no longer need to distinguish between different resource types, but between their connectivity. From this perspective, the different notions of distributed systems fall together as a complex system consisting of computational resources connected over a communication link. Modern systems are therefore no longer building up a strict von Neumann architecture, but a modular platform that connects any amount of von Neumann like units. In particular from a programming perspective, we can therefore derive a modified von Neumann architecture as depicted in Figure 2.1, in particular the I/O moves closer to the processing unit (PU) thus allowing for connectivity between multiple PUs without an explicit single central instance. We also distinguish between processing and memory units (MU), which are connected through a dedicated I/O. It must be noted that current multicore systems do not make use of an explicit I/O unit for core-2-core communication – whilst this is expected to change, the according I/O will still most likely differ from the one connecting with the outside of the processor. It must also be mentioned that cache access is not linked to an I/O as such, but to a memory controller – again, long-term developments (e.g. ring over caches) will impact here.

All this however, has no impact on the programming layer. In this view, clusters, multicore processors and cloud or grid based systems are essentially identical. So far, this model is realised through according means

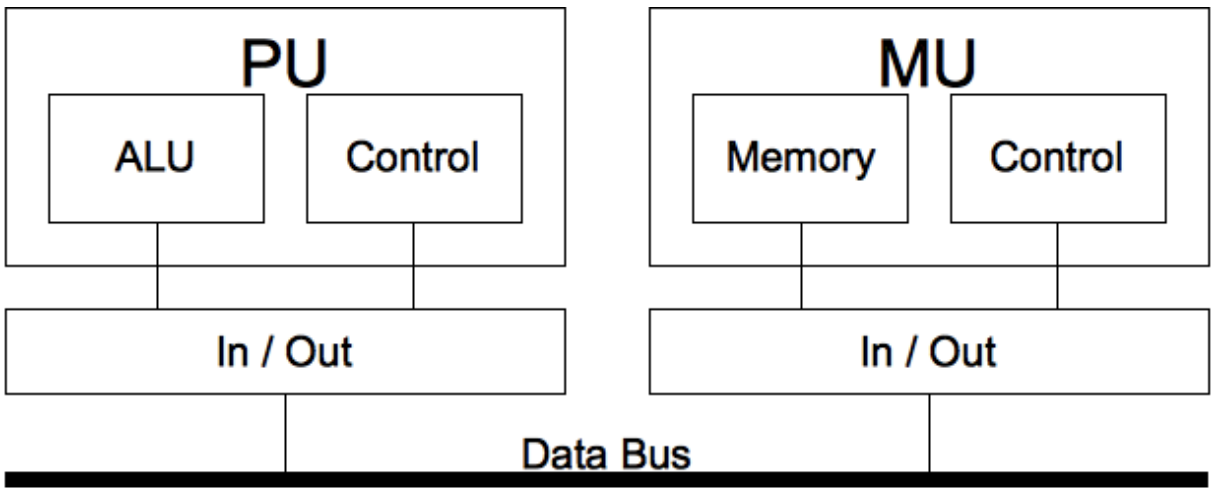

Fig. 2.1: A modified von Neumann architecture where I/O is directly coupled to multiple compute units and memory units, respectively.

on the middleware level (with the highest level of abstraction being represented through distributed workflows, e.g. [22]).

2.1. Data Management in the "new Model". The massive distribution of data over multiple machines (i.e., storage points) as provided by this new architecture fundamentally changes current data management concepts, too, in particular data generation, exchange and storage. As data usage grows faster than bandwidth (and even faster than storage), any form of data movement implicitly consumes a large amount of time and resources, respectively. To compensate for this restriction, new mechanisms, strategies and tools are required that manage the movement of particular data sets (similar to [9]) in segments across a large storage hierarchy. Ideally, data movement is restricted to the minimum amount needed at any specific time and maintained in a way so as to reduce the distance between sender and recipient. Similar to the code (see below), data must therefore be managed in a more intelligent fashion, according to the points of access and the degree of synchronisation, respectively coherency across usage points (see also [10]). As such, data that is frequently used should be moved to highly parallel dynamic storage (e.g. parallel file systems like Oracle's Lustre[1] or virtual distributed file systems like GFS [11] or a combination of both), while archived data should reside in "passive" storage devices (e.g. low-cost storage devices or robotic tape libraries).

To allow for this separation, a fundamental requirement is the ad-hoc allocation, use and release of particular storage space. Hence, algorithms should automatically request necessary space but also track and delete unused data from these dynamic storages, so as to minimise storage costs and increase throughput. This is not only relevant for any storage device but also main memory and caches are affected. Both are extremely useful to achieve fast processing of data sets and thus higher throughput, too. It should be noted here that the latter units are most effective and relevant for multicore processor rather than distributed systems. However, the entire data management hierarchy ranging from caches and main memory to any (external and / or remote) storage device has to be considered and optimised in order to achieve better performance and thus reduce latency. Of particular relevance is also that data source and applications are not necessarily directly coupled. Collections of data sets should be organised as hierarchical directories. Such abstraction will essentially change the way the I/O is expressed by applications and will involve data exchange and storage management in a form that maps data sets into physical devices without affecting the application's behaviour.

Similarly, replication of data represents a key issue to increase data availability through locality. Management of replicas has to carefully deal with lifetime issues to remove outdated pieces immediately, so as to not unnecessarily block storage space. New replicas should be dynamically created, distributed and / or destroyed

[1] http://www.lustre.org/

based on users' and applications' needs but also according to technical requirements [12]. In shared environments, i.e. where multiple access points require the same data, serious consistency issues across nodes have to be addressed [13].

3. Weaving Resource Fabrics. The classical approach towards dealing with distributed systems consists in providing a form of middleware that translates function invocation, instantiation etc. into a set of remote procedure calls or web service calls. The actual details depend not only on the realisation but also on the domain applied to.

In multicore systems, the actual transaction logic is encapsulated and realised via the hardware; supercomputer clusters employ some form of dedicated communication programing model (with either explicit (e.g. MPI[2]) or implicit (e.g. PGAS[3]) messaging) that is translated into ports and sockets at compile time. The grid / cloud support typically builds on HTTP protocols that are typically translated into ports and sockets at runtime. In other words, the actual type of communication bridge is transparent to the user, though he will still have to use it in different ways, depending on use case and environment. Though there are ways of controlling and configuring the communication link, there is little possibility to exploit this dynamically for maintaining distributed code – in other words, parallel programming models that use synchronisation for controlling the execution behaviour base on the assumption that the underlying infrastructure is effectively homogeneous, or at least give this impression to the user. In reality however neither the resource infrastructure, nor the program is effectively homogeneous: even in the strong scaling based HPC domain, the actual algorithm consists of both parallel and sequential segments that interchange during execution.

What is more, e.g. in typical (unstructured) numerical grid algorithms (such as blood-flow simulation etc.), the communication relationship between the parallel processes is not symmetrical, meaning that the code would benefit from a distribution on the infrastructure aligning the connectivity between compute units with the neighbourhood model of the code. The classical means to developing parallel applications consists in either segmenting the work or the data by identifying natural partitions of either of those [14]. Whilst this is the most common approach, it suffers from the drawback that it a) requires good knowledge about the program and the concurrency in work and data, necessitating additional development work; b) the partitioning may be too small or to big to be efficient, in particular if the communication exchange between segments is not considered properly; c) the specific capabilities of a heterogeneous system are mostly unused – what is more, if the according effort is undertaken to adapt code to the specific system, it will become less portable; and finally d) not all kind of segments can be identified this way. Heterogeneity and architecture of the system are typically regarded as obstacles, but program execution can actually benefit from this structure by reflecting the "natural" behaviour of an application.

We can identify the following key aspects of any algorithm:

1. Concurrency – some functions are executed without explicitly sharing data or resources. In this case these segments can be executed in parallel.
2. Parallelism – similar to concurrency, some functions operate on a common data set, but the actions they perform are not directly dependent on one another. This is typical for loops over large grids ("loop unrollment").
3. Interactivity – in particular in desktop applications, the interface towards the user can be easily defined by the connectivity to external input resources.
4. Background Tasks – are tasks waiting for specific events to execute and with little relationship to the main execution (regarding data dependency).

Similarly, the communication needs are not the same between all these parts, though there is no general statement about the relationships possible: concurrency and parallelism do not necessarily imply high connectivity (low latency), as e.g. embarrassingly parallel applications show. On the other hand, events processed in the background may require immediate response, and interactivity does not mean that all processing on the input data has to be executed immediately (cf. word count and spell-checking in modern word editors: even though they react "immediately" to any input, the delay until the actual results are available is of no concern to the user and hence typically not even noticeable). Again, keep in mind that if the result is in some form relevant (in the sense of often checked by the user), the time frame for maintaining a flow of work is 1 second [7].

[2] http://www.mcs.anl.gov/research/projects/mpi/
[3] http://pgas.org/

3.1. The Structure of Applications. The structure of an application can therefore be used as an indicator for its distributability (and, to a degree, parallelisability). The (runtime) behaviour provides additional information about the actual connectivity between the individual segments and thus its requirements towards the communication model, i.e. the relationship of latency versus bandwidth. Implicitly, runtime behaviour effectively provides more information about the potential code and data distribution than the programmer can currently encode in the source code. This is simply due to the fact that this is not in-line with our current way of writing programs and is implicitly not directly supported by programming models. The foundation is however laid out by integration of remote processes (web services) and dedicated synchronisation points in parallel processes – this does not always reflect the best distribution though, as the according invocations are mainly functionality- rather than communication-driven. A way of identifying and exploiting the "behavioural" structure of the application for distribution purposes consists in runtime analysis and annotation of the code and data memory to produce a form of dependency graph (cf. Figure 3.1) which depicts the invocations of memory locations (code) and read / write operations on data.

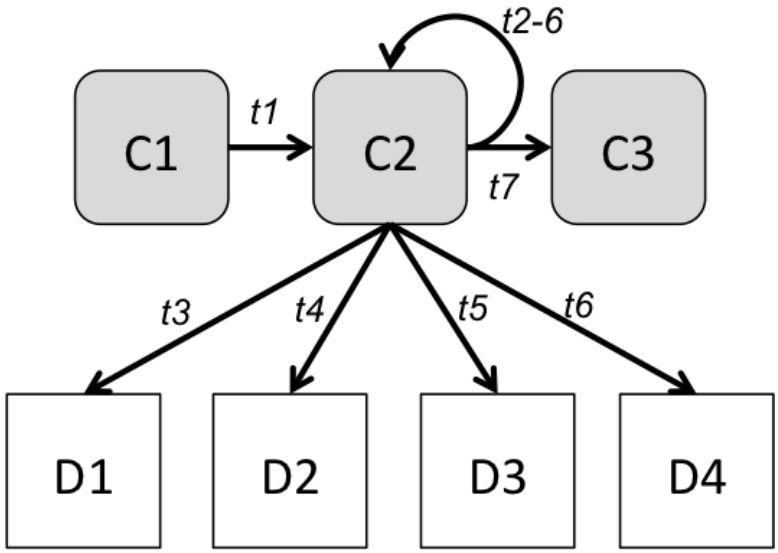

Fig. 3.1: A simple code (C) and data (D) dependency graph of a sequential application with a loop over an array (C2). The graph denotes a sequence of actions with t(x) representing the xth transition, respectively access action.

In order to acquire the according code and data blocks, the basic starting point consists in identifying jumps and non-consecutive data accesses. Figure 3.1 exemplifies how the graph information can be used to identify an unrollable loop (C2) that consecutively accesses unrelated memory blocks to produce a result. In this simplified case there is no data dependency between C1 and C2, or even between C2 and C3, which means that the unrolled C2 blocks do not even have to be synchronised. An example of such a loop would be

C2: for (int i = 0; i < 4; i++) a[i] = 0;

As opposed to this, Figure 3.2 depicts the example of a loop that can not be unrolled due to dependency issues in C2. This loop could look like

*C2: for (int i = 1; i < 3; i++) a[i] = a[i-1] * 2;*

Notably, the examples given are not sensible for parallelisation, as the actual work load per iteration is too small to compensate the overhead for spreading out, communication etc. All analysis so far is based on the simple fact of data relationship and dependency – not unlike the approach pursued in StarSS [15], which takes a directive based approach to task parallelisation on top of C, C++ and Fortran. The runtime takes care of dependency

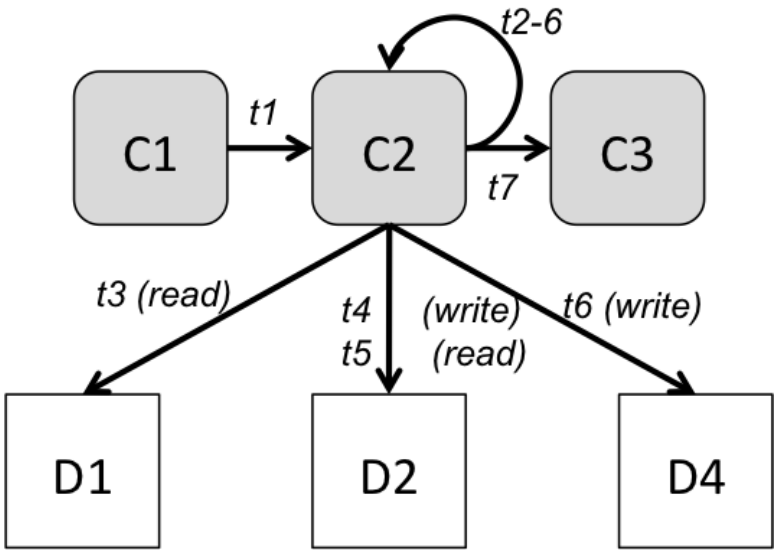

Fig. 3.2: Example of a loop that is not unrollable and its representation in a dependency graph.

tracking, synchronisation, and scheduling. Source code annotations allow the programmer to designate tasks for concurrent execution along with their data dependencies. During runtime, such tasks are placed into a dependency tree and scheduled for execution by a number of working threads a soon as the data dependencies are met.In a nutshell, the StarSs directives 1) designate specific code functions / subroutines to be executed concurrently as so call tasks by the runtime; 2) specify the direction of a function subroutines parameters, i.e. input, output, or inout, which is later used to infer the inter-dependencies of task in a task-dependency graph.

Accordingly, even though a pattern based approach like the one presented above does principally provide information about the distribution and parallelisabilty, it does not per se increase the execution performance. In order to not only identify principle points of distribution and parallelisation, but to also make code execution more efficient, so as to exploit the specific benefits of parallel architectures and infrastructures, further information about the code behaviour is needed – in particular the "strength" of code and data relationships, and the size of the segment. Note that timing can be derived through more detailed sequential information (i.e., in Figure 3.1 or Figure 3.2 by storing actual timestamps in t_n). Strength of relationship is thereby proportional to amount of invocations divided by the full execution time.

With this information, we can derive a graph (Figure 3.3) where the strength of the relationship (e.g. C2 calls C3 less often than C1 calling C2) and the size of the underlying code and data is encoded as weights of vertices and edges. For simplicity reasons we left out timing information in the figure and concentrated on a very simple code structure (without loops or similar).

The dependency information in this graph, in combination with the size information can be used to extract different segments in the form of subgraphs according to nearness (connection strength) and combined size. Or to put it in computational terms again: according to the number of memory accesses, whereby fewer accesses imply a potentially good cutting point.

We can thereby distinguish between different types of data dependencies that relate to the "strength" of coupling, in the sense of the speed required, or rather assumed for executing the according transfer. In other words, the strength represents the implicit cost for execution performance if the according link should be delayed. Obviously, the strongest link is therefore the exchange of values via the register set of the processor, this is followed by memory access and finally any external I/O. It may come as a surprise that the implicit linkage through sequential execution of operations is similar to the normal workflow. This is simply due to the fact that effectively any code can be split and, more importantly, even be executed in parallel, given that this does not affect any values that are processed in the overall algorithms. In other words, if the two segments or operations are concurrent, each partition reflects the code to be executed on one compute unit (core, node etc.) and connections across segments need to be realised through a cross-process communication link. Information

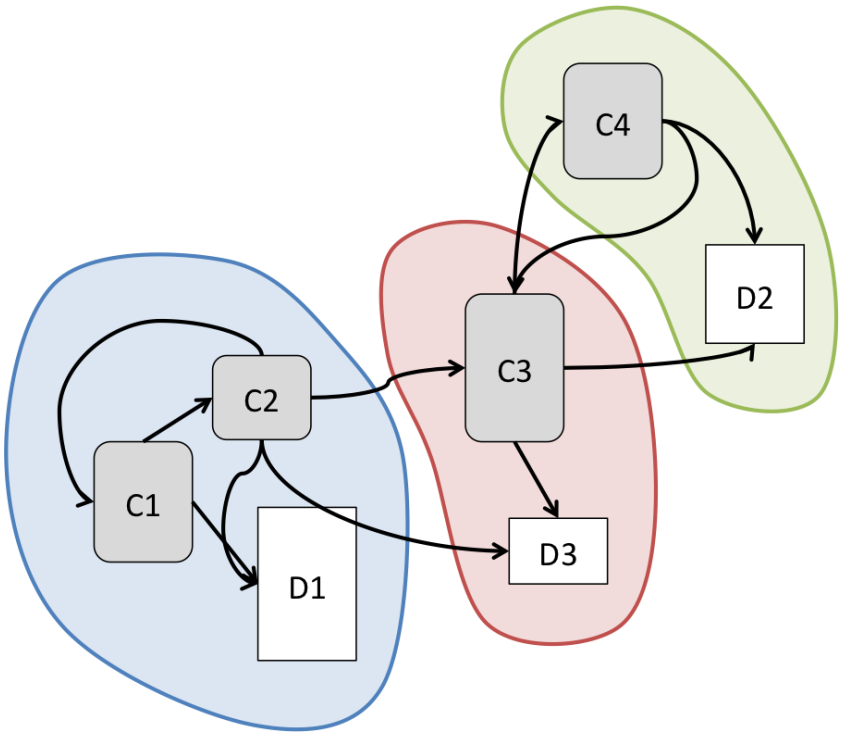

Fig. 3.3: A dependency graph with implicit relationship strength (length of the vector) and size information (size of the block). The three areas depict potential areas of segmentation.

about the bandwidth and latency of the system's specific communication links can thereby serve as an indicator for cutting point identification, if it is respected that access across segments is effectively identical to data passing – accordingly, the temporal sequence and dependency of access constrains the segments' independency and hence concurrency.

We can regard this as a specific case of the maximum flow theorem, where the maximum communication between any two given nodes in a flow network (represented by a graph) with limited capacities (bandwidth) is searched for. In our specific case, the maximum flow of a network denotes the code instances / parts that should least be cut: If we interpret "flow" as the amount of data shared between two instances, than "capacity" is the restriction imposed by the type of data exchange. By segmenting the network we imply a further reduction of capacity at the cutting point – we therefore try to select the points where this impact is minimised, i.e. the minimum cut. The min cut algorithm applied therefore looks for the edges with the minimal capacity and flow [23].

As the goal of this process consists in exploiting concurrency between different code logic segments, distance between nodes (and therefore capacity and flow) are also particularly influenced by the potential degree of execution overlap. Execution overlap between two segments is effectively represented by the distance between the connecting nodes, i.e. the number of operations executed between the two connected access commands. For example, assume that a program writes a variable at the beginning of its execution, but will actually only use this variable after multiple seconds of execution. If no further relationships in between these two commands exists, the program can be split at any place in between and these two segments can be executed in parallel at the same time, as long as the first write command is executed before any further data manipulations take place. We thereby want to maximise the degree of execution overlap to make best use of the resources and thus minimise the total execution time.

The degree of potential overlap (i.e., the maximum degree of overlap between two potential code segments) thus obviously impacts on the weight between two nodes in the dependency graph. This means that the higher the degree of overlap, the less relevant the factor of capacity and flow. In other words, if there is enough time to communicate the according value by any means, the fact that the program assigns a specific strength to

the access plays no role anymore. The segmentation process must therefore respect this when calculating the minimum cut. It must be noted in this context that with the explanation above, we ignore the time for actually communicating the according data, which impacts on the degree of partial overlap, too.

The full algorithm would exceed the scope of this paper and is therefore not elaborated further here. It is obvious that one of the major problems for an efficient segmentation consists in the right data gathering granularity: whilst too fine data blocks will cause memory and algorithm to go over bounds, too coarse information will lead to too strong relationships and too large segments, so that the effective gain through distribution is counterweighted by the overhead for communication and memory swaps.

3.2. Lifecycle of Applications in a Resource Fabric. As noted, the major part of the information about the code is acquired at actual runtime – implicitly, the distribution information may change during execution, leading to potential instabilities and dependencies, respectively. In this section we will examine the full lifecycle of an application executed in such an environment and implicitly the steps involved to better exploit a scalable (and potentially dynamic) environment can be captured:

1. Analysis of the application behaviour ("Analyse"): In an initial step (first time the application is executed), there is little to no dependency information available, unless an according programming extension (such as StarSS) has been applied. This means that initially the application is executed locally in a virtual memory environment that logs the runtime behaviour and hence the dependencies between code partitions and memory segments. Implicitly first time invocation is effectively identical to sequential execution – even though user and or compiler (e.g. using the OpenMP[4] model) provided parallelisation (if any) can still be exploited.
2. Identification of appropriate resources ("Match"): The dependencies in the application graph reflect the communication needs between segments and thus implicitly indicate the required infrastructure architecture, including type and layout of interconnects. But also specific core types can be exploited to a degree – e.g. Figure 3.1 shows clear vectorisable behaviour and other microarchitecture specific patterns can be identified.
3. Distribution and adaptation of code and data ("Distribute"): When appropriate resources could be identified, the code segments can be distributed across the infrastructure accordingly. In the simplest case, all partitions will be uploaded prior to actual execution, in which case no additional data has to be distributed at runtime (besides for the data transported during communication). However, in principle, it is possible to distribute the segments in their order of invocation, though this runs the risk of potential delays.
4. Execution and runtime analysis ("Execute"): Actual execution is principally identical to any distributed program execution with explicit communication between process instances. However, as opposed to an explicitly developed parallel program, the source code in this model has not been altered and the communication points and tasks are unknown to the algorithm itself. Accordingly, the infrastructure has to take care of communicating the right data at the appropriate time. From the development perspective, this is principally identical to the PGAS (Partitioned Global Address Space) approach, which provides a virtual shared (global) memory and deals with the communication necessary to enable remote data access. Effectively the system for enabling distributed execution in the model proposed here must enact the same tasks directly on the (virtual) memory environment of the operating system (OS), rather than on the programming level. During execution, the system may continue analysing behaviour (cf. step 1 "Analyse") in order to further improve the segmentation information and granularity. As program behaviour changes according to code dependency, the main issue in this phase consists in ensuring and maintaining an efficient stability of the distribution. Monitoring and segmentation must therefore not only consider the dependencies, but also higher-level parameters that implicitly define the stability of a given segmentation. Though there is some relationship to SLA based monitoring, these parameters should not be confused with common quality metrics.
5. Information storing ("Store"): Behaviour and distribution information should be stored after execution in order to be retrieved in the next iteration, thus allowing to skip step 1 and thus improving the process.

The main issues in the lifecycle consist obviously data exchange and synchronisation of the segments which are treated as distributed processes. If data and execution are not carefully aligned, inconsistency may lead

[4]http://openmp.org/wp/

to serious crashes, deadlocks, or serious efficiency issues due to overhead. It is therefore not only relevant, from where data is accessed, but also when and in which order. Ideally, data is being transported in the "background" whilst the segment is mostly inactive. We will not elaborate these issues here – suffice to say that if the segments can be treated completely isolated from each other and all memory is swapped during execution passover, coherency and consistency is ensured.

4. A Middleware for Resource Fabrics. In order to realise an environment that enables such distributed execution, it must accordingly provide some capabilities to capture memory access and intervene with code progression so as to pass the execution point at the boundary of the individual segments. Effectively this means that the system provides a virtual environment in which to host and execute the code - a full virtual system however would impact on execution performance. Whilst this may be acceptable for some type of applications where simplicity of development ranks higher than performance, in particular in scalable environments, efficiency typically ranks higher.

However, the system does not necessarily need to provide a full virtual machine, but in particular a virtual memory environment and a set of interfaces to access (shared) resources – in other words, an operating system. Since effectively in all execution parts memory access needs to be controlled, a centralised operating system approach would create a serious non-scalable bottleneck and additional delays due to message creation, synchronisation and communication would turn out a major performance stopper. This relates to the major reasons why monolithic, centralised operating systems show bad horizontal scaling capabilities [16]. In order to achieve better scalability the individual compute units hence need some local support to reduce overhead and enable virtual memory management in a form that can also identify and handle segment passover.

The S(o)OS project[5] funded by the European Commission investigates into new operating system architectures that can deal with exactly this type of scenarios. The approach thereby essentially bases on a concept of distributed microkernel instances that fit into the local memory of a compute unit (processor core) without obstructing it, i.e. leaving enough space for code and data stack. We can essentially distinguish between two types of OS instances in a resource fabric: firstly, the main instance that deals with initiating execution and distribution, as well as scheduling the application as a whole. Secondly, the local instances that effectively only deal with communication and virtual memory management. In effect, all instances could have the same capabilities [16], which would make them larger and more complex to adapt to specific environments though. In S(o)OS the principle is extended with on-the-fly adaptation of local OS instances according to application requirements and resource capabilities – this way, the kernel can even support adaptation to resource specifics without the central instance having to cater for that [17].

In Figure 4.1 we depict the principle of such an operating system with respect to the relationship between cores (or compute units, comes to that): typically one selected unit will take over the initial responsibility for code and data, i.e. loading it from storage, analysing it (respectively retrieving the annotations) and distributing it accordingly. The segmentation information (i.e., the memory structure derived from the dependency graph) will be passed with the code segments, allowing the local instance to build up the local virtual memory and implant system invocations at the appropriate time (during segment passover).

4.1. Distributed Execution. Pure distributed execution without any execution overlap is effectively similar to context switching during time-sharing, only that the "new" state is not uploaded from local memory, but provided over the communication link between the initiating unit and the one taking over. However, parts of the state (code and base data) can principally already be available at the destination site due to predistribution according to the dependency graph (cf. above).

Similar to context switching, the application itself does not have a dedicated point at which to perform the switch (or even enable it), though with additional programming effort context switching can be avoided (thus leading to single-task execution, up to a point). As opposed to multi-tasking operating systems, however, a S(o)OS like system must initiate the context switch (execution passover) at a dedicated point, according to the segmentation analysis (cf. Figure 3.3). This point is effectively the end of the local segment and can thus be initiated by a simple jmp (jump) command into the virtual memory address space of the new segment and can thus simply be appended to the partition. Just like with any access to remote data (be that due to branching or data access), the operating system has to intercept the request prior to its execution, similar to classical virtual memory management. The MMU (Memory Management Unit) of modern processors supports this task

[5] http://www.soos-project.eu/

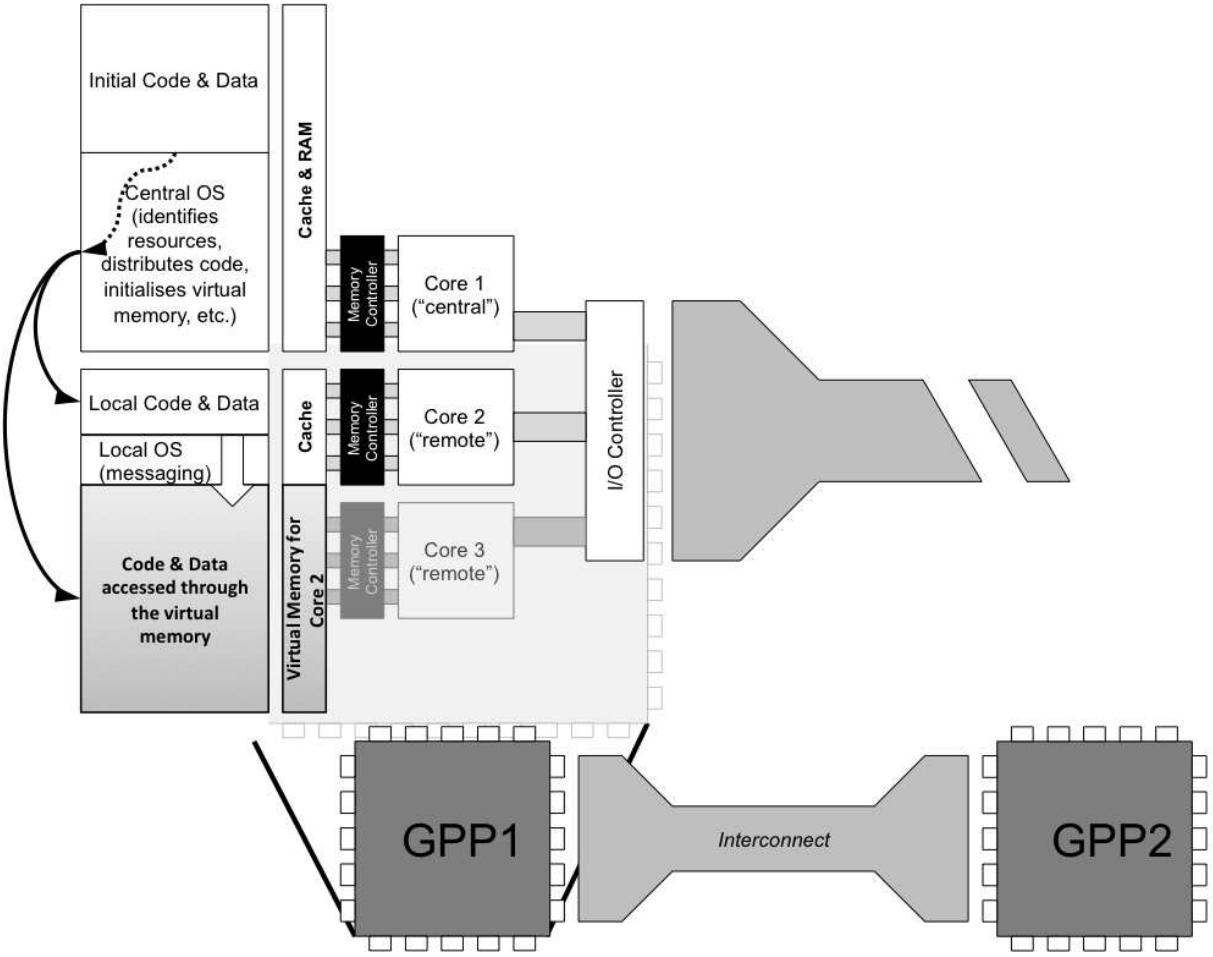

Fig. 4.1: A distributed micro operating system and its relation to the individual compute units in the system. "Remote" cores can be hosted within the same processor or principally any remote machine.

already, but would need to be extended to also cater for the specific needs of a distributed (cross-unit) memory virtualisation.

In other words, as soon as the instruction pointer leaves the area of available, and more importantly, designated code operations, the system needs to be able to judge how to handle the according situation. In the traditional, unhandled fashion, the system would interpret this similar to a cache miss for data access and initiate fetching further instructions from memory. In the distributed case, this should however trigger an execution passover to the remote processor / memory unit. As the point of passover is defined by the segmentation process as detailed above, it seems appropriate to extend the machine code operations with a set of instructions for handling the passover at the cutting point.

Notably, code segments in such a resource mesh are essentially treated as independent, separate processes, that can principally be executed in parallel. The execution restrictions are given by the data dependencies that can implicitly be regarded as execution triggers. From this perspective, execution passover is hence not so much a question of identifying the next appropriate segment, but of validating that all relevant preconditions for execution of the respective segment are met.

In the straight-forward approach we could therefore maintain the data dependency graph as a workflow description of execution, whereas dedicated flags indicate whether an edge is "satisfied". In this case, this means that the according data access operation that forms the precondition for execution of the second segment has been met and that hence the execution of the dependent process can start, once all these preconditions have been satisfied. Programmatically this means that flags per datum / edge are maintained which are constantly

checked by a scheduler which selects the appropriate process to execute on basis of these flags.

Whilst this approach is nice due to its simplicity, it leads to insufficient resource usage – in particular since the partial overlap between execution segments are not exploited. This can be easily demonstrated in the case already introduced above, where an algorithm assigns a variable early in its execution, but only uses the variable comparatively late. With the partial overlap condition in the segmentation process as described above, we can assume without restriction of generality that the write operation will be executed in the middle of the first segment, whilst the read operation will occur in the last third of the second segment, so as to increase resource utilisation. If however, execution of the second segment is delayed until the first segment has reached the point of the according write operation, the core dedicated to the second segment will idle for at least half of the execution time. This would lead to an execution performance that is only minimally better than the sequential case, even though the situation would allow for twice the execution speed.

The obvious alternative consists in executing the processes till the point of data access and stall further processing until the data becomes available. As discussed below, this could be achieved by having the process actively query / access the data source process according to its data dependency specification in the segmentation graph. Whilst this approach obviously performs well for the case presented above, it may nonetheless lead to a similarly weak resource utilisation when the first process writes data quite late and the second tries to read early, i.e. if the execution overlap is small (respectively non-existent). This is obviously the case for segments that are intended to be executed at a later point in the process in parallel with other segments than this first one.

Even though this latter case creates weak resource utilisation, it must nonetheless be noted that the access delay does not create the same execution delay as in the first case. This is due to the fact that the maximum overlap is still exploited in the case of delayed access. However, to exploit this approach implicitly required that a large amount of otherwise unused resource are available. Since we cannot assume that (in particular from an energy-saving perspective), we must instead try to improve the utilisation.

Accordingly, the segments in the dependency graph must be sorted so as to form a workflow that respects two crucial aspects: (1) availability of resources in principle, and (2) maximising the total degree of overlap and thereby reducing the total execution time. The second aspect implicitly prevents, respectively reduces the delay time for data access.

These concepts are implicitly closely related to classical virtualisation concepts, even though in this case, it has to act on a much lower level than typical virtualisation approaches – namely in between hardware and machine code. The main task of virtualisation thereby also does not consist in exposing different hardware capabilities (even though it may be exploited to this end), but more importantly in maintaining an integrated view on a dispersed environment.

4.2. Data Maintenance. Nonetheless, it must still be taken into consideration that the actual transfer of the according data consumes time, too. This specific delay was part of our initial considerations of where and how to perform the code segmentation. In addition, the execution time of a process can only be roughly estimated and it may therefore happen often enough that a dependent process overtakes the data source process and therefore attempts to access data prior to its availability.

In order to reduce the impact from communication and delay, data must be handled intelligently by the system. We can note thereby that the OS instance has principally three options to deal with data access:

1. preemptive distribution: provide the data to all requestors, before they actually try to access it;
2. context switch: pass all status data when passing over the execution point to a remote instance (does not apply to parallel processes);
3. on demand: make data available and accessible at the moment the remote process requests it.

The actual decision will not only depend on time of access (in particular for parallel processes), but also on size of the data and on outstanding tasks of the processing segment. For example, if the data will only be ready by the end of the segment's processing, there is no point in distributing it earlier. However, if the data size is too large for single provisioning without introducing unnecessary delays, partial data updates according to the data segments (cf. above) may be distributed ahead of time (preemptive). Note that current processor architecture does not allow for easy background data transmission and in most cases the communication will stall execution of the main process. As noted, a particular issue in this context consists in ensuring data coherency across segments and avoiding deadlocks. Intel's MESIF protocol over the Quick Path Interconnect [18] is one means to ensure cache coherency in a distributed environment at the cost of access delays. The basic principle of

this approach consists in checking the consistency of a datum at access time by verifying it against all other replicated instances. Obviously this protocol does not scale very well and requires a specific type of architecture that will most likely not be supported in future large scale environments any more [19].

5. Exploiting Resource Fabrics. Since the system primarily caters for distribution of an application across a (potentially large-scale) environment for effectively sequential execution, how would this approach help solve the problem of dealing with future infrastructures? The system offers two major contributions that will be discussed in more detail in this section, namely support for high performance computing but also common web applications.

5.1. Supporting High Performance Computing. Though the primary concern is distribution and not parallelisation, the features and principles provided by a S(o)OS like environment deal with essential issues in large scale high performance computing, namely:

1. exploiting cache to its maximum;
2. providing a scalable operating system;
3. matching the code structure with the infrastructure architecture;
4. managing communication and synchronisation in a virtual distributed shared memory environment.

Whilst the system does not explicitly provide a programming model that deals with scalability over heterogeneous, hierarchical infrastructures, it does support according models by providing additional and enhanced features to deal with such infrastructures. In particular, it implements essential features as pursued e.g. by StarSS and PGAS: the concurrency information extracted from the memory analysis is consistent with the dependency graph that StarSS tries to derive [15] and could be used as an extension along that line. The main principle behind PGAS on the other hand consists in providing a virtual shared memory to the programmer, with the compiler converting the according read / write actions into remote procedure calls, access requests etc.

5.2. Office@World. A slightly different use case supported by the S(o)OS environment consists in the current ongoing trend of resource outsourcing into the web (cf Sec. 1). As has been noted before, an essential part for web exploitation consists in maintaining interactivity whilst offloading demanding tasks. As we have shown, this does not only affect code, but equally data used in an application – accordingly, the features are of particular relevance in a future environment where most code and personal data will be stored in the web. As discussed in detail in [6], the system thus not only allows that code and data becomes accessible and usable from anywhere at any time (given internet connectivity), but also virtually increases local performance of the system. This enables in particular frequent travelers to exploit a local system environment for their applications whilst using data and code from the web – future meeting places could thus offer compute resources that can be exploited by a laptop user, but also the other way round, i.e. a home desktop machine could replicate the full environment of the laptop. A low-power, portable device could thus turn into a highly efficient system by seamlessly integrating into the "Resource Fabric" without requiring specific configuration or development overhead [6]. This is similar to carrying an extended secure full work, respectively private profile with you.

6. Conclusions. We have presented in this paper an approach to dealing with future distributed environments that span across the Internet, multiple resources and multiple cores (i.e., computational units), but also across different types of usage and applications (such as High Performance Computing and Clouds). "Resource Fabrics" are thereby an effective means to integrate compute units non-regarding their location and specifics.

We have shown how a general view on computing resources can serve as basis for such a development, enabling higher scalability. To this end, a more scalable operating system is required that enables the explicit usage of resource fabrics as discussed here. The according OS is still in a highly experimental stage, and more experiments are needed for concrete performance estimations. However, the basic principle has already been implicitly proven by the Cloud Platform as a Service movement, where the dichotomy of interactive and "executive" parts of the application is used to exploit remote resources and services.

Many issues still remain speculative, though, for example regarding the execution of parallel applications where time-critical alignment is crucial. Along this line in particular the concurrent exploitation of remote resources, leading to potential time sharing of individual units (and their resources, such as I/O) still needs to be assessed more critical and deeply. With the general movement towards multicore systems, it can however generally expected that time-sharing during execution time is no longer a valid execution model. It will be noticed however, that we did not even touch upon the issues of security and privacy, which will be a major

concern in scenarios such as the "Office@World" one. So far, we did not touch upon this as other more technological issues are more relevant for the time being – according concepts are expected in the near future.

REFERENCES

[1] M.A. Rappa, *The utility business model and the future of computing services*, IBM Systems Journal, 43,1 (2004), pp. 32–42.

[2] W. Fellows, *The State of Play: Grid, Utility, Cloud* Presentation at CloudScape 2009, OGF Europe. Available at http://old.ogfeurope. eu/uploads/Industry%20Expert%20Group/FELLOWS_Cloudscape Jan09-WF.pdf, 2009.

[3] L. Schubert, K. Jeffery, B. Neidecker-Lutz, and others, *Cloud Computing Expert Working Group Report: The Future of Cloud Computing*, European Commission. Available at: http://cordis.europa.eu/fp7/ict/ssai/docs/cloud-report-final.pdf, 2010.

[4] I. Foster and C. Kesselman, *The Grid. Blueprint for a New Computing Infrastructure*, Morgan Kaufmann Publishers, 1998.

[5] J. Waldo, G. Wyant, A. Wollrath, and S. Kendall, *A Note on Distributed Computing*, Sun Microsystems Technical Report, Available at: http://labs.oracle.com/techrep/1994/smli_tr-94-29.pdf, 1994.

[6] L. Schubert, A. Kipp, B. Koller, and S. Wesner, *Service Oriented Operating Systems: Future Workspaces*, IEEE Wireless Communications, 16 (2009), pp. 42–50.

[7] R.B. Miller, *Response time in man-computer conversational transactions*, In: Proceedings AFIPS Fall Joint Computer Conference, 33, ACM, New York (1968), pp. 267–277.

[8] J. Young and S. Smith, *Akamai and JupiterResearch Identify '4 Seconds' as the New Threshold of Acceptability for Retail Web Page Response Times*, Akamai Press Release. Available at: http:// www.akamai.com/html/about/press/releases/2006/press_110606.html, 2006.

[9] D. Yuan, Y. Yang, X. Liu, and J. Chen, *A data placement strategy in scientific cloud workflows*, Future Generation Computer Systems, 26,8 (2010) pp. 1200–1214.

[10] D. Nikolow, R. Slota, and J. Kitowski, *Knowledge Supported Data Access in Distributed Environment*, In: Proceedings of Cracow Grid Workshop - CGW'08, October 13-15 2008, ACC-Cyfronet AGH, Krakow, (2009), pp. 320–325.

[11] S. Ghemawat, H. Gobioff, and S. Leung, *The Google file system*, ACM SIGOPS Operating Systems Review, 37 (2003), pp. 29–43.

[12] G. Aloisio and S. Fiore, *Towards Exascale Distributed Data Management*, International Journal Of High Performance Computing Applications, 23 (2009), pp. 398–400.

[13] S. Grottke, A. Koepke, J. Sablatnig, J. Chen, R. Seiler, and A. Wolisz, *Consistency in Distributed Systems*, TKN Technical Report TKN-08-005, Technische Universitaet Berlin. Available at http://www.tkn.tu-berlin.de/publications/papers/consistency_tr.pdf, 2007.

[14] I. Foster, *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering*, Addison Wesley, 1995.

[15] J. Planas, R.M. Badia, E. Ayguade, and J. Labarta, *Hierarchical Task-Based Programming With StarSS*, International Journal of High Performance Computing Applications, 23,3 (2009), pp. 284–299.

[16] A. Baumann, P. Barham, P.E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schuepbach, and A. Singhania, *The multikernel: a new OS architecture for scalable multicore systems*, In: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles, ACM (2009), pp. 29–44.

[17] 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. Amsterdam: IOS Press (2009), pp. 238–249.

[18] Intel Corporation, *An Introduction to the Intel QuickPath Interconnect*, Intel Whitepaper. Available at: http://www.intel.com/technology/quickpath/introduction.pdf, 2009.

[19] F. Petrot, A. Greiner, and P. Gomez, *On Cache Coherency and Memory Consistency Issues in NoC Based Shared Memory Multiprocessor SoC Architectures*, In: Proceedings of the 9th EUROMICRO Conference on Digital System Design. IEEE Computer Society (2006), pp. 53–60.

[20] L. Schubert, M. Assel, and S. Wesner, *Resource Fabrics: The Next Level of Grids and Clouds*, In: M. Ghanza and M. Paprzycki (Eds.), Proceedings of the International Multiconference on Computer Science and Information Technology (2010), pp. 677–684.

[21] 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, Springer (2009), pp. 167–185.

[22] D. Churches, G. Gombas, A. Harrison, J. Maassen, C. Robinson, M. Shields, I. Taylor, and I. Wang, *Programming scientific and distributed workflow with Triana services*, Concurrency and Computation: Practice and Experience, 18 (2005), pp. 1021–1037.

[23] M. Stoer and F. Wagner, *A simple min-cut algorithm*, J. ACM 44,4 (1997), pp. 585–591.

Edited by: Dana Petcu and Marcin Paprzycki
Received: May 1, 2011
Accepted: May 31, 2011