Self-managed Microkernels: From Clouds towards Resource Fabrics

Lutz Schubert[1], Stefan Wesner[1], Alexander Kipp[1], and Alvaro Arenas[2]

[1] HLRS – Höchstleistungsrechenzentrum Universität Stuttgart,
Nobelstr. 19, 70569 Stuttgart, Germany
{schubert,wesner,kipp}@hlrs.de
[2] STFC Rutherford Appleton Laboratory, e-Science Centre,
Didcot, OX11 0QX, UK
alvaro.arenas@stfc.ac.uk

Abstract. Cloud Computing provides a solution for remote hosting of applications and processes in a scalable and managed environment. With the increasing number of cores in a single processor and better network performance, provisioning on platform level becomes less of an issue for future machines and thus for future business environments. Instead, it will become a major issue to manage the vast amount of computational resources within the direct environment of each process – across the web or locally. Future resource management will have to investigate in particular into dynamic & intelligent processes (re)distribution according to resource availability and demand. This paper elaborates the specific issues faced in future "cloud environments" and proposes a microkernel architecture designed to compensate these deficits.

Keywords: distributed operating systems, SOA, multi-core systems, large-scale HPC, heterogeneous systems.

1 Misconceiving the Cloud?

Cloud Computing is often considered the future of computing platform provisioning: reliable application hosting over the web allows easy accessibility from everywhere to everything. Notably, however, this is a slight misconception of the actual working focus of "cloud computing", which focuses primarily on the manageability and scalability aspects of hosting. Remote hosting as such (i.e. reliable server farms) is not in itself a novelty and has been supported by multiple providers for a long time now – with remote access such as enabled by VMWare[1] or Remote Desktop[2], and replicated virtual machines, this already provided most of the capabilities associated today with Clouds. Only increased network and computational performance, as well as the advent of simple web "APIs" have allowed the sudden success of this approach.

Virtualisation, enhanced routing, on-the-fly replication, reconfigurable resources etc. are the core features of modern clouds and thus lead to other, more commercially

[1] http://www.vmware.com/
[2] http://www.microsoft.com/windows/windows-vista/features/remote-desktop-connection.aspx

D.R. Avresky et al. (Eds.): Cloudcomp 2009, LNICST 34, pp. 167–185, 2010.

oriented use cases which make use of the more innovative features of cloud computing. This includes aspects such as hosting of web "services" (e-Commerce) with demand-specific scalability and thus availability, as well as improved reliability – in other words, the application and data is highly available, independent of problems with the resources and amount of concurrent invocations. This becomes particularly interesting for e-Commerce environments with a high amount of customers, such as Amazon or eBay, which notably belong to the first entities actually making use of cloud-like environments internally.

Many users mistake cloud computing with high performance computing and whilst the same principles can principally be applied in the HPC environment, machine restrictions and requirements of the respective applications only allow for a certain degree of scalability and manageability, as replication is not easily achieved with the amount of resources in use, and scalability in the context of HPC is dependent on the algorithm, not the amount of requests.

Considering the current development in processor architectures and in network performance, future systems will effectively incorporate a cloud *environment* within a single machine. Due to their nature, these machines effectively allow for both: distributed / parallelised process execution (current HPC), as well as scalable and reliable application hosting. It should be noted in this context that "cloud computing" is not a technology as such, but rather a concept, respectively a paradigm. This paper will therefore examine the specific requirements put forward towards hosting applications in future environments, and elaborate an approach to address these requirements using approaches from Cloud Computing, Grid and SOA.

2 From Historical to Future Systems

The current development in computing system clearly indicates that the amount of cores being integrated into a single processor / machine will steadily increase in future years, whilst the speed of individual cores will increase only minimally. Implicitly, the system will not become more efficient regarding individual (single-thread) applications, but will provide an improved *overall* performance by allowing for parallel execution of multiple processes or threads concurrently.

Such systems are effectively identical to what was considered computer farms a few years back, where multiple computers are hosted within the same environment and can communicate with each other in order to coordinate and distribute processes. The Grid and P2P computing emerged from such environments, in order to maximize usage of otherwise unused resources (machines), e.g. during lunch-break or when no applications are running on the respective machines. Whilst the Grid has moved towards a different scope of distributed computing, one can still clearly see the relationship to Grid, SOA and in particular clouds: managing applications in a distributed environment so as to ensure reliability and higher performance. In particular in the P2P environment, one particular task consisted in replicating the same application with different configuration settings so as to produce a set of "integratable" results in the end: this only worked for "embarrassingly parallel" tasks, but still allowed for a definite increase in overall execution performance.

The tasks of such systems are similar to what modern operating systems (OS) have to face in multi-core environments: distribution of processes, according to individual schedules, as well as integration of results and management of cross-machine calls. As opposed to P2P systems with typically little to no requirements towards synchronization of the tasks, Grid systems investigated into coordinated execution of processes in distributed environments, whilst finally clouds are little concerned with distributed execution, but with distribution and scheduling of individual processes.

An efficient multi-core operating system should obviously not be restricted to parallel execution of standalone processes (thus reducing the scheduling problem), but should particularly support parallelized and highly scalable (multi-thread) processes. Accordingly, such a system needs to draw from all of the paradigms and concepts above in order to provide the necessary scalability, reliability and manageability of distributed processes in distributed environments.

2.1 Classical Approaches

In order to identify the specific capabilities to be fulfilled by future systems, it is recommendable to examine the classical concepts towards managing distributed environments in more detail so as to make best use of the multi-core capabilities:

Grid Systems. The modern grid integrates different resource types on a *service* level, i.e. principally follows the concepts of Virtual Organisations [1, 2], where the combination of individual services leads to enhanced capabilities. However, the Grid does provide means for common interfaces that allow the coordinated integration of heterogeneous resources for higher, abstract processes and applications.

Distributed Applications. Some computational algorithms can execute logical parts in parallel, so as to improve the overall process through multiple instantiation of the same functional block. One may distinguish between optimal parallel code (no data exchange between the blocks) and distributed applications that share some kind of data. Of particular interest thereby is the capability to control communication and to deal with the scheduling issues involved in multiple resource exploitation.

Cloud Environments. In a world of high connectivity, not only scalability of individual (distributed) applications is relevant, but also scalability in the sense of accessibility to a specific service / resource, i.e. replication of individual processes according to demand. This requires enhanced control over the resources and maintenance of multiple, potentially coupled instances of processes and data.

2.2 Scoping Future Multi-core Systems

As described above, the current cloud approach is insufficient to address the requirements of future multi-core systems, respectively might become obsolete with the capacities of such systems. However, in order to exploit the capabilities of multi-core systems, and in order to address the respective requirements towards future applications, clouds and related approaches provide a strong conceptual basis to realize such future support.

In light of the development of middleware and hardware, multi-core systems *should* be able to support the following capabilities:

Concurrency. The most obvious capability (to be) fulfilled by multi-core systems consists in the "real" concurrent execution of processes and applications, i.e. running (at least) one process per core so that they can be executed in real parallel instead of constant switching – however, each core may host multiple processes which are executed in a multitasking manner. The scheduling mechanism will thereby decide how to distribute processes across cores so that e.g. higher priority jobs compete with fewer processes on the same core, or get more time assigned than other jobs.

This feature is a simple extension to classical multitasking operating systems that assign jobs with different time slots in the overall execution schedule according to their respective priority. All current main stream operating systems choose this approach to exploit the multi-core feature for performance improvement, yet this approach only improves the net performance of the whole system, not of single processes.

Parallelism. More importantly than distributing individual processes to single cores, an application or job may be separated into *parallel* threads which can be executed concurrently at the same time. As opposed to concurrent individual processes, parallel processes share communication and information directly with each other – depending on the actual use cases either at nominated integration points, "offline" (i.e. via a common stack) or at even based, at random, unpredicted points in time. This poses additional constraints on timing and distribution of job instances / threads in the environment in order to ensure communication, respectively to reduce latency. Individual infrastructures thereby have a direct impact on this issue.

Typically, it is up to the developer to respect all this aspects when coding distributed applications. However, the requirements put forward to the developer will increase in future systems due to multiple reasons: *heterogeneous* resources will require dedicated code; *concurrent* processes will put additional strain on communication management (see above); processes and applications will *compete* with each other over resources; *latencies* will differ between setup and may thus lead to different communication strategies to be employed.

As the computing system grows and the complexity increases, the developer needs a simpler way to exploit the infrastructure with his / her code. Implicitly the infrastructure needs to provide stronger means and support the parallelization work.

Scalability. Parallel processes require that part of the code / a thread will be executed multiple times concurrently – in some cases the number is directly defined by the infrastructure (number of computing cores available) and not (only) by the application. In addition to this, in particular in the server domain, the same process may have to be instantiated and executed multiple times concurrently, e.g. when multiple invocations are executed at the same time.

Multi-core processors allow real parallel execution of one instance per core. Obviously, the system is restricted by the number of cores and the processing speed, with any number of instances higher than the number of cores impacting more and more on the reaction. With additionally concurrent jobs competing for the computational resources, managing scalability becomes a complicated aspect of both cloud and server provisioning, but also for specific common user cases, in particular where the processes have high computational requirements.

This aspect also strongly relates to data management issues involved in parallelism (cf. above), as some instances will have to share data between them, whilst others will host their own data environment (often also referred to as "stateless" vs. "stateful").

Reliability. Server architectures often use mechanisms of data and process replication in order to increase the respective reliability. Additional approaches include dedicated checkpointing and rollback. Whilst in classical "common" usage scenarios the cost for reliability was too high for the benefit gained from it, in particular cloud, server and HPC environments strongly require reliability features.

Depending on the relevance of the application and data, multi-core platforms should hence be able to support reliability.

Dynamicity. With multiple processes competing over the same resources (instead of, as in most cloud, server and HPC use cases typically only hosting one dedicated job), different resources will become available and unavailable over time, to which the distribution of processes must adapt. This ranges from simple (re)distribution of processes across the infrastructure to up- and downscaling of specific instances (see parallelism and scalability).

Notably, the degree of necessity per requirement and the degree of support by the system itself depends on the actual usage scenario. Nonetheless, in order in particular to ensure *portability* of applications across platforms and systems, i.e. in order to allow developers to provide their code equally as service, process or web application, it is mandatory that the essential basis of the system is identical.

3 The Monolithic Mistake

The current approach to dealing with multiple computational resources in a *tightly coupled* system consists in one central instance controlling all processes across these resources, i.e. all scheduling and communication is essentially centralised. It is notable that *loosely coupled* systems typically host communication support and essential system control features per node (as opposed to core), whilst only overall scheduling is centralised in the cluster. This decision is basing primarily on communication latency which will seriously impact on the performance of HPC systems and even though latency is much diminished in tightly coupled systems, the central instance will act as a bottleneck that potentially can lead to clashes, unnecessarily stalling the individual processes:

Monolithic kernels are often said to scale well with the amount of processes on many processors (see e.g. [16]). It should be noted though that this is not identical to scaling well with the amount of processors. Most tests are executed on a limited number of cores where the increment in the number of processes effectively shows similar behaviour in single-core machines, i.e. scalability is primarily restricted by memory and processor-speed, not by the operating system itself, as the degree of concurrency and hence the additional strain on process management is comparatively low.

The main reason for this consists in the fact that the OS primarily deals with system requests, context switches and device access, not with the process itself. In other words, as long as the processes do not require something from the OS and whilst the scheduler does not demand a context switch between processes, the OS' tasks are not

affected by the amount of processes. Obviously with an increase in the number of jobs, the amount of requests increase – notably, in a single core system the average amount of context switches does not increase as they are defined by the scheduling algorithm and only indirectly by the amount of processes (depending on the scheduling strategy).

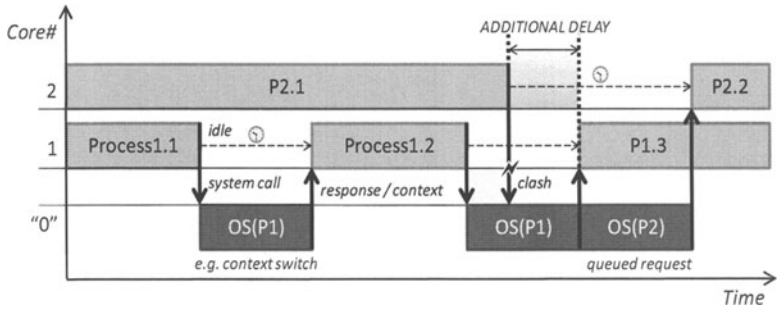

Fig. 1. System requests of concurrent cores may clash if they occur within the same timeframe (time-relationships exaggerated)

With the increasing amount of cores the operating system in particular has to deal with more system requests – however, this alone would not impact drastically on the performance, as system requests are comparatively few and quick as opposed to process execution. Hence, scalability would only be affected if more system requests need to be handled than a single core can execute. More drastic, however, is the impact of system request clashes which arises from the concurrent nature of process execution: as depicted in Fig. 1, a second core may request an operation from the operating system whilst the latter is still dealing with a request from the first core.

Under normal conditions these clashes hardly affect the overall performance, as they occur rarely and as the delay caused by it is comparatively short. However, with the number of cores rising to a few thousands, clashes become more regular, thus leading to a significant overall delay in processing and hence decreasing the effective performance per core.

Fig. 2 depicts this issue in an exaggerated fashion for the sake of visibility[3]: most monolithic kernels (and in particular most developers) assume that processes are executed in a fashion similar to Fig. 2 above, i.e. with short gaps between processes caused by context switches, respectively by other system requests. In reality, however, these requests overlap and causing the OS to queue the messages and execute them in sequential fashion, thus delaying process execution even further. Fig. 2 below indicates how these overlaps summarise during a given timeframe, whereas dark blocks depict the delay caused in addition to the (expected) system request execution time and the arrows reflect the accumulated delay per core within the timeframe. Note that we assume in both cases that a full core ("0") is dedicated to OS execution for the sake of simplicity.

[3] Actual figures will be published in a separate paper – please contact the authors for more information.

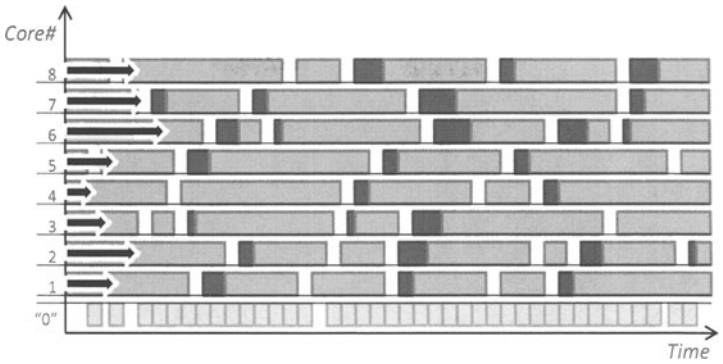

Fig. 2. Multiple processes executed in real parallel lead to significant process delays due to overlaps in system requests – the dark blocks denote *additional* delays, the arrows reflect the full delay in the timeframe. This figure assumes that one core ("0") is designated completely to the operating system.

Obviously, this impact depends directly on the amount of cores and the number of processes running per core. With an expected number of thousands of cores in the near future, the monolithic kernel will become a bottleneck for concurrent processes.

In order to overcome this effect, each core must hence maintain enough information to allow execution of main and repeating system requests. This puts additional constraints on the scheduling and the memory management system – in particular since the actual memory per core is still comparatively small in common multicore systems. With the current communication structure in multicore processors, it is also impossible for individual cores to access the memory extension (L2 cache) without going via the main controller, and thus automatically blocking access for other processes, so that the same clash situation arises again (see e.g. [3]). Even though parallel memory access is being researched, a good strategy for exploiting the level 1 cache is still required in order to maintain a low latency.

Of course, there are further issues that impact on the performance of monolithic systems – particularly worth mentioning are distributed scheduling in centralised systems and the tight hardware binding: in heterogeneous, large-scale systems, additional overhead has to be put on the main instance, in order to maintain processes and resources. In [7] we discuss the concepts of application execution across distributed resource fabrics (similar to clouds), with a particular focus on aspects related to scheduling and dynamic infrastructures (as opposed to the kernel structure).

4 Moving on to Micro-kernels

It has often been claimed that the messaging overhead caused by the component-based segmentation of the micro-kernel approach impacts stronger on performance than the centralistic approach pursued by monolithic systems [4]. This is generally true, if one takes an essential centralistic approach with the microkernel architecture too. In essence, such an approach is identical to a monolithic system with all communication having to be routed via a central instance – with the additional overhead of

complicated messaging protocols. However, this is essentially a specific use case of the microkernel architecture where the monolithic kernel is basically structured according to the Object-Oriented Programming (OOP) and Service Oriented Architectures (SOA) paradigm. It does not take the full consequences from the microkernel approach though:

4.1 SOA and Segmentation

Though SOA and OOP are related, one of the core differences consists in the communication connection between components: in general, OOP assumes that all components are hosted locally on the same machine, whilst SOA is not restricted to specific communication models – in fact, there is a certain tendency to assume that components are deployed on different resource. With respect to microkernel architectures, this implies in particular that functionalities can be separated not only "methodologically" but also with respect to their distribution across resources. Or more specifically: each core can host part of the operating system.

Typically, in modern processor architectures, one must distinguish between hierarchical internal memory (L1 & L2 cache) and external memory. Even though external memory is fast, its latency is too high for efficient computation (the processor being faster than the memory) and it brings in yet another bottleneck factor, as the cores cannot directly access the memory individually, but have to be routed via a processor-central controller (cf. Fig. 3). Future systems will allow for more flexibility with this respect, i.e. by granting parallel access to the external memory [5] – however, the main issue, latency, will still apply.

To reduce latency and thus improve performance of the system, the full execution environment should be available in level 1 cache, so that calls and jumps can be processed locally without requiring access to external memory. This is the ideal approach for single core systems, where changes in the memory structure do not affect other processes (on other cores). However, the main problem is not posed by the synchronisation between individual memory views, but in particular by the restriction in size per L1 cache – in particular with the growing amount of cores, cache memory impacts heavily on the price of the processor. In order to host the *full* execution context, however, the cache would have to cater for a) the full process code, b) the application data and c) the operating system or at least all exposed functions and methods. Together, this exceeds the limits of the cache size in almost all cases.

This is a well-known problem in High Performance Computing, where a particular challenge consists in identifying the best way(s) to distribute and access application specific data. As the cache in supercomputing nodes is way larger than the one in common multi-core systems, the thread or code part is typically fully hosted in the cache, without having to think about further split-ups. As opposed to this, however, system calls will all be routed to the main node, as this is the classical monolithic OS approach (cf. above).

The main idea of Service Oriented Architectures, similar to OOP, consists in splitting up the main process into individual methods, functionalities and sub-processes that can principally be hosted in different locations. The main challenge thereby consists in finding a sensible block size that is not too small so as to create messaging overhead and not too big so as to impact on flexibility again – typically a logical

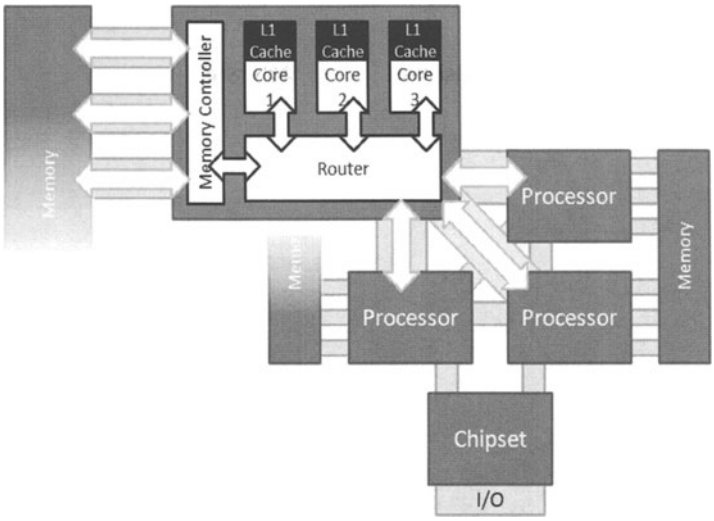

Fig. 3. The architecture of a multi-core & -processor system (adapted from [3])

segmentation provides the best results in this context. The same principle can be applied to data segmentation and is principally applied in distributed data management, though typically the segmentation criterion is comparatively arbitrary and not related to data analysis.

By applying SOA paradigms to both code *and* data, the core cache can be filled with smaller parts rather than with the full execution environment, which would exceed the available space. Obviously, this is not a general solution though, as it immediately poses the following problems:

1. Dynamicity: during normal execution, the process will jump between methods of which only parts are loaded in memory, so that constant loading and unloading has to take place.
2. Dependencies: code and data stand in a direct relationship, i.e. data access has to be considered when separating code and data blocks.
3. Integrity: with multiple code segments accessing the same data blocks and potential replications of the same data, updates need to be communicated in order to ensure integrity of the process' behaviour
4. Distribution: segmented code is not necessarily executed and loaded in a strict sequential fashion anymore – accordingly, multiple cores may host parts of the code, replicate data etc. In order to ensure integrity, dependencies and so as to actually improve performance, this distribution needs to respect the process' restrictions, requirements and capabilities.

4.2 SOA and Operating Systems

As noted, micro-kernel operating systems principally follow an object (or service) oriented approach where functionalities are segmented into libraries with flexible

communication interfaces. This allows on-demand loading of libraries according to need, as well as distribution across multiple cores for more efficient execution. In other words, each core's cache may host part of the OS' functionality according to the respective processes' needs. This effectively distributes the load of the operating systems on cache and core across the system and, at the same time, increases the availability of system functionalities for the executed processes, thus improving performance and reducing the risk of clashes caused by procedure calls (cf. section 3).

Since segments can be replicated, essential, recurring functionalities (such as virtual memory management) can be hosted on each core at the same time so that no bottleneck issue arises directly. However, any access to remote resources and in this case including "external" memory (cf. section 4.1), will be subject to the same message queuing problems (and thus bottlenecks) as calls to a centralized operating system. Regarding actual physical devices (such as printers, hard drive, network etc.), the according latency is typically so high that delays are expected anyway. As for resources with "lower" latencies (such as external memory in this case), replication and background updating strategies reduce the risk of bottlenecks and improve access. By estimating *future* data access, data can be loaded in the background thus further reducing the delay caused by loading and unloading memory.

Fig. 4 illustrates the assignment of logical process *blocks* and data *segments* to the cache of individual computing units of a multi-core processor (cf. Fig. 3). Note that a full distribution is not necessarily the most efficient way to handle a single, non-parallel process: as all code blocks are executed in a sequential fashion, cores would either idle whilst they wait for the respective block to get invoked, or switch between different assigned and scheduled process blocks of the respective core.

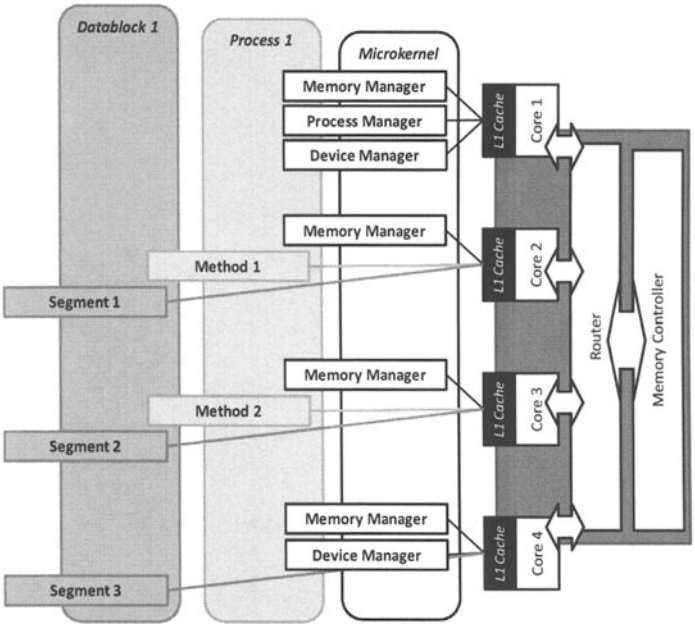

Fig. 4. Distributing Operating System, process and data block across cores

Hence, it is most crucial to find the best distribution of a) a single process' code and data blocks with respect to their interaction with one another, their invocation frequency and their respective resource requirements (see below), b) operating system unit with respect to their relationship to the code blocks, i.e. which functionalities are required by the respective process part(s) and finally c) overall processes and operating system capabilities to make the most of common requirements (e.g. towards capabilities) and adhere to the overall scheduling and prioritization criteria.

5 Principles of the Service-Oriented Operating System

Though we focus particularly on the multi-core, i.e. tightly coupled use case here in this paper, the principle communication modes between the distributed components actually depends on the setup, where obviously higher latency communication impacts on the distribution of blocks across the infrastructure (in order to meet the interaction requirements).

In this section, we will discuss the principle behaviour of SOA based microkernels, with a specific focus on the segmentation of code and data according to relationship information, requirements and restrictions.

5.1 Microkernel Base Structure

As noted, the microkernel structure is component-based, i.e. segmented into logical functional units where each "component" fulfils essential capabilities for specific tasks. For example, virtual memory management, device management, execution management etc. all build units of their own, that may even be further sub-segmented, respectively that can be adapted according to specific parameters – likewise, e.g. a local virtual memory manager instance only needs to maintain information relevant for hosted process parts and the device manager only needs to provide interfaces to devices actually required by the local processes etc.

At process load time, the requirements of the respective process are retrieved respectively analysis is initiated (cf. below) and the according operative components will be shifted to the core along with initial data and assigned code block. Note that if microkernel components are already assigned to the respective core, that adaptations may be needed to reflect the new requirements. In principle, each context switch could rearrange the local microkernel component arrangement – obviously, this would cause unnecessary load and the main task in identifying potential segmentations consists in reducing such overhead.

The space in this document is insufficient to represent the full architecture of a SOA based microkernel operating system (short S(o)OS: Service-oriented Operating System) – for more details please refer to [7, 9]. Instead, we will focus on one of the core components only, namely the virtual memory manager:

The virtual memory manager is hosted on almost all cores – it is responsible for virtualising the infrastructure per process (execution environment) and for analyzing the code behavior. In essence, it is a dynamic routing mechanism which forwards requests to and from the code to the respective location in the external memory.

Distributed process manager maintains a high level overview over the processes and control distributed execution (i.e. passing the execution points between cores whilst maintaining the execution context).

Micro schedulers replace the centralized scheduler and are responsible for scheduling the processes *per* resource, rather than for the full system. Micro schedulers are aligned to the overall priority and scheduling assignment.

Virtual device controller provides a virtual interface to resources of any kinds to allow the process to access resources without having to implement the protocol details – this is similar to e.g. the Hardware Abstraction Layer of Microsoft systems, but acts on top of the I/O manager to allow remote integration independent of the underlying communication protocol.

I/O manager, like in any other operating system, provides the communication interface between resources. It incorporates different communication layers, thus integrating tightly e.g. into the distributed virtual memory (see above).

5.2 Relationship Analysis and Distribution

The main important feature to enable service oriented microkernels as described above consists in the capability to split code and data into meaningful blocks that can be hosted by individual cores, respectively fit into their cache. As this segmentation must be dynamic, to meet the (changing) requirements and constraints of the execution system, the according distribution depends only secondarily on the information provided by the developer, even though programming models such as MPI [6] foresee that individual methods can be distributed and that specific communication modes exist with and between these segments. In order to increase performance and capability of such distributed models, new programming paradigms will be needed – as this is of secondary relevance for this paper, the according findings will be published in a separate document (see also [7]). We therefore assume in the following that no additional information has been provided by the developer, even though the model described below principally allows for extended programming annotations.

Code and data segmentation follows the principle of graph partitioning whereas nodes represent code / data blocks and edges their relationship with one another. As the code has already been compiled, i.e. since the source code is not available for structural analysis, segmentation must base on "behavioral" blocks rather than methods and class structure. At the same time, this provides better relationship information than pure code analysis, as frequency of invocation is often determined by environmental conditions, events, parameters etc. In order to analyze and obtain this kind of information, all code is enacted within a virtual memory environment, where access to data and other code areas is routed via extended paging information. This is principally identical to the way any modern operating system treats memory.

By applying a divide and conquer approach, the virtual memory is divided into logical blocks that represent the code's "typical" execution path and its relationship to data, system calls and other processes (cf. Figure 5, left). Such information is gained by following the calls and read / write access via the virtual memory. This relationship information can be represented as a directed graph (cf. Figure 5, right), whereas an edge between code nodes implies invocations, respectively jumps, whilst an edge

to a data node represents a write action, respectively an edge from such a node represents read access. By analyzing access, invocation and access frequency, the graph can furthermore be annotated with a weight (*w*) representing the likelihood of one node calling / accessing another, as well as a frequency (*f*) that designates how often the respective code is accessed during a given timeframe at all (note that this information can principally be derived from a full invocation graph and the according weights of the nodes).

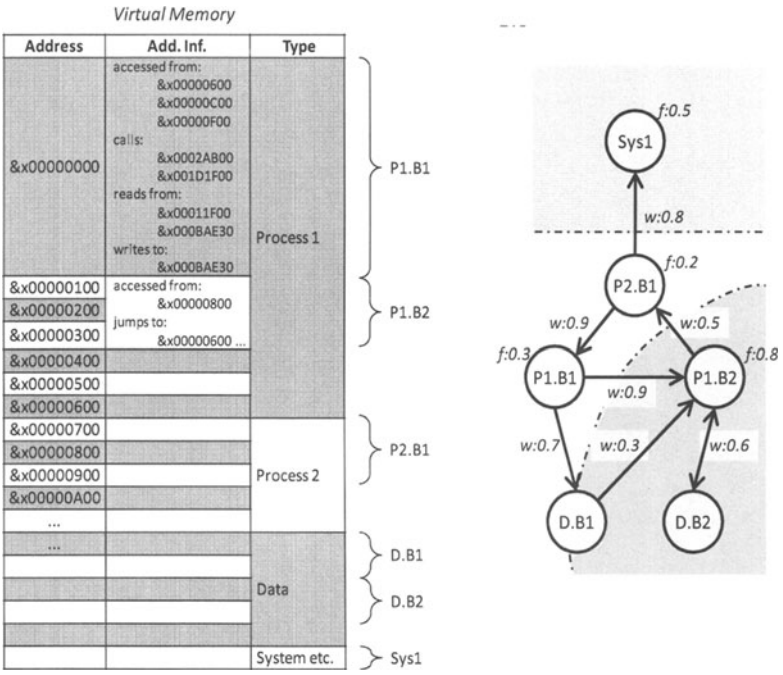

Fig. 5. Annotated memory and relationship analysis. *f* stands for the "frequency" of execution in a given timeframe and *w* for the likelihood that the caller invokes the respective node.

Implicitly, the information exactness increases over the amount of executions and during the time actually using the respective processes or applications. It is therefore recommended to expect a minimal number of invocations or wait until a certain stability of the graph is reached before actually applying the segmentation and distribution – even if this means that the infrastructure cannot be optimally exploited in the beginning. Otherwise, there is a high risk that additional code movements will produce more overhead than gain.

Principally, such annotation data could be provided by the developer (cf. comments above), but this would exceed the scope of this paper.

5.3 Code and Data Segmentation

As stated, code and data needs to be segmented in a fashion that meets multiple requirements and constraints, such as cache size, relationship with data and other code

(including system calls) etc. so that the unnecessary overhead on the core is reduced. Such overhead is caused in particular by loading and unloading context information, processing message queues due to centralisation and so on. Ideally, all processes, all their contexts and all according system data fit into the cache of the respective core – this, however, is most unlikely. Therefore, the segmentation must find a distribution, where common requirements of concurrent processes are exploited and where relationships between codes and data are maintained to a maximum.

Figure 5, right side designates such a potential segmentation given the relationship as stated in the table (Figure 5., left) and the temporal information represented by f (frequency of execution in a given timeframe) and w (likelihood that one code calls another code block, respectively accesses a specific data area). The figure already indicates some of the major concerns to be respected in this context, such as shared data segments, concurrent invocations, cross-segment communication etc.

As it is almost impossible for the core cache to hold all the code blocks, all related data (including global variables) and the according system processes at the same time, the micro kernel has hence to account for the following potential issues:

- Dynamic (un)loading of process blocks is normal behavior for all operating systems executing more processes than fit into memory. It involves all the issues of context switching plus overhead for load / memory management.
- Replication and hence consistency management of shared data across different caches. Background and / or dedicated synchronization needs to be executed in order to keep consistency. Timing vs. potential inconsistency is important in this context and the relationship analysis information can be employed to identify the *least* amount of synchronization points. Data consistency is covered substantially in literature though and will not be elaborated here (e.g. [14] [15]).
- Concurrent usage of access limited resources (e.g. hard-drive) pose issues on consistency and cause delays in the executing process. In order to reduce delay, the process is often handled by separate threads – in the case of multi-core processors, these threads can be handled like separate processes with the additional relationship information in the respective process graph.
- Queuing and scheduling is in principle no different to other OS [8] – however self-adapting microkernels have the additional advantage that they can rearrange themselves to process queues faster, given that they do not compete for restricted / limited resources.
- Cross-segment communication, as opposed to the single-core approach, requires dedicated communication points, channeling of messages, as well as their queuing etc. Similar to limited resources, data consistency etc. communication between segments may cause delays due to dependencies.

The main issue in executing segmented code and that also causes problems in manual development of distributed programs consists in the delays caused by communication between threads – partially due to latency, but also due to the fact that processes do not send / require information exchange at exactly the same point, so that delays in response, respectively in reception. MPI (Message-passing Interface) [10] is one of the few programming models dedicated to handling the communication model between blocks and similar principles must be applied in the case of automated segmentation.

Efficiency may be slightly increased by executing other processes whilst the respective thread(s) are put into a waiting state – accordingly, the amount of communication has to be kept at a minimum. In segmented (as opposed to parallelised) code, the main communication within a single process consists in passing the execution environment between blocks, and system calls. As opposed to this, cross-process communication is comparatively seldom.

5.4 Self-adaptive Microkernels

As noted, the main issue to be addressed by the OS (respectively the kernel), consists in reducing the communication and the context switching overhead, respectively keeping it at a minimum. Since the two main causes for this overhead consists in passing the execution point between code segments and making system calls – and thus implicitly accessing resources, including the virtual memory – the most strongly related code parts should be made locally available, whereas lower-level cache is preferable over higher-level one, as latency increases over distance (level).

The segmentation must therefore find the best distribution of code blocks according to size of cache and their latency – in other words, frequent invocations and strong relationships should be located closer than loosely coupled blocks. This does not only apply to process specific code and data, but implicitly also to system calls – in particular since essential capabilities (virtual memory, messaging etc.) are required by almost all processes to execute smoothly in a potentially dynamic environment where locations (in particular in memory) are subject to change.

In the classical OS approach, as noted, the main kernel instance (located on any one core) is responsible for handling such requests, leading to additional messaging overhead, conflicts and extensive delays. With the more advanced dynamic approach as suggested here, the kernel can provide *partial* functionalities to the individual core's environment, where it sees fit. This segmentation is basing on the relationship information as described above – however, since the kernel is more sensitive to execution faults and since it also requires that specific functionality is available and cannot be routed to another code location, such as the capability to route in the first instance, some segments need to be made available together. Furthermore, since the virtual memory is enacted by the kernel itself, relationship information is generally not maintained about the kernel in order to reduce overhead.

Instead, the kernel is structured in a fashion that adheres to the main principles of SOA: atomic, logical functionality groups; minimal size; common interfaces and protocol-independent communication. By identifying the direct entry points of the process into the system kernel (i.e. system procedure calls), the segmentation method can identify the system capabilities that need to be provided in addition to base capabilities, such as virtual memory and communication handling. Depending on the system calls needed by the process, additional segments can be identified that need / should be provided with the sub-kernel in the respective core's cache – the primary restriction consisting in the size of the cache.

Sub-kernels will only maintain memory information related to the specific processes, in order to reduce the memory size required. Similarly, only essential, frequently required functionalities will be hosted in the same cache. The according selection of kernel methods bases primarily on predefined architectural relationship

similar to the one depicted in Figure 5 – the fully detailed kernel architecture relation graph will be published in a separate document, as it exceeds the scope of the current paper.

Context switches are particularly critical with OS methods, as no higher-level management system (i.e. the kernel) can supervise the process at this level. As switches on this level add to the delays caused by context switches per core, the amount of changes in the sub-kernel infrastructure per core should be kept to a minimum. Implicitly, the distribution of processes across cores does not only depend on relationships between segments and the size restrictions of the according cache, but more importantly on the functional distribution of sub-kernel segments. In other words, the relationship to system procedure calls and the according distribution across cores plays an essential role in the segmentation process, whereby the amount of switches between sub-kernel routines should be kept to a minimum.

Each system procedure call can therefore lead to one of the following three types of invocation:

1. Local processing using the cache of the respective core – this is the most efficient and fastest call, but leads to the same consistency issues as segmented processes do
2. Local processing with context switching – in this case the call is executed by the same core that processes the invoking procedure, but must load the system procedure from central memory (or another location). This reduces the consistency problem, as the context switches can update the memory, but it leads to increased delays in the execution of the invoking procedure
3. Call forwarding to the main kernel's core – system procedure calls can also be forwarded to the main kernel instance, just like in monolithic instances. Obviously this loses the advantage coming from a distributed kernel, namely obstructing message queues and concurrent call handling. By reducing the average number of "centralised" system calls, however, the risk of conflicts decreases accordingly (cf. Section 3). Since such call handling comes at the cost of higher latency, it is generally recommended to reserve this for background calls (that can be executed in parallel and may be identified in the dependency graph). In all cases, the OS must be able to precedence "active" processes over "waiting" ones, e.g. through an event-based system – a detailed discussion of these mechanisms will be published separately.

6 Local Private Clouds (or Micro-Clouds)

As has been mentioned in the initial chapters, current approaches towards cloud systems all take a high-level approach towards resource management, i.e. they assume that the operating system handles simple multi-core platforms and that main cloud features act over multiple instances (PCs, Servers) rather than over multiple cores as such. Implicitly, most cloud systems only address horizontal elasticity – process / data replication on multiple systems – and only little vertical elasticity – extending the amount of resources vested into a single instance, though notably the according scale will have to be applied to all horizontal replications too.

The biggest business motivation for outsourcing to clouds at the moment being that equipment and maintenance of a local resource infrastructure (private cloud) is too

costly. However, such assessments forget about the current development in current Microsystems leading to unprecedented resource availability even in desktop PCs. This poses three issues: 1) outsourcing to public clouds will only be of interest for large scale applications, 2) applications and services must foresee their own (vertical) scalability already at development time, whereas only little "common" programming models are available to this end, and 3) scalable execution on local infrastructures requires new OS models.

This paper presented an approach to exploit the specific features of multi-core systems in a way that enables cloud-specific capabilities on a single (multi-core) machine:

6.1 Elasticity in Self-managed Microkernels

The core feature of selfmanaged microkernels as presented in this paper consists in its capability to adjust the distribution of code and data segments according to resource requirements and availability. By updating the relationship graph frequently and relating individual graphs (per process) with one another, the system can adjust the vertical scale to reflect the current requirement of the process in alignment with other processes and resource availability. Since the principle of service oriented operating systems also enables enhanced programming models, vertical scalability can both be exploited for more efficient data throughput, as well as for multiple instantiation of individual threads with shared, as well as distributed memory. Such threads can be dynamically instantiated and destroyed by the system, but the process itself must still be capable to deal with a dynamic number of concurrent threads. Optimally parallelizable code, i.e. algorithms that execute calculations on separate data instances and which results' are integrated only after execution, are ideal for such usage – typical examples for such applications are 3d renderers, protein folding etc. [11] [12] [13].

Horizontal scalability in a multi-core environment is only limited by the number of cores – similar to the limitation of yesterday's web servers that merged multiple motherboards ("blades") into a single interface. As discussed, multiple instantiation automatically leads to the problem of consistency maintenance, which has to be compensated by complex data management mechanisms which lead to additional latencies, as they act on a higher level than the processes themselves. Even though service oriented operating systems cannot handle complex differentiation and merging strategies, they can nonetheless support data consistency management through background synchronization thus ensuring that multiple instances have access to principally the same data body.

6.2 Open Issues

Service-oriented operating systems and self-managed microkernels are still research issues and as such, many challenges remain incompletely solved, such as security aspects and reliability:

Security: since service oriented operating systems act below the level of virtual machines (but on top of virtual resources), they implicitly do not support segregation into secure, individual execution environments. All top layer security can be provided in the same fashion as in classical, non-SOA operating systems, though kernel-near security (message encryption etc.) may need further investigation, considering the dynamic distribution of processes and sub-kernel modules across cores.

Reliability: self-managed microkernels can principally increase reliability through improved data and code management which allows even dynamic (re)distribution of code, thus dealing with potential issues. However, main reliability issues arise from hardware faults which cannot be foreseen, therefore typically being addressed by means of replication mechanisms. Though service oriented OS support replication mechanisms, it is typically the whole system that goes down and not just a single core, so that cross-system mechanisms need to be employed. In [9] we discuss the principles of a distributed virtual memory to enable distributed execution and indicate how replication across systems may be realized – however, such mechanisms are still subject to research.

6.3 Summary

The self-managed microkernel approach as presented in this paper is taking cloud concepts to a core level in future tightly coupled systems, thus providing elasticity for large scale systems, as well as means to deal with dynamic and heterogeneous infrastructures. This will not only allow common users and providers to make use of cloud features in simple, smaller sized infrastructures, but also enable new means to write and execute distributed applications in dynamic environments.

Multicore systems for common usage are comparatively new on the market and distributed computing platforms so far have mostly been an issue for high performance computing developers. With the trend of integrating more and more cores into a single system, the average developer is now faced with similar issues than HPC programmers were before and who have realized their own specific programming models to realize these issues. The self-managed microkernel approach simplifies this problem by providing new means to develop distributed applications that allow for a certain degree of self-management, namely cloud capabilities.

At the same time, many issues have not yet been fully researched in this area and since furthermore most approaches only consist of conceptual models so far, actual benchmarks still have to validate the approach and, what is more, define the fine-grained parameters to identify cut-off points in code / data segmentation, as well as the according dynamicity.

Business benefits for such a system are obvious, yet not all of the according requirements have been addressed so far, since many of them require that a stable base system exists first. It is e.g. not sensible to elaborate authorization mechanisms yet, when not all implications from code segmentation have been fully elaborated – as such, security could be tightly coupled with the main kernel instance, or be dynamically distributed like other sub-kernel modules.

References

1. Saabeel, W., Verduijn, T., Hagdorn, L., Kumar, K.: A Model for Virtual Organisation: A structure and Process Perspective. Electronic Journal of Organizational Virtualness, 1–16 (2002)
2. Schubert, L., Wesner, S., Dimitrakos, T.: Secure and Dynamic Virtual Organizations for Business. In: Cunningham, P., Cunningham, M. (eds.) Innovation and the Knowledge Economy - Issues, Applications, Case Studies, pp. 1201–1208. IOS Press, Amsterdam (2005)
3. Intel, Intel White Paper. An Introduction to the Intel® QuickPath Interconnect (2009), `http://www.intel.com/technology/quickpath/introduction.pdf`
4. Lameter, C.: Extreme High Performance Computing or Why Microkernels Suck. In: Proceedings of the Linux Symposium (2007)
5. Wray, C.: Ramtron Announces 8-Megabit Parallel Nonvolatile F-RAM Memory (2009), `http://www10.edacafe.com/nbc/articles/view_article.php?section=ICNews&articleid=714760`
6. Gropp, W.: Using MPI: Portable Parallel Programming with the Message-passing Interface. MIT Press, Cambridge (2000)
7. Schubert, L., Kipp, A., Wesner, S.: Above the Clouds: From Grids to Resource Fabrics. In: Tselentis, G., Domingue, J., Galis, A., Gavras, A., Hausheer, D., Krco, S., et al. (eds.) Towards the Future Internet - A European Research Perspective, pp. 238–249. IOS Press, Amsterdam (2009)
8. Tanenbaum, A.S.: Modern Operating Systems. Prentice Hall PTR, Upper Saddle River (2001)
9. Schubert, L., Kipp, A.: Principles of Service Oriented Operating Systems. In: Vicat-Blanc Primet, P., Kudoh, T., Mambretti, J. (eds.) Networks for Grid Applications, Second International Conference, GridNets 2008. Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering, vol. 2, pp. 56–69. Springer, Heidelberg (2009)
10. Gropp, W.: Using MPI: Portable Parallel Programming with the Message-passing Interface. MIT Press, Cambridge (2000)
11. Anderson, D.: Public Computing: Reconnecting People to Science. In: Conference on Shared Knowledge and the Web. Residencia de Estudiantes, Madrid, Spain (2003)
12. Menzel, K.: Parallel Rendering Techniques for Multiprocessor Systems. In: Computer Graphics, International Conference, pp. 91–103. Comenius University Press (1994)
13. Foster, I.: Designing and Building Parallel Programs: Concepts and Tools for Parallel Software. Addison-Wesley, Reading (1995)
14. Tanenbaum, A.: Modern Operating Systems. Prentice-Hall, Englewood Cliffs (1992)
15. Deitel, H.: An Introduction to Operating Systems. Addison-Wesley, Reading (1990)
16. Lameter, C.: Extreme High Performance Computing or Why Microkernels Suck. In: Proceedings of the Linux Symposium (2007)