PROGRAMMING MULTI-CORE AND MANY-CORE COMPUTING SYSTEMS

PROGRAMMING MULTI-CORE AND MANY-CORE COMPUTING SYSTEMS

A JOHN WILEY & SONS, INC., PUBLICATION

Published by John Wiley & Sons, Inc., Hoboken, New Jersey.
Published simultaneously in Canada.

Library of Congress Cataloging-in-Publication Data:

Title, etc
Printed in the United States of America.

10 9 8 7 6 5 4 3 2 1

CONTENTS

CHAPTER 1

OPERATING SYSTEM AND SCHEDULING FOR FUTURE MULTI-CORE AND MANY-CORE PLATFORMS

TOMMASO CUCINOTTA[1], GIUSEPPE LIPARI[1] AND LUTZ SCHUBERT[2]

[1]Real-Time Systems Laboratory, Scuola Superiore Sant'Anna, Pisa, Italy
[2]HLRS - University of Stuttgart, Stuttgart, Germany

1.1 INTRODUCTION

Computing systems are experiencing nowadays a complete paradigm shift. On the old-fashioned single-processor platforms, sequential programming used to constitute an easy and effective way of coding applications, and parallelism was used merely to easily realise independent or loosely coupled components. True parallel and distributed programming used to constitute a domain reserved to only a relatively small number of programmers dealing with high-performance computing (HPC) systems and commercial high-end mainframes. However, recently, multi-core systems have become the de-facto standard for personal computing and are being increasingly used in the embedded domain as well. Furthermore, in the context of servers, it is more and more common to see multi-processor and multi-core systems realising up to 8 - 12 cores. The according process just continues: already experimental multi-core processors, such as the Polaris by Intel, reach up to 80 cores and specialised processors, such as the Azul Vega, go up to 54 cores, yet the mass market commercial exploita-

title, edition. By author

tion of these processors will still take a few years. Along that line, high-performance and massively parallel systems are undergoing a tremendous architectural shift that promises to move towards an unimaginable number of interconnected cores and other hardware elements such as local memory/cache elements, within the same chip. As a consequence, in the short future, parallel and distributed programming paradigms need to become more and more widespread and known across the whole base of developers, comprising not only the high-performance computing domain, but also the general-purpose one.

However, the entire ensemble constituting the software stack of nowadays computing systems, comprising Operating Systems, programming languages, libraries and middleware, are still too much influenced by the former non-concurrent era, and are slow at adapting to the new scenario. Furthermore, when dealing with real-time and more generally time-sensitive applications, the support of a General-Purpose OS suffers of various drawbacks: it is often merely limited to priority-based scheduling and some kind of priority inheritance mechanism, it does not provide temporal isolation across concurrently running applications; it cannot properly deal with interactive tasks with performance and end-to-end latency requirements; etc. On the other hand, the use of a full RTOS for complex time-sensitive applications (e.g., multimedia) is prohibitive due to the lack of complex and commonly needed functionality (e.g., high-level communication protocols, middleware, encoding, etc.).

A few GP OSes (e.g., Linux) have been extended for addressing the requirements of complex, distributed real-time applications. However, being developed in the domain of real-time and embedded systems, they lack essential scalability capabilities needed for coping with large-scale systems. Also, various OS extensions and middleware components have been proposed to cope with large-scale distributed systems designed for GRID, High-Performance, or Cloud Computing domains. However, these approaches usually rely on traditional kernel architectures which are not capable of handling many-core nodes.

In this chapter, an overview is made on the limitations of the nowadays Operating System support for multi-core systems, when applied to the future and emerging many-core, massively parallel and distributed platforms. Also, an overview is done of the most promising approaches which have been proposed to deal with such platforms. The discussion is strongly focused on the kernel architecture models and kernel-level mechanisms, and the needed interface(s) towards user-level code.

1.1.1 Organisation of this chapter

This chapter is organised as follows: in Section 1.2, various works proposed in the literature about Operating System architectures and kernel models for multi-core and many-core systems are overviewed. In Section 1.3, the focus is specifically on the critical problem of scheduling in multi-processor and distributed systems, comprising scheduling of applications with precise timing requirements.

1.2 OPERATING SYSTEM KERNEL MODELS

Various models of Operating Systems and kernels have been proposed in the literature. In what follows, an overview of the most significant works is provided.

1.2.1 Linux Scalability

It must be noted that the Linux kernel developers community is focusing more and more on the issue of scalability of the kernel in the number of underlying cores. This is witnessed by the announcement[1] by Linus Torvalds accompanying the 2.6.35 kernel release, mentioning the imminent merge of the VFS scalability patch by Nick Piggin into the mainline kernel. This patch aims to remove many bottlenecks at the kernel level that hinder the performance of Linux file-system operations when being deployed on multi/many-core machines. Also, the scheduler subsystem is already designed since long ago with scalability in mind: distributed per-core runqueues and cpusets [22] allow for keeping a limited sharing of data among different cores.

Furthermore, there exist various projects for improving the Linux scalability of the kernel even further. The Linux Scalability Effort project[2] , running within years 2001 and 2004, aimed to improve scalability of various subsystems of the kernel. More recently, during the 2006 Kernel Summit[3] , Christoph Lameter gave a comprehensive talk about the current status of scalability issues on the Linux kernel, highlighting the importance of robustness to failures. On a related note, Deputovitch et al. presented [21] a mechanism that allows one core to recover a kernel panic occurred onto another core, with applications potentially able to recover seamlessly from kernel crashes. During the Linux Kongress 2009, Kleen presented [40] various known bottlenecks for the scalability of the kernel, and workarounds for avoiding them. Boyd-Wickizer et al. investigated [13] on scalability issues of the Linux OS on a 48-core machine, identifying various related bottlenecks at the kernel-level arising from the use of seven different application benchmarks. Interestingly, the authors conclude with: "a speculative conclusion from this analysis is that there is no scalability reason to give up on traditional operating system organizations just yet." On a related note, Lelli et al. [47] showed how to improve the scalability of a deadline-based scheduler in Linux by using proper data structures and lock-based synchronization.

1.2.2 Microkernels

Microkernels have been proposed in the research literature as an alternative to monolithic kernels [49, 81, 66]. In this architecture, a small code base (microkernel) provides essential services, such as physical memory management and protection, interrupt handling and task scheduling. Most of the classical services provided by a monolithic operating system, like network protocols, I/O device drivers, file sys-

[1]More information is available at: `http://lwn.net/Articles/398371/`.
[2]More information is available at: `http://lse.sourceforge.net/`.
[3]More information is available at: `http://lwn.net/Articles/191929/`.

tems, virtual memory management, are provided by special server processes running in user-space. In this way, the operating system is extremely modular, robust and secure. In microkernel based OSes, all services interact with each other and with applications through message passing.

Several microkernel architectures have been proposed in the research literature, most notably the Minix system by the A. Tanenbaum Group[4] and the L4 microkernels [49]. Unfortunately, only a few researchers have investigated the performance and scalability of microkernels on multicore systems. In L4 [49], the basic IPC mechanism is designed to be highly optimised for uni-processors. Moreover, it is not clear how to make the sharing of internal shared data structures, namely for interrupt forwarding and virtual memory, scalable in many-core environments.

Uhlig [81] presented an adaptive mechanism for sharing data structures between microkernels running on different cores, which is a combination of coarse and fine grain locking and Remote Procedure Call (RPC). Also, remote resources are treated differently from local resources.

The use of microkernel based OSes has been investigated also in the domain of supercomputing on multi-processor systems, like happened with the Amoeba [79], Mach [66] and Chorus [67] OSes. These investigations were born from the observation that monolithic OSes used to carry on lot of unneeded functionality on each and every node of a parallel machine, introducing unneeded overheads. Also, the standard communication primitives used on traditional OSes used to be highly inefficient in the context of a multi-processor machine. On the other hand, the approach typical of the HPC domain used to consist in not having a real OS, but rather a small run-time environment provided in the form of libraries. This was too minimalistic and used to lack potentially useful capabilities. So, the adoption of a microkernel based OS was considered a good trade-off between these worlds [78].

A few commercial operating systems were inspired by the microkernel architecture. Windows NT borrowed some of the ideas from the microkernel environment [80], however NT should be considered a hybrid between a monolithic kernel and a microkernel. Also, the Mac OS-X kernel, a.k.a., XNU [73], derives from the fusion of the Mach [66] and FreeBSD kernels[5]. However, only some kernel-level primitives and paradigms of Mach were kept, whilst the microkernel architecture has been dropped.

QNX Neutrino [77] was one of the first RTOSes to be available on embedded multicore platforms with a structure similar to a microkernel, with some of the essential scheduling services implemented directly by the kernel for efficiency reasons. OSE[6] is another microkernel-based RTOS. OSE was developed by ENEA, a spin-off of Ericsson AB. In OSE, real-time tasks are implemented as processes that communicate among them mainly through message passing paradigm. Since memory protection is enforced between processes, the OSE kernel is known for its fault-tolerant and high availability features, and it is widespread in telecommunication applications.

[4]More information is available at: `http://www.minix3.org`.
[5]More information is available at: `http://www.freebsd.org/`.
[6]More information is available at: `http://www.enea.com`.

1.2.3 Single-System Image

Single-System Image (SSI) Operating Systems have been designed in the context of cluster computing, for the purpose of making the usability and programmability of clusters easy. An SSI OS gives the application programmer the illusion that a cluster is a single computing system with a higher performance. The programmer writes parallel applications composed of many processes which communicate to each other by means of standard IPC mechanisms. Actually, the OS is capable of seamlessly distributing the workload over a distributed set of homogeneous machines interconnected by a network, migrating applications and data as required in order to make an efficient use of the underlying physical resources. The SSI Operating System concept has been implemented for example in Kerrighed [59], openMosix[7] (initially based on MOSIX [6], the project was officially closed in 2008) and OpenSSI[8]. All of them constitute variants of Linux, which add to the kernel the fundamental lacking features. A comparison among these approaches can be found for example in [53].

SSI systems aim to realise high-performance computing clusters preserving a local programming model that is unaware of the actual distribution of the load within the network. This allows parallel applications initially thought for multi-processor (or multi-core) systems to easily take advantage of the additional computing resources made available across the network, without any need to explicitly code the distribution logic. While being one of the main advantages of this kind of systems, it also constitutes its very limitation. The actually obtainable performance speed-up depends strongly on the communication patterns among the processes composing an application. However, the assumptions of the programmer about locality of data and processes are subverted when the application is deployed in an SSI cluster. The interaction overheads (now implying networking latencies) may sometimes nullify the potential advantages due to the increased available overall computing power, unless the application is carefully coded considering the deployment environment. This kind of problems may be mitigated by proper monitoring and migration strategies at the SSI kernel level, e.g., by trying to keep those processes which interact too frequently on the same machine.

It is also noteworthy to mention that there are approaches to exposing a SSI runtime to applications in a cluster which do not require any specific OS/kernel adaptation. For example, this has been done for the Java language, to allow for the seamless deployment of large parallel Java applications, with many threads but non-necessarily distributed, over a cluster of physical machines [87, 3, 86]. These approaches require the realization of proper mechanisms into the run-time of the language itself, and they usually do not require any special support from the OS/kernel.

[7]More information is available at: `http://www.openmosix.org`.
[8]More information is available at: `http://www.openssi.org`.

1.2.4 Operating Systems for Multi-Many Cores

Research on multiprocessor operating systems is active since a long time, much before the multi-core paradigm became so successful. Two broad classes of kernel organisation have been proposed by Lauer and Needham [45]: message-based and procedure-based approaches. In a procedure-based kernel, there is no fundamental distinction between a process in user space and kernel activities: each process performs kernel operations via system calls, and kernel resources are represented by shared data structures between processes. Conversely, in a message-based kernel, each major kernel resource is handled by a separate kernel process, and typical kernel operations require message exchanges. The procedure-based approach closely mimics the hardware organisation of Symmetric Multiprocessors (SMP) with Uniform Memory Access (UMA): this is the basic organisation underlying monolithic kernels. The message-based approach closely mimics the hardware organisation of a distributed memory multicomputer, and this is the basic organisation of microkernels. Chaves et al. [38] compared remote memory access versus remote invocation for kernel-kernel communication in a NUMA machine without cache coherency. In the first case, access to shared resources is performed by accessing remote memory using a remote locking mechanism; in the second case, it is achieved through invoking an operation on the remote node. The work is outdated, because of the advances in hardware architectures, however, some of the basic findings are of general validity: in particular, remote invocation is preferable for long operations, while remote memory access is preferable for short critical sections.

Many different papers [38, 45] have insisted on the tension between lock-based communication and synchronisation versus remote invocation. Depending on the underlying hardware architecture and on the different structure of the operating system, and depending on the requirements of the applications (performance, security, scalability, etc.) sometimes the lock-based approach seems to be the most appropriate, sometimes the remote-invocation approach proves to be the best approach.

Andrew Baumann et al. recently proposed an OS model called Multikernel [25], which advocates for independent kernel instances on the individual cores, allowed to interact solely via message passing. All kernel-level information and status data that needs to be shared among multiple cores is therefore replicated between them, and kept synchronised by explicit protocols. This way all communications among cores and processors need to be explicitly coded, and this naturally leads to asynchronous communication patterns, largely used in distributed systems, which enhance the possibility for the system to parallelise and pipeline activities, rather than having cores stall waiting for implicit cache coherence protocols run by the underlying hardware. Interestingly, in the Multikernel view, the fact that the OS does not rely on shared data does not preclude applications to be developed with a shared memory paradigm.

Also, Multikernel envisions a hardware-neutral OS model, where for example the part of a CPU driver in charge of handling the communications between different cores, may actually take advantage of available low-level information about the cores topology and their interconnection infrastructure. For example, different hardware-level mechanisms may be exploited in order to send messages among cores sharing a

L3 cache, as compared to the ones needed to send messages among cores that do not share such a cache, or reside on different processors.

The Multikernel model has been implemented as the Barrelfish[9] prototype, and preliminary measurements seem promising, especially on the side of scalability of certain critical operations involving all the cores (e.g., TLB shootdown). However, the experimental results available so far are to be considered as preliminary, due to the still incomplete implementation of the Multikernel concept.

Yuan et al. proposed GenerOS [85], a variation of the Linux kernel explicitly addressing heterogeneous multi-core systems. In GenerOS, the cache contention on the same core due to different types of activities going on within a system is reduced by means of partitioning the activities among the available cores: application cores, dedicated to running applications and exclusively user-space code; kernel cores dedicated to running exclusively the kernel-space part of the system calls invoked by the applications; interrupt cores dedicated to servicing interrupt requests. A set of modifications to the kernel are required to allow system calls to execute on a core different from the application core invoking the functionality. Also, kernel cores run one or more kernel servers. Each kernel server is dedicated to one or more system calls, it waits continuously for requests of that particular set of system calls from the application cores, and serializes their execution, avoiding any context switches among requests from different applications.

The fact that kernel-space code is handed over to different cores than the ones where the main applications code is running, together with the serialisation of kernel-space system call executions by the kernel servers, causes a decrease of the contention in accessing the cache, when compared with a plain Linux system, as shown by the experimental results performed by the authors. However, the serialisation of system calls execution is somewhat against the current trend in the Linux kernel: from the ancient ages in which the kernel-space code was non-preemptible, in recent years a lot of effort has been dedicated just for increasing preemptibility of kernel code, which is well-known to reduce latencies and improve responsiveness of the system. Even if the presence of multiple cores may mitigate such problem, the situation is not expected to be tremendously different when high workloads are in place with a nearly saturated system. Therefore, more investigations would be needed in order to understand what is the impact of the proposed OS model on various application classes, especially on interactive and real-time ones. An issue of the GenerOS model is constituted by the proper OS configuration in terms of balancing among the various types of cores, as well as the number of kernel servers and how system calls are distributed across them. In the initial prototype, the authors used a static configuration, but they also observed that this is one of the troublesome issues to be faced in their proposed OS model.

Boyd-Wickizer et al. proposed Corey [12], an Operating System designed from scratch around the need for allowing applications to make an efficient use of massively parallel and multi-core hardware. The authors highlight that kernel-level shared data structures may cause unneeded overheads when accessed from multiple cores, even

[9]More information is available at: `http://www.barrelfish.org`.

when the applications that are being run would not need to share any data. For example, process and file descriptor tables are potentially at risk of being contended among multiple cores, even when the applications running on them are accessing independent processes and files. Therefore, it is proposed to delegate the responsibility to decide what is shared across which cores as much as possible to the application. This is done via a specialised API allowing applications to define and control three main elements: shares are areas of scope either local to a core or global for a set of identifiers; address ranges are memory segments explicitly assigned to shares; kernel cores are cores dedicated to the execution of kernel code, i.e., interrupt handlers and the kernel-side part of system calls (which are handed over from the application cores to the kernel cores). Experimental results seem promising, in that a reduction of the overheads due to contentions on kernel-level data structures is achievable, at the cost of a little complexity for the application developer, who needs to properly set-up shares and address ranges.

Wentzlaff and Agarwal proposed the Factored Operating Systems (fos) [84], an OS model that builds on concepts taken from the distributed computing world, in order to reduce contention on kernel-level shared data structures. Specifically, the fos architecture foresees the partitioning of cores between applications and kernel services. Each kernel service is implemented by one or more specialised servers that run on kernel cores, and bits of service-based computing are reused for allowing each kernel service in distributing its workload to the available kernel servers, similarly to load-balancing techniques in web servers. The fos Operating System is still under development, and it is being entirely redesigned from scratch, based on a microkernel structure, where the concept of relegating OS functionality within specialised servers that communicate by message-passing mechanisms to each other and to applications is already in place. Also, in fos, each kernel core runs a single kernel server that enqueues requests in an input queue and services them in a serialised, non-preemptible way. This removes (in the opinion of the authors of this approach) the need for having traditional temporal scheduling of kernel cores. Actually, it is foreseen to have a form of cooperative scheduling, by which a kernel core, while serving a request, can yield explicitly the core so that it can serve other requests, while it waits for some device and/or other kernel servers to respond. The way kernel servers service applications requests is planned to be based on stateless protocols, so that subsequent requests of the same application can be potentially handed over to different kernel servers for a better load distribution across kernel cores.

Interesting investigations in this area have also been carried out recently by Schubert et al. [68, 69, 70, 71]. In [68], a Service-Oriented Operating System (SOOS) model is proposed, constituting an enabling technology for future distributed collaboration scenarios called Future Workspaces. In this work, it is suggested that the Operating System should possess a distributed and heterogeneous nature. A Main OS instance is the one offering the most complex services to applications, comprising process management, virtual memory management, I/O, networking, and a graphical user interface, whilst other OS instances exhibit a limited set of functionality focused/specialized on the capabilities locally available on the nodes they are running on. Specifically, it is envisioned that an Embedded Micro OS instance should

be used to directly control remote resources and devices, while a Standalone Micro OS instance should be used to expose access to virtualized resources made available through virtualization technology by some further Operating System. This structure will enable an unforeseen enhanced level of experience for mobility, where the actual resources (computational power, storage, data) will be maintained remotely through dedicated corporate server farms, thus greatly reducing administration efforts.

One advantage of such a structure is that the Operating System may easily embed the additional features needed by GRID applications [69], which usually are made available today by means of specialized extensions to the OS. SOOS introduces a novel resource provisioning concept [70] that differs from existing approaches of Grids and Clouds, in that it aims for making applications and computers more independent of the underlying hardware and for increasing mobility and performance.

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

Also, in a cloud environment, a vast amount of computational resources will be at reach of each process across the web or locally. Therefore, the SOOS concept calls for deep investigations into dynamic and intelligent processes (re)distribution policies according to resource availability and demand, and it proposes [71] a micro-kernel OS architecture model designed to compensate these deficits.

All these elements are under investigation in the context of the Service-oriented Operating Systems European project[10].

All of the above mentioned approaches to the (re-)engineering of the OS kernel model for dealing with massively parallel systems are of extremely interesting. However, these approaches are at a quite preliminary and conceptual stage, with only some of them having experimental prototype implementations. Therefore, these do not constitute consolidated approaches proved to be industrially viable, feasible and understandable for a wide audience. More research needs to be performed on this side, addressing scalability, efficiency and programmability issues for all of the sub-components of an OS kernel, and investigating on the achievable trade-offs between overall system and individual applications performance and responsiveness.

[10]More information is available on the project website: `http://www.soos-project.eu/`.

Finally, still there are researchers showing how traditional OS kernel architectures, e.g., as found in Linux, can be improved from a scalability viewpoint (see Section 1.2.1).

1.2.5 Operating Systems for GRIDs

Proposals have appeared in the literature for Operating Systems specifically targeted at supporting GRID systems. For example, Padala and Wilson proposed [62] a GRID OS that has the goal of providing a minimum set of services which are common to all GRID middleware infrastructures, still building on a traditional OS like Linux with as few changes as possible (in fact, additional features required at the kernel level are provided through Linux loadable kernel modules). The core functionality that needs to be added to the OS, according to the authors, is: high-performance I/O and networking, for example relying on copy-free communication primitives and fine-tuning of network stack parameters such as the TCP/IP window size; communication primitives with a better support for such mechanisms as MPI; resource management features allowing for resource discovery, allocation, monitoring; process management capabilities supporting for example global identifiers for processes, which may be used in the mentioned communication primitives for distributed IPC. The point that is made by the authors, and validated by the presented experimental results, is that implementing such services merely at the middleware level, outside the kernel, as commonly done in existing GRID middleware solutions, constitutes a bottleneck in the potentially achievable performance.

More recently, Puri and Abbas conceptualized [64] a GRID OS aimed to support GRID applications, by means of embedding within the OS itself such capabilities as: fault-tolerance and check-pointing, transparent access to distributed resources in a location-independent fashion, load balancing by means of migration of processes and virtualization, and scalability. However, the paper remains at a very abstract level, and it does not discuss practical implications of the envisioned architecture, such as what is required to be supported at the kernel level and what can be delegated to the OS middleware.

The XtreemOS European Project[11] produced XtreemOS [58], a variation of the Linux OS enhanced with Grid capabilities. The XtreemOS extensions to Linux include: LinuxSSI, a single-system image version of Linux based on Kerrighed [59] which allows to register into the XtreemOS Grid a cluster of systems virtually seen as a single, more powerful machine; XtreemFS [37], a networked file-system supporting automatic replication and high-availability of data; process checkpointing; a middleware for management of Grid nodes and submission of tasks; XOSAGA [26], an application-level API for Grid applications which constitutes an implementation of the abstract language-independent SAGA [29] specification, with some XtreemOS specific extensions.

[11] More information is available at: `http://www.xtreemos.eu/`.

Starting from release 10.4, the Mac OS-X Operating System embeds a simple Grid management middleware called Xgrid [31], which is immediately available on all installations of the OS, if the corresponding service is activated. Xgrid has a three-tier architecture: clients submit jobs to controllers, which in turn hand them over to agents for the actual processing. Clients may submit jobs to the Grid by means of either a command-line tool, or a dedicated API which is part of the OS foundations API. In order to share large amounts of data among Grid tasks, it is possible to use one of the available distributed filesystems, such as NFS. Xgrid is targeted to an easy set-up of Grids with no strong requirements on the number of interconnected nodes and complexity of the submitted jobs. For example, only linear workflows are supported, whereas for more complex Grid settings one can install on the OS one of the other more complex Grid management middleware solutions. Mac OS-X 10.5 introduced Xgrid 2, with some enhancements on the job scheduling decisions, such as the so called Scoreboard, i.e., a customizable scoring script that may be provided by clients in order to drive the decisions made by controllers about what agent nodes to choose when multiple ones are available. The script may base its score on the availability of particular capabilities of the agent node, or the connection/connectivity conditions, etc. This is useful for increasing the performance of the deployed applications.

1.2.6 Operating Systems for HPC

As detailed in section V. on high performance computing, in classical cluster systems, each processor (in the sense of smallest compute unit) hosts its own operating system environment. HPC jobs typically run on the system in an exclusive way, in order to achieve maximum performance. This means that the respective execution environment does not have to deal with scheduling issues, scale management etc. Essentially, the operating system therefore primarily serves the same purpose as a virtualisation system, i.e. it abstracts from the underlying hardware and deals with I/O of the system, in particular for accessing shared resources and communication between threads, respectively processes. This means implicitly that many of the functions in a general purpose operating system are obsolete for HPC usage and can (should) be removed from the kernel, in order not to produce unnecessary overhead.

As noted, it is thereby of particular relevance for efficient parallel computing that the specific characteristics of the hardware are exploited to their maximum potential, such as the memory architecture of the systems. Therefore, the operating system is typically specifically adapted to the environment, so as to reduce performance loss due to misalignment.

Essentially, most HPC providers therefore reduce the operating system to an essential minimum and adapt the kernel to the specific environment. Obviously, Linux is the primary choice in such cases, due to its open source nature. Microsoft Windows and even Apple Mac OS-X have been demonstrated to work for HPC clusters, too[12], yet their performance and the overhead for adaptation typically does not fulfill the

[12]More information is available at: `http://hpc.sourceforge.net/`.

expectations - as such, there are for example 475 Linux / UNIX based systems on the Top 500[13] list (in which systems are ranked by their performance on the LINPACK Benchmark [24, 63]), but only 5 Windows based ones (and none for Mac OS-X)[14]. With the increasing heterogeneity and hence divergence between supercomputer setups and at the same time the growing scale of affordable high performance machines, it is likely that this distribution will change slightly in the near future.

Currently, the most widely used Linux distributions are probably[15] Red Hat Linux, SUSE Linux Enterprise, Scientific Linux and CentOS. Red Hat Linux[16] and SUSE Linux Enterprise[17] are widely used[18] mainly due to the range of system architectures they support (namely: x86 32 and 64 bit, Itanium IA-64, PowerPC 32 and 64 bit), even though official Red Hat Linux releases are comparatively rare (latest official release was in 2003). The Scientific Linux[19]. distribution is typically preferred over the Red Hat distribution, which is 100% compatible with Red Hat Linux. As opposed to the Red Hat version, Scientific Linux however is available for free and adaptations to individual systems are maintained by the community rather than by Red Hat. Even though reliability is decreased this way, the distribution adapts quicker to new systems. Scientific Linux supports the following architectures: x86 32 and 64 bit, Itanium IA-64. CentOS[20] is another popular Linux distribution which bases on Red Hat Linux and is available free of charge. Like Scientific Linux it is mainly maintained by the community, yet the latest versions only support the x86 architectures thus making it less interesting in the future. At the time of writing, 7 machines in the top 500 made use of CentOS.

Even UNIX based operating systems are still in use on HPC clusters, as they generally scale quite well. With a few exceptions, they are mostly commercially distributed which makes them less attractive than Linux in particular in academic circles. The number of UNIX systems in the top 500 basically decreases, with the particular exception of AIX which ships with the IBM machines and supports in particular the PowerPC and the IA-64 architecture, making it attractive for the according clusters.

Though Lameter claims that Linux scales well enough for future large scale platforms [44], it must be noted that this is mostly true for the number of processes but not the number of processors [71, 25]. The authors furthermore claim that due to messaging overhead, a microkernel approach is not a feasible alternative to Linux, yet as Barham et al. could show, the messaging approach scales better than the Linux monolithic structure [25].

[13] More information is available at: `http://www.top500.org/`.
[14] More information is available at: `http://www.top500.org/stats/list/35/osfam`.
[15] More information at: `http://www.clusterbuilder.org/software/operating-system.php`.
[16] More information is available at: `http://www.redhat.com`.
[17] More information is available at: `http://www.novell.com/linux`.
[18] At the time of writing this, 18 machines in the top 500 use either distribution.
[19] More information is available at: `http://www.scientificlinux.org`.
[20] More information is available at: `http://www.centos.org`.

1.3 SCHEDULING

One of the core features of an Operating System is its ability to multiplex the access to the available physical resources to multiple processes/threads that run at the same time onto the system. Some resources may be managed in an exclusive way, i.e., only one application at a time is granted access to it. Other resources, and particularly processor(s) and disks, are managed in a shared way, so that the competing applications alternate in accessing the resource(s) according to some scheduling policy. For the CPU, General-Purpose OSes usually provide Round-Robin based scheduling, where the ready tasks alternate each other after a certain time-slice which may be fixed or dynamically changing. GPOSes have usually scheduling policies designed so as to achieve a high overall system throughput, and to serve processes on a Best-Effort (BE) basis, i.e., no guarantees can be provided to the individual applications. On the other hand, Real-Time OSes have usually scheduling policies which are capable of providing precise scheduling guarantees to the competing applications. To this purpose, RTOSes undertake an *admission-control* phase, in which a new process is accepted into the system only if its timing requirements may be fulfilled, and the ones of the already accepted processes are not disrupted.

1.3.1 Scheduling on Multiprocessor

When deciding which kind of scheduler to adopt in a multiple processor system, there are two main options: partitioned scheduling and global scheduling. In a partitioned scheduler, there are multiple ready queues, one for each processor in the system, and it is possible to leave a processor in idle state even when there are ready tasks needing to be executed. The placement of tasks among the available processors is a critical step, and doing it optimally is equivalent to the bin-packing problem, which is known to be NP-hard in the strong sense [43, 5]. This complexity is typically avoided using sub-optimal solutions provided by polynomial and pseudo-polynomial time heuristics (e.g., First Fit, Best Fit, etc.) [23, 48, 46, 52].

With a global scheduler, tasks are extracted from a single system-wide queue and scheduled onto the available processors. The load is thus intrinsically balanced, since no processor is idled as long as there is a ready task in the global queue. A class of algorithms, called Pfair schedulers [7], is able to ensure that the full processing capacity can be used, but unfortunately at the cost of a large run-time overhead.

Complications in using a global scheduler mainly relate to the cost of inter-processor migration, and to the kernel overhead due to the necessary synchronisation. Even if there are mechanisms that can reduce the migration cost, it could nevertheless cause a significant schedulability loss when tasks have a large associated context (i.e. data overhead). Therefore, the effectiveness of a global scheduler is rather dependent on the application characteristics and on the architecture in use.

In addition to the above classes, there are also intermediate solutions, like hybrid- and restricted-migration schedulers [14, 10], that limit the number of processors among which a task can migrate, or the possibilities that a task has to migrate (by

disabling preemption). This way, fewer cache misses are expected, and the cost of migration and context changes is lower.

1.3.2 Real-Time Scheduling

The traditional real-time scheduling research area focuses mainly on hard real-time systems, where deadlines are considered to be critical, in the sense that deadline misses cannot be tolerated, because they lead to the complete system failure and possible catastrophic consequences (i.e. losses of life).

However, real-time theory and methodologies are gaining applicability in the field of soft real-time systems, where applications possess precise timing and performance requirements, but occasional failures in meeting them may be easily tolerated by the system, causing a graceful degradation in the quality of the provided service.

1.3.2.1 Scheduling real-time task sets on multiprocessor platforms Only recently multiprocessing is receiving a significant attention from the real-time community, thanks to the increasing industrial interest in such platforms. While the scheduling problem for uni-processor systems has been widely investigated for decades, few of the results obtained for a single processor generalise directly to the multiple processor case [51].

Unfortunately, predicting the behaviour of a multiprocessor system requires in many cases a considerable computing effort. To simplify the analysis, it is often necessary to introduce pessimistic assumptions. This is particularly needed when modelling globally scheduled multiprocessor systems, in which the cost of migrating a task from a processor to another can significantly vary over time. The presence of caches and the frequency of memory accesses have a significant influence on the worst-case timely parameters that characterize the system. To bind the variability of these parameters, often real-time literature focuses on platforms with multiple processors but with no caches, or whose cache miss delays are known. Also, the cost of preemption and migration on multi-processor systems is a very important issue that still needs to be properly considered in real-time methodologies. Some research in the domain of hardware architectures moves towards partially mitigating such issues. Recently, a few architectures have been proposed that limit penalties associated to migration and cache misses, for example the MPCore by ARM. Some researchers have recently proposed hardware implementations of some parts of the operating system, allowing one to reduce the scheduling penalties of multiprocessor platforms [74].

1.3.2.2 Soft real-time scheduling Different scheduling algorithms have been proposed to support the specific needs of soft real-time applications. A first important class approximates the Generalized Processor Sharing concept of a fluid flow allocation, in which each application using the resource marks a progress proportional to its weight. Among the algorithms of this class, we can cite Proportional Share [75] and Pfair [7]. Similar are the underlying principles of a family of algorithms known as Resource Reservation schedulers [65]. In the Resource Kernels project [65],

the resource reservation approach has been successfully applied to different types of resources (including disk and network). The Resource Reservation framework has been adapted to partitioned multiprocessor systems in [8] and [9] for the Constant Bandwidth Server (CBS) and the Total Bandwidth Server (TBS) algorithms, respectively. Also, it has been proposed to let the scheduler automatically self-tune the best parameters for a running real-time application [17].

1.3.2.3 Scheduling of distributed real-time applications

The problem of designing scheduling parameters for distributed real-time applications has received a constant attention in the past few years. In [4], the authors introduce a notion of transaction for real-time databases characterized by periodicity and end-to-end constraints and propose a methodology to identify periods and deadlines of intermediate tasks. In [27], the activation periods of the intermediate tasks that comply with end-to-end real-time requirements are found by an optimization problem. In [39], the authors use performance analysis techniques to decide the bandwidth allocated to each task that attain a maximum latency and a minimum average throughput for a chain of computations.

Concerning modelling of timing requirements of real-time applications, usually models similar to synchronous data-flow networks [61] are used. As shown in [11], these models lend themselves to an effective code generation process, in which an offline schedule is synthesized that minimizes the code length and the buffer size. The models used in [19], [20] and [28] are also a special cases of synchronous data-flow, but, due to the inherently distributed and dynamic nature of the considered applications, the aim is not an optimized offline scheduling of activities, but rather an efficient on-line (run-time) scheduling mechanism. Finally, in [41] the problem of optimum deployment, over a physical heterogeneous network, of distributed real-time applications with computing and networking requirements subject to end-to-end response-time constraints is tackled by introducing a formalisation in terms of a Mixed-Integer Non-Linear Programming optimisation program, both in a deterministic and a probabilistic form.

1.3.3 Scheduling and Synchronisation

Synchronisation is an essential problem of concurrent programming, and it has received a great attention from the research community. The problems of concurrent access and of providing a consistent view of shared data structures can be solved in different ways, depending on the abstraction level and on the basic organization of the operating system and programming paradigm. The basic properties that correct concurrent programs must possess were described by Herlihy and Wing [36], and solutions have been proposed both for shared-memory and message-passing. Also, synchronisation is tightly coupled with scheduling.

When multiple tasks need to access a shared resource or data structure, they need to synchronise each other, in order to avoid performing the access at the same time. The basic synchronisation means is constituted by a binary semaphore, which, if already taken, causes the process attempting to acquire the lock to be suspended by

the scheduler, and be woken up later when the lock owner exits the critical section. However, a great research effort has been done in two very important domains: in the literature of real-time scheduling, it is important to ensure that the amount of time a process has to wait before acquiring a lock may be some how kept under control; also, in multi-processor and multi-core scheduling, if the acquired resource is held by a task on another processor, and the critical section is expected to be very short (like it happens quite often in the kernel of an OS), then suspending the current task and performing a context-switch might lead to unnecessary overheads, whilst other policies may be more convenient (e.g., spin-locking). Interestingly, the PREEMPT_RT branch of the Linux kernel has an option [54] for turning (almost) every spin-lock primitive used inside the kernel into a mutex.

In the shared-memory paradigm, all processors can access the same memory, uniformly (UMA machines) or non uniformly (NUMA machines). A lot of work has been done to improve the basic locking mechanisms. Mellor-Crummey and Scott [57] solved the problem of reducing contention on the shared bus by using separate spin variables for each processor on which each core performed busy waiting. Many papers have proposed improvements on this basic mechanism, e.g., to introduce time-out [72], to reactively change the lock behaviour [50], to balance overhead vs. latency using a mixed coarse/fine-grain locking strategy [83]. Mukherjee and Schwan [60] proposed an adaptive locking protocol that chooses between mutex locks and spin locks depending on the characteristics of the application.

A different approach consists in making a local copy of the data structure (or of part of it), modify it, and later try to commit the changes in the global copy without disrupting the linearisability property [36]. This class of approaches is usually referred to as lock-free or non-blocking or wait-free [34]. Many wait-free algorithms have been proposed for common data structures, like priority queues [76], or stacks [33]. Transactional memory has been proposed as a hardware-level support for wait-free mechanisms [35].

For data structures where reading is more frequent than writing/updating, special mechanisms have been proposed to reduce contention, such as the Read-Copy Update (RCU) mechanism [56], widely adopted within the Linux kernel for accessing critical shared lists. McKenney [55] provides a comparison of several locking techniques in the form of patterns, from different points of view: latency, memory bandwidth, memory size, granularity, fairness and read-write ratio. There is no clear winning strategy and every mechanism has its advantages and disadvantages. A similar comparison has been carried out by Anderson [2].

In the message-passing paradigm, each resource is assigned a server thread which exclusively performs operations on the data structures of the resource. Other threads (clients) must request the operation by issuing a remote invocation via an IPC. This organisation mimics distributed systems where nodes do not share memory. The MPI interface [30] is based on this paradigm.

Chavez et al. [16] perform a comparison of shared-memory communication based on spin-locks vs. remote invocation in a NUMA multiprocessor architecture. Another comparison has been proposed by Chandra et al. [15]. They highlight that the performance is highly dependent on the underlying hardware structure and memory

hierarchy (UMA or NUMA, presence of cache coherency, etc.). However, it can be generally said that message passing is more adequate for long critical operations on processors with high memory access delay, while shared memory is preferable on short critical section on local data structures. Clearly, it is possible to mix shared memory locking and remote invocation [42, 32]. Recently, Uhlig [82] proposed to use IPC or shared memory locking depending on the locality of the data structure with respect to the current node, and to use an adaptive locking mechanism depending on the level of contention.

Due to space reasons, we cannot overview the protocols to arbitrate the exclusive access to shared resources for real-time task sets. However, the interested reader can refer to [18] for more information.

1.3.4 Shared resources protocol in the Linux kernel

The design criteria and performance metrics mainly adopted by Linux kernel developers is overall system throughput and fairness. However, although real-time behaviour and predictability have not been a primary concern until now, Linux embeds a few features that are commonly included in real-time kernels, and the synchronization sub-system does not constitute an exception to that.

In a Linux system, support for mutual exclusive access to shared memory areas is provided at both kernel and user level. In the latter case, this is achieved by means of system libraries (e.g., the `glibc` and `pthreads` libraries).

Inside the kernel, critical sections can be protected mainly by spinlocks, mutexes and RT-mutexes. There are other means of regulating the access to sensible code, such as read-writer locks and RCU locks, but describing them in details is out of the scope of this chapter.

The Linux kernel includes the POSIX [1] RT-mutexes, which support the Priority Inheritance (PI) protocol for avoiding Priority Inversion, a well-known problem of systems scheduled under priority-based policies.

In mainline Linux, the only subsystem which uses RT-mutexes is the Fast Userspace muTEXes (`futex`) interface. A futex is a special implementation of locking primitives which, exploiting atomic instructions and memory coherence available on the underlying hardware, manages to handle the synchronisation entirely at the user-space level in those cases in which there is no contention. When, instead, task blocking and unblocking is needed, then futexes involve kernel-level operations.

In PREEMPT_RT, a kernel branch maintained by a small developer group led by Ingo Molnar, with the aim of making the kernel suitable for very low latency and real-time applications, things are quite different: in fact, when this patch is applied, most of the spinlocks and mutexes are turned into RT-mutex. Basically, at the cost of sacrificing part of the overall system performance, this branch of the kernel reduces significantly the duration of non-preemptable sections and enforces priority inversion avoidance, reducing the performance and predictability gap between Linux and classical real-time kernels.

REFERENCES

1. Ieee standard for information technology Ã¢ portable operating system interface (posix), ieee std 1003.1, 2004 edition. Available online at: `http://www.opengroup.org/onlinepubs/009695399`.
2. T. E. Anderson. The performance of spin lock alternatives for shared-memory multiprocessors. *IEEE Trans. Parallel Distrib. Syst.*, 1:6–16, January 1990.
3. Yariv Aridor, Michael Factor, and Avi Teperman. cjvm: A single system image of a jvm on a cluster. In *Proceedings of the 1999 International Conference on Parallel Processing*, ICPP '99, pages 4–, Washington, DC, USA, 1999. IEEE Computer Society.
4. N. C. Audsley, A. Burns, M. F. Richardson, and A. J. Wellings. Data consistency in hard real-time systems, 1993.
5. Neil C. Audsley and Konstantinos Bletsas. Fixed priority timing analysis of real-time systems with limited parallelism. In *16th Euromicro Conference on Real-Time Systems (ECRTS 2004)*, pages 231–238, Catania, Italy, 2004.
6. Amnon Barak, Shai Guday, and Richard Wheeler. The mosix distributed operating system, load balancing for unix. In *Lecture Notes in Computer Science*, volume 672. Springer-Verlag, 1993.
7. S. K. Baruah, N. K. Cohen, C. G. Plaxton, and D. A. Varvel. Proportionate progress: a notion of fairness in resource allocation. In *Proceedings of the twenty-fifth annual ACM symposium on Theory of computing*, STOC '93, pages 345–354, New York, NY, USA, 1993. ACM.

8. Sanjoy Baruah and Giuseppe Lipari. Executing aperiodic jobs in a multiprocessor constant-bandwidth server implementation. In *Proceedings of the 16th Euromicro Conference on Real-Time Systems*, pages 109–116, Washington, DC, USA, 2004. IEEE Computer Society.
9. Sanjoy Baruah and Giuseppe Lipari. A multiprocessor implementation of the total bandwidth server. *Parallel and Distributed Processing Symposium, International*, 1:40a, 2004.
10. Sanjoy K. Baruah and John Carpenter. Multiprocessor fixed-priority scheduling with restricted interprocessor migrations. *J. Embedded Comput.*, 1:169–178, April 2005.
11. Shuvra S. Battacharyya, Edward A. Lee, and Praveen K. Murthy. *Software Synthesis from Dataflow Graphs*. Kluwer Academic Publishers, 1996.
12. Silas Boyd-Wickizer, Haibo Chen, Rong Chen, Yandong Mao, Frans Kaashoek, Robert Morris, Aleksey Pesterev, Lex Stein, Ming Wu, Yuehua Dai, Yang Zhang, and Zheng Zhang. Corey: An operating system for many cores. In *8th USENIX Symposium on Operating Systems Design and Implementation*, 2008.
13. Silas Boyd-Wickizer, Austin T. Clements, Yandong Mao, Aleksey Pesterev, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. An analysis of linux scalability to many cores. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10, pages 1–8, Berkeley, CA, USA, 2010. USENIX Association.
14. John M. Calandrino, James H. Anderson, and Dan P. Baumberger. A hybrid real-time scheduling approach for large-scale multicore platforms. In *Proceedings of the 19th Euromicro Conference on Real-Time Systems*, pages 247–258, Washington, DC, USA, 2007. IEEE Computer Society.
15. Satish Chandra, James R. Larus, and Anne Rogers. Where is time spent in message-passing and shared-memory programs? In *Proceedings of the sixth international conference on Architectural support for programming languages and operating systems*, ASPLOS-VI, pages 61–73, New York, NY, USA, 1994. ACM.
16. Eliseu M. Chaves, Prakash Ch. Das, Thomas J. Leblanc, Brian D. Marsh, and Michael L. Scott. Kernel-kernel communication in a shared-memory multiprocessor. *Concurrency: Practice and Experience*, 5(3):171–191, 1993.
17. Tommaso Cucinotta, Fabio Checconi, Luca Abeni, and Luigi Palopoli. Self-tuning schedulers for legacy real-time applications. In *Proceedings of the* 5^{th} *European Conference on Computer Systems (Eurosys 2010)*, Paris, France, April 2010. European chapter of the ACM SIGOPS.
18. Tommaso Cucinotta, Giuseppe Lipari, Dario Faggioli, Fabio Checconi, Sunil Kumar, Rui Aguiar, JoÃ£o Paulo Barraca, Bruno Santos, Javad Zarrin, Jan Kuper, Christiaan Baaij, Lutz Schubert, Hans-Martin Kreuz, and Vincent Gramoli. S(o)os project deliverable d6.1 - state of the art. Available on-line on the S(o)OS website: `http://www.soos-project.eu/`., 7 2010.
19. Tommaso Cucinotta and Luigi Palopoli. Feedback scheduling for pipelines of tasks. In *Proceedings of the 10th international conference on Hybrid systems: computation and control*, HSCC'07, pages 131–144, Berlin, Heidelberg, 2007. Springer-Verlag.
20. Tommaso Cucinotta and Luigi Palopoli. Qos control for pipelines of tasks using multiple resources. *IEEE Transactions on Computers*, 59:416–430, 2010.
21. Alex Depoutovitch and Michael Stumm. Otherworld: giving applications a chance to survive os kernel crashes. In *Proceedings of the 5th European conference on Computer systems*, EuroSys '10, pages 181–194, New York, NY, USA, 2010. ACM.

22. Simon Derr, Paul Jackson, Christoph Lameter, Paul Menage, and Hidetoshi Seto. Cpusets. Available on-line at: `http://www.kernel.org/doc/Documentation/cgroups/cpusets.txt`.

23. Sudarshan K. Dhall and C. L. Liu. On a real-time scheduling problem. *OPERATIONS RESEARCH*, 26(1):127–140, 1978.

24. J. Dongarra, J. Bunch, C. Moler, and G. W. Stewart. *LINPACK Users Guide*. SIAM, Philadelphia, 1979.

25. Andrew Baumann et al. The multikernel: a new os architecture for scalable multicore systems. In *SOSP*, 2009.

26. Thilo Kielmann et al. Xtreemos project deliverable d3.1.6: Second prototype of xtreemos runtime engine, version 1.0.3, 1 2009.

27. Richard Gerber, Seongsoo Hong, and Manas Saksena. Guaranteeing real-time requirements with resource-based calibration of periodic processes. *IEEE Trans. Softw. Eng.*, 21:579–592, July 1995.

28. Steve Goddard and Kevin Jeffay. Managing latency and buffer requirements in processing graph chains. *The Computer Journal*, 44:200–1, 2001.

29. Tom Goodale, Shantenu Jha, Hartmut Kaiser, Thilo Kielmann, Pascal Kleijer, Gregor von Laszewski, Craig Lee, Andre Merzky, Hrabri Rajic, and John Shalf. Saga: A simple api for grid applications. high-level application programming on the grid. *Computational Methods in Science and Technology*, 12(1):7–20, 2006. Available on-line at: `Onlineat: http://wiki.cct.lsu.edu/saga/`.

30. William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. A high-performance, portable implementation of the mpi message passing interface standard. *Parallel Comput.*, 22:789–828, September 1996.

31. Olivier Michielin Hamid Hussain-Khan. Xgrid, a "just do it" grid solution for non it's. *EMBnet.news*, 11(3), 9 2005.

32. John Heinlein, Kourosh Gharachorloo, Scott Dresser, and Anoop Gupta. Integration of message passing and shared memory in the stanford flash multiprocessor. *SIGPLAN Not.*, 29:38–50, November 1994.

33. Danny Hendler, Nir Shavit, and Lena Yerushalmi. A scalable lock-free stack algorithm. In *Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures*, SPAA '04, pages 206–215, New York, NY, USA, 2004. ACM.

34. Maurice Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13:124–149, January 1991.

35. Maurice Herlihy and J. Eliot B. Moss. Transactional memory: architectural support for lock-free data structures. *SIGARCH Comput. Archit. News*, 21:289–300, May 1993.

36. Maurice P. Herlihy and Jeannette M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12:463–492, July 1990.

37. Felix Hupfeld, Toni Cortes, Björn Kolbeck, Jan Stender, Erich Focht, Matthias Hess, Jesus Malo, Jonathan Marti, and Eugenio Cesario. The xtreemfs architecture - a case for object-based file systems in grids. *Concurr. Comput. : Pract. Exper.*, 20:2049–2060, December 2008.

38. Eliseu M. Chaves Jr., Prakash Das, Thomas J. LeBlanc, Brian D. Marsh, and Michael L. Scott. Kernel-kernel communication in a shared-memory multiprocessor. *Concurrency - Practice and Experience*, 5(3):171–191, 1993.

39. Dong-In Kang, Richard Gerber, and Manas Saksena. Parametric design synthesis of distributed embedded systems. *IEEE Transactions on Computers*, 49:1155–1169, 2000.

40. Andi Kleen. Linux multi-core scalability. 16th Linux Kongress. Available on-line at: `http://www.linux-kongress.org/2009/abstracts.html#3_4_1`., 10 2009.

41. Kleopatra Konstanteli, Dimosthenis Kyriazis, Theodora Varvarigou, Tommaso Cucinotta, and Gaetano Anastasi. Real-time guarantees in flexible advance reservations. *Computer Software and Applications Conference, Annual International*, 2:67–72, 2009.

42. David Kranz, Kirk Johnson, Anant Agarwal, John Kubiatowicz, and Beng-Hong Lim. Integrating message-passing and shared-memory: early experience. In *Proceedings of the fourth ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPOPP '93, pages 54–63, New York, NY, USA, 1993. ACM.

43. Pramote Kuacharoen, Mohamed A. Shalan, and Vincent J. Mooney III. A configurable hardware scheduler for real-time systems. In *in Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms*, pages 96–101. CSREA Press, 2003.

44. C. Lameter. Extreme high performance computing or why microkernels suck. In *Proceedings of the Linux Symposium*, 2007.

45. H.C. Lauer and R.M. Needham. On the duality of operating system structures. *ACM SIGOPS Operating System Review*, 13(2):3–19, 4 1979.

46. Sylvain Lauzac, Rami Melhem, and Daniel Mossé. An improved rate-monotonic admission control and its applications. *IEEE Trans. Comput.*, 52:337–350, March 2003.

47. Juri Lelli, Giuseppe Lipari, Dario Faggioli, and Tommaso Cucinotta. An efficient and scalable implementation of global edf in linux. In *(to appear) Proceedings of International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT 2011)*, Porto, Portugal, July 2011.

48. Jörg Liebeherr, Almut Burchard, Yingfeng Oh, and Sang H. Son. New strategies for assigning real-time tasks to multiprocessor systems. *IEEE Trans. Comput.*, 44:1429–1442, December 1995.

49. Jochen Liedtke. On μ-kernel construction. In *15th ACM symposium on Operating Systems Principles (SOSP)*, pages 237–250, 12 1995.

50. Beng-Hong Lim and Anant Agarwal. Reactive synchronization algorithms for multiprocessors. In *Proceedings of the sixth international conference on Architectural support for programming languages and operating systems*, ASPLOS-VI, pages 25–35, New York, NY, USA, 1994. ACM.

51. C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20:46–61, January 1973.

52. J. M. López, J. L. Díaz, and D. F. García. Utilization bounds for edf scheduling on real-time multiprocessor systems. *Real-Time Syst.*, 28:39–68, October 2004.

53. Renaud Lottiaux, Benoit Boissinot, Pascal Gallard, Geoffroy VallÃ©e, and Christine Morin. OpenMosix, OpenSSI and Kerrighed: A Comparative Study. Technical Report 5399, INRIA, 11 2004.

54. Paul McKenney. A realtime preemption overview. Available on-line at: `http://lwn.net/Articles/146861/`.

55. Paul E. McKenney. Selecting locking primitives for parallel programming. *Commun. ACM*, 39:75–82, October 1996.

56. Paul E. McKenney and John D. Slingwine. Read-copy update: Using execution history to solve concurrency problems. Available on-line at: `http://www.liblfds.org/wikipedia/index.php/White_Papers`.

57. John M. Mellor-Crummey and Michael L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Syst.*, 9:21–65, February 1991.

58. Christine Morin, Yvon Jégou, Jérôme Gallard, and Pierre Riteau. Clouds: A New Playground for the XtreemOS Grid Operating System. *Parallel Processing Letters (PPL)*, 19:435–449, 2009.

59. Christine Morin, Renaud Lottiaux, Geoffroy VallÃ©e, Pascal Gallard, David Margery, Jean-Yves Berthou, and Isaac D. Scherson. Kerrighed and data parallelism: Cluster computing on single system image operating systems. In *6th IEEE International Conference on Cluster Computing*, pages 277–286, San Diego, California, 9 2004.

60. B.C. Mukherjee and K. Schwan. Improving performance by use of adaptive objects: experimentation with a configurable multiprocessor thread package. In *High Performance Distributed Computing, 1993., Proceedings the 2nd International Symposium on*, pages 59 –66, July 1993.

61. Walid A. Najjar, Edward A. Lee, and Guang R. Gao. Advances in the dataflow computational model. *Parallel Comput.*, 25:1907–1929, December 1999.

62. Pradeep Padala and Joseph N. Wilson. Gridos: Operating system services for grid architectures. In *Proceedings of the International Conference on High-Performance Computing (HiPC 2003)*, 2003.

63. A. Petitet, R. C. Whaley, J. Dongarra, and A. Cleary. *HPL - A Portable Implementation of the High-Performance Linpack Benchmark for Distributed-Memory Computers*, 9 2008.

64. Sanjeev Puri and Dr. Qamas Abbas. Grid operating system: Making dynamic virtual services in organizations. *International Journal of Computer Theory and Engineering*, 2(1):96–102, 2 2010.

65. Raj Rajkumar, Kanaka Juvva, Anastasio Molano, and Shuichi Oikawa. Readings in multimedia computing and networking. chapter Resource kernels: a resource-centric approach to real-time and multimedia systems, pages 476–490. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.

66. Richard Rashid, Daniel Julin, Douglas Orr, Richard Sanzi, Robert Baron, Alessandro Forin, David Golub, and Michael Jones. Mach: A System Software kernel. In *Proceedings of the 34th Computer Society International Conference (COMPCON 89)*, 2 1989.

67. M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrmann, C. Kaiser, S. Langlois, P. LÃ©onard, and W. Neuhauser. Overview of the CHORUS Distributed Operating Systems. *Computing Systems*, 1:39âŁ“–69, 1991.

68. L. Schubert, A. Kipp, B. Koller, and S. Wesner. Service oriented operating systems: Future workspaces. *IEEE Wireless Communications*, 16:42–50, 2009.

69. Lutz Schubert and Alexander Kipp. Principles of service oriented operating systems. In Ozgur Akan, Paolo Bellavista, Jiannong Cao, Falko Dressler, Domenico Ferrari, Mario Gerla, Hisashi Kobayashi, Sergio Palazzo, Sartaj Sahni, Xuemin Shen, Mircea Stan, Jia Xiaohua, Albert Zomaya, Geoffrey Coulson, Pascale Vicat-Blanc Primet, Tomohiro Kudoh, and Joe Mambretti, editors, *Networks for Grid Applications*, volume 2 of *Lecture*

Notes of the Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering, pages 56–69. Springer Berlin Heidelberg, 2009.

70. Lutz Schubert, Alexander Kipp, and Stefan Wesner. *Above the Clouds: From Grids to Service-oriented Operating Systems*, pages 238 – 249. IOS Press, 2009.

71. Lutz Schubert, Stefan Wesner, Alexander Kipp, and Alvaro Arenas. Self-managed microkernels: From clouds towards resource fabrics. In Ozgur Akan, Paolo Bellavista, Jiannong Cao, Falko Dressler, Domenico Ferrari, Mario Gerla, Hisashi Kobayashi, Sergio Palazzo, Sartaj Sahni, Xuemin (Sherman) Shen, Mircea Stan, Jia Xiaohua, Albert Zomaya, Geoffrey Coulson, Dimiter R. Avresky, Michel Diaz, Arndt Bode, Bruno Ciciani, and Eliezer Dekel, editors, *Cloud Computing*, volume 34 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, pages 167–185. Springer Berlin Heidelberg, 2010.

72. Michael L. Scott and William N. Scherer. Scalable queue-based spin locks with timeout. In *Proceedings of the eighth ACM SIGPLAN symposium on Principles and practices of parallel programming*, PPoPP '01, pages 44–52, New York, NY, USA, 2001. ACM.

73. Amit Singh. What is Mac OS X? Available on-line at: `http://osxbook.com/book/bonus/ancient/whatismacosx/arch\_xnu.html`., 12 2003.

74. J. Starner, J. Adomat, J. Furunas, and L. Lindh. Real-time scheduling co-processor in hardware for single and multiprocessor systems. *EUROMICRO Conference*, 0:0509, 1996.

75. I. Stoica, H. Abdel-Wahab, K. Jeffay, S. K. Baruah, J. E. Gehrke, and C. G. Plaxton. A proportional share resource allocation algorithm for real-ti time-shared systems. In *Proceedings of the 17th IEEE Real-Time Systems Symposium*, RTSS '96, pages 288–, Washington, DC, USA, 1996. IEEE Computer Society.

76. Håkan Sundell and Philippas Tsigas. Fast and lock-free concurrent priority queues for multi-thread systems. *J. Parallel Distrib. Comput.*, 65:609–627, May 2005.

77. QNX Software Systems. The qnx neutrino microkernel. Available on-line at: `http://www.qnx.com/developers/docs/6.3.2/neutrino/sys_arch/kernel.html`.

78. Andrew S. Tanenbaum. A comparison of three microkernels. *Journal of Supercomputing*, 9, 1995.

79. Andrew S. Tanenbaum, M. Frans Kaashoek, Robbert Van Renesse, and Henri E. Bal. The amoeba distributed operating system âŁ“ a status report. *Computer Communications*, 14:324âŁ“–335, 1991.

80. Microsoft TechNet. Windows nt 4.0 workstation architecture. Available on-line at: `http://technet.microsoft.com/en-us/library/cc749980.aspx`.

81. Volkmar Uhlig. The mechanics of in-kernel synchronization for a scalable microkernel. *Operating Systems Review*, 41(4):49–58, 2007.

82. Volkmar Uhlig. The mechanics of in-kernel synchronization for a scalable microkernel. *SIGOPS Oper. Syst. Rev.*, 41:49–58, July 2007.

83. Ronald C. Unrau, Orran Krieger, Benjamin Gamsa, and Michael Stumm. Experiences with locking in a numa multiprocessor operating system kernel. In *Proceedings of the 1st USENIX conference on Operating Systems Design and Implementation*, OSDI '94, Berkeley, CA, USA, 1994. USENIX Association.

84. David Wentzlaff and Anant Agarwal. Factored Operating Systems (fos): The Case for a Scalable Operating System for Multicores. *ACM SIGOPS Operating System Review:*

Special Issue on the Interaction among the OS, Compilers, and Multicore Processors, 4 2009.

85. Qingbo Yuan, Jianbo Zhao, Mingyu Chen, and Ninghui Sun. GenerOS: An Asymmetric Operating System Kernel for Multi-core Systems. In *IEEE International Parallel & Distributed Processing Symposium*, Atlanta, USA, 4 2010.
86. Hua Zhang, Joohan Lee, and Ratan Guha. Vcluster: a thread-based java middleware for smp and heterogeneous clusters with thread migration support. *Softw. Pract. Exper.*, 38:1049–1071, August 2008.
87. Wenzhang Zhu, Cho-Li Wang, and Francis C. M. Lau. Jessica2: A distributed java virtual machine with transparent thread migration support. *Cluster Computing, IEEE International Conference on*, 0:381, 2002.