2015 IEEE International Conference on Cloud Engineering

A Container-Based Approach to OS Specialization for Exascale Computing

Judicael A. Zounmevo*, Swann Perarnau*, Kamil Iskra*, Kazutomo Yoshii*,
Roberto Gioiosa†, Brian C. Van Essen‡, Maya B. Gokhale‡, Edgar A. Leon‡
*Argonne National Laboratory {jzounmevo@, perarnau@mcs., iskra@mcs., kazutomo@}anl.gov
†Pacific Northwest National Laboratory. Roberto.Gioiosa@pnnl.gov
‡Lawrence Livermore National Laboratory. {vanessen1, maya, leon}@llnl.gov

***Abstract*—Future exascale systems will impose several conflicting challenges on the operating system (OS) running on the compute nodes of such machines. On the one hand, the targeted extreme scale requires the kind of high resource usage efficiency that is best provided by lightweight OSes. At the same time, substantial changes in hardware are expected for exascale systems. Compute nodes are expected to host a mix of general-purpose and special-purpose processors or accelerators tailored for serial, parallel, compute-intensive, or I/O-intensive workloads. Similarly, the deeper and more complex memory hierarchy will expose multiple coherence domains and NUMA nodes in addition to incorporating nonvolatile RAM. That expected workload and hardware heterogeneity and complexity is not compatible with the simplicity that characterizes high performance lightweight kernels. In this work, we describe the Argo Exascale node OS, which is our approach to providing in a single kernel the required OS environments for the two aforementioned conflicting goals. We resort to multiple OS specializations on top of a single Linux kernel coupled with multiple containers.**

***Keywords*-OS Specialization; Lean; Container; Cgroups**

I. Introduction

Disruptive new computing technology has already begun to change the scientific computing landscape. Hybrid CPUs, manycore systems, and low-power system-on-a-chip designs are being used in today's most powerful high-performance computing (HPC) systems. As these technology shifts continue and exascale machines close in, the *Argo* research project aims to provide an operating system and runtime (OS/R) designed to support extreme-scale scientific computations. It aims to efficiently leverage new chip and interconnect technologies while addressing the new modalities, programming environments, and workflows expected at exascale.

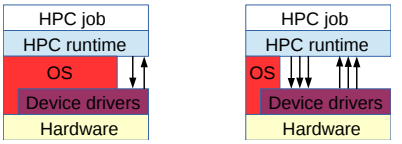
At the heart of the project are four key innovations: dynamic reconfiguring of node resources in response to workload changes, allowance for massive concurrency, a hierarchical framework for power and fault management, and a cross-layer communication protocol that allows resource managers and optimizers to communicate and control the platform. These innovations will result in an open-source prototype system that runs on several architectures. It is expected to form the basis of production exascale systems deployed in the 2018–2020 timeframe.

The design is based on a hierarchical approach. A global view enables Argo to control resources such as power or interconnect bandwidth across the entire system, respond to system faults, or tune application performance. A local view is essential for scalability, enabling compute nodes to manage and optimize massive intranode thread and task parallelism and adapt to new memory technologies. In addition, Argo introduces the idea of "enclaves," a set of resources dedicated to a particular service and capable of introspection and autonomic response. Enclaves will be able to change the system configuration of nodes and the allocation of power to different nodes or to migrate data or computations from one node to another. They will be used to demonstrate the support of different levels of fault tolerance—a key concern of exascale systems—with some enclaves handling node failures by means of global restart and other enclaves supporting finer-level recovery.

We describe here the early stages of an ongoing effort, as part of Argo, to evolve the Linux kernel into an OS suitable for exascale nodes. A major step in designing any HPC OS involves reducing the interference between the OS and the HPC job. Consequently, we are exposing the hardware resources directly to the HPC runtime, which is a user-level software layer. However, the increase in resource complexity expected for the next-generation supercomputers creates a need for some hardware management that is best left to the OS. The resource complexity comes from the heterogeneous set of compute cores and accelerators, coupled with deeper and more complex memory hierarchies with multiple coherence and NUMA domains to cope with both the CPU heterogeneity and the massive intranode parallelism. Our design is based on the concept of OS Specialization—splitting the node OS into several autonomous components each managing its own subset of CPU cores, a memory region, etc. We introduce the concepts of *ServiceOS* which is a fully-fledged Linux environment meant for node management and legacy application execution, and *Compute Containers* which realize *lean* OS environments meant for running HPC applications with little interference from the OS.

The rest of the paper is organized as follows. Section II provides motivation for putting forth OS specialization in the Argo NodeOS. Section III describes our approach to OS specialization using containers, as well as how Compute Containers can achieve various levels of leanness over the same fully-fledged Linux kernel used by the ServiceOS. Section IV discusses related work and Section V concludes.

978-1-4799-8218-9/15 $31.00 © 2015 IEEE
DOI 10.1109/IC2E.2015.78



(a) Traditional HPC OS with limited OS-bypass

(b) Lean HPC OS with substantial OS-bypass

Figure 1. Traditional HPC OS vs. lean OS with most resources exposed to the HPC runtime

II. Establishing the Need for OS Specialization

A. Lean OS

Recent 3.x Linux kernels have more than 15 millions lines of code (LoC). With only about 60,000 LoC, the IBM Compute Node Kernel (CNK) [1] of Mira, a Blue Gene/Q supercomputer at Argonne National Laboratory, contrasts with Linux by being barely more than a thin layer between the hardware and the HPC job. The CNK LoC number includes the kernel, the management network stack, all the supported system calls, and limited file system facilities. CNK is tailored to achieving the minimum possible interference between the HPC job and its hardware usage; in doing so, it provides the maximum possible hardware efficiency to the HPC job. As shown for collective operations [2], low OS interference can make a noticeable difference in HPC job performance at extreme scales; in that respect, the lightweight nature of kernels such as CNK is a compelling characteristic for an exascale OS.

Contemporary HPC stacks already selectively bypass the OS for direct access to certain devices such as RDMA-enabled network cards (Fig. 1(a)). The Argo project goes further by exposing most hardware resources to the HPC runtime which executes as part of the application (Fig. 1(b)). On top of fulfilling the minimal interference goal, offloading the hardware resource management of the application from the OS to the HPC runtime is justified by the runtime being more knowledgeable about the needs of the HPC application than the OS. The OS as seen by the HPC application running in the Argo environment is said to be *lean*.

B. Provisioning for Legacy Applications

A look at the list of the 500 most powerful supercomputers (Top500) [3], as of November 2014, shows that such systems are overwhelmingly using Linux. In fact, Fig. 2 shows that more than half of the Top500 systems have consistently been using Linux for the past decade, and the share has kept growing. This observation implies that there is a substantial amount of HPC code that assumes the existence of familiar programming APIs such as POSIX, as well as most of the system calls that Linux programmers take for granted. A purely lean OS such as CNK is not a flexible environment that would easily host the massive amount of existing legacy code. For instance, CNK does not support process forking; in fact, the 63 system calls offered by CNK represent only a very limited subset of what is offered by any Linux kernel of the last 10 years. CNK does not provide any complex or dynamic virtual memory to physical memory mapping. Similarly, it does not provide all the thread preemption scenarios of a vanilla Linux kernel. In particular, the absence of time quantum-like sharing can prevent the well-known approach to nonblocking operations that is fulfilled by hidden agents backed by a middleware-level thread that runs in addition to the application threads [4].

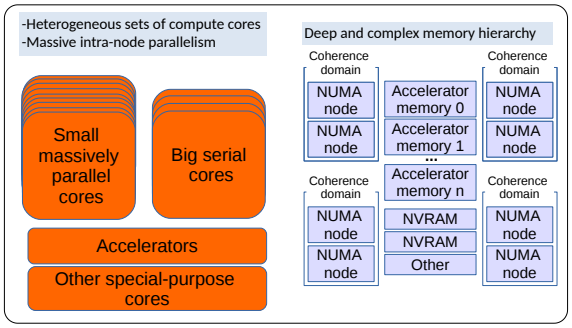


Figure 3. Resource heterogeneity and complexity in exascale nodes

While the overall Argo NodeOS seeks the leanness of a lightweight kernel, there is provision for supporting legacy applications that need a full Linux environment. An aspect of the OS specialization is to provide side-by-side in the same node:

- an OS environment for HPC applications that require a fully-fledged Linux kernel,
- an OS environment that is lean. As much as possible, that environment would allow the HPC runtime to get direct access to the hardware resources with little or no OS interference.

C. Provisioning for Heterogeneous Resources

Substantial changes in hardware are expected for exascale systems. The deeper and more complex memory hierarchy will expose multiple coherence domains and NUMA nodes in addition to incorporating NVRAM. The envisioned multiple coherence and NUMA domains are the consequence of the expected increase in CPU core density and processor heterogeneity at the node level. That resource complexity (Fig. 3) requires a hardware management rigor that is best left to a fully-fledged kernel such as Linux. Hardware drivers are not always open-source, and hardware vendors are reluctant to provide drivers for niche operating systems. However, hardware drivers, especially for HPC-directed devices, are routinely provided for Linux because it is a well-established OS in HPC [3]. As a consequence, the Argo NodeOS requires a fully-fledged Linux kernel as its underlying OS but also as one of its OS specializations.

D. Provisioning for Different Compute Needs

Nob all HPC jobs have the same needs, and their underlying processes do not exhibit the same behavior. HPC jobs could differ by communication patterns, I/O requirements, storage needs, computation-intensiveness, etc. For instance, a job that performs large numbers of small message

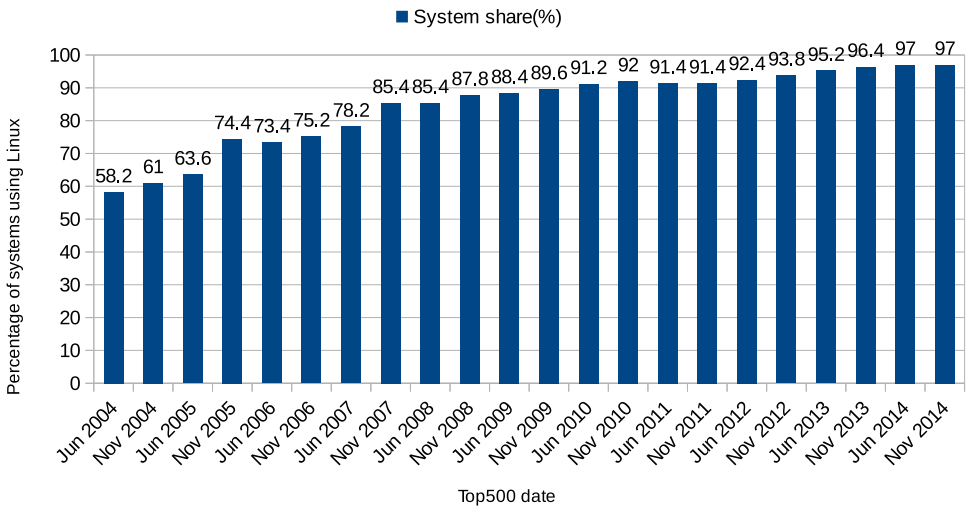


Figure 2. Percentage of the Top500 supercomputers using Linux

exchanges would be more sensitive to the latency or the leanness of the network stack than one that does heavier data transfers. While some jobs require massive external storage for the sake of reading input or generating output, other expect to start and complete entirely in memory. Some HPC applications could require the ability to oversubscribe the node with more threads than CPU cores so as to efficiently achieve an optimal balance between blocking operations and asynchronous needs. Taking that disparity of needs into account, there is no single set of lean OS characteristics that could fulfill all the possible compute needs. Actually, any such single set of characteristics is bound to lead to over-provisioning of features that could hardly keep the OS lean. As a result, there is not just a need for distinction between a fully-fledged Linux environment and a leaner OS environment providing low-interference HPC; there is also a need for specializing between various lean compute environments. Thus, the OS specialization in the Argo project allows OS features to be enabled or disabled according to the desired level of leanness expressed by the HPC job being run.

Distinct compute needs exist because different jobs are dissimilar, but distinct compute needs can exist for a single, composite job as well. While it is uncommon in HPC to collocate two or more distinct jobs on the same compute node, and most jobs have the same compute requirements for all their processes on the same node, there is a growing trend of modular HPC jobs where a compute aspect collaborates with a post-processing aspect in real time. An example of such post-processing could be co-visualization. As the compute and the co-visualization aspects do not have the same needs, in spite of being part of the same job, their respective optimal compute environments could differ widely. The same job can therefore require its processes to be distributed over several distinct OS specializations, even within the same node.

III. OS Specialization via Compute Containers

As shown in Fig. 4, the NodeOS is made of a unique ServiceOS and of one or more Compute Containers. The ServiceOS—which is the default OS specialization—boots the node, initializes the hardware resources, and provides most of the legacy API and services expected by an HPC job that assumes a fully-fledged Linux environment. An HPC application with lean OS requirements does not execute over the ServiceOS; instead, it executes over one or multiple Compute Containers which are OS specializations distinct from the ServiceOS.

A single-kernel approach is used to achieve the specialization. The ServiceOS is the default host OS, and the Compute Containers are the guest entities based on Linux cgroups. Compute Containers get their hardware allocation in bulk from the ServiceOS and then expose these resources, with little interference, to the user-space HPC runtime. For instance, scheduling policies, CPU core allocation, and memory management are to a large extent delegated to the runtime. The ServiceOS hosts a resource manager which manipulates cgroups and resource controllers. The ServiceOS also deals with storage management, job management, and system call forwarding for select calls that are by default disabled in the Compute Containers.

Compute Containers have exclusive ownership of certain resources that they are granted by the ServiceOS; even the ServiceOS refrains from using these resources during the lifetime of the owning Compute Containers. A Compute Container can be as fully-fledged as the ServiceOS since the same kernel is shared between the two kinds of OS specializations. However, by default, a Compute Container is created with many features disabled, so as to achieve a certain default level of leanness (Fig. 5). Further features can be selectively enabled or disabled by providing parameters to the resource manager either at the creation time of a

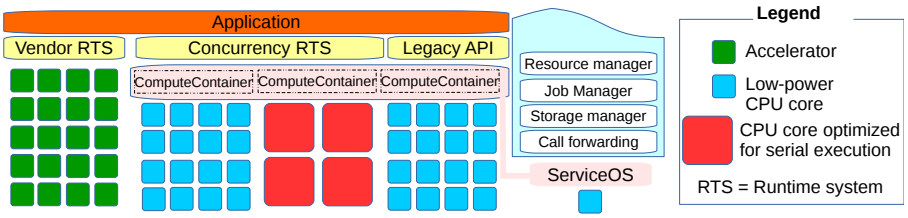


Figure 4. OS specialization

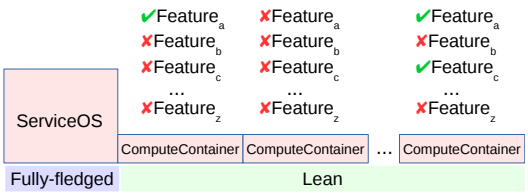


Figure 5. Differentiated behavior between ServiceOS and Compute Containers

Compute Container or during its lifetime.

The concept of a container on Linux usually comes with boundaries similar to virtual machine isolation. Compute Container, as put forth by the Argo NodeOS, does not put any emphasis on namespace isolation. Processes running on distinct Compute Containers share the same PID space, file system, and network domains. In fact, the NodeOS proactively needs Compute Containers to be permeable, so the same job using multiple Compute Containers can seamlessly have its processes communicate without crossing any virtual node boundary. The emphasis in this work is put on resource partitioning and differentiated behaviors by means of feature activation/deactivation.

A. Provisioning for Compute Container Feature Selection

The mechanism depicted in Fig. 5 is already partially feasible on a vanilla Linux kernel. For instance, by disabling load balancing in the cpuset resource controller for a cgroup, it is possible to reduce the OS interference over the subset of CPU cores dedicated to a Compute Container. When the ServiceOS is kept away from the Compute Container CPU cores via the isolcpus kernel parameter, the deactivation of load-balancing noticeably reduces OS interference and leaves the CPU core management to the HPC runtime. These available features were already an important steps in realizing a certain level of Compute Container leanness. The level of feature activation/deactivation offered by a vanilla Linux kernel is insufficient though, and we are developing additional mechanisms to further Compute Container Specialization.

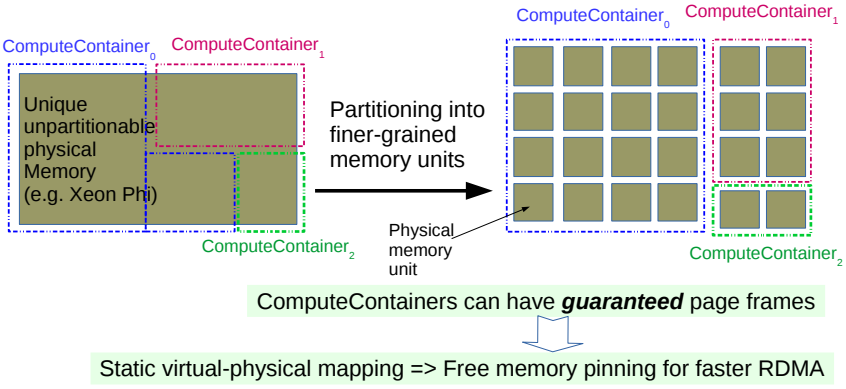
A new scheduling class is under development for Compute Containers. It is optimized for small per-core process and thread counts. It disables load balancing and replaces process preemption with cooperative scheduling, thus providing a considerably more predictable performance to workloads optimized for a single software thread per hardware thread, as is common in HPC.

The memory registration required by Remote Direct Memory Access (RDMA) is known to be costly [5]. However, if a process can have its virtual address space pre-registered, memory registration would become virtually free at run time and would lead to an overall performance boost for the RDMA operations that are common in supercomputing inter-node communication. For a process to benefit from such static mapping, it must own the physical memory frames exclusively. However, with the existing cpuset resource controller, the physical memory ownership granularity is the NUMA node. Thus, in a uniform memory access (UMA) architecture such as the Xeon Phi, where the whole memory is considered a single memory node, it becomes impossible for a collocated Compute Container to achieve the aforementioned exclusive memory ownership. We are adding a new resource controller that breaks down physical memory into smaller units that can be granted exclusively to Compute Containers (Fig. 6). The units are sized in multiples of a page size. For a UMA architecture such as Xeon Phi, Fig. 6(a) shows how the new feature can allow memory partitioning. The equivalent of the same partitioning for NUMA architectures is shown in Fig. 6(b). For NUMA architectures, the finer-grained memory units of a Compute Container do not have to be contiguously allocated from the same NUMA node. For instance, in Fig. 6(b), ComputeContainer$_1$ does not have to get all its finer-grained memory units from NUMA node 2, especially if it hosts a process that needs to use interleaved memory allocation policy.

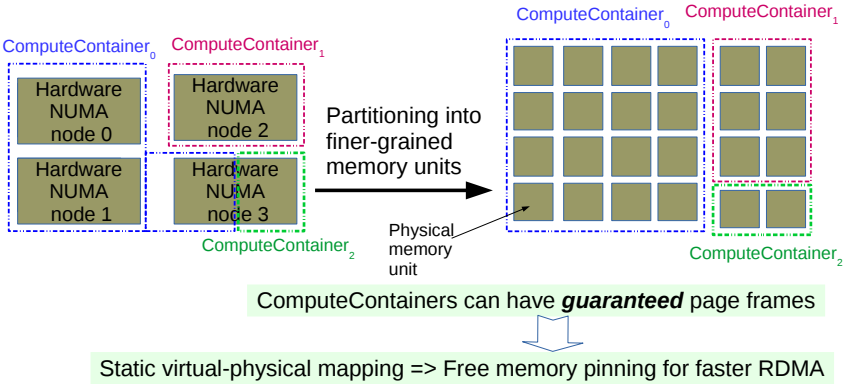
For certain threads or processes, the runtime can have very low tolerance for preemption; and would therefore expect to be hosted, as much as possible, in a Compute Container that prevents the scheduling of OS-related kernel threads. We are providing an OS specialization knob that can disable select per-CPU core kernel threads on the cores used by any such Compute Container. This OS specialization feature is appropriate when the functionality provided by a disabled per-CPU core kernel threads is not required for the set of functionalities needed for a lean Compute Container.

IV. Related Work

While the idea of isolating processes from each other is not new (see `chroot` or `jails` from BSD), containers are becoming a popular solution in the commercial space as a lightweight virtualization technology. Docker [6] or LXC [7] provide a framework to specify, launch, and control an environment isolated from the base system through various



(a) Finer memory unit (UMA)



(b) Finer memory unit (NUMA)

Figure 6. Breaking physical memory nodes in finer logical nodes that can exclusively be owned by Compute Containers

namespaces and to manage resources between those environments. However, these technologies aim at different goals than us. Their containers are focused on isolation—a process cannot access resources outside its container—and resource sharing management—how much memory or CPU to give to a given container—while we focus on exclusive partitioning of heterogeneous resources and OS specialization inside containers. Furthermore, in our context, all processes running in a single node are part of a single application. Thus, the job does not benefit from these processes being hosted in isolated containers where communication must cross virtual node boundaries instead of using straighforward efficient means such as shared memory.

CoreOS [8] is a minimalist OS based on the Linux kernel, using Docker at its core to run each application inside a container and providing additional software to control a cluster deployment. As HPC systems usually depend on architecture- and vendor-specific base system images, the Argo project's intent is to build loosely integrated components (Fig. 4) that can later be deployed and configured on a large variety of machines instead of a tightly integrated solution.

In recent years, several research projects have studied radical changes to the OS stack to adapt to large, heterogeneous or homogeneous resources on single nodes. Corey [9] put forth a single kernel that delegates specific kernel functions to specific cores, and allowing applications to specify when kernel data structures should be shared and across which part of the system. Multikernels are also being investigated and put forth by many others. Barrelfish [10] for instance is implemented as a distributed system, with one kernel instance per core and efficient communication between the different kernel instances. Another instance of multikernel approach is presented by Baumann et al. [11], with the goal, among other concerns, of showing that multiple OSes over multiple partitions of the node hardware resources can offer a means of enforcing intra-node scalability by learning from inter-node distributed systems. Mint [12] and Popcorn [13] are two other multikernels which are meant for managing parallel intra-node resources with perfectly autonomous kernels. Popcron Linux allows the independent kernels to provide a seamless feeling of single system image where processes can easily cross kernel boundaries. Finally, Twin-Linux [14] appears as an approach to OS specialization via multiple fully-fledged kernels meant for different workloads. We argue here for a middle-ground approach, with Compute Containers using dedicated resources, having special configurations, and some kernel functions forwarded to the ServiceOS. Furthermore, by using the Linux kernel as a basis, we ensure compatibility with vendor-controlled

architectures. One crucial difference between the aforementioned multikernels and the Argo NodeOS is the need to host the same job over multiple OS specializations at low cost.

V. Conclusion

The extreme scale of next generation supercomputers will noticeably benefit from reduced OS interference in the HPC job execution. Furthermore, hardware resource micromanagement is best offloaded from the OS to the HPC runtime, for the latter is more informed of the needs of the HPC job. These two goals are reminiscent of lightweight kernels such as the Blue Gene CNK. However, the complex and heterogeneous nature of the next generation supercomputing systems and the need to support the massive amount of legacy applications that assume fully-fledged POSIX APIs and a variety of system services establish the relevance of an existing and well-adopted OS like Linux. The NodeOS aspect of the Argo project is an ongoing effort to simultaneously provide the leanness of a lightweight kernel for extreme performance and the richness required to support legacy applications and disparate hardware. We are leveraging the cgroups and resource controller interface of Linux to implement OS specialization on top of a single kernel. New resource controllers are being added to complement those that are provided by the mainline Linux kernel.

Acknowledgements

This work was supported by the Office of Advanced Scientific Computer Research, Office of Science, U.S. Department of Energy, under Contract DE-AC02-06CH11357.

References

[1] T. Budnik, B. Knudson, M. Megerian, S. Miller, M. Mundy, and W. Stockdell, "Blue Gene/Q resource management architecture," in *Many-Task Computing on Grids and Supercomputers (MTAGS), 2010 IEEE Workshop on*, Nov. 2010.

[2] P. Beckman, K. Iskra, K. Yoshii, and S. Coghlan, "The influence of operating systems on the performance of collective operations at extreme scale," in *Cluster Computing, 2006 IEEE International Conference on*, Sep. 2006, pp. 1–12.

[3] "Top500," http://www.top500.org/.

[4] T. Hoefler and A. Lumsdaine, "Message progression in parallel computing – to thread or not to thread?" in *Proceedings of the 2008 IEEE International Conference on Cluster Computing (Cluster)*, 2008, pp. 213–222.

[5] F. Mietke, R. Rex, R. Baumgartl, T. Mehlan, T. Hoefler, and W. Rehm, "Analysis of the memory registration process in the Mellanox InfiniBand software stack," in *Euro-Par 2006 Parallel Processing*, ser. Lecture Notes in Computer Science, W. Nagel, W. Walter, and W. Lehner, Eds. Springer Berlin Heidelberg, 2006, vol. 4128, pp. 124–133.

[6] "Docker," http://www.docker.com/.

[7] "LXC: Linux container tools," http://www.linuxcontainers.org/.

[8] "CoreOS: Linux for massive server deployments," http://www.coreos.com/.

[9] S. Boyd-Wickizer, H. Chen, R. Chen, Y. Mao, F. Kaashoek, R. Morris, A. Pesterev, L. Stein, M. Wu, Y. Dai, Y. Zhang, and Z. Zhang, "Corey: An operating system for many cores," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI '08)*, 2008, pp. 43–57.

[10] A. Schüpbach, S. Peter, A. Baumann, T. Roscoe, P. Barham, T. Harris, and R. Isaacs, "Embracing diversity in the Barrelfish manycore operating system," in *Proceedings of the Workshop on Managed Many-Core Systems*, 2008.

[11] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania, "The Multikernel: A new OS architecture for scalable multicore systems," in *Proceedings of the 22nd ACM SIGOPS Symposium on Operating Systems Principles (SOSP '09)*, 2009, pp. 29–44.

[12] Y. Nomura, R. Senzaki, D. Nakahara, H. Ushio, T. Kataoka, and H. Taniguchi, "Mint: Booting multiple Linux kernels on a multicore processor," in *Broadband and Wireless Computing, Communication and Applications (BWCCA), 2011 International Conference on*, Oct. 2011, pp. 555–560.

[13] A. Barbalace, B. Ravindran, and D. Katz, "Popcorn: A replicated-kernel OS based on Linux," in *Proceedings of the Linux Symposium*, 2014, pp. 123–138.

[14] J. Adhiraj, P. Swapnil, N. Mandar, R. Swapnil, and P. Kiran, "Twin-Linux: Running independent Linux kernels simultaneously on separate cores of a multicore system," in *Proceedings of the Linux Symposium*, 2010, pp. 101–107.