Seminar HPC Trends
Winter Term 2017/2018

# New Operating System Concepts for High Performance Computing

Fabian Dreer
Ludwig-Maximilians Universität München
dreer@cip.ifi.lmu.de

January 2018

## Abstract

When running large-scale applications on clusters, the noise generated by the operating system can greatly impact the overall performance. In order to minimize overhead, new concepts for HPC OSs are needed as a response to increasing complexity while still considering existing API compatibility.

In this paper we study the design concepts of heterogeneous kernels using the example of *mOS* and the approach of library operating systems by exploring the architecture of *Exokernel*. We summarize architectural decisions, present a similar project in each case, *Interface for Heterogeneous Kernels* and *Unikernels* respectively, and show benchmark results where possible.

Our investigations show that both concepts have a high potential, reduce system noise and outperform a traditional Linux in tasks they are already able to do. However the results are only proven by micro-benchmarks as most projects lack the maturity for comprehensive evaluations at the point of this writing.

## 1 The Impact of System Noise

When using a traditional operating system kernel in high performance computing applications, the cache and interrupt system are under heavy load by e.g. system services for housekeeping tasks which is also referred to as noise. The performance of the application is notably reduced by this noise.

Even small delays from cache misses or interrupts can affect the overall performance of a large scale application. So called *jitter* even influences collective communication regarding the synchronization, which can either absorb or propagate the noise. Even though asynchronous communication has a much higher probability to absorb the noise, it is not completely unaffected. Collective operations suffer the most from propagation of jitter especially when implemented linearly. But it is hard to analyse noise and its propagation for collective operations even for simple algorithms. Hoefler et al. [5] also suggest that "at large-scale, faster networks are not able to improve the application speed significantly because noise propagation is becoming a bottleneck." [5]

Hoefler et al. [5] also show that synchronization of point-to-point and collective communication and

OS noise are tightly entangled and can not be discussed in isolation. At full scale, when it becomes the limiting factor, it eliminates all advantages of a faster network in collective operations as well as full applications. This finding is crucial for the design of large-scale systems because the noise bottleneck must be considered in system design.

Yet very specialized systems like BlueGene/L [8] help to avoid most sources of noise [5]. Ferreira et al. [3] show that the impact is dependent on parameters of the system, the already widely used concept of dedicated system nodes alone is not sufficient and that the placement of noisy nodes does matter.

This work gives an overview of recent developments and new concepts in the field of operating systems for high performance computing. The approaches described in the following sections are, together with the traditional Full-Weight-Kernel approach, the most common ones.

The rest of this paper is structured as follows. We will in the next section introduce the concept of running more than one kernel on a compute or service node while exploring the details of that approach at the example of *mOS* and the *Interface for Heterogeneous Kernels*. Section 3 investigates the idea of library operating systems by having a look at *Exokernel*, one of the first systems designed after that concept, as well as a newer approach called *Unikernels*. Section 4 investigates *Hermit Core* which is a combination of the aforementioned designs. After a comparison in Section 5 follows the conclusion in Section 6.

# 2 Heterogeneous Kernels

The idea about the heterogeneous kernel approach is to run multiple different kernels side-by-side. Each kernel has its spectrum of jobs to fulfill and its own dedicated resources. This makes it possible to have different operating environments on the partitioned hardware. Especially with a look to heterogeneous architectures with different kinds of memory and multiple memory controllers, like the recent Intel Xeon Phi architecture, or chips with different types of cores and coprocessors, specialized kernels might help to use the full potential available.

We will first have an in-depth look at *mOS* as at this example we will be able to see nicely what aspects have to be taken care of in order to run different kernels on the same node.

## 2.1 mOS

To get a light and specialized kernel there are two methods typically used: The first one is to take a generic Full-Weight-Kernel (FWK) and stripping away as much as possible; the second one is to build a minimal kernel from scratch. Either of these two approaches alone does not yield a fully Linux compatible kernel, which in turn won't be able to run generic Linux applications [4].

Thus the key design parameters of *mOS* are: full linux compatibility, limited changes to Linux, and full Light-Weight-Kernel scalability and performance, where performance and scalability are prioritized.

To avoid the tedious maintenance of patches to the Linux kernel, an approach inspired by FUSE has been taken. Its goal is to provide internal APIs to coordinate resource management between Linux and Light-Weight-Kernels (LWK) while still allowing each kernel to handle its own resources independently.

"At any given time, a sharable resource is either private to Linux or the LWK, so that it can be managed directly by the current owner." [11] The resources managed by LWK must meet the following requirements: i) to benefit from caching and reduced TLB misses, memory must be in physically contiguous regions, ii) except for the ones of the applications no interrupts are to be generated, iii) full control over scheduling must be provided, iv) memory regions are to be shared among LWK processes, v) efficient access to hardware must

be provided in userspace, which includes well-performing MPI and PGAS runtimes, vi) flexibility in allocated memory must be provided across cores (e.g. let rank0 have more memory than the other ranks) and, vii) system calls are to be sent to the Linux core or operating system node.

*mOS* consists of six components which will be introduced in one paragraph each:

According to Wisniewski et al. [11], the Linux running on the node can be any standard HPC Linux, configured for minimal memory usage and without disk paging. This component acts like a service providing Linux functionality to the LWK like a TCP/IP stack. It takes the bulk of the OS administration to keep the LWK streamlined, but the most important aspects include: boot and configuration of the hardware, distribution of the resources to the LWK and provision of a familiar administrative interface for the node (e.g. job monitoring).

The LWK which is running (possibly in multiple instantiations) alongside the compute node Linux. The job of the LWK is to provide as much hardware as possible to the applications running, as well as managing its assigned resources. As a consequence the LWK does take care of memory management and scheduling [11].

A transport mechanism in order to let the Linux and LWK communicate with each other. This mechanism is explicit, labeled as *function shipping*, and comes in three different variations: via shared memory, messages or inter-processor interrupts. For shared memory to work without major modifications to Linux, the designers of *mOS* decided to separate the physical memory into Linux-managed and LWK-managed partitions; and to allow each kernel read access to the other's space. Messages and interrupts are inspired by a model generally used by device drivers; thus only sending an interrupt in case no messages are in the queue, otherwise just queueing the new system call request which will be handled on the next poll. This avoids floods of interrupts in bulk-synchronous programming. To avoid jitter on compute cores, communication is in all cases done on cores running Linux [11].

The capability to direct system calls to the correct implementor (referred to as *triage*). The idea behind this separation is that performance critical system calls will be serviced by the LWK to avoid jitter, less critical calls, like signaling or `/proc` requests handles the local Linux kernel and all operations on the file system are offloaded to the operating system node (OSN). But this hierarchy of system call destinations does of course add complexity not only to the triaging but also to the synchronization of the process context over the nodes [11].

An offloading mechanism to an OSN. To remove the jitter from the compute node, avoid cache pollution and make better use of memory, using a dedicated OSN to take care of I/O operations is already an older concept. Even though the design of *mOS* would suggest to have file system operations handled on the local linux, the offloading mechanism improves resource usage and client scaling [11].

The capability to partition resources is needed for running multiple kernels on the same node. Memory partitioning can be done either statically by manipulating the memory maps at boot time and registering reserved regions; or dynamically making use of hotplugging. These same possibilities are valid for the assignment of cores. Physical devices will in general be assigned to the Linux kernel in order to keep the LWK simple [11].

We have seen the description of the *mOS* architecture which showed us many considerations for running multiple kernels side-by-side. As the design of *mOS* keeps compatibility with Linux core data structures, most applications should be supported. This project is still in an early development stage, therefore an exhaustive performance evaluation is not feasible at the moment.

## 2.2 Interface for Heterogeneous Kernels

This project is a general framework with the goal to ease the development of hybrid kernels on many-

core and accelerator architectures; therefore *attached* (coprocessor attached to multi-core host) and *builtin* (standalone many-core platform) configurations are possible. It follows the two design principles of keeping the interface minimal on the one hand, and providing a requisite utility library for kernels on the other hand.

Similar to *mOS* yet less strict, IHK defines the requirements for a hybrid kernel approach to be i) management of kernels and an interface to allocate resources, ii) resource partitoning and, iii) a communication mechanism among kernels. In IHK it is assumed that one kernel manages at most one processor.

The framwork consists of the following components:

IHK-Master has the ability to boot other kernels. The mechanisms needed to do so are the same as discussed in the architecture description of *mOS* about partitioning resources. The master kernel also makes the user interface available.

IHK-Slave defines an interface for slave kernels to work with each other. Kernels of this type only run in their assigned space, retrieve that information from, and are booted by, the master kernel.

The IHK-IKC (communication model) provides rudimentary functions for the use of channels, where a channel is a pair of message queues. Master and slave kernels have an interface to use the IKC. This slightly differs from what we've seen in *mOS*, as IHK provides a library for using interrupts and queuing where the exact implementation is free. The included IKC library provides functions to setup a client-server layout among the kernels with a master channel to share control messages [10].

For easier development of LWKs, a bootstrap library is part of IHK. Currently an implementation for x86_64 is available. The delegation of system calls works in concept exactly like we've seen in *mOS*, with the difference that there is no operating system node where file system operations can be sent to.

"While IHK/McKernel is in a more advanced phase than *mOS* at this moment, both projects are too early in their development cycle for doing an exhaustive performance study." [4]

To still have an idea what can be expected from the heterogeneous kernel approach, Figure 1 shows benchmark results from *FusedOS* [9], which was the first prototype incorporating the idea. The work of Wisniewski et al. [11] is also based on *FusedOS*, therefore the overall tendency should be comparable.

The x-axis of Figure 1 shows time while the y-axis shows the number of iterations performed during a certain quantum of time. We see the performance of the FusedOS PECs (Power-Efficient-Cores) — which can be thought of as the LWKs of *mOS* — in purple above the red Linux. High-frequency noise as well as occasional large spikes can be seen in the Linux curve. Especially these large spikes are detrimental to the performance on large-scale clusters. In comparison, the *FusedOS* PEC curve has the form of a straight line, thus not displaying any spikes; for that reason we would tend to believe that the behavior of the application running on *FusedOS* is deterministic [9].

To sum up, even though some prototypes of heterogeneous kernels are still in their early phases, the concept itself looks promising. Light-Weight-Kernels run almost completely deterministically and show superior noise properties.

# 3 Library Operating Systems

The job of a traditional OS is to abstract away the hardware and isolate different processes, owned by potentially multiple users, from one another, as well as from the kernel. This abstraction is commonly realized by the differentiation, and therefore separation, into kernel space and user space.

But as this makes the abstraction fix, this concept can also limit performance and freedom of implementation. As a result applications are denied the possibility of domain-specific optimizations; this

presetting also discourages changes to the abstractions.

Conceptually a library operating system (libOS) is built around an absolutely minimalistic kernel, which exports all hardware resources directly via a secure interface. The operating system, which implements higher level abstractions, uses this interface.

Therefore the (untrusted) OS lives entirely in user space, which effectively moves the whole resource management to user space. This has the advantage that parts like e.g. virtual memory are user defined and offer more flexibility as well as specialization to the application. Another strong point of this design is the reduction of context switches for privileged operations the kernel would normally have to execute [2].

In contrast to the previously described heterogeneous kernel approach, libraryOS concepts work with exactly one kernel which is then used by multiple libOSs. Similar to our investigation of the heterogeneous kernel approach, we will discuss first the concept of *Exokernel* in detail, then take a look at the younger variants named *Unikernels*.

## 3.1 Exokernel

The challenge for the exokernel approach is to give the libOSs maximal freedom while still secluding them in such a way that they do not affect each other. A low-level interface is used to separate protection from management. The kernel performs the important tasks of i) keeping track of resource ownership, ii) guarding all resource binding points and usage as well as, iii) revoking resource access.

In order to protect resources without managing them at all, the designers of *Exokernel* decided to make the interface in such a way that all hardware resources could be accessed as directly as possible because the libOS knows best about its needs. This is supposed to be possible by exposing allocation, physical names, bookkeeping data structures and revocation. Additionally a policy is needed to handle competing requests of different libOSs. In case of an exokernel the decisions to make are all about resource allocation and are handled in a traditional manner with e.g. reservation schemes or quotas. In the following paragraphs we will have a closer look at some other mechanisms of the exokernel [2] architecture.

Exokernel uses a technique referred to as *secure bindings* in order to multiplex resources so that they are protected against unintended use by different libOSs. The point in time where a libOS requests allocation of a resource is called *bind time*, subsequent use of that resource is known as *access time*. By doing authorization checks only at *bind time* this mechanism improves efficiency. Another aspect is that this way of handling checks strengthens the separation of management and protection as the kernel does not need to know about the complex semantics of resources at *bind time*. *Secure bindings* are implemented with hardware mechanisms, caching in software and the download of application code into the kernel. As this downloading mechanism is not as common as the other two, an example can be found in the paragraph about network multiplexing in this section.

The multiplexing of physical memory is done with *secure bindings* as well. When the libOS requests a page of memory, the kernel creates a binding for that page with additional information on the capabilities of this page. The owner of a page is allowed to manipulate its capabilities, and these capabilities are used to determine the access rights for that memory page. Therefore applications can grant memory access to other applications which makes resource sharing easier.

Multiplexing of network resources efficiently is rather hard with the design philosophy requiring separation of protection, which includes delivering packets to the correct libOS, and management, e.g. creating connections and sessions. To deliver the packets correctly it is necessary to understand the logic of their contents. This can be done by either requesting each possible recipient (every libOS), or, more efficiently, with the use of downloaded application code in the packet filter to handle the packet.

This code can be run with immediate execution on kernel events which avoids costly context switches or otherwise required scheduling of each application. As this code is untrusted, it should be combined with security mechanisms like sandboxing [2].

Finally there must be a way to reclaim allocated resources. For this a *resource revocation protocol* has been implemented. Typically a kernel does not inform the OS when physical memory is allocated or deallocated. But the design of exokernels strives to give the libOS the most direct access to hardware possible. Therefore the revocation is visible for most resources which allows the libOS to react to a revocation request accordingly by e.g. saving a certain state. In cases where the application becomes unresponsive there is an *abort protocol* which can be understood as issuing orders instead of requests. Still if the libOS cannot comply, *secure bindings* to allocated resources must be broken by force. To complement this behavior, the libOS actively releases resources no longer needed.

In sum we have seen that exokernel approaches do not provide the same functionality as a traditional OS, but offer a way of running specialized systems with high performance implemented mostly in user space. Engler et al. [2] already show that the concept of exokernels and their implementation can be very efficient, and that it is efficient as well to build traditional OS abstractions on application level. Yet, as the experiments are quite a few years old already, we will investigate more recent approaches based on this concept in the next subsection.

## 3.2 Unikernels

As it is difficult to support a wide range of real-world hardware with the exokernel approach, libOSs have never been widely deployed. This problem can be solved with the use of virtualization hypervisors which are already very popular today especially in cloud environments [7].

The key difference between the previously shown exokernel architecture and a unikernel is that unikernels are single-purpose, single-image and single-address-space applications that are, at compile-time, specialized into standalone kernels. To make the deployment of unikernels easier, the configuration is integrated into the compilation process. In comparison, Linux distributions rely e.g. on complex shell scripting to pack components into packages. When deployed to e.g. a cloud platform, unikernels get sealed against modifications. In return they offer significant reduction in image size, improved efficiency and security, and should also reduce operational costs [7].

In the following paragraphs we will now have a closer look at the architecture of such a single-purpose application, often referred to as an *appliance*.

Application configurations are usually stored in dedicated files, one for each service. The view of services in the unikernel architecture is not the one of independent applications but are seen as libraries of one single application. As a result the configuration is either done at build time for static parameters or with library calls for dynamic ones. This concept eases the configuration of complex layouts and also makes configurations programmable, analyzable and explicit.

Another advantage of linking everything as a library, even functions that would normally by provided by an operating system, results in very compact binary images. The system can by optimized as a whole without including unnecessary functionalities. And the static evaluation of linked configurations helps to eliminate dead code segments. This compile time specialization is also a measure of security, especially effective in the combination with the isolating hypervisor and possibly, as it is the case for the work of Madhavapeddy et al. [7], type-safe languages.

A special protection mechanism made possible by the design of unikernels is *sealing*. When the appliance starts up, it allocates all the memory it needs in a set of page tables with the policy that no page is writable and executable. After this a call to the hypervisor is used to seal these pages, which in turn makes the heap size fixed. The hypervisor has to be

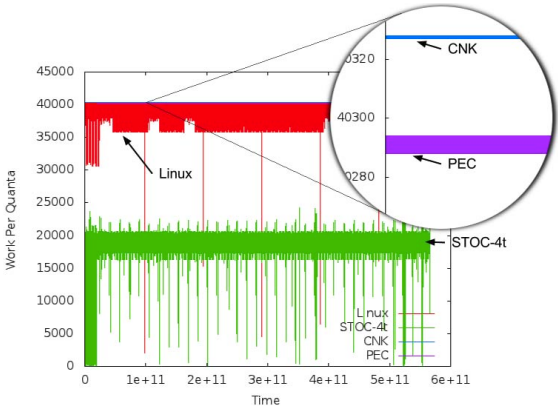



Figure 1: Fixed Time Quanta benchmark for Linux and FusedOS; adapted from Park et al. [9].

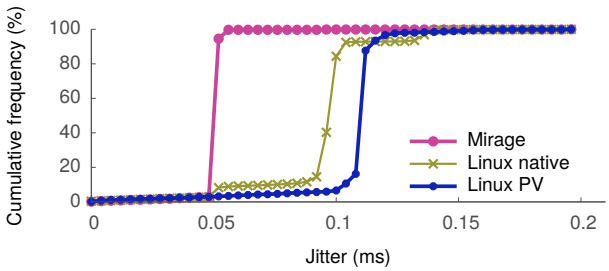



Figure 2: Figure 2: CDF of jitter for $10^6$ parallel threads in Linux and Mirage OS; adapted from Madhavapeddy et al. [7].

modified in order to provide the sealing mechanism. This action makes all code injection attacks ineffective. The use of this technique is optional. The second security mechanism for unikernels is possible because most of the time a reconfiguration requires recompilation. Therefore the address space layout can be randomized at compile time [7].

Madhavapeddy et al. [7] show with their *Mirage* prototype that CPU-heavy applications are not affected by the virtualization "as the hypervisor architecture only affects memory and I/O." [7]

According to Briggs et al. [1] the performance of Mirage OS, the prototype of Madhavapeddy et al. [7], is not easily evaluated as e.g. the DNS server as well as the HTTP server are still example skeletons. They are missing important features or are unstable. But the evaluation of the Mirage DNS server nevertheless showed that it is able to handle much higher request rates than a regularly used one on Linux. Mirage OS might need some more time to mature but shows other advantageous results, such as lower and also more predictable latency which can be seen in Figure 2 [7].

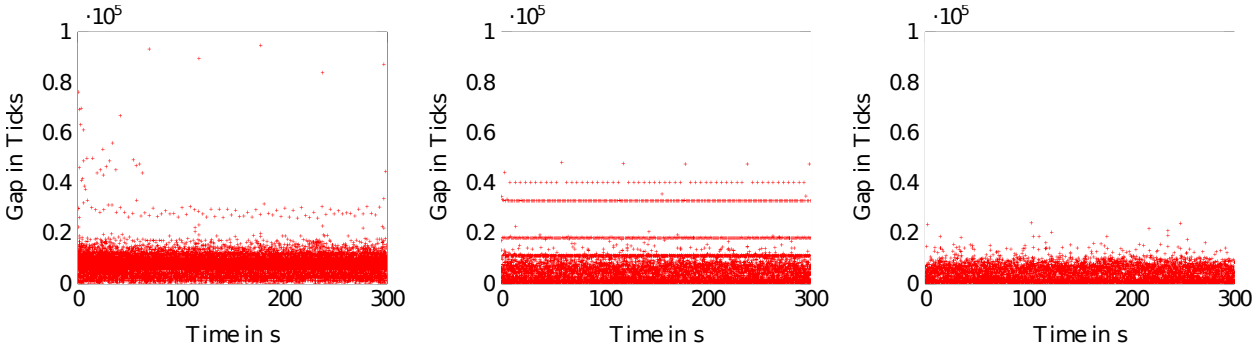
In conclusion, the drawback of the Mirage prototype is that it only runs specifically ported applications written in OCaml, like the system itself. But by this design it provides "type-safety and static analysis at compile-time." [1]

# 4 Hermit Core

*Hermit Core* is the combination of the approaches we have seen above. It combines a Linux kernel with a unikernel and promises maximum performance and scalability. Common interfaces and non-performance critical tasks are realized by Linux [6]. But as this project is focused on HPC programming models (e.g. MPI, OpenMP), performance has been improved in exchange for full POSIX compliance.

*Hermit Core* is extremely versatile. It can be run as a heterogeneous kernel, standalone like a pure unikernel, in a virtualized environment as well as directly on hardware as single- or multi-kernel. It can be booted without a Linux kernel directly by virtualization proxies, but in multi-kernel mode a special loader is required.

When running as a multi-kernel, one instance of *Hermit Core* runs on each NUMA node abstracting this fact so the application is presented a traditional UMA architecture.




(a) Linux (b) Hermit Core (c) Hermit Core without IP thread

Figure 3: Scatter plots observing OS noise in different configurations adapted from Lankes et al. [6].

The communication between the kernel instances are realized either by using a virtual IP device or via a message passing library. Also the communication with the Linux kernel in the heterogeneous setup happens over an IP connection as well.

Just as in the unikernel design previously seen, some parameters, like the number of system threads, are set at compile time. As a result internal data structures can be built as static arrays, which provides fast accesses and good cache-usage. For tasks like garbage collectors in the runtime of managed languages it is still necessary for *Hermit Core* to provide a scheduler. Yet the scheduling overhead is reduced by the fact that the *Hermit Core* kernel does not interrupt computation threads. Computation threads run on certain cores and are not using a timer which would be a source for OS noise.

For building applications a slightly modified version of the *GNU binutils* and *gcc* is used with the build target *x86_64-hermit*. By this design, hermit core applications can be built in any language supported by *gcc*. But it is even possible to use a different C-compiler by including the special *Hermit Core* header files instead of the Linux ones.

Figure 3 shows scatter plots from the *Hourglass* benchmark. The gaps in the execution time are used to indicate points in time where the operating system took time from an application process for maintenance tasks.

In comparison to a standard Linux, which can be seen in Figure 3a, the *Hermit Core* with a networking thread, seen in Figure 3b, shows significantly smaller gaps than the Linux. But *Hermit Core* is designed to spawn only one thread for handling IP packets, therefore all computation threads run with a profile similar to Figure 3c which shows the smallest noise distribution. Lankes et al. [6] also show that their prototype is approximately twice as fast, with regard to basic system services on the Intel Haswell architecture, as Linux.

We have seen *Hermit Core*, which is a versatile, well performing symbiosis of Linux and unikernels designed for HPC. The benchmark results still show OS noise to be present, but on a much smaller scale than on Linux.

# 5 Comparison

As the *mOS* project is still in a prototype phase, and the *Interface for Heterogeneous Kernels* as well is in an early stage, the stability of the projects has still to show as they both progress. As the concept of library operating systems is much older, the systems investigated in this work seem to be stable yet are facing different issues.

The development of applications to run on a heterogeneous kernel should be not much different than for traditional Full-Weight-Kernel systems as both projects presented set Linux compatibility as one of their goals. Additionally the symbiotic system presents itself as a single system to applications. Even with a stable exokernel already present, the implementation of an application together with a specialized OS suited for it involves much more manual work than with the heterogeneous kernel concept. Virtualization helps to cope with hardware abstractions for the exokernel, but the libOS has to be written for every application. *Hermit Core* provides the combination of the aforementioned concepts by uniting a libOS in form of a unikernel with a Full-Weight-Kernel. It makes it possible to use high-level languages for application development and provides an adapted software collection with familiar build tools.

If we take the performance evaluation of *FusedOS* into account, as *mOS* as well as IHK are not ready yet for macro-benchmarks, the Light-Weight-Kernels run with much less jitter and deterministic behavior. Results of the unikernel prototype *Mirage OS* benchmarks are all at the micro-level as most applications for it are still stubs or have limited functionality. Yet this approach as well shows a clear reduction in jitter and more predictable behavior can be expected. As can be seen in Figure 3, *Hermit Core* offers a considerable reduction in OS noise. Additionally the performance for basic system services and scheduling has been shown to be higher compared to Linux.

# 6 Conclusion

On the way to exascale computing there is a need for new concepts in HPC OS design in order to make full use of the hardware. One step in this direction is the elimination of jitter.

In this paper we introduced the two currently most popular concepts for new operating system designs focusing on high performance computing. To sum up, both approaches show great promise for performance and the reduction of OS noise. Even their combination is possible and the so constructed prototype system performs equally well. Yet all projects are missing comprehensive evaluation results as a consequence of their youth. Heterogeneous kernel concepts seem to have high potential yet are not mature enough to be considered at the moment. Application development should be straightforward and compatibility with already existing ones should be provided. The concept of library operating systems has been around for a long time and it might be a good option if the performance boost compensates the cost for virtualization layers, but more manual work is involved in order to write a tailored libOS in addition to the desired application. A combination of both concepts is possible and seems to have excellent properties.

# References


[1] Ian Briggs, Matt Day, Yuankai Guo, Peter Marheine, and Eric Eide. A performance evaluation of unikernels. 2015.

[2] Dawson R Engler, M Frans Kaashoek, et al. *Exokernel: An operating system architecture for application-level resource management*, volume 29. ACM, 1995.

[3] Kurt B Ferreira, Patrick G Bridges, Ron Brightwell, and Kevin T Pedretti. The impact of system design parameters on application noise sensitivity. *Cluster computing*, 16 (1):117–129, 2013.

[4] Balazs Gerofi, Masamichi Takagi, Yutaka Ishikawa, Rolf Riesen, Evan Powers, and Robert W Wisniewski. Exploring the design space of combining linux with lightweight kernels for extreme scale computing. In *Proceedings of the 5th International Workshop on Runtime and Operating Systems for Supercomputers*, page 5. ACM, 2015.

[5] Torsten Hoefler, Timo Schneider, and Andrew Lumsdaine. Characterizing the influence of system noise on large-scale applications by simulation. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–11, Washington, DC, USA, 2010. IEEE Computer Society. ISBN 978-1-4244-7559-9. doi: 10.1109/SC.2010.12. URL `https://doi.org/10.1109/SC.2010.12`.

[6] Stefan Lankes, Simon Pickartz, and Jens Breitbart. Hermitcore: A unikernel for extreme scale computing. In *Proceedings of the 6th International Workshop on Runtime and Operating Systems for Supercomputers*, ROSS '16, pages 4:1–4:8, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4387-9. doi: 10.1145/2931088.2931093. URL `http://doi.acm.org/10.1145/2931088.2931093`.

[7] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. Unikernels: Library operating systems for the cloud. In *ACM SIGPLAN Notices*, volume 48, pages 461–472. ACM, 2013.

[8] José Moreira, Michael Brutman, Jose Castano, Thomas Engelsiepen, Mark Giampapa, Tom Gooding, Roger Haskin, Todd Inglett, Derek Lieber, Pat McCarthy, et al. Designing a highly-scalable operating system: The blue gene/l story. In *SC 2006 Conference, Proceedings of the ACM/IEEE*, pages 53–53. IEEE, 2006.

[9] Yoonho Park, Eric Van Hensbergen, Marius Hillenbrand, Todd Inglett, Bryan Rosenburg, Kyung Dong Ryu, and Robert W Wisniewski. Fusedos: Fusing lwk performance with fwk functionality in a heterogeneous environment. In *Computer Architecture and High Performance Computing (SBAC-PAD), 2012 IEEE 24th International Symposium on*, pages 211–218. IEEE, 2012.

[10] Taku Shimosawa, Balazs Gerofi, Masamichi Takagi, Gou Nakamura, Tomoki Shirasawa, Yuji Saeki, Masaaki Shimizu, Atsushi Hori, and Yutaka Ishikawa. Interface for heterogeneous kernels: A framework to enable hybrid os designs targeting high performance computing on manycore architectures. In *High Performance Computing (HiPC), 2014 21st International Conference on*, pages 1–10. IEEE, 2014.

[11] Robert W Wisniewski, Todd Inglett, Pardo Keppel, Ravi Murty, and Rolf Riesen. mos: An architecture for extreme-scale operating systems. In *Proceedings of the 4th International Workshop on Runtime and Operating Systems for Supercomputers*, page 2. ACM, 2014.