# mOS: An Architecture for Extreme-Scale Operating Systems

Robert W. Wisniewski[†] Todd Inglett[†] Pardo Keppel[†]
Ravi Murty[†] Rolf Riesen[†]

Linux®, or more specifically, the Linux API, plays a key role in HPC computing. Even for extreme-scale computing, a known and familiar API is required for production machines. However, an off-the-shelf Linux distribution faces challenges at extreme scale. To date, two approaches have been used to address the challenges of providing an operating system (OS) at extreme scale. In the Full-Weight Kernel (FWK) approach, an OS, typically Linux, forms the starting point, and work is undertaken to remove features from the environment so that it will scale up across more cores and out across a large cluster. A Light-Weight Kernel (LWK) approach often starts with a new kernel and work is undertaken to add functionality to provide a familiar API, typically Linux. Either approach however, results in an execution environment that is not fully Linux compatible.

mOS (multi Operating System) runs both an FWK (Linux), and an LWK, simultaneously as kernels on the same compute node. mOS thereby achieves the scalability and reliability of LWKs, while providing the full Linux functionality of an FWK. Further, mOS works in concert with Operating System Nodes (OSNs) to offload system calls, e.g., I/O, that are too invasive to run on the compute nodes at extreme-scale. Beyond providing full Linux capability with LWK performance, other advantages of mOS include the ability to effectively manage different types of compute and memory resources, interface easily with proposed asynchronous and fine-grained runtimes, and nimbly manage new technologies.

This paper is an architectural description of mOS. As a prototype is not yet finished, the contributions of this work are a description of mOS's architecture, an exploration of the tradeoffs and value of this approach for the purposes listed above, and a detailed architecture description of each of the six components of mOS, including the tradeoffs we considered. The uptick of OS research work indicates that many view this as an important area for getting to extreme scale. Thus, most importantly, the goal of the paper is to generate discussion in this area at the workshop.

[†]Intel Corporation
Linux® is the registered trademark of Linus Torvalds in the U.S. and other countries.

*ROSS* '14, June 10, 2014, Munich, Germany
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2950-7/14/06 ...$15.00.
http://dx.doi.org/10.1145/2612262.2612263

## 1. INTRODUCTION

As the system software community moves forward to exascale computing and beyond, there is the oft debated question of how revolutionary versus how evolutionary the software needs to be. Over the last half decade, researchers have pushed in one direction or the other. We contend that both directions are valid and needed *simultaneously*. Throwing out *all* current software environments and starting over would be untenable from an application perspective. Yet, there are significant challenges getting to exascale and beyond, so revolutionary approaches are needed. Thus, we need to simultaneously allow the evolutionary path, i.e., in the OS context, a Linux API, to coexist with revolutionary models supportable by a nimble LWK. The focus of mOS is extreme-scale HPC. mOS, which simultaneously runs a Linux and an LWK, supports the coexistence of evolutionary and revolutionary models. In the rest of the introduction, we describe the following motivations for mOS:

- simultaneously support the existing Linux API with LWK performance, scalability, and reliability;
- resolve the tension between FWK and LWK approaches;
- nimbly incorporate new technologies such as hybrid memory and specialized cores, as well as new models such as fine-grained threading and asynchronicity; and
- provide a hierarchy for system call implementation.

The architecture of mOS is depicted in Figure 1. We describe it in more specificity in Section 4, but include it early so the following description has context.

**Simultaneously support legacy and new paradigms**
Linux is crucial to a broad user base across many computing areas. As such, proposed modifications to Linux need to meet a high bar in terms of their applicability. An LWK has greater flexibility to target the specific needs of high-end HPC. mOS, by incorporating Linux and an LWK, leverages the strengths of Linux and provides a test bed to explore new technologies allowing Linux to incorporate them when they have demonstrated broader-based appeal. For example, large pages were used in HPC LWKs a decade ago, and over time, as the usage models were validated, Linux has started including large page support. Thus, in mOS, Linux and the LWK, are not in competition but are symbiotic with each leveraging the strengths of the other. From a pragmatic point of view, vendors need to deliver solutions that work for their existing customer base, but also need to innovate to maintain competitiveness. mOS provides the ability to simultaneously support legacy and emerging paradigms.

**Resolve tension in FWK and LWK approaches**
Historically, high-end operating systems have been approached either from an FWK or an LWK perspective. In

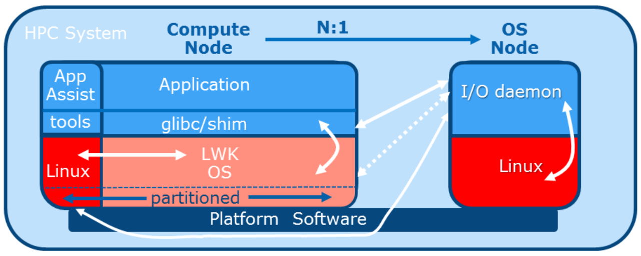

**Figure 1: mOS architecture ©Intel**

an FWK approach, an OS, typically Linux, forms the starting point, and work is undertaken to remove features from the environment so that it will scale across more cores and out across more nodes in a large cluster. An LWK approach often starts with a new kernel and work is undertaken to add functionality to provide a familiar API, typically something close to that of a general purpose operating system such as Linux. Either approach's end point however, is an environment that is not fully Linux compatible. The LWK approaches Linux, but some Linux functionality, either because of time constraints or intentionally, is not implemented. An FWK approach, strips enough from Linux so that it too, no longer supports generic Linux applications. Thus, some vendors offer two solutions: a fully compatible cluster HPC Linux that performs well but not at extreme scale, and a configured patched offering for extreme scale, but that does not provide full Linux compatibility. As an example, the Cray®Linux Environment provides extreme scale and cluster compatibility capability [3]. mOS runs both Linux and an LWK simultaneously. Therefore, Linux can be more Linux, it does not need to be stripped down. And the LWK can be more lightweight, it does not need to include support for services that should be supported on the compute node, but would be better left out of an LWK, e.g., Java, Python. Further, mOS allows Linux functionality to be achieved with minimal or no patching of Linux. The goal of upstreaming modifications can be achieved due to the narrow interface needed to interact with Linux, and provides for better sustainability due to not needing to maintain Linux patches.

**Nimbly support new hardware and software needs**
A primary motivation for including an LWK in mOS is that the LWK portion of mOS is a small piece of code that is easily modifiable for the needs of a targeted class of applications. Because it is a small, a single developer can internalize the whole code base. That improves reliability because there are reduced misunderstandings about component interactions and simply because there is less code. LWKs have been shown to have low noise, high performance, and scale well. An LWK also allows easier implementation of specialized features. One example that Linux probably would not entertain, but can be done in an LWK, is to remove all dynamic memory mapping and statically map in all memory. Other examples include specialized placement of MPI communicating threads nearest the network, having zero ticks, and automatic use of large pages.

In the long term, as frequency is not increasing and specialized cores proliferate, we contend the right way to manage the other cores is with an LWK on them where that "kernel" is targeted to that special application or device. In particular, HPC systems are increasingly heterogeneous, with nodes containing a combination of cores optimized for single thread performance, ones optimized for power efficiency, ones designed for special purposes but being used more generally, e.g., GPUs, programmable logic (FPGAs), and fixed-function application-specific hardware. Thus, an mOS-like approach will be increasingly important compared to either FWK or traditional LWK approaches, as mOS's architecture is suited to leverage special cores. It allows placing key services as close as possible to raw hardware, while at the same time keeping jittery and other system services isolated. Further, because mOS provides non-critical services by calling Linux, the mOS LWK is smaller than a conventional LWK; in turn, this makes it simpler to port and target mOS to novel hardware.

**Provide a hierarchy for system calls**
As machines increase in size – both node count across the machine, and core count per node – a scalable method for handling the proliferation is to move to hierarchical mechanisms. This has been done for scalable job launch, and I/O. Cray's Linux [3] and the IBM®CNK [9] use a hierarchy for I/O. By design, mOS leverages a hierarchy of places to implement system calls. First, performance critical calls are serviced by the LWK on the compute cores running mOS. Second, calls that benefit from a short latency, require node-local information, or require a Linux kernel, are serviced on the Linux running on the compute node. Third, calls with higher-latency or those requiring resources not available on the compute node, are off-loaded to an OS node. While we describe three levels of hierarchy for simplicity, we envision offload can be to more than just a single node.

This paper is an architectural description of mOS. As a prototype is not fully implemented, the contributions of this work are 1) a description of mOS's architecture, 2) an exploration of the tradeoffs we considered and the value of the mOS approach as a way to satisfy the above listed motivations. The motivations and requirements are representative of the needs for OS kernels for high-end HPC systems, and 3) a description of mOS's six components and the tradeoffs we considered.

There was significant operating system research in the 1990s, and then in the 2000s there was less emphasis on new operating structure and more work on Linux enhancements. More recently there has been resurgence of OS work [6, 7, 5, 15, 8, 16, 17, 14]. Thus, most importantly, the goal of this paper is to provide a basis for a timely and active discussion in this area at the workshop.

The rest of the paper is structured as follows. In Sections 2 and 3, we start by describing the design space for mOS and then describe the related work. Section 4 describes the architecture of mOS. It is broken into six subsections corresponding to each of the components in mOS. We provide concluding remarks in Section 5.

## 2. DESIGN SPACE

Designing an OS involves trade-offs, consideration of the target system, and the OS's use. This is particularly true for extreme-scale OSes where seemingly insignificant features may render the final product unusable at scale or too slow. For the mOS project, we identified three competing requirements as depicted in Figure 2. Each corner of the triangle represents one of the key design parameters of mOS: Linux compatibility, limited changes to Linux, and full LWK scalability and performance.

We are trying to achieve all three, but prioritize perfor-

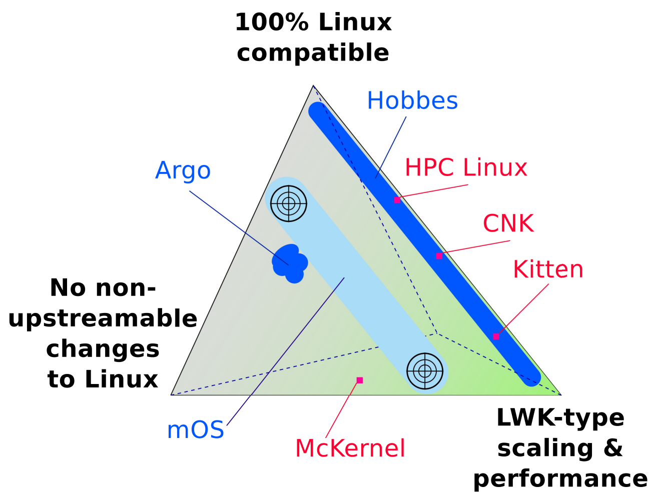
100% Linux compatible
Hobbes
Argo
HPC Linux
CNK
Kitten
No non-upstreamable changes to Linux
mOS
McKernel
LWK-type scaling & performance

**Figure 2: Design space for mOS ©Intel**

mance and scalability, followed by compatibility, then minimizing Linux changes for sustainability. Because mOS consists of two different kernels, it is depicted in an area with two target symbols: a Linux kernel in the upper part to give us Linux compatibility, and an LWK in the lower part to provide performance and scalability.

To evaluate and compare mOS with other approaches, we placed more markers in Figure 2. They indicate where we believe other projects place their priorities and efforts given the triumvirate design space. Research projects like Hobbes and Argo cover a larger area in our design space triangle than a single OS "point" because these projects encompass more than just an OS kernel. For example, Hobbes contains a hypervisor specialized for extreme-scale systems. This node virtualization layer can run OSes and kernels, such as Linux or Kitten, alongside each other and allows to combine OS capabilities to adapt the system software environment to an application's needs.

The primary goal of Figure 2 is to convey the tensions in the design space. Being positioned closer to a vertex indicates a greater capability in that dimension. A pure LWK would be located near the lower right vertex. That implies that it would have little Linux compatibility. Since it is not Linux, there are also no changes to Linux. Placement of projects and OSes is subjective, but it helps with comparisons by showing where project teams place their priorities.

When we refer to Linux compatibility, it is from the application's perspective. System tools written specifically for Linux would need to be adapted to work with the LWK. However, for other administrative calls, our goal is to handle those in Linux to provide compatibility. For example, administrative calls that support clock manipulation for an NTP daemon will not be supported by the LWK. Nevertheless, these calls are in Linux, so NTP will continue to operate unmodified on mOS.

To avoid maintaining patches, which is costly and error prone, any modifications need to be sufficiently non-invasive and have broad appeal to the Linux community, so they are upstreamed. One approach to achieve this is inspired by FUSE. We call it "Cooperative Agent Kernel Extensions" or CAKE. Its goal is to provide internal APIs for resources that Linux manages and to coordinate resource management between Linux and LWKs. Using clean resource partitioning allows Linux and the LWKs each to manage their own partitions, yet safely share the managed resources. At any given time, a sharable resource is either private to Linux or the LWK, so that that it can be managed directly by the current owner.

## 3. RELATED WORK

As mOS is a combination of techniques, there are several areas of related work pertinent to understanding where mOS fits it. The first is the tradeoff of LWK versus FWK for delivering HPC functionality and performance. The second area is that of providing an infrastructure to allow targeted HPC services to be instantiated either through a microkernel or virtualized environment. The third is work that looks at combining multiple kernels to provide full OS functionality.

**LWK versus FWK approach**

Work on a lightweight design at Sandia and the University of New Mexico has been ongoing for over two decades [18]. That collaboration produced operating systems for several top computer systems including Catamount [12] for Red Storm. Catamount was designed to be lightweight to facilitate scalability. Its strategy of providing only the features necessary to support the application, and then giving control of the processor and hardware to the application yields a low-noise kernel. Over three generations, CNK [9] on Blue Gene®, provided an increasingly enriched operating environment, with more flexibility and functionality, by leveraging and integrating Linux code e.g., glibc, NPTL. On-going work on Kitten [17] in combination with Palacios [14] has similar goals of providing a richer environment that more easily supports existing codes.

Another approach, taken by ZeptoOS [2], is to start with a Linux image and make modifications such as reserving memory during boot before Linux accesses it, and removing auxiliary daemons to reduce the noise. By doing so, ZeptoOS achieved good performance results and scaled well. A similar study undertaken by Shmueli et al. [19] looked at the issues that caused Linux not to scale to tens of thousands of nodes. They too observed that by modifying the memory management system of Linux, reducing the number of daemons, and synchronizing the remaining daemons, it was possible to get Linux to scale. Both of these approaches inherit the structure and algorithms of Linux, and though they can be modified, Linux is a moving target and is not focused on the high-end HPC space.

Other groups have taken the approach of providing a full-feature Linux. Both the SGI®Altix®ICE$^{TM}$Pleiades Linux machine installed at NASA [1], and the Cray Compute-Node Linux (CNL) [11, 21] on Titan at ORNL, take this approach. The goal behind a full CNL approach is to provide the rich set of operating system services and system calls that users and developers expect, and that their applications may require. These groups provide this functionality by reducing daemons and working on memory allocation issues, and try to upstream their work so the end system can be built by config options. However, the modifications to Linux, such as Cray's Extreme Scale Linux [10, 3] needed to scale to machines the size of Titan, break full Linux compatibility and cause challenges with upstreaming. HPC has tended to lead the technology trend, often actively employing new hardware mechanisms sooner than the mass workstation community. Because the Linux community focuses on the latter, it is not always receptive to all the changes needed for HPC, so the Linux approach often requires maintenance of patches.

**Microkernels and virtualization for targeted HPC system services**

mOS aims at providing revolutionary capability while maintaining an evolutionary path supporting existing computing paradigms, i.e., the Linux API. K42 [13] allowed the application to “reach around” Linux APIs and call native K42 interfaces. However, its approach involved significant entanglement at the implementation level causing considerable effort to keep K42 tracked with the latest Linux. To avoid that, LibraOS [4] separated the Linux implementation by placing a Linux instance in a side-car instantiation, and used a hypervisor to direct performance-insensitive calls from the performance critical OS services compiled into the application. This approach incurred the cost of the hypervisor and was a static binding of those services. Kitten and Palacios [14] took the next step. Palacios is an open source Virtual Machine Monitor (VMM) and embedded within the LWK Kitten. Palacios allows Kitten to host Linux as a guest OS. While this approach provides Linux functionality to Kitten applications with high performance, the Linux functionality is limited by what capability the VMM provides. For example, Palacios does not currently support symmetric multiprocessors (SMPs), thus Linux in this environment is not SMP capable. Although such capability could be added, it highlights the limitation afforded by this approach. Furthermore, Linux and Kitten can not be called simultaneously to maintain transparent name spaces.

Hobbes is a on-going DOE OS/R (Operating System and Runtime) project. The central theme of Hobbes [7] is that of application composition. Instead of using sequences of simulation and analysis tasks that communicate via long-term storage, applications use a compositional approach consisting of combinations of coupled codes, data services, and tools. This model is largely driven by the expected constraints of power and the limitations of the I/O system (co-locating applications to avoid data movement) in the future. Hobbes utilizes virtualization to create multiple “virtual nodes” to support sharing of physical resources between applications in distinct enclaves.

**Multiple kernels for specialized services**

The Tesselation project [15] from UC Berkeley takes a similar approach to mOS called Space-Time Partitions. It subdivides the available cores into groups called cells. Each cell is responsible for an application or set of system services. This creates a two-level scheduling hierarchy allowing the node-level resource manager to assign cores to cells and allows the system service or application to have fine-grained control over the resources they were assigned.

NIX [5] is a closely related effort to our own work. Its focus is primarily on role assignment of cores. The work explores four different types of roles for the cores: time-shared cores (TCs) that run similar to a general purpose OS, kernel cores (KCs) that run only in supervisor mode and handle system services and device drivers as well as system service requests from applications, application cores (ACs) that run an application and remote system calls to a kernel mode process in the TC or a KC, and exclusive cores (XCs) that run a single application thread and any kernel services it requires.

Argo [6] is another DOE OS/R project targeted at applications with complex workflows. Like Hobbes, they envision using OS and runtime specialization on the compute node. In their architecture, each node may contain a heterogeneous set of compute resources, a hierarchy of memory types with different performance (bandwidth, latency) and power characteristics. Given such a node architecture, Argo expects to use a ServiceOS like Linux to boot the node and run management services. It then expects to run different ComputeOS instances that cater to the specific needs of the application. Similar to mOS, kernels cooperate and are trusted, but do not form a single system image. Unlike Hobbes, virtualization is not used on a node.

McKernel [8, 20] is similar to mOS. It combines Linux and a LWK on the same many-core node or can run over PCI. As in mOS, coordination, data sharing, and function shipping among the running kernels, as well as off-node data transfers, are focus research topics. McKernel began life in a heterogeneous system with a traditional host CPU and PCI-bus-attached Intel Xeon Phi co-processors. Because of this heritage, McKernel has a hierarchical memory view and uses a hardware abstraction layer to let heterogeneous kernels communicate and share data. McKernel leverages Linux to natively manage process context such as file descriptors and, unlike mOS, currently does not incorporate offload to external OS nodes.

FusedOS’s [16] approach is similar to mOS with the objective of addressing core heterogeneity between system and application cores while providing a standard operating environment. However, in FusedOS, the LWK runs at user level on the compute cores. In the FusedOS prototype, the kernel code on the application core is a stub that relays system calls to a corresponding user-level proxy process called CL. Therefore, the entire LWK is implemented within the CL user proxy process on Linux. This provides the same functionality as the Blue Gene CNK from which CL was derived, but the system could benefit from the low-latency communication for function-shipping to Linux, and it opened up the possibility for shared memory and other Linux features to interoperate between the LWK and Linux. The FusedOS work demonstrated that Linux noise can be isolated to the Linux core and not interfere with the LWK cores or application running on them. This result carries forward to mOS. The FusedOS project also concentrated on an effort that avoided changes to Linux.

## 4. ARCHITECTURE

Realizing mOS requires several components. The architecture of mOS is depicted in Figure 1. We divide mOS into six components described in this section. The first two components are the LWK and Linux. A third component is the transport mechanism that connects the LWK and Linux. A fourth piece is the capability to triage, or appropriately direct a system call made by the application to the correct kernel in the hierarchy. A fifth component is the offload mechanism, both on the compute node and its pair on the OS node. The sixth mechanism is the capability to partition resources between the LWK and Linux.

### 4.1 LWK

In the mOS architecture, one or more instantiations of an LWK runs alongside a Linux kernel. The goal is to provide as much of the compute hardware resources as possible to the HPC applications. The amount of memory, cores, hardware threads, etc., dedicated to the LWK is configurable. Depending on the workload and how much Linux functionality is required, more or less resources may be allocated to the LWK.

The LWK is directly responsible for the management of the resources it has been allocated. In particular, memory management and process scheduling policies are the domain of the LWK. Unlike traditional LWKs, much of the other OS administrative work is done by the local Linux kernel.

Traditionally, LWKs like SUNMOS, Puma, Catamount, and CNK, were responsible for booting a node, initializing its hardware resources, including the NIC, and loading processes. In mOS, much of the hardware of a node will be booted and initialized by Linux. This allows us to design a simpler more streamlined LWK, allowing us to more effectively deliver on its goals of making the performance and scalability of the underlying hardware accessible to application processes.

The mOS LWK is based on the following requirements for the resources for which it is responsible:

1. Manage memory as physically contiguous regions
   - Important for page table caching and large pages to reduce or avoid TLB misses
2. Generate no interrupts
   - Except those setup by the application
   - Run in a cooperative multi-tasking mode; i.e., no quantum timer
3. Provide full control of scheduling
4. Share memory regions across LWK processes
5. Ship system calls to the Linux core and OSNs
6. Provide efficient and user-level access to the hardware
   - Yield high performance MPI and PGAS runtimes
   - Allow scalable application load
7. Allow flexibility across cores in allocated memory
   - Rank 0 may need more memory than the remaining ranks of a parallel job

Unlike a pure LWK model, we intentionally simplify mOS's LWK by leveraging the on-node Linux. We have identified the following features:

1. Boot and configure the hardware
2. Direct interrupts to Linux instead of the LWK
3. Hand off cores, memory, and other resource to the LWK to manage

## 4.2 Linux

The primary role of the Linux kernel component is to provide Linux functionality in mOS. From a microkernel perspective, the Linux kernel component may be thought of as a service that provides Linux functionality. This functionality includes capability that the LWK may otherwise have had to implement. Examples include compute node TCP/IP sockets and non-I/O file descriptor operations such as an epoll on any of a timer, inotify, signal, or event descriptors. Secondary goals include providing an execution environment for various application assistance and infrastructure services and providing a familiar administrative interface to the compute node. The former would include daemons typically found on Linux clusters for job launch and monitoring, but also might include specialized application framework daemons that support new workflow-based programming models.

There are several possibilities for selection of a Linux kernel for mOS. The main criteria is that it should be a standard HPC cluster Linux. It does not need development or GUI packages, but it should have standard HPC execution packages. High performance paths will be handled

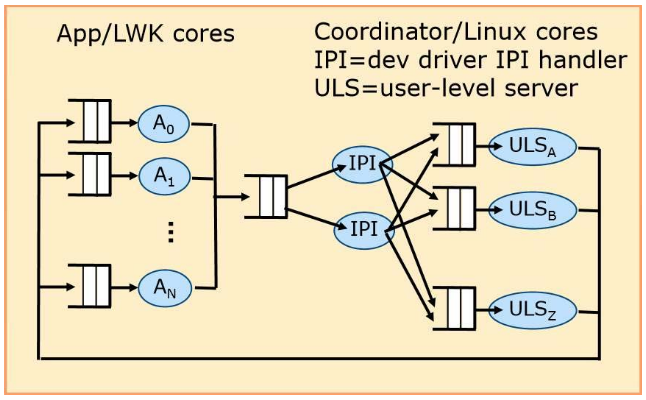

**Figure 3: Schematic of in-node shipping ©Intel**

by the LWK, and application memory will be managed by the LWK. Memory is a precious resource on compute nodes, so Linux is configured for minimal memory use, and without disk paging.

Our goal is to minimize the changes mOS requires of Linux to maximize the likelihood of upstreaming the modifications. The mOS operating environment relies on OS nodes and other external OS infrastructure, thus we plan to use the same Linux on other nodes as on mOS. However, different distributions of Linux may be desired on different systems, and if we succeed in upstreaming the mOS patches, it will be relatively easy to substitute a Linux of choice.

## 4.3 The In-Node LWK-Linux Connection

Communication between Linux and LWK is explicit. We refer to moved functionality as "function shipping". The LWK is structured so some LWK components may be implemented as Linux device drivers and others as user-level servers. Communication between Linux and the LWK is done on the cores running Linux to avoid jitter on application cores even though the LWK is initiating a request.

Figure 3 shows a general schematic of one node in mOS. On the left are LWK cores that may be running one or several threads; on the right are user-level servers. Requests from LWK cores are sent via messages with delivery as discussed below.

User-level servers in mOS take several forms. The servers run on Linux at user level on Linux cores. They are leveraged by the LWK and are structured according to needs of the specific service, including that some may utilize kernel drivers. One simple structure is a user-level server thread per LWK thread, and a user-level server process per LWK process. Specific services may, however, go directly to other servers.

Communication between LWK and Linux is accomplished by one of three mechanisms: shared memory, messages, and inter-processor interrupts (IPIs). The Linux kernel directly accesses user level with loads and stores. This makes full marshalling a problem without invasive Linux changes as cross-space accesses are not fully abstracted. To overcome this, mOS partitions physical memory into a Linux-managed part and an LWK-managed part, where each kernel is allowed to access the other's memory.

System calls are triaged (see 4.4) to determine which kernel should implement them. Calls that are forwarded to Linux have parameters assembled in a message along with context information such as the hardware thread and core

```
read(fd, buf, len):
    payload = { READ, fd, buf, len }
    msg = { route=pid, context=lwk_pid, payload }
    was = linux_channel.q.insert(msg)
    if (!was):
        linux_channel.receiver.send_ipi()
    lwk_channel.wait_for_ack()
```

**Figure 4: Message send example.**

```
ipi_handler():
    msg_list = linux_channel.q.get_list()
    while ((msg = msg_list.pop()) != NULL):
        pid = msg→route
        channel[pid].q.insert(msg)
        sched_run(pid)
```

**Figure 5: IPI receive/dispatch example.**

initiating the syscall, indicating where the syscall acknowledgment (ACK) should be sent. Fig. 4 shows an example. The Linux and the LWK trust each other. For debugging or more-protected designs, each incoming message can be checked against consistency rules.

There are latency and load tradeoffs in the function shipping implementation. At one extreme, an always-polling thread would be most responsive, but could monopolize resources and so may induce delays. At the other extreme, per-message interrupts only use resources on demand, but with a BSP programming model can generate a "storm" of requests. As is common in device drivers, we use the approach in Figure 4, which sends IPIs only if there are no other queued requests, else the mechanism piggy-backs on IPIs of prior not-yet-serviced requests. This slightly compromises latency at the lowest load, but significantly reduces IPIs at high load.

Each message has routing information indicating the eventual destination, as shown by route= in Fig. 4. The IPI handler examines routing information, re-queues the message, and if needed wakes the queue reader, as shown in Fig. 5.

The description assumes message sends are performed by the LWK in supervisor mode, but if a small relaxation of the Unix protection model is tolerable, only send_ipi() needs privilege. Note, user-level queuing would allow a user to lie about context and thus cause service to be performed on behalf of the wrong actor. However, user-level service does not mean giving up protection entirely. Instead, channel is in shared memory, and shared memory is set up under kernel control. Thus, only a subset of tasks could write to a given channel. If context is outside of the tasks with channel access, an error is signaled.

Messages from Linux to LWK (both ACKs and Linux requests for LWK service) go by similar channels, but LWK channels allow LWKs to tell senders to skip IPIs. When IPIs are skipped, the LWK promises to eventually poll for pending messages and/or re-enabled IPIs. Skipping IPIs reduces receiver (LWK) jitter, at the expense of delayed response.

## 4.4 Triaging System Calls

A system call can be serviced by the LWK, the Linux on the compute node, or the Linux running on the OSN. The goal of triage is to route requests to the correct implementer.

The first point of interception for system calls is within the C library. The LD_PRELOAD dynamic linking mechanism for glibc makes this relatively easy. User-level interception avoids kernel overhead and offers a way to bind streamlined (caller-specific) implementations of services. It provides a rich proramming and friendly debug environment. Triage may be staged. Depending on the syscall and parameters, triage might perform a simple table lookup for cases to offload to an OSN via a user-level aggregator, with remaining cases serviced via trap to the LWK.

Currently, all calls that are not offloaded are directed to the LWK. It may be possible to have some shipped directly to the Linux core. This would increase the complexity of providing transparency, but would simplify the LWK and improve its performance. System calls that are performance sensitive for HPC applications are implemented by the LWK to provide high performance with no jitter, and the other calls (the large majority by number), are shipped to Linux.

Because Linux is available on the node, the LWK can remain lightweight with system calls that need to be handled directly by Linux. The ability to direct needed but non-performance-critical calls is an advantage of mOS not possible with a solely-LWK approach.

System calls directed to Linux are those considered less performance critical, including event, signal mechanisms, some of the date/time interfaces, /proc requests, and /sys requests. The local Linux in mOS is also where the industry standard networking layer is implemented. Rather than function-ship the socket syscalls to the OS node, these are implemented by the local Linux so that data paths remain short and optimized. These network interfaces can be used by application assist code and tools running on the Linux core. The HPC application will continue to access the network directly through high-performance messaging libraries such as MPI.

Triage of the system calls requires each layer maintain some information of what the other layers are doing in order to make the correct decisions. For example, the open system call operates in the context of a current working directory. The chdir system call that changes the working directory needs to notify other layers of the triage so they can operate correctly.

The disposition of many of the system calls will be decided at build or even implementation time, but others may be decided at runtime. One reason to decide at runtime is that the syscall may be requesting service on a resource currently managed by the LWK, but that resource may at a future time move to be managed by Linux, in which case the syscall also should be handled by Linux. Secondly, applications will exhibit different patterns of syscall behavior, and thus it may make sense to tailor where certain syscalls are serviced. This is currently under study.

## 4.5 Offloading to an OS Node

A challenge in extreme-scale systems is the increasing scalability requirements placed on the parallel filesystem (PFS). A mitigation that has been successfully leveraged in previous systems is to offload PFS operations from a set of compute nodes to an I/O node reducing the number of clients by one or two orders of magnitude. Not only does this reduce the number of PFS clients, but it also removes the jitter caused by the client by removing it from the compute node. It also reduces memory usage and cache pollution.

In the mOS architecture, the PFS would naturally exist in the Linux operating environment of the compute node. This addresses the jitter caused by the PFS, but it does not address the client scaling of the PFS, nor does it address the CPU and memory resource needed by the PFS. Therefore, an offload solution remains an important component of mOS. Because there will be tools and programs for application assist running on the Linux core(s), see Figure 1, it will also be important to provide offload capability for Linux as illustrated by the white bidirectional arrow from Linux to the OSN.

System call offload to a remote OS node is accomplished with a mechanism similar to a remote procedure call. The OS node implementation could be kernel or user-level based. A kernel-mode implementation could be done using existing kernel extension mechanisms, but user level is preferred due to the ease of development and availability of efficient user-level access to the network hardware. Another motivation to pursue a user-level offload solution for user processes is that Linux already tracks the necessary context. A growing problem with offload is the non-linear growth in number of compute tasks a particular OS node is expected to serve. Future systems may keep the approximate OSN to compute node ratio, but the number of cores, threads, and tasks within a compute node is growing substantially. Therefore, it will likely be important to aggregate I/O requests on the compute node before offloading them, which also motivates a user-level solution.

The mOS solution also introduces additional complexity from previous offload solutions. Previous solutions, such as Blue Gene/Q CNK, assigned only a single offload destination for its function-shipping solution. The mOS solution now allows some system calls to be directed to the local Linux and others to the remote OS node. This adds complexity in tracking process context (working directories, uid/gid, umask, etc,), which must be common across the independent Linux environments.

## 4.6 Partitioning Resources Between the LWK and Linux

The LWK and Linux run on the same node so a mechanism is needed to share resources between the two kernels. The resources of interest are: 1) cores, 2) memory, and 3) devices, e.g., HFIs (Host Fabric Interface), global timer, IOAPIC, IOMMU. Each kernel depicted in Figure 1 needs to be given a set of resources. This can be done statically, requiring a reboot to change, or dynamically, using Linux's support for CPU and memory hotplug. There are more challenges when attempting to do this dynamically, but it provides more flexibility. We may relax the implementation from static to dynamic as we explore the needs of the applications. In the rest of the section we describe how to handle memory, core, and device partitioning.

**Memory partitioning**

Static partitioning of memory between the Linux and LWK is relatively straightforward. The platform BIOS provides Linux on the first core that is booted with an ACPI or E820 table that is a map of usable and reserved memory regions. Since this table describes all usable physical memory, we restrict Linux to use a fraction of the total memory, e.g., by using the Linux kernel command line parameter mem=nn[KMG], and then build a second E820 map that describes the remaining physical memory for the LWK. In this E820 map, the physical memory already used by the Linux is marked reserved. When the LWK is started, it parses the table provided in memory and builds its list of physical pages without touching pages allocated to Linux. Using this mechanism the size of each partition remains fixed throughout an application's execution. To resize the memory, Linux would need to be restarted.

Dynamic partitioning of memory can be achieved by using the logical memory hotplug support in Linux, i.e., via memory onlining/offlining sysfs entries. This memory can then be described using an E820 table as described above. This method allows resizing the partitions without rebooting. Another enhancement we are considering is allowing the LWK to dynamically integrate this memory.

In addition to partitioning physical memory there are two other issues. Frist, each hardware thread starts out in real-mode where only the lower 1MB of physical memory is addressable (segment + offset = 20 bits), thus, a region of memory must be reserved to store the trampoline code for booting other Linux processors and the LWK processors. Second, some shared physical memory must be reserved between the two kernels for messaging and notification, e.g., for system call shipping. This is in addition to the mapping memory in a manner to allow it to be visible to the application on Linux and LWK.

**Core partitioning**

The MADT in the ACPI table describes a list of CPUs and interrupt sources to the Linux kernel. Just as the kernel can be limited to use a fraction of total memory, the kernel command line *maxcpus=n* can be used to limit the maximum number of CPUs it will boot. The remaining CPUs are left in a state where a set of IPIs and startup code can take them from real-mode to 64-bit long mode with paging. This simple mechanism can be used to statically partition the CPUs between the Linux and the LWK. Alternatively, like the memory partitioning mechanism described above, CPU offlining/onlining can be used to dynamically partition the number of CPUs allocated to each kernel without requiring a reboot of Linux.

**Device partitioning**

- HFI: The HFI is initialized by Linux as per the above text on simplifying the LWK, but it is made available to the LWK and to the user-level application running on the LWK.
- Other physical devices: Since our goal is to avoid interrupting the LWK, any other physical devices, e.g., PCH, will be assigned to Linux, which will provide the appropriate drivers to manage them.
- IPIs: Care must be taken to avoid using IPI shortcuts like all and all-but-self to prevent IPIs from crossing over Linux/LWK boundary.
- Global timer for local APIC calibration: This is handled by providing a kernel command line similar to the loops_per_jiffy in Linux.

## 5. CONCLUSIONS

We presented an architectural description of mOS and its components, and provided insight into the tradeoffs we considered. mOS represents a methodology for combining a Linux kernel that provides support for existing execution models, with an LWK that provides support for emerging ones. We described the advantages that such an architecture provides. While that OS structure may be a good design across a wide variety of future areas, we are focusing mOS on extreme-scale HPC. Some of the next steps involve finishing our prototype and measuring the overheads associated with the triaging of system calls. Also, while previous work [16] indicated success in isolating noise between Linux and the LWK, we will need to verify we can achieve the same with mOS's architecture.

## Acknowledgments

It is wonderful to have a group of smart people who help push your ideas and ensure their validity. We want to thank Kurt Alstrup, Steve Hampson, Mike O'Hara, and Evan Powers for countless conversations that resulted in a stronger, more fleshed-out architecture. We thank Kevin Pedretti for help in better understanding Kitten and Hobbes.

## 6. REFERENCES

[1] NASA's Pleiades Machine. http://www.nas.nasa.gov/Resources/Systems/pleiades.html.

[2] ZeptoOS: The small Linux for big computers. http://www.mcs.anl.gov/research/projects/zeptoos/.

[3] Cray Linux Environment™ (CLE) 4.0 software release overview. `http://docs.cray.com/books/S-2425-40/S-2425-40.pdf`, Mar. 2011.

[4] G. Ammons, J. Appavoo, M. Butrico, D. D. Silva, D. Grove, K. Kawachiya, O. Krieger, B. Rosenburg, E. V. Hensbergen, and R. W. Wisniewski. Libra: A library operating system for a JVM in a virtualized execution environment. In *VEE (Virtual Execution Environments)*, San Diego CA, June 13-15 2007.

[5] F. J. Ballesteros, N. Evans, C. Forsyth, G. Guardiola, J. McKie, R. Minnich, and E. Soriano-Salvador. Nix: A case for a manycore system for cloud computing. *Bell Labs Technical Journal*, 17(2):41–54, 2012.

[6] P. Beckman. An exascale operating system and runtime research project.

[7] R. Brightwell, R. Oldfield, A. B. Maccabe, and D. E. Bernholdt. Hobbes: Composition and virtualization as the foundations of an extreme-scale OS/R. In *Proceedings of the 3rd International Workshop on Runtime and Operating Systems for Supercomputers*, ROSS '13, pages 2:1–2:8, New York, NY, USA, 2013. ACM.

[8] B. Gerofi, A. Shimada, A. Hori, and Y. Ishikawa. Partially separated page tables for efficient operating system assisted hierarchical memory management on heterogeneous architectures. In *Cluster, Cloud and Grid Computing (CCGrid), 2013 13th IEEE/ACM International Symposium on*, pages 360–368, May 2013.

[9] M. Giampapa, T. Gooding, T. Inglett, and R. W. Wisniewski. Experiences with a lightweight supercomputer kernel: Lessons learned from Blue Gene's CNK. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing (SC10)*, New Orleans, LA, Nov. 2010.

[10] S. D. Hammond, G. R. Mudalige, J. A. Smith, J. Davis, S. A. Jarvis, J. Holt, J. A. H. I. Miller and, and A. Vadgama. To upgrade or not to upgrade? Catamount vs. Cray Linux Environment. In *Large Scale Parallel Processing Workshop 2010 (LSPP10), 2010 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW)*, pages 1–8, April 2010.

[11] L. S. Kaplan. Lightweight Linux for high-performance computing, Dec. 2006.

[12] S. Kelly and R. Brightwell. Software architecture of the lightweight kernel, Catamount. In *Cray Users' Group Annual Technical Conference*, Albuquerque, New Mexico, June 2005.

[13] O. Krieger, M. Auslander, B. Rosenburg, R. W. Wisniewski, J. Xenidis, D. D. Silva, M. Ostrowski, J. Appavoo, M. Butrico, M. Mergen, A. Waterland, and V. Uhlig. K42: Building a real operating system. In *Proceedings of EuroSys'2006*, pages 133–145. ACM SIGOPS, April 2006.

[14] J. Lange, K. Pedretti, T. Hudson, P. Dinda, Z. Cui, L. Xia, P. Bridges, A. Gocke, S. Jaconette, M. Levenhagen, and R. Brightwell. Palacios: A new open source virtual machine monitor for scalable high performance computing. In *IPDPS*, April 2010.

[15] R. Liu, K. Klues, S. Bird, S. Hofmeyr, K. Asanović, and J. Kubiatowicz. Tessellation: Space-time partitioning in a manycore client OS. In *Proceedings of the First USENIX Conference on Hot Topics in Parallelism*, HotPar'09, pages 10–10, Berkeley, CA, USA, 2009. USENIX Association.

[16] Y. Park, E. Van Hensbergen, M. Hillenbrand, T. Inglett, B. Rosenburg, K. D. Ryu, and R. 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, Oct 2012.

[17] K. T. Pedretti, M. Levenhagen, K. Ferreira, R. Brightwell, S. Kelly, P. Bridges, and T. Hudson. LDRD final report: A lightweight operating system for multi-core capability class supercomputers. Technical report SAND2010-6232, Sandia National Laboratories, Sept. 2010.

[18] R. Riesen, R. Brightwell, P. G. Bridges, T. Hudson, A. B. Maccabe, P. M. Widener, and K. Ferreira. Designing and implementing lightweight kernels for capability computing. *Concurrency and Computation: Practice and Experience*, 21(6):793–817, Apr. 2009.

[19] E. Shmueli, G. Almasi, J. Brunheroto, J. Castanos, G. Dozsa, S. Kumar, and D. Lieber. Evaluating the effect of replacing CNK with Linux on the compute-nodes of Blue Gene/L. In *ICS '08: Proceedings of the 22nd annual international conference on Supercomputing*, pages 165–174, New York, NY, USA, 2008. ACM.

[20] M. Si, Y. Ishikawa, and M. Tatagi. Direct MPI library for Intel Xeon Phi co-processors. In *Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2013 IEEE 27th International*, pages 816–824, May 2013.

[21] D. Wallace. Compute node Linux: Overview, progress to date & roadmap. In *Proceedings of the Cray User Group (CUG)*, 2007.