Exploring the Design Space of Combining Linux with Lightweight Kernels for Extreme Scale Computing

Balazs Gerofi[†], Masamichi Takagi[†], Yutaka Ishikawa[†], Rolf Riesen[‡], Evan Powers[‡], Robert W. Wisniewski[‡]
[†]RIKEN Advanced Institute For Computational Science
[‡]Intel Corporation
{bgerofi, masamichi.takagi, yutaka.ishikawa}@riken.jp, {rolf.riesen, evan.powers, robert.w.wisniewski}@intel.com

ABSTRACT

As systems sizes increase to exascale and beyond, there is a need to enhance the system software to meet the needs and challenges of applications. The evolutionary versus revolutionary debate can be set aside by providing system software that simultaneously supports existing and new programming models. The seemingly contradictory requirements of scalable performance and traditional rich programming APIs (POSIX, and Linux[1] in particular) suggest that approach, and has lead to a new class of research. Traditionally, operating systems for extreme-scale computing have followed two approaches: they have either started with a full-weight kernel (FWK), typically Linux, and removed features which were impeding performance and scalability, or they started with a light-weight kernel (LWK), and added capability to provide Linux compatibility. Neither of these approaches, succeed in retaining full Linux compatibility and achieving high scalability.

To overcome this problem, we have been exploring the design space of providing LWK performance while retaining the Linux APIs and Linux environment. Our hybrid solution is to run Linux and an LWK side-by-side on the same node. HPC applications execute on top of the LWK, but the system selectively provides OS features by leveraging the Linux kernel. In this paper, we discuss two possible methods of achieving the symbiosis between the two kernels and the trade-offs between them. Specifically, we detail and contrast two particular approaches, Intel's mOS project and IHK/McKernel, an effort lead by RIKEN Advanced Institute for Computational Science.

Categories and Subject Descriptors

D.4 [**Operating Systems**]: Organization and Design

[1]Linux is the registered trademark of Linus Torvalds in the U.S. and other countries

ROSS '15, June 16, 2015, Portland, Oregon, USA

http://dx.doi.org/10.1145/2768405.2768410

Keywords

High Performance Computing; Multi kernels; Hybrid kernels

1. INTRODUCTION

With the increasing complexity of high-end supercomputers, the current system software stack faces significant challenges as we look forward to exascale computing and beyond. The necessity to deal with extreme amounts of parallelism, heterogeneous architectures, multiple levels of memory hierarchy, power constraints, etc. suggests the need for operating systems that can rapidly adapt to new hardware and software requirements, and that can support novel programming paradigms and runtime systems. Applications have also become considerably more complex, with an increasing demand for components such as in-situ analysis, elaborate monitoring and performance tools. This complexity relies on rich features of POSIX, and the Linux API in particular.

Traditionally, operating systems in high-end computing followed two approaches to deliver scalable performance and the reliability needed for extreme scale. In the full weight kernel (FWK) approach [14, 16, 20], a full Linux environment is taken as the basis, and features that inhibit achieving HPC scalability and performance requirements are removed. The light-weight kernel (LWK) approach [9, 8, 2] starts from scratch and effort is undertaken to add sufficient functionality so that it provides a familiar API, typically something close to that of a general purpose operating system such as Linux, while at the same time retaining the desired scalability and reliability attributes. Neither of these approaches yields a fully Linux compatible environment.

An alternative approach pursued by Intel's mOS project and RIKEN Advanced Institute for Computational Science's IHK/McKernel effort is to run Linux simultaneously with a light-weight kernel on the same node[19, 17]. The application is run primarily on the LWK to achieve the needed scalability and reliability, and Linux is leveraged judiciously to achieve API compatibility. The projects independently identified the following goals for such a symbiotic system:

- **Scalability and performance**: The system has to scale and deliver the parallel performance needed in an extreme-scale machine, with the goal of achieving LWK scalability and reliability.
- **Nimbleness:** The system should be easily adaptable to new hardware features and to specific software needs by applications or new programming models.

- **Maintainability**: The system must be highly maintainable, especially with respect to tracking Linux kernel changes. The goal is to be as isolated from Linux version-to-version changes as possible.

- **Linux compatibility**: The system needs to support POSIX and Linux APIs and a Linux environment, thereby enabling tools that run on top of Linux.

While not all these goals are easily quantifiable, they serve as high-level guiding principles. The primary goal is to enable scalable, parallel applications to run and perform with high efficiency on extreme-scale systems. By means of a small and easy to understand LWK code base, the system should nimbly incorporate new technologies. At the same time, since Linux compatibility is also required, this can only be achieved if the overall system is maintainable and able to track Linux. Finally, while Linux compatibility is one of the goals, parts of it may have to be limited, if doing otherwise would interfere with the other primary goals.

Despite the fact that from an application point of view, this hybrid/multi-kernel system presents itself as a single system image, resources (e.g., CPU cores and physical memory) are explicitly partitioned between Linux and the LWK. This separation is required so that the LWK can deliver scalable, consistent performance without suffering from any disturbance on the Linux side. Accordingly, HPC applications primarily run on the LWK and thus, performance-sensitive OS features relevant for HPC should be provided by the LWK, leaving only non-performance sensitive services to be provided by Linux. The following are key questions that need to be addressed to be successful: Where is the border between the LWK and Linux? How do the two kernels interact with each other so that they can provide transparent and unified node resource management? Where do tools run, and how do they interact with applications?

We discuss two approaches. First, the *proxy model*, where for each LWK application there exists a dedicated Linux process to serve as a context for delegated system calls. IHK/McKernel uses this method. Second, the *direct model*, where the LWK passes kernel data structures directly to Linux to obtain OS services. This is the approach currently studied as a possible implementation method for mOS. We then compare these two approaches from various angles and present the key trade-offs.

The rest of this paper is organized as follows. Section 2 provides background information and motivation, Section 3 discusses the design space and challenges of combining LWKs with a full weight Linux kernel. Section 4 provides early experimental results, Section 5 surveys related work, and finally, Section 6 concludes the paper.

2. BACKGROUND AND MOTIVATION

Before discussing actual design alternatives, we describe our requirements in more detail.

As described earlier, CPU cores and physical memory of the compute node are explicitly partitioned between Linux and the light-weight kernel. An important aspect of achieving the high-level goal of enabling scalability, is performance isolation. LWKs have been shown to exhibit low OS noise/jitter, which is essential for large scale execution of bulk synchronous parallel applications. A multi-kernel approach must preserve the low-noise profile of an LWK. Therefore, the LWK has to be sufficiently isolated from any negative performance impact of Linux. Furthermore, the LWK should also use as little memory and CPU resources as possible, and make the rest available to the application.

Another important requirement for an LWK, is the ability to nimbly be adapted to new technologies. Therefore, even in a hybrid configuration, the LWK code base should be small so that it can serve as a vehicle for rapid prototyping and implementation of services needed to support leading edge next-generation hardware and software features. In fact, in a multi-kernel approach, the LWK should be able to be even smaller than if it needed to manage a node's resource by itself. The need for supporting next-generation hardware becomes apparent when considering the anticipated architectural changes of the future, such as new memory technologies and deep memory hierarchies, heterogeneous core architectures, possibly multiple cache coherence domains, as well as new software models such as fine-grained threading and asynchronicity.

An important aspect of the proposed symbiotic kernel architecture is to keep the non-up-streamed Linux modifications and dependence on Linux kernel changes minimal. This goal is motivated by the desire to minimize the development effort of tracking Linux changes.

Finally, full Linux compatibility is highly desired. Performance and monitoring tools, debuggers, etc. are essential for application development and tuning. Linux exposes a number of interfaces leveraged by tools. Various system information via the /proc and /sys file systems, the availability of hardware performance counters (e.g., the PAPI interface [13]), the ability to track processes and system calls through the ptrace mechanism should all be available and work for applications running in the LWK partition.

Depending on the integration technique between Linux and the LWK some of these requirements are easier or harder to meet, as we will see in the following sections.

3. DESIGN EXPLORATION

This section describes two approaches to implementing the required interaction between Linux and an LWK. The *proxy model* for which we use McKernel as an example, and the *direct model*, which is the approach currently being investigated for mOS.

3.1 The Proxy Model

The basic idea of the proxy model is that for each application executed on the LWK, a corresponding proxy process (also referred to as *ghost process*) on the Linux side is spawned. This architecture is shown in Figure 1.

Because the LWK implements only a subset of the Linux services, i.e., the performance sensitive system calls, the rest of the OS services need to be executed on Linux. Essentially, the proxy process provides the execution context on behalf of the application (running on the LWK) so that the offloaded calls can be directly invoked. The proxy process also serves the purpose of ensuring that Linux maintains certain state information that would have to be otherwise kept track of in the LWK. As we will see later, McKernel has no notion of file descriptors, but rather it simply returns the number it receives from the proxy process when a file is opened. The actual set of open files (i.e., file descriptor table, file positions, etc..) are managed by the Linux kernel. On the other hand, holding state in Linux implies a certain degree

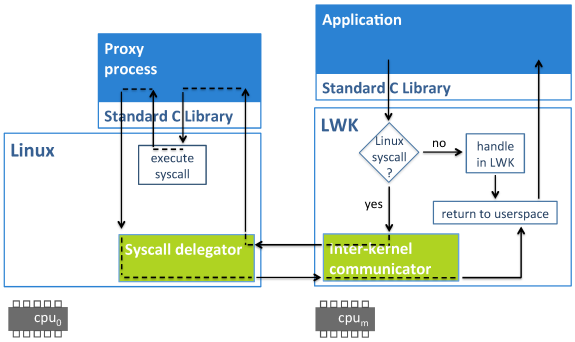

Figure 1: Overview of the proxy model and the system call delegation mechanism.

of synchronization between the LWK and the Linux state, e.g., the unified address space described below.

3.1.1 IHK and McKernel

We give a brief introduction to the main components of McKernel to provide a basis for further discussion [6, 17]. McKernel relies on a low-level software infrastructure called Interface for Heterogeneous Kernels (IHK) [17]. IHK is a general framework that provides capabilities for partitioning resources in a many-core environment (i.e., partitioning CPU cores and physical memory). It enables management of lightweight kernels (i.e., loading, booting, etc.) and it also provides an Inter-Kernel Communication (IKC) layer, upon which system call delegation can be implemented. McKernel is a lightweight kernel designed for HPC workloads. It can only be booted from IHK, and it requires the presence of Linux for running actual applications.

When it comes to designing such a hybrid LWK system, one of the most important questions is: which kernel features should the LWK itself implement? According to the requirements discussed in Section 2, McKernel provides native support only for a minimal set of kernel features, the ones that are either performance critical or change the local processor's state. It has its own memory management, it supports processes and multi-threading with a simple round-robin co-operative scheduler, and it implements signaling. It also allows inter-process memory mappings and it provides interfaces to hardware performance counters. McKernel has no native support for disk device drivers, file systems, etc., and all these services are available with the help of Linux.

In terms of system calls, our approach is that McKernel implements only the very necessary set of calls so that the aforementioned services are attainable. The rest of the system calls are offloaded to Linux (also detailed in Figure 1). In Linux, a delegator kernel module handles IKC channels for system call delegation between McKernel and the proxy process that performs the calls on behalf of the actual application. During system call delegation, McKernel sends a message to Linux via a dedicated IKC channel. As mentioned earlier, for each application on the LWK, a corresponding proxy process exists in Linux. The proxy process waits for system call requests via an ioctl() call into the delegator kernel module. The delegator kernel module's IKC interrupt handler wakes up the ghost process when it receives a system call request, passing the information necessary to execute the system call (i.e., system call number and its arguments). The ghost process then executes the system call and requests that the delegator module sends the result back to McKernel, which simply passes the return value to user-space.

A problem arises, however, as to how the ghost process on Linux can access the memory of the application running on McKernel and how the virtual addresses in arguments can be resolved. The problem arises because certain system call arguments may be just pointers (e.g., the buffer argument of a read() system call).

3.1.2 Handling Pointers in Delegated System Calls

There are multiple ways to tackle the pointer problem. One solution would be that McKernel rewrites the pointer arguments in system calls to physical addresses. Linux could then map McKernel's physical memory to its local virtual memory, and perform the system call by rewriting again the address of the physical pointers to Linux virtual addresses. Note, that in an SMP environment the physical addresses are the same both on the Linux side and for the LWK. However, this requires the knowledge of which arguments are pointers for all system calls. This is cumbersome (and may not be possible) because certain system calls can have context dependent semantics of their arguments, such as *ioctl()*. Furthermore, this solution would also require that mappings are contiguous in physical memory.

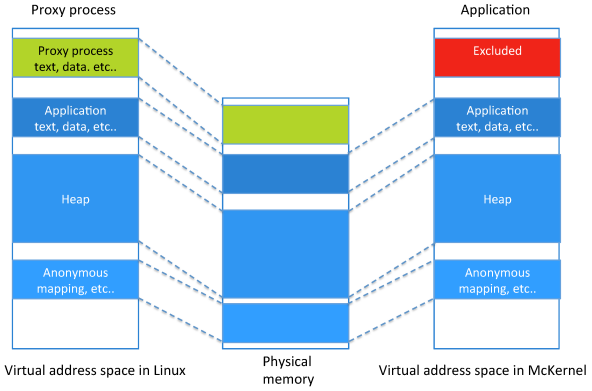

Figure 2: Unified virtual address space of the proxy process on Linux and the corresponding application on the LWK.

The second solution, the one McKernel utilizes, is that the proxy process employs the same virtual to physical mappings as the actual application, as illustrated in Figure 2. This so-called *unified address space layout* allows the ghost process to access the memory area of the application using the same virtual addresses. The code and data segments specific to the proxy process are mapped in an address range which is explicitly excluded from McKernel's user space region.

The benefit is that there is no need to recognize which arguments of a system call are addresses. Moreover, any side effects of a system call (e.g., modifications to user-space data carried out by the Linux kernel) can naturally proceed.

The proxy does not need to know in advance which virtual address is mapped to which physical page. This is because Linux uses a special mapping that covers the entire McKernel user space virtual address range, and every time an

unmapped address is accessed, the page fault handler consults the page tables corresponding to the application on the LWK. As described earlier, this requires that the mappings are occasionally synchronized, for instance, when the application calls *munmap()* or modifies certain mappings.

3.2 The Direct Model

In the direct model, an LWK sends requests directly to the Linux kernel without going through a user-space proxy process. The mOS team is currently investigating a mechanism called *Evanescence* to accomplish this. It plays a similar role as McKernel's IKC described in Section 3.1.1, but uses the Linux process affinity mechanism. When an LWK process makes a system call for which mOS will leverage Linux to service, the processing of that request is moved to the Linux core instead of explicitly moving data (e.g., system call arguments). This means the LWK and Linux are more tightly integrated with each other thereby allowing the LWK to use Linux functionality that would otherwise have to be written in the LWK.

The Evanescence mechanism works as follows. When an LWK task makes a system call that will be handled by Linux, the task is temporarily moved to the Linux core (without any additional communication between the kernels), the system call is processed there and then the task is moved back to the LWK. The data, arguments and buffer contents, logically remain on the LWK side. Since the LWK and Linux share memory, Linux can access that data from its CPU core and no data movement or marshaling is necessary. Due to hardware caching, some data may migrate from the cache of one core to another and back.

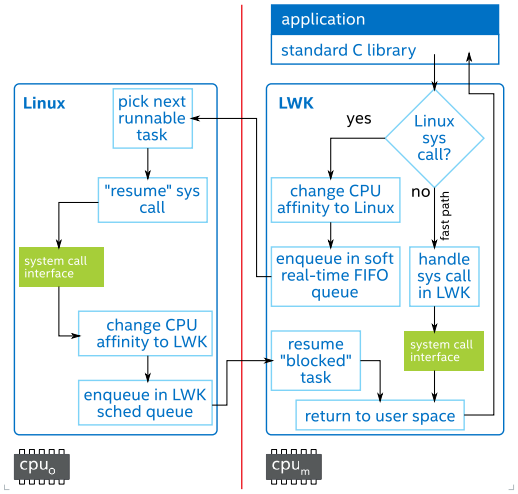

Figure 3: System call execution using the Evanescence mechanism.

The *Evanescence* mechanism is shown in Figure 3. When a task on the LWK side makes a system call and requests a service that Linux will provide, it is moved (briefly vanishing or fading away) from the LWK side to be processed on Linux. It will return once Linux has done the work needed by the system call requested by the process.

3.2.1 mOS

To explore Evanescence, we implemented a prototype in mOS [19]. In the Evanescence model, there is a closer integration between Linux and the LWK than the proxy model. In mOS' current prototype, the LWK code is compiled directly into the Linux image, as opposed to McKernel, where the LWK is an independent ELF binary. This tight integration allows mOS to take advantage of Linux functionalities in a more straightforward fashion, which will be demonstrated in Section 3.3.

mOS employs techniques to ensure that Linux does not interfere with the LWK resources (i.e., CPU cores and memory assigned to the LWK) when responding to requests. However, it allows Linux to be aware of their existence. These techniques involve isolating the LWK CPU cores from the Linux kernel scheduler, reserving physical memory to the LWK, and ensuring that no Linux interrupts are delivered to LWK cores. The LWK code base primarily provides process and memory management, and a simple scheduler.

The way Evanescence implements system call delegation is through the task's processor affinity and is performed as follows. After the LWK decides to hand-off a particular call, it saves the affinity mask of the process and changes the active mask so it only contains Linux CPUs. The original mask only stores LWK CPUs. Since the LWK scheduler does not migrate processes unless explicitly told so by the application, in most cases only one of the LWK bits is set.

After saving and modifying the process affinity mask, the process is enqueued in the soft real-time FIFO queue of the Linux scheduler. That is the highest priority queue on the Linux side and will give LWK processes preferred treatment. This may impact interactive response time when many LWK processes demand the attention of the Linux kernel, but that is acceptable behavior for a compute node in a high-end HPC system. Moreover, the same issue exists in the proxy model.

The Linux scheduler will pick the next runnable task from the queue and resume the system call that had started on the LWK side. When the call finishes, its affinity mask is set back to the original LWK-only mask, the process is transferred back to the LWK side, and control is returned to user space. During the system call execution, the LWK thread sits idle, waiting for the system call to complete and the task to migrate back.

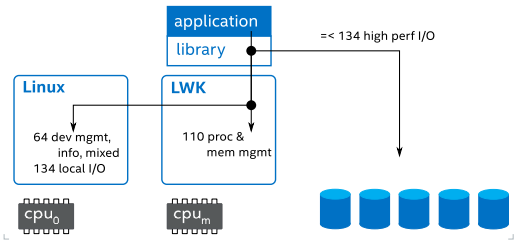

Figure 4: System call routes in mOS with the corresponding number of calls for each path.

There are three routes an mOS system call can take. If it is a high-performance I/O call, it may get intercepted in user space and routed to the parallel file system on the I/O nodes. The second route is taken if the call is to be processed by the LWK, and the third route is used if the Linux kernel is to handle the call. Figure 4 illustrates this and shows the approximate number of different calls implemented by each route.

Although this is not specific to the direct model, we discuss the mOS's intended system call distribution. There are

134 system calls related to I/O. Many of those, will be intercepted in user space and directed to the parallel file system. If they are directed at local devices and pseudo devices such as RAM disks or procfs, then those 134 calls are handled by Linux on the node.

There are 110 system calls that deal with process and memory management. They are handled by the LWK and comprise the main pieces of the LWK. There are 64 calls that deal with device management, information, or are mixed; e.g., calls like *mmap()* which deal with memory and I/O. These 64 calls need to be handled by Linux and, in some cases, also by the LWK.

3.3 Comparison

This section offers a comparison between the two models.

3.3.1 POSIX Compliance and Tools Support

One of the major concerns with the Linux+LWK integration is to maintain full POSIX and Linux compatibility also on the LWK side. This involves multiple components, such as correct system call API and execution, valid signaling behavior, and access to various statistical information, such as the /proc, /sys file systems or other Linux specific interfaces. Furthermore, the availability of the ptrace interface, upon which multiple tools rely on, as well as access to performance counters, such as the PAPI interface [13], are also required.

Providing POSIX compliance in the proxy model is generally a significantly bigger effort than in the direct one, mainly because the LWK in the proxy model is completely isolated from Linux kernel code. McKernel for instance implements a Linux ABI compatible syscall interface, signaling and the ptrace infrastructure entirely from scratch. Tools that rely on the ptrace interface therefore should either execute the directly on the LWK, or an additional component (such as the GDB server) would be required.

Accessing the /proc and /sys file systems is enabled in McKernel via redirections from the syscall delegation kernel module, which means for each /proc entry, the underlying implementation has to consult the LWK to obtain the right information. Performance counters, at the time of writing this paper, are only available via special system calls that expose direct access to the counters from user-space, but PAPI is part of our future plans.

In contrast, because mOS maintains Linux compatibility on the level of core kernel data structures, /proc and /sys entries can directly fetch the necessary information from the kernel without an additional indirection. Because mOS is in a considerably early phase at this moment, it is unclear whether signaling and ptrace will also be available without any extra development, but hopes are high that they will be. Accordingly, most of the tools are expected to run in the Linux partition. Obviously, the syscall interface and PAPI can be directly leveraged from Linux.

Another interesting example is virtual dynamic shared objects (VDSO). McKernel implements its own VDSO page, which at this point simply falls back to an actual system call, while mOS can take advantage of the Linux implementation. In summary, the proxy model needs to implement substantially more features by itself than the direct model, but in turn it has more control over the implementation details of those features.

3.3.2 LWK Flexibility

Full control over the kernel source code as well as the features implemented in the LWK is one of the major arguments in favor of a light-weight kernel. While it has been shown earlier that with careful modifications Linux can also attain scalable performance on a large-scale system [14, 16, 20], rapidly prototyping exotic features, such as support for unconventional hardware is substantially easier on a smaller code base.

As we mentioned earlier, even the near term anticipated changes in architectures, such as heterogeneous cores, multiple levels of memory hierarchy, etc. raise concerns of Linux' ability to adapt. Considering the possibility of more disruptive changes in the future (e.g., elimination of CPU ring zero, configurable cache coherency domains or a much simplified, segmentation-like virtual memory system) are even more alarming.

In theory, an LWK in the proxy model (and McKernel in particular) runs its own ELF image and thus it is in full control of its code base. There is nothing that prevents its feature set as well as their implementation details to be considerably more simple and easy to modify, which we have already demonstrated in some of our previous studies [18, 7].

mOS, on the other hand has slightly tighter integration with Linux, because it needs to be able to pass Linux kernel compatible data structures in its system call offloading mechanism and makes use of Linux macros and utility functions when appropriate. Theoretically, an LWK in the direct model could still diverge from using Linux compatible data structures internally, but it would have to carefully translate them back when offloading system calls. Consequently, the need for such conversion could impose certain restrictions on the degree of nimbleness an LWK could possibly attain. However, practical implications of this restriction remains to be seen.

It is also worth noting that the LWK's separate nature in case of the proxy model allows a proprietary kernel image to be deployed. Because the direct model compiles the LWK code directly into Linux, it automatically inherits its GPL restrictions.

3.3.3 Required Linux Modifications

As we outlined above, taking either path of integration between Linux and an LWK, it is highly desirable to keep Linux changes minimal. The Linux code base is a rapidly evolving target and keeping patches up-to-date with the latest kernel changes can be a major development effort.

The proxy model in principle requires no changes to Linux and can work in the form of a stand-alone kernel module. IHK/McKernel currently demonstrates this, although IHK relies on accessing unexported kernel symbols for resource partitioning. While this is not the intended usage of kernel modules in general, the Linux community seems to accept it and modules with similar mechanisms (e.g., the BLCR checkpoint/restart library [5]) are part of major Linux distributions.

Evanescence requires a minimal set of patches to Linux, and mOS at this time comprises approximately 150 lines of code changes to the Linux kernel. These are mostly related to changes to the system call wrapper macros, which enable Evanescence's offloading mechanism. Of course, mOS' objective is to keep the number of changes small and implement most LWK features isolated from the Linux codebase.

3.3.4 LWK Reboot Capability

The reboot capability of an LWK is a somewhat controversial issue with diverse opinions across the HPC OS community. While a reboot may require extra time, and technically it is possible to correct any issues that are overcome by rebooting, the reality is that rebooting can fix many problems that are difficult to diagnose and may provide features that reduces complexity of the overall system.

We believe that the ability of rebooting an LWK with relatively low overhead may provide the following advantages:

- A reboot can ensure that the LWK starts from a clearly defined, clean state with contiguous chunks of free memory.
- Rebooting allows easy deployment for different versions of an LWK image, depending on the specific kernel features an application may require.
- Application performance tuning is difficult and adding variables of previous job execution history is not helpful, rebooting on the other hand leads to more predictable behavior.
- The reboot every time design might highly simplify the LWK itself, it may not need concepts such as switching between applications or between users.

The proxy model, and IHK/McKernel in particular, allows warm-rebooting LWK cores. Unless Linux kernel state is corrupted, a warm reboot of LWK cores may solve problems where otherwise rebooting the whole node would be required, such as if the LWK is trapped in an inconsistent state.

The direct model, using Evanescence, is built upon a much tighter symbiosis between Linux and the LWK and directly rebooting LWK cores is not supported in its strict sense. In Evanescence, LWK cores do not go through their own trampoline code, nevertheless, the Linux CPU hotplug system can be used to unplug and re-plug cores. However, because the hotplug system assumes a correct CPU state in the first place, it may not be able to handle reboot requests to the same degree as the proxy model.

4. EVALUATION

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.

Nevertheless, we provide some comparative results on the cost breakdown of offloading an empty system call. The measurements were carried out on an Intel Xeon Ivy Bridge (E5-2670) system and for both the McKernel and mOS scenarios the same set of CPU cores were partitioned between Linux and LWK.

Figure 5 indicates the results, averaged over 1,000,000 samples. Considering the fact that IHK/McKernel involves significantly more components (i.e., the delegator kernel module, the proxy process, etc.) we found it surprising that the two mechanisms perform similarly. As seen, in McKernel's case the most expensive component is the IKC communication, which involves IPIs, the cost of the IRQ handlers, and waking up the proxy process. On the other hand, the numbers on mOS clearly show that reaffinitization is the heaviest component. Because mOS is an early prototype phase, we are hoping that this number can be significantly improved in the future.

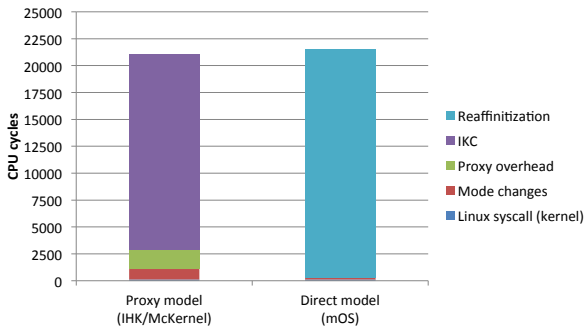

Figure 5: Cost breakdown of system call delegation.

5. RELATED WORK

Taking the hybrid nature of the proposed kernel architectures into account, a large number of related projects could be discussed. Without striving for completeness, we pinpoint some of the most relevant studies on HPC lightweight kernels, existing hybrid solutions, and kernels for many-cores in general.

5.1 Lightweight Kernels

Lightweight kernels explicitly designed for HPC workloads date back over 20 years now. Catamount [9] from Sandia National Laboratories was one of the notable systems which has been developed from scratch and successfully deployed on a large scale supercomputer. The IBM BlueGene line of supercomputers have also been running an HPC targeted lightweight kernel called CNK [8]. Nevertheless, CNK borrows a significant amount of code from Linux (e.g., glibc, NPTL) so that it can comply with elaborate Unix features, which have been increasingly demanded by the growing complexity of nowadays' HPC applications. The most current in Sandia National Lab's lightweight compute node kernels line of effort is Kitten [2], which distinguishes itself from their prior LWKs by providing a more complete Linux-compatible user environment. It also features a virtual machine monitor capability via Palacios [11] that allows full-featured guest OSs. However, with the ever growing appetite for full Unix/-POSIX feature compatibility from the application side, it has become increasingly difficult to support all these features without compromising the primary goal of LWK performance.

On the other end of the lightweight kernel spectrum are kernels which originate from Linux, but have been heavily modified to meet HPC requirements ensuring low noise, scalability and predictable application performance. Cray's Extreme Scale Linux [14, 16] and ZeptoOS [20] follow this approach. They often employ techniques, such as eliminating daemon processes, simplifying the scheduler or replacing the memory management system. There are mainly two problems with the Linux approach. First, the heavy modifications occasionally break Linux compatibility, which is highly undesirable. Second, because HPC tends to follow (or rather dictate) rapid hardware changes that need to be reflected in kernel code, Linux often falls behind with the necessary updates which results in an endless need for main-

taining Linux patches. In contrast, both mOS and IHK/McKernel aim at full Linux compatibility without sacrificing LWK performance.

5.2 Kernels for Multi/Many-cores

K42 [10] was a research OS designed from the ground up to be scalable. Similarly how mOS and IHK/McKernel selectively implement a set of performance sensitive system calls on the LWK side, K42 allowed the application to circumvent the Linux APIs and call native K42 interfaces. However, it involved a significant entanglement with Linux which made it difficult to keep track of the latest kernel changes. While mOS and McKernel also rely on Linux, as discussed above, one of their important design criteria was to minimize the engineering effort required to keep them up-to-date with the rapidly evolving Linux kernel code base.

Multiple kernels such as Tessellation [12] and Multikernel [3] are built upon the observation that modern node hardware resembles a networked system and so the OS should be modeled as a distributed system as well. The Tessellation project [12] follows a resource partitioning approach called Space-Time Partitions. It divides CPU cores into groups called cells, where each cell is responsible for a particular application or some system services. Since applications and system services can be assigned to separate cells, Tessellation's structure resembles both mOS and IHK/McKernel, where HPC workloads are explicitly assigned to LWK cores while system daemons reside on the Linux partition.

Multikernel [3] runs a small kernel on each CPU core and the OS is built as a set of cooperating processes, where each process is running on one of the kernels and communicating via message passing. Similarly to Multikernel, the IHK/McKernel model employs a message passing facility which enables communication between the two types of kernels and thus, between the actual application and its proxy process.

Zellweger et. al have recently proposed decoupling CPU cores, kernels and operating systems [21]. Their system enables applications to be seamlessly migrated over to a separate OS node while the kernel is updated on a particular CPU core. In Evanescence, process representation in LWK retains compatibility with Linux kernel data structures so that it can directly migrate processes for system call offloading. This mechanism is similar to the idea of decoupling the application state from the OS, as proposed in [21], although Evanescence's purpose is to execute certain system calls in a different kernel context where a richer set of kernel features is available.

5.3 Hybrid Kernels for HPC

FusedOS [15] was the first proposal to combine Linux with an LWK. It's primary objective was addressing core heterogeneity between system and application cores and at the same time providing a standard operating environment. Contrary to mOS and McKernel, FusedOS runs the LWK at user level. In the FusedOS prototype, the kernel code on the application core is simply a stub that offloads system calls to a corresponding user-level proxy process called CL. The proxy process itself is similar to that in IHK/McKernel, but in FusedOS the entire LWK is implemented within the CL process on Linux. This provides the same functionality as the Blue Gene CNK from which CL was derived. The FusedOS work was the first to demonstrate that Linux noise can be isolated to the Linux cores and avoid interference with the HPC application running on the LWK cores. This property has been one of the main driver for both mOS and McKernel models.

Hobbes [4] is one of the DOE's on-going Operating System and Runtime (OS/R) framework for extreme-scale systems. The central theme of the Hobbes design is to explicitly support application composition, which is emerging as a key approach for applications to address scalability and power concerns anticipated with coming extreme-scale architectures. Hobbes makes use of virtualization technologies to provide the flexibility to support requirements of application components for different node-level operating systems and runtimes. At the bottom of the software stack, Hobbes relies on Kitten [2] as its LWK component, on top of which Palacios [11] is in charge to serve as a virtual machine monitor.

Argo [1] is another DOE OS/R project targeted at applications with complex work flows. Like Hobbes, they envision using OS and runtime specialization inside the compute node. In Argo's 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 and McKernel, Argo's kernels cooperate and are trusted, but do not form a single system image. Unlike Hobbes, virtualization is not used on a node.

6. CONCLUSION AND FUTURE WORK

Moving to exascale and beyond, there is a need to enhance system software so that it meets the needs and challenges of applications. Both Intel and RIKEN AICS have been investigating a hybrid Linux plus LWK solution, with the goal of providing LWK performance while retaining the Linux APIs. Through our collaboration, we have explored alternative mechanisms for interaction between the LWK and Linux and in this paper we detailed the approaches and their tradeoffs.

In summary, the direct model, due to its tight integration with Linux, requires less development effort for providing Linux compatibility and for supporting tools. On the other hand, the proxy model can retain full control over the code base of the LWK and thus has a higher degree of flexibility for nimbly incorporating new technologies.

We will continue working closely together and exchanging experiences on a regular basis. We are excited to see how the two approaches evolve and to what extent they will be successful.

Acknowledgment

This work is partially funded by MEXT's program for the Development and Improvement for the Next Generation Ultra High-Speed Computer System, under its Subsidies for Operating the Specific Advanced Large Research Facilities.

We acknowledge Tomoki Shirasawa and Gou Nakamura from Hitachi for their McKernel development efforts. We also want to thank Pardo Keppel, Todd Inglett, Kurt Alstrup, Steve Hampson and David Van Dresser from Intel for their participation in countless conversations on mOS.

7. REFERENCES

[1] Argo: An Exascale Operating System (Accessed: Jan, 2015). http://www.mcs.anl.gov/project/argo-exascale-operating-system.

[2] Kitten: A Lightweight Operating System for Ultrascale Supercomputers (Accessed: Jan, 2015). https://software.sandia.gov/trac/kitten.

[3] Baumann, A., Barham, P., Dagand, P.-E., Harris, T., Isaacs, R., Peter, S., Roscoe, T., Schüpbach, A., and Singhania, A. The multikernel: a new OS architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles* (2009), SOSP '09, pp. 29–44.

[4] Brightwell, R., Oldfield, R., Maccabe, A. B., and Bernholdt, D. E. 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* (2013), ROSS '13, pp. 2:1–2:8.

[5] Duell, J. The design and implementation of Berkeley Lab Linux Checkpoint/restart. Technical report, Lawrence Berkeley National Laboratory, 2000.

[6] Gerofi, B., Shimada, A., Hori, A., and Ishikawa, Y. 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* (may 2013).

[7] Gerofi, B., Shimada, A., Hori, A., Masamichi, T., and Ishikawa, Y. CMCP: A Novel Page Replacement Policy for System Level Hierarchical Memory Management on Many-cores. In *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing* (New York, NY, USA, 2014), HPDC '14, ACM, pp. 73–84.

[8] Giampapa, M., Gooding, T., Inglett, T., and Wisniewski, R. W. 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, Networking, Storage and Analysis* (2010), SC '10, pp. 1–10.

[9] Kelly, S. M., and Brightwell, R. Software architecture of the light weight kernel, Catamount. In *In Cray User Group* (2005), pp. 16–19.

[10] Krieger, O., Auslander, M., Rosenburg, B., Wisniewski, R. W., Xenidis, J., Da Silva, D., Ostrowski, M., Appavoo, J., Butrico, M., Mergen, M., Waterland, A., and Uhlig, V. K42: Building a Complete Operating System. *SIGOPS Oper. Syst. Rev. 40*, 4 (Apr. 2006), 133–145.

[11] Lange, J., Pedretti, K., Hudson, T., Dinda, P., Cui, Z., Xia, L., Bridges, P., Gocke, A., Jaconette, S., Levenhagen, M., and Brightwell, R. Palacios and Kitten: New high performance operating systems for scalable virtualized and native supercomputing. In *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on* (April 2010), pp. 1–12.

[12] Liu, R., Klues, K., Bird, S., Hofmeyr, S., Asanović, K., and Kubiatowicz, J. Tessellation: Space-time Partitioning in a Manycore Client OS. In *Proceedings of the First USENIX Conference on Hot Topics in Parallelism* (2009), HotPar'09, pp. 10–10.

[13] Mucci, P. J., Browne, S., Deane, C., and Ho, G. PAPI: A Portable Interface to Hardware Performance Counters. In *In Proceedings of the Department of Defense HPCMP Users Group Conference* (1999), pp. 7–10.

[14] Oral, S., Wang, F., Dillow, D. A., Miller, R., Shipman, G. M., Maxwell, D., Henseler, D., Becklehimer, J., and Larkin, J. Reducing Application Runtime Variability on Jaguar XT5. In *In Proceedings of Cray User Group* (2010), CUG'10.

[15] Park, Y., Van Hensbergen, E., Hillenbrand, M., Inglett, T., Rosenburg, B., Ryu, K. D., and Wisniewski, R. 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* (Oct 2012), pp. 211–218.

[16] Pritchard, H., Roweth, D., Henseler, D., and Cassella, P. Leveraging the Cray Linux Environment Core Specialization Feature to Realize MPI Asynchronous Progress on Cray XE Systems. In *In Proceedings of Cray User Group* (2012), CUG'12.

[17] Shimosawa, T., Gerofi, B., Takagi, M., Nakamura, G., Shirasawa, T., Saeki, Y., Shimizu, M., Hori, A., and Ishikawa, Y. Interface for Heterogeneous Kernels: A Framework to Enable Hybrid OS Designs targeting High Performance Computing on Manycore Architectures. In *High Performance Computing (HiPC), 2014 21th International Conference on* (Dec 2014), HiPC '14.

[18] Soma, Y., Gerofi, B., and Ishikawa, Y. Revisiting Virtual Memory for High Performance Computing on Manycore Architectures: A Hybrid Segmentation Kernel Approach. In *Proceedings of the 4th International Workshop on Runtime and Operating Systems for Supercomputers* (2014), ROSS '14.

[19] Wisniewski, R. W., Inglett, T., Keppel, P., Murty, R., and Riesen, R. mOS: An Architecture for Extreme-scale Operating Systems. In *Proceedings of the 4th International Workshop on Runtime and Operating Systems for Supercomputers* (New York, NY, USA, 2014), ROSS '14, ACM, pp. 2:1–2:8.

[20] Yoshii, K., Iskra, K., Naik, H., Beckmanm, P., and Broekema, P. C. Characterizing the Performance of Big Memory on Blue Gene Linux. In *Proceedings of the 2009 International Conference on Parallel Processing Workshops* (2009), ICPPW '09, IEEE Computer Society, pp. 65–72.

[21] Zellweger, G., Gerber, S., Kourtis, K., and Roscoe, T. Decoupling Cores, Kernels, and Operating Systems. In *11th USENIX Symposium on Operating Systems Design and Implementation* (Broomfield, CO, Oct. 2014), OSDI '14, pp. 17–31.