A Multi-Kernel Survey for High-Performance Computing

Balazs Gerofi[†], Yutaka Ishikawa[†], Rolf Riesen[‡],
Robert W. Wisniewski[‡], Yoonho Park[§], Bryan Rosenburg[§]
[†]RIKEN Advanced Institute For Computational Science
[‡]Intel Corporation
[§]IBM T. J. Watson Research Center
{bgerofi, yutaka.ishikawa}@riken.jp, {rolf.riesen, robert.w.wisniewski}@intel.com, {yoonho,rosnbrg}@us.ibm.com

ABSTRACT

In HPC, two trends have led to the emergence and popularity of an operating-system approach in which multiple kernels are run simultaneously on each compute node. The first trend has been the increase in complexity of the HPC software environment, which has placed the traditional HPC kernel approaches under stress. Meanwhile, microprocessors with more and more cores are being produced, allowing specialization within a node. As is typical in an emerging field, different groups are considering many different approaches to deploying multi-kernels.

In this paper we identify and describe a number of ongoing HPC multi-kernel efforts. Given the increasing number of choices for implementing and providing compute node kernel functionality, users and system designers will find value in understanding the differences among the kernels (and among the perspectives) of the different multi-kernel efforts. To that end, we provide a survey of approaches and qualitatively compare and contrast the alternatives. We identify a series of criteria that characterize the salient differences among the approaches, providing users and system designers with a common language for discussing the features of a design that are relevant for them. In addition to the set of criteria for characterizing multi-kernel architectures, the paper contributes a classification of current multi-kernel projects according to those criteria.

Categories and Subject Descriptors

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

Keywords

High Performance Computing; Multi kernels; Hybrid kernels

1. INTRODUCTION

Historically, 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 effort is expended in disabling, removing, tuning, and sometimes reimplementing OS features so that the system scales up across more cores and out across larger clusters. A Light-Weight Kernel (LWK) approach often starts with a new kernel and effort is then expended in adding more and more functionality to more closely emulate an established API, typically that of Linux. There are challenges with each approach due to the performance - compatibility tension. As the role of the classic high-performance computing (HPC) machine is broadening to include uncertainty quantification, visualization, and big-data analytics, with more sophisticated workflows on the horizon, the need for greater Linux compatibility at scale is increasing. With either the pure LWK or FWK approach, this need conflicts with the HPC requirement for performance at scale.

Fortunately, another trend that has been occurring has been an increase in the number of cores in a node. This capability has allowed the emergence of an approach to compute node kernels that involves simultaneously running multiple kernels on each node. The promise of a *multi-kernel* approach, which most commonly involves running a Linux kernel and a LWK together, is to deliver high performance and scalability, but at the same time to maintain a full-function Linux API. Some approaches focus on performance isolation, some on application composition, and others triage system calls, with the LWK handling performance-sensitive calls and Linux handling the rest for compatibility. The vision of this approach is compelling given the requirements placed on the compute node OS and the capability offered by modern hardware, but it is not without challenges and tradeoffs. There have been a series of new projects undertaken to explore this approach [17, 2, 21, 20, 7, 1, 24]. These projects have taken different approaches as to, for example, whether the LWK is run in privileged mode or user space, whether the LWK and Linux are running the same or different address spaces, and the mechanisms by which the LWK makes requests of Linux.

In addition to having a significant impact on the implementation of a multi-kernel, these tradeoffs affect applications, runtimes, and especially, tools. For example, the function to read `/proc/cpuinfo` may produce different answers when run on the Linux kernel and on the LWK. `/proc/cpuinfo` might show only the cores Linux controls when run on Linux and the cores the LWK owns when run on the LWK. However, it may be more subtle; code on the LWK side or even the Linux side could be modified to recognize it was in a multi-kernel environment. Or maybe, the LWK or Linux, or

ROSS'16, June 1 2016, Kyoto, Japan
Copyright 2016 ACM 978-1-4503-4387-9/16/06 ...$15.00.
http://dx.doi.org/10.1145/2931088.2931092

both, should report only the cores that are available to the application. This example illustrates that the "right" answer is not always clear, even for simple interfaces, and it highlights potential differences among multi-kernel approaches. Some questions include, should `cpuinfo` on either side (LWK or Linux) report the number of cpus available in its kernel or to the application? Some multi-kernel approaches allow applications to span the LWK and Linux. For those, should the answer be different than for systems that do not allow applications to span the kernels? In the same vein, when a tool looking at the load on a node reports a number, should it include cpus owned by the LWK, by Linux, or by both? This example serves to show that in a multi-kernel design, differences in approach as well as in implementation can yield seemingly different environments on the same node.

In this paper we will describe different multi-kernel projects with the intent to compare and contrast them. While different definitions are possible, we have chosen to define the domain of HPC multi-kernel designs to include just those that are focused on HPC environments and that run multiple kernels on individual compute nodes at the same time, with applications able to use either of the kernels. Our hope is that the comparison can benefit kernel researchers by highlighting the differences between the approaches and their implications for multi-kernel development, can help application developers understand the different usage models, and can assist tools and middleware developers in supporting multi-kernel environments. A performance comparison is beyond the scope of this paper. There are other closely related classes of work that we describe in the related work section but that are not included as part of the analysis in the paper. The first is the lineage from which multi-kernels arise, namely LWKs and Linux. Other classes of related projects include virtualization and container technology. Containers have some of the characteristics of multi-kernels, but underneath are reliant on a single kernel for providing system functionality, and thus have different degrees of freedom for what can and cannot be considered for implementing system requests.

The paper consists of three primary contributions. First is a unified summary of the HPC multi-kernel approaches that have been taken to date. Second is an extensible set of criteria or a language to describe the differences between the multi-kernels we have analyzed. Third is a mapping of the known salient features of the described multi-kernels to the criteria. The mapping provides illustrative examples that highlight system differences from the perspectives (user/application, tool developer, kernel developer) described above.

2. MULTI-KERNEL PROJECTS

In this paper we look at five projects that combine an LWK or micro-kernel with Linux.

2.1 FusedOS

FusedOS was the first implementation to combine Linux with an LWK. The FusedOS design assumed a heterogeneous hardware architecture that includes both fullweight and lightweight cores. The lightweight cores have an ISA similar to that of the full cores, but, for space and power reasons, lack functionality such as a supervisor mode.

Linux runs on the full cores and partitions the hardware resources between itself and LWKs running as user-level processes. To run an application, the LWK first requests hardware resources such as lightweight cores and memory from the FWK. It then loads and starts the application through a hardware interface to the lightweight cores. All system calls and exceptions generated by the application are forwarded to Linux and handled by the LWK process.

A FusedOS prototype has been implemented on IBM's Blue Gene/Q and is available on GitHub.[1] The LWK is CNK, which is well known for its low noise signature and scalability. A small supervisor-state monitor runs on the LWK cores to emulate the lightweight core hardware interface. The monitor is the only code besides the application code that runs on the these cores. The monitor and LWK communicate through a shared memory area. For example, when an LWK application makes a system call or encounters an exception, the monitor stores the system call or exception information in the shared memory area and then passes control to the LWK. After the LWK services the system call or handles the exception, it resumes the LWK application.

A significant advantage of the FusedOS approach is the small number of changes required in the Linux kernel to support an LWK as a user-level process. Also, much of the LWK code base, CNK in this case, can be reused.

2.2 IHK/McKernel

IHK/McKernel is a true multi-kernel approach running Linux and LWK(s) side-by-side on compute nodes. At the heart of the stack is a low-level software infrastructure called Interface for Heterogeneous Kernels (IHK) [20]. IHK provides capabilities for dynamically partitioning resources in a many-core environment (e.g., CPU cores and physical memory) and it enables management of lightweight kernels. It also provides an Inter-Kernel Communication (IKC) layer, upon which system call delegation is implemented.

McKernel is a lightweight kernel written from scratch. It is designed for HPC and it can be booted only from IHK [7, 8]. McKernel retains a binary compatible ABI with Linux. It implements only a small set of performance-sensitive system calls and delegates the rest to Linux. Most importantly, McKernel has its own memory management, it supports processes and multi-threading with a simple round-robin cooperative scheduler, and it implements signaling.

HPC applications run primarily on McKernel, but for each LWK process there is a corresponding *proxy-process* created on Linux. The proxy process's central role is to facilitate system call offloading. Essentially, it provides an execution context on behalf of the application so that offloaded calls can be directly invoked in Linux. The proxy process also enables transparent access to kernel drivers in Linux (such as networking, file systems, etc.).

IHK/McKernel runs certain tools (e.g., GDB) directly on McKernel, but it is worth noting that this is not inherent from the multi-kernel design. Because McKernel is completely isolated from Linux, pseudo file system support comes at the price of significant development effort. On the other hand, the strict isolation enables full control over LWK kernel mechanisms and it also ensures that Linux OS jitter can not propagate to the LWK.

2.3 mOS

The *mOS* project creates a multi-kernel by embedding an LWK into the Linux kernel. This enables the LWK to make use of any Linux code when desired but still maintain

[1]https://github.com/ibm-research/fusedos

isolation at runtime. That is accomplished by running the LWK code on the compute cores, while letting Linux manage the remaining cores.

In a sense, this creates an architecture that was used in previous supercomputers [19] with strong partitioning of compute resources and service and I/O partitions. In *mOS*, the cores running Linux service `ssh` requests and run tools and other utilities that are needed to maintain a highly parallel system.

Tools, device drivers, and pseudo file systems work because LWK processes are visible to Linux as any other process. Code modifications to Linux isolate LWK processes from being managed by Linux.

The system administrator decides at node boot time which CPU and memory resources to assign to the LWK. They will not be managed by Linux. At application launch time, the user reserves the resources needed for the duration of the job. During thread creation or memory requests, resources from this reserved pool are assigned to the requester. At each step, the available resources are partitioned with appropriate protection mechanisms and made available to the requester. There is no demand paging or time-sharing. Requests exceeding the available resources fail.

2.4 FFMK (L4)

The Fast and Fault-tolerant Microkernel-based System for Exascale Computing (FFMK) [1] project investigates the feasibility of a microkernel-based hybrid OS for HPC. Specifically, it relies on the L4 microkernel and a para-virtualized Linux instance (a.k.a., L4Linux).

The basic idea of FFMK is to run HPC application directly on L4, but to provide transparent access to Linux features via L4Linux. In order to achieve this, FFMK exploits the underlying software architecture of L4 and the paravirtualized Linux, where from L4's perspective not only regular user-space applications, but also the Linux kernel itself, run as regular user processes. This allows L4Linux user processes to be decoupled from the Linux kernel by moving the underlying L4 thread to another core. Because the process's address space (from the hardware's point of view) is managed by L4, the virtual-to-physical mappings are valid and the process can simply execute user code. From Linux' point of view the process is blocked as UNINTERRUPTIBLE in the kernel scheduler. When the process makes a (POSIX) system call, execution is transferred back to Linux. Currently, every single system call requires a transition back to Linux.

One of the limitations of this mechanism stems from how L4 processes grant access to parts of their address spaces and how the L4 address space corresponding to the L4Linux' kernel space is mapped internally to user processes via L4's `mmap()` mechanism. L4's memory mapping mechanism requires that any memory region (i.e., any set of physical addresses) a user process maps has to be visible in the L4Linux kernel's address space as well. For example, a Linux device driver can not map an arbitrary PCI address into user space unless the given address is mapped inside the kernel. Since the IB driver initiates this kind of mapping, IB support on L4Linux requires some modification to the device driver code.

In summary, *FFMK* provides undisturbed execution of user-space code, but it does not use native L4 implementations for any system calls. In terms of system call execution, *FFMK*'s offloading mechanism is similar to that of *mOS*, while its policy to handle all system calls in Linux resembles that of *FusedOS*.

2.5 Hobbes (Kitten, Pisces, and Palacios)

Hobbes is one of the DOE funded OS/R projects that aims to address the needs of an exascale system software stack [6]. Hobbes is centered around the idea of application composition, and its node OS piece primarily stands on the following three pillars: the Pisces node resource manager (i.e., a Linux kernel module that enables partitioning of node resources) [16], the Kitten lightweight kernel [2], and the Palacios virtual machine monitor [11]. Hobbes provides a range of configurations using these components.

For example, Kitten can be booted either standalone or on top of a resource partition configured by Pisces. At the same time, Palacios can run on top of both Linux and Kitten, regardless of whether Kitten is booted standalone or in a Pisces partition. In this paper we will focus on two specific configurations of the Hobbes stack, which we believe are the two most representative.

2.5.1 Pisces/Kitten

The *Pisces/Kitten* configuration boots Linux on compute nodes and the Pisces resource manager is utilized to partition CPU cores, physical memory, and PCI devices so that a Kitten instance can be booted on top of a resource subset. Just as in IHK/McKernel, resources are strictly space partitioned in this model. In fact, Pisces plays a very similar role to IHK, although it is worth noting that contrary to McKernel, Kitten runs completely isolated from Linux and it manages its own devices at the PCI level. Moreover, at the time of writing this paper, Pisces/Kitten does not employ proxy processes for applications running on Kitten, although such directions are being considered for the purpose of gaining transparent access to device drivers in Linux.

2.5.2 Kitten/Palacios

The *Kitten/Palacios* configuration boots Kitten first and the Palacios VMM is leveraged to run a general purpose operating system, such as Linux, in a virtual machine. This configuration enables running applications that require the full Linux APIs in the VM, and it also allows time sharing with native Kitten processes. It is worth pointing out that this configuration bears noticeable similarities with FFMK since Kitten is in full control of the hardware resources (identically to L4) and Linux runs in a virtualized environment. Different from FFMK, however, processes are not allowed to migrate between Kitten and Linux.

3. DEFINING CHARACTERISTICS

To provide structure for this survey of multi-kernels for HPC, we need a set of characteristics that will help us differentiate the various OSes. This is early work and we are still working on finding the hierarchy and relationships that will allow us to create a full taxonomy in future work.

In this paper we focus on defining an initial set of characteristics. We begin with four possible viewing angles of these types of systems. Each is important for a different, possibly overlapping, subset of OS developers, users, application writers, and system administrators.

3.1 System Administrator Perspective

Multi-kernel systems need additional configuration to standard OS management procedures and this section enumerates aspects such as the standalone nature of LWK images, machine booting and resource partitioning.

Criteria 1.1 Standalone LWK: *Is the LWK a separate binary from Linux, and does it boot the cores it runs on?*

A bootable LWK requires code, expertise, and detailed hardware specifications - which are not always available to the public - to create and maintain the boot image.

A standalone LWK has more flexibility to support futuristic hardware which Linux may not support. However, it is also more difficult to write and maintain such an LWK.

Criteria 1.2 Node boot: *Which kernel is booted by the BIOS/Firmware of the node?*

Being able to boot a CPU core is a challenge; booting the entire node even more so, but it offers the greatest freedom to innovate. A small OS team may not have the resources to compete with the world-wide Linux community which often supports new hardware very quickly. Leveraging that body of work may be a faster path to success than letting the LWK do it all.

Criteria 1.3 Resource partitioning: *How and when are node resources partitioned?*

The systems we are looking at in this paper all have Linux as one of their kernels. Linux is not capable to share the management of resources with another kernel. Therefore, resources must be partitioned and managed by one kernel or the other in a multi-kernel system. One of the kernels boots first and at some point resources for the second kernel need to be handed over. How this is done exactly depends on the multi-kernel architecture.

We distinguish systems that partition *statically* or *dynamically* and the method of partitioning that can be done *early* during boot or *late* after the node is up.

3.2 Application Perspective

Ideally, when an application executes on a multi-kernel, it has full access to all the features and services Linux offers, but achieves the scalability and performance of running on an LWK. Many of the LWK advantages are only possible by compromising Linux compatibility. This creates tension and applications will need to decide to restrict themselves to the capabilities of the LWK or must be willingly to give up some of the LWK benefits.

Criteria 2.1 POSIX compatibility: *What is the level of POSIX support on the LWK?*

LWK of the 1990s offered only limited POSIX compatibility. I/O functions were shipped off node while most other functions were not available. With multi-kernels it is now possible to execute applications on an LWK and still access Linux functionality on the same node. This criteria measures the coverage of POSIX system calls.

Criteria 2.2 Pseudo file system support: *Is the Linux pseudo file system visible and fully supported on the LWK side?*

The pseudo file system in Linux plays an important role. Without full support, most tools and many libraries will not work. Unfortunately, this "second API" to Linux changes quickly. Maintaining Linux compatibility is therefore not easy for an LWK.

Criteria 2.3 Access method to Linux functionality: *How does an application access Linux functionality?*

HPC applications are expected to perform few or no system calls during the performance critical phase. However, the set of calls that must be handled reasonably fast and with high Linux compatibility is not small. And only a Linux kernel can provide *full* Linux compatibility.

Some multi-kernels employ a *proxy* process in the Linux partition that make system calls on behalf of the LWK processes, while some migrate the calling task to the Linux partition and let it execute the system call. Others provide *no* access to Linux while the application is running on the LWK.

Criteria 2.4 What is the system call overhead?:

All systems surveyed that allow system calls into Linux have a an additional cost of doing so.

Criteria 2.5 Shared memory between the two kernels: *Can an LWK and a Linux process share memory?*

If Linux gets called upon to deliver data to an LWK process, it is most efficient, if the data can be deposited directly into LWK memory. If tools and utilities are meant to run in the Linux partition and work with LWK processes, then sharing may be a requirement.

Criteria 2.6 Multi-kernel processes: *Can a single process with multiple threads span Linux and the LWK?*

Conceptually, running a main process on Linux with all features available, and running sub-tasks or worker threads on LWK cores seems like a natural usage model. However, this is difficult to implement and may place unwanted restrictions on the LWK because this feature would require binding the two kernels much more tightly than is currently done. This criteria looks at multi-kernel processes.

Criteria 2.7 NUMA support: *Does the LWK support NUMA?*

Modern many-core processors are NUMA architectures. Scattering processes and their data throughout such a system, instead of minding NUMA boundaries, has a huge performance impact. It is important for applications and runtime systems to align with these boundaries, but the LWK must provide the necessary information and offer default assignments.

Criteria 2.8 Performance isolation: *How is Linux limited from interfering with the LWK?*

One of the key goals of an LWK in a multi-kernel is performance isolation from Linux; i.e., the system must ensure scalable and predictable application performance and Linux must not introduce noise or otherwise interfere with the LWK.

3.3 Linux Perspective

The multi-kernels we cover in this paper all run Linux as one of their components. This section iterates some of the Linux specific aspects of this symbiosis.

Criteria 3.1 Linux tools and LWK processes: *Are LWK processes visible to standard tools like* `ps` *and* `top`*?*

One of the reasons for considering Linux for high-end HPC is the large number of tools available. For these tools to work with LWK processes, these processes need to be visible on the Linux side. An alternative approach is to run the tools in the LWK partition.

Criteria 3.2 Linux kernel modifications: *Are modifications to the Linux kernel necessary?*

A changing Linux kernel should have minimal impact on the LWK. On the other hand, for some multi-kernels to work, they need to change the Linux kernel. Extensive changes to the rapidly evolving Linux code base would be difficult to maintain.

Criteria 3.3 Linux kernel update impact: *Do Linux kernel code changes propagate to the LWK?*

Some LWK make direct use of Linux code. Changes by the Linux community to these functions propagate to the LWK. This has the advantage of being immediately up-to-date, but brings the danger of unwanted behavior.

3.4 LWK Perspective

The LWK in a multi-kernel is meant to provide a high-performance environment with excellent scalability. It also needs to provide the desired Linux features. Doing that brings up questions of LWK design and the level of Linux compatibility.

Criteria 4.1 Code isolation: *How well is the LWK code base isolated from Linux?*

Multi-kernels have the contradictory goals of maintaining isolation from Linux code changes while providing access to the latest Linux features. This criteria measures the degree of separation and the impact of Linux code changes to the LWK.

Criteria 4.2 Impact of Linux changes: *How difficult is it for the LWK to track Linux changes?*

Keeping up with Linux changes is important for a multi-kernel. What is the cost of adapting the LWK to the latest Linux kernel? Low cost makes tracking Linux kernel development easier. As the LWK and the Linux kernels evolve, the complexity to track may increase with each added feature. Not tracking Linux will lead to obsolescence of the multi-kernel.

Criteria 4.3 Development effort: *What is the cost writing and maintaining the LWK?*

One of the promises of an LWK is that a small team can write and maintain it. In multi-kernels, there is the added cost of maintaining Linux compatibility. The Linux kernel can also reduce cost by providing functionality that would otherwise need to be built into a stand-alone LWK.

Criteria 4.4 LWK code size and complexity: *How large and complex is the LWK code?*

A small LWK is important to maintain nimbleness; i.e., adding new features and porting it to novel hardware. The size of the LWK is determined my the multi-kernel architecture. The more Linux functionality it can leverage, the less code is required in the LWK.

Criteria 4.5 Physical memory management: *How much control does the LWK have over physical memory*

We distinguish between *full* which means the LWK controls the memory after Linux has released it, and *total* which means the LWK determines at boot time which physical memory blocks to manage.

Criteria 4.6 Memory type management: *How does the LWK manage the deeper and more complex memory hierarchy of modern devices?*

Letting the application manage memory is important, but providing good defaults for legacy applications is equally important.

Criteria 4.7 Virtual address management: *Which kernel decides what virtual address ranges to use?*

Linux uses a specific virtual memory address system. Some libraries, for example glibc, have some dependencies on that layout. Letting Linux manage virtual space offers the greatest level of compatibility, but also limits what the LWK can do to offer novel features.

Criteria 4.8 Process scheduling: *What scheduling policy does the LWK provide?*

The scheduler is a crucial component to provide a noise free LWK environment. Unlike the common time-sharing paradigm, LWK usually provide only cooperative, non-preemptive scheduling.

Criteria 4.9 Device drivers: *Do device drivers need to be re-implemented in the LWK?*

Taking advantage of the world-wide Linux device driver development effort is equally important to maintaining the capability to let an LWK control a device entirely.

4. COMPARISON

In this section we summarize similarities and differences among the OSes we introduced in Section 2. Our main findings are compiled in Table 1, where each project is categorized based on the criteria enumerated previously. We emphasize that the table reflects the conditions of the kernels as they are at the time of writing. These projects are evolving and will undergo modifications in the future.

We first consider the viewpoint of system administration. As seen, except for *FusedOS* and *mOS* all projects provide a standalone LWK image. An isolated LWK executable can have both advantages and disadvantages. For example, systems that support isolated LWK images can also deploy proprietary executables, while a Linux integrated code base needs to be open. Less integration with Linux also implies higher degree of control over what can and cannot be implemented, which we will discuss below under the aspect of standalone LWK source code. As for booting the host machine, besides configurations where Linux is run in a virtual machine, basically all projects rely on Linux. Except *FFMK*, which boots L4 natively, the machine is either directly booted by Linux or booting is built on top of existing Linux code. For example, although *Kitten/Palacios* boots the LWK on the node first, it borrows the Linux boot sequence to perform this task.

With respect to resource partitioning, there are mainly two approaches. *mOS* and *FusedOS* follow a static solution, which reserves LWK resources at boot time. On the other hand, *McKernel*, *Kitten/Palacios*, *Pisces/Kitten* and *FFMK* allow dynamic repartitioning of resources and most of them rely either on hot-swap capabilities of Linux or simply do time sharing in the form of hosting a virtual machine.

From an application point of view, some of the central questions of these systems are how Linux functionality is obtained and to what extent a Linux compatible environment on the LWK is provided. We distinguish two main aspects of a Linux-like environment. The POSIX system call interface, and other Linux specific APIs, such as Linux pseudo filesystems. From a portability point of view, availability of the full POSIX system call interface is highly desired and one of the preeminent promises of a multi-kernel architecture is the ability to execute applications on an LWK, but at the same time to access Linux functionality in a transparent fashion. Full POSIX syscall support is provided by most of the projects, except the ones based on Kitten. *Kitten/Palacios* and *Pisces/Kitten* isolate the LWK from Linux entirely and force applications that require the full POSIX API to run directly on Linux. Other projects provide a POSIX execution environment on the LWK by means of Linux proxy processes and system call forwarding (i.e., *McKernel* and *FusedOS*) or by directly migrating threads into Linux (such

Table 1: **Comparison of HPC multi-kernels.**

Project/ Property	**FusedOS**	**IHK/ McKernel**	**mOS**	**Pisces/ Kitten**	**Kitten/ Palacios**	**FFMK (L4)**	**Criteria**
Standalone LWK image	No	Yes	No	Yes	Yes	Yes	1.1
Node booted by	Linux	Linux	Linux	Linux	Kitten	L4	1.2
Resource partitioning	Static	Dynamic	Static	Dynamic	Dynamic	Dynamic	1.3
	Late	Late	Early	Late	Late	Late	
POSIX compatibility on LWK	Yes	Yes	Yes	No	No	Yes	2.1
`/proc`, `/sys` support on LWK	No	No	Yes	No	No	No	2.2
Access method to Linux features	Proxy	Proxy	Migrate	No	No	Migrate	2.3
Linux sys call overhead	High	High	High	–	–	High	2.4
Inter-kernel shared memory	Yes	Yes	Yes	Yes	Yes	Yes	2.5
Multi-kernel processes	No	No	No	No	No	Possible	2.6
NUMA support on LWK	No	No	Yes	No	No	No	2.7
Kernel level performance isolation	Yes	Yes	Yes	Yes	Yes	Yes	2.8
Linux tools for LWK tasks	Yes	Yes	Yes	No	No	Yes	3.1
		LWK side	Linux side	–	–	Linux side	
Unmodified Linux kernel	No	Yes	No	Yes	Yes	No	3.2
Linux updates propagate to LWK	No	No	Yes	No	No	No	3.3
Isolated LWK code base	Yes	Yes	No	Yes	Yes	Yes	4.1
Impact of Linux changes	Minimal	Minimal	Code merge	Minimal	No	L4Linux port	4.2
Development effort	Small	Significant	Ideally small	Significant	Significant	Significant	4.3
Code size (kLOC)[2]	150	65	12	213 (entire Hobbes)		32	4.4
LWK control over physical memory	Full	Full	Full	Full	Total	Total	4.5
LWK memory type control	Full	Full	Full	Full	Total	Total	4.6
Virtual address space control	Total	Total	Linux	Total	Total	Linux	4.7
Scheduler in LWK	Cooperative	Cooperative	Cooperative	Cooperative	Cooperative	Time sharing	4.8
Device driver transparency	No	Yes	Yes	No	No	No	4.9

as in *mOS* or in *FFMK*). The degree of support for pseudo file systems (i.e., `/proc` and `/sys`), which are crucial components of a full Linux runtime environment, is also uneven. Except *mOS*, most of the kernels provide very little availability of these features. *McKernel* employs a technique which overlaps some of the Linux pseudo files with LWK specific content, but support is far from complete.

Depending on whether system calls are selectively shipped or not, Linux specific calls can have a higher associated cost. Function shipping can be built on top of shared memory between Linux and the LWK, but regardless if shipping is performed or not, support for sharing memory appears to be a fundamental property of multi-kernel environments. As Linux functionality is obtained by temporally moving execution to Linux, the question of multi-kernel processes arises. Although most of these systems do not allow threads executing across the two kernels to co-operate explicitly, *FFMK* doesn not exclude the possibility of doing so.

The status of supporting non-uniform memory access architectures across these projects is also disparate. Except for *mOS*, most projects are either unaware of, or simply make no NUMA information available for applications. One exception, Kitten, provides a unique interface for discovering NUMA topology, however, at this time it is not integrated into standard tools and libraries.

Finally, another key aspect of application level concerns is performance isolation; i.e., the ability of a multi-kernel system to contain Linux OS jitter and provide consistent and predictable HPC application performance. The degree and difficulty of achieving this differs among the OS projects, but all claim a high level of isolation at this point.

Let us turn our attention to more Linux specific aspects of a multi-kernel system now and discuss support for Linux based tools first. Tools, such as debuggers or performance profilers, are of great importance in HPC systems. There are differences across the surveyed projects in how and to what extent they support execution of tools. Ideally, tools would run on Linux so that they will not expend application resources, but they would still have access to information regarding LWK processes. By architectural design, *mOS*, *FusedOS* and *FFMK* have the ability to run tools in this fashion. On the other hand, *McKernel* runs certain tools (e.g., GDB) directly on the LWK, which comes at the price of explicit support for interfaces like `prtrace()`, `prctl()`, and `perf_event_open()`.

Another important Linux perspective is whether or not Linux kernel code is modified. 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, thus keeping modifications minimal is a great concern. *McKernel* and *Pisces/Kitten* are confined to Linux kernel modules. On the other hand, both *mOS* and *FusedOS* require a small set of changes to the Linux kernel. *FFMK* utilizes a paravirtualized version of Linux. While these three projects do modify the Linux kernel code base, adapting those changes to different Linux versions has been up till now reportedly straightforward.

A tight integration with Linux can be beneficial. Linux is continuously being updated to support new CPU features

[2]The source trees of *FusedOS* and *Hobbes* (i.e., *Pisces/Kitten* and *Kitten/Palacios*) borrow a significant amount of code from Linux.

which need special initialization code. *mOS* can naturally exploit those updates, while other projects need new code in the LWK to enable them.

We will now iterate through various aspects of a multi-kernel system from the LWK's point of view. The first two criteria we consider are the relationship of the LWK code base and Linux and the impact of Linux kernel changes on a given system. As a matter of fact, these two angles are closely related. Except for *mOS*, the source code of all LWKs are completely independent from Linux, although *McKernel*'s IHK and the Pisces resource manager of *Pisces/Kitten* are somewhat entangled with Linux. Consequently, the impact of Linux kernel changes divide these systems in a similar pattern, the most sensitive being *mOS*, which requires adapting the LWK component every time a new Linux version is utilized. It is also worth noting that one of the primary motivations behind developing LWKs is their potential to support rapid experimentation with unusual software or hardware features, also referred to as the nimbleness of the kernel. From this aspect, an isolated LWK code base is highly beneficial because it provides a higher degree of freedom and control over what exactly can be explored. On the other hand, the standalone nature usually comes at the price of more significant development effort and an increased LWK code size, which is well reflected in the Table 1.

LWKs in all considered projects generally manage only the physical memory which they are given, although in case of *Kitten/Palacios* and *FFMK* the LWK has control over the machine's entire physical memory. Virtual address space management is also an interesting aspect. *mOS* currently lets Linux manage the virtual address space, while *McKernel* simply remains binary compatible with Linux. It also follows a unified address space model between the proxy process and the application so that system call offloading can proceed naturally. *Kitten/Palacios* and *FFMK* adopt their own virtual address space layout. Scheduling in most LWKs takes a simple, co-operative (i.e., non preemptive) approach, although the L4 microkernel in *FFMK* provides time sharing as well.

The last aspect we consider is device driver support. Large scale HPC environments usually rely on a relatively limited set of devices, with high speed interconnects being the most important. However, access to Linux device drivers is certainly a great benefit. *mOS* and *McKernel* provide transparent access to Linux device drivers, while *Pisces/Kitten* and *FFMK* require porting drivers to the LWK codebase. As a matter of fact, *FFMK* can utilize Linux device drivers in L4Linux, but there are minor modifications required. Similarly, *Kitten/Palacios* may also take advantage of Linux device drivers executing in the Palacios VM, however, at this time Kitten processes have no transparent access to those.

5. RELATED WORK

This section covers related studies on HPC lightweight kernels, container technologies, hybrid HPC solutions that are similar to multi-kernels, but do not strictly run heterogeneous kernels, and multi-kernels targeting commercial workloads.

Lightweight kernels explicitly designed for HPC workloads date back over two decades now. Catamount [10] from Sandia National Laboratories was one of the notable systems which has been developed from scratch and successfully deployed on large scale supercomputers. The IBM BlueGene line of supercomputers have also been running an HPC targeted lightweight kernel called CNK [9]. While most of the above mentioned kernels provide a very small set of the Linux APIs, CNK borrows a significant amount of code from Linux (e.g., from glibc) so that it can comply with more elaborate Unix features. The most current in Sandia National Laboratories' lightweight compute node kernels line of effort is Kitten [2], which we studied extensively in this paper. Kitten distinguished itself from their prior LWKs by providing a more complete Linux-compatible user environment. 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. Cray's Extreme Scale Linux [15, 18], Fujitsu's Linux on the K Computer [14] and ZeptoOS [22] 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. 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 maintaining Linux patches. Many of the above described issues served as motivation to investigate multi-kernel architectures for HPC environments.

Container technologies, such as Docker [13] or rkt [4], also bare some resemblance to LWKs in multi-kernel configurations. Although containers primarily address the problem of packaging an application along with all of its dependencies so that they can be run smoothly in disparate environments, they heavily rely on the Linux kernel's Control Groups (`cgroups`) facility, which implements resource accounting and limiting. Resource limiting plays the same role as dedicated resource sets in multi-kernel environments, however, they are implemented inside the Linux kernel. As a consequence, containers have less direct control over resource management.

Nevertheless, *ARGO*, one of two large OS and runtime projects (alongside Hobbes) funded by the US DOE, investigates whether or not container technologies are feasible for HPC. By modifying various aspects of the Linux `cgroups` behavior (e.g., fine grained exclusive reservation of physical memory), introducing a specialized scheduler for HPC, etc., *ARGO* attempts to reach similar goals as the multi-kernel projects in Section 2 in a less disruptive manner [24].

Multi-kernels have also been considered in the commercial domain. Tessellation [12] and the Multikernel [5] 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 that divides CPU cores into groups, where each group is responsible for a particular application or some system services. This structure resembles many of the HPC projects discussed earlier, where HPC workloads are explicitly assigned to LWK cores while system daemons reside in the Linux partition. Multikernel [5] runs a small kernel on each CPU core and the OS is built as a set of cooperating processes that communicating via message passing, similarly to the proxy model. Popcorn Linux [3], a recent effort of running multiple Linux instances on a single node, targets heterogeneous ISAs, while Zellweger et. al have recently proposed decoupling CPU cores, kernels and operating systems [23]. Their system enables applications to be seamlessly migrated over to a separate OS node while the kernel is updated on a particular CPU core, similarly how threads migrate in *mOS* or *FFMK*.

6. CONCLUSION AND FUTURE WORK

With the recent emergence of multi-kernel OS projects for high-performance computing there is a need to provide a basis for understanding their fundamental properties and defining characteristics. To this end, we have surveyed a number of efforts and compiled a set of criteria that underlie them. Mapping each OS to those criteria provides us not only a better understanding of the particular characteristics of these efforts, but it also exposes a clearer view of the multi-kernel design space itself.

In the future, we will further extend our exploration, hoping that we can establish a more complete taxonomy of these systems.

Acknowledgment

We acknowledge Adam Lackorzynski from TU Dresden and Kevin Pedretti from Sandia National Laboratories for their invaluable discussions on L4 and Kitten, respectively.

7. REFERENCES

[1] FFMK: A Fast and Fault-tolerant Microkernel-based System for Exascale Computing (Accessed: Mar, 2016). http://ffmk.tudos.org/.

[2] Kitten: A Lightweight Operating System for Ultrascale Supercomputers (Accessed: Mar, 2016). https://software.sandia.gov/trac/kitten.

[3] Popcorn Linux (Accessed: Mar, 2016). http://www.popcornlinux.org/.

[4] rkt - App Container runtime (Accessed: Mar, 2016). https://github.com/coreos/rkt.

[5] 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 SOSP'09*, pp. 29–44.

[6] 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 ROSS'13*, pp. 2:1–2:8.

[7] 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 *Proceedings of CCGrid'13*.

[8] 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 HPDC'14* (New York, NY, USA), ACM, pp. 73–84.

[9] 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 SC'10*, pp. 1–10.

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

[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 *Processing of IPDPS'10*, 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 HotPar'09*, pp. 10–10.

[13] Merkel, D. Docker: Lightweight Linux Containers for Consistent Development and Deployment. *Linux Journal 2014*, 239 (Mar. 2014).

[14] Moroo, J., Yamada, M., and Kato, T. Operating System for the K computer. In *Fujitsu Journal of Science and Technology* (2012), pp. 295–301.

[15] 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.

[16] Ouyang, J., Kocoloski, B., Lange, J. R., and Pedretti, K. Achieving Performance Isolation with Lightweight Co-Kernels. In *Proceedings of HPDC'15* (New York, NY, USA), ACM, pp. 149–160.

[17] 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 *Proceedings of SBAC-PAD'12*, pp. 211–218.

[18] 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.

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

[20] 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 *Proceedings of HiPC'14*.

[21] Wisniewski, R. W., Inglett, T., Keppel, P., Murty, R., and Riesen, R. mOS: An Architecture for Extreme-scale Operating Systems. In *Proceedings of ROSS'14* (New York, NY, USA), ACM, pp. 2:1–2:8.

[22] 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 ICPPW'09*, IEEE Computer Society, pp. 65–72.

[23] Zellweger, G., Gerber, S., Kourtis, K., and Roscoe, T. Decoupling Cores, Kernels, and Operating Systems. In *Proceedings of OSDI'14* (Broomfield, CO), pp. 17–31.

[24] Zounmevo, J. A., Perarnau, S., Iskra, K., Yoshii, K., Gioiosa, R., Essen, B. C. V., Gokhale, M. B., and Leon, E. A. A Container-Based Approach to OS Specialization for Exascale Computing. In *Proceedings of WoC'15*.