Toward Full Specialization of the HPC Software Stack:

Reconciling Application Containers and Lightweight Multi-kernels

Balazs Gerofi
RIKEN Advanced Institute For Computational Science
JAPAN
bgerofi@riken.jp

Rolf Riesen
Intel Corporation
USA
rolf.riesen@intel.com

Robert W. Wisniewski
Intel Corporation
USA
robert.w.wisniewski@intel.com

Yutaka Ishikawa
RIKEN Advanced Institute For Computational Science
JAPAN
yutaka.ishikawa@riken.jp

ABSTRACT

Application containers enable users to have greater control of their user-space execution environment by bundling application code with all the necessary libraries in a single software package. Lightweight multi-kernels leverage multi-core CPUs to run separate operating system (OS) kernels on different CPU cores, usually a lightweight kernel (LWK) and Linux. A multi-kernel's primary goal is attaining LWK scalability and performance in combination with support for the Linux APIs and environment. Both of these technologies are designed to address the increasing hardware complexity and the growing software diversity of High Performance Computing (HPC) systems. While containers enable specialization of user-space components, the LWK part of a multi-kernel system is also a form of software specialization, but targeting kernel space.

This paper proposes a framework for combining application containers with multi-kernel operating systems thereby enabling specialization across the software stack. We provide an overview of the Linux container technologies and the challenges we faced to bring these two technologies together. Results from previous work show that multi-kernels can achieve better isolation than Linux. In this work, we deployed our framework on 1,024 Intel Xeon Phi Knights Landing nodes. We highlight two important results obtained from running at a larger scale. First, we show that containers impose zero runtime overhead even at scale. Second, by taking advantage of our integrated framework, we demonstrate that users can transparently benefit from lightweight multi-kernels, attaining identical speedups to the native multi-kernel execution.

CCS CONCEPTS

• **Operating Systems** → Organization and Design;

KEYWORDS

High Performance Computing; Multi-kernels; Containers

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ROSS '17, Washingon, D.C., USA
© 2017 ACM. 978-1-4503-5086-0/17/06...$15.00
DOI: http://dx.doi.org/10.1145/3095770.3095777

ACM Reference format:
Balazs Gerofi, Rolf Riesen, Robert W. Wisniewski, and Yutaka Ishikawa. 2017. Toward Full Specialization of the HPC Software Stack: . In *Proceedings of ROSS '17, Washingon, D.C., USA, June 27, 2017,* 8 pages.
DOI: http://dx.doi.org/10.1145/3095770.3095777

1 INTRODUCTION

Containers have emerged as a lightweight virtualization alternative for efficient application deployment offering the "build once and deploy everywhere" development model. In scientific computing, application containers have been receiving attention due their potential for improving productivity, portability, and reproducibility of experimental results. Application containers enable users to have greater control of their user-space execution environment by bundling application code with the necessary libraries in a single software package. This provides a convenient way for effective distribution of the increasingly non-trivial scientific computing software stack, as well as to incorporate specialized dependencies needed by HPC environments [12]. Indeed, the increasing prevalence of Big Data, analytics, and machine-learning has diversified the workloads of high-end computing systems to such a degree that some speculate containers may become the new narrow waist of the HPC system software stack [5].

At the same time, lightweight multi-kernels have also been receiving considerable attention [10]. Lightweight multi-kernels leverage today's many-core CPUs to run multiple operating system (OS) kernels, typically a lightweight kernel (LWK) and a Linux kernel, simultaneously. The LWK provides high performance and scalability, usually through specialization targeting particular hardware features or HPC application needs, while Linux provides the necessary POSIX and Linux compatibility. The multi-kernel promise is to be able to meet tomorrow's computing needs of big data, machine and deep learning, and multi-tenancy, while at the same time, to provide the strong isolation yielding high performance and scalability needed by classical HPC applications.

Containers and multi-kernels help address the increasing hardware complexity and the growing software diversity of HPC. While containers enable specialization of user-space components, the LWK part of a multi-kernel system is also a form of software specialization, but targeting kernel space.

Our previous work [9] shows that multi-kernels can achieve better isolation than the capabilities that Linux provides. In this paper,

we focus on combining application containers with multi-kernel operating systems with the intention of allowing specialization across the software stack. Specialization is being explored in the cloud space as well. Unikernels provide specialized, single address space images, constructed by using library operating systems [21]. Unikernels target primarily virtualized environments (i.e., virtual machines), and thus require rebuilding the application. Instead, we leverage the Linux compatibility of lightweight multi-kernels and run unmodified Linux containers on top of the LWK.

This paper makes the following contributions:

- We propose a framework for combining application containers with multi-kernel operating systems, thereby enabling specialization across the entire software stack.
- We provide an overview of the Linux container technologies and describe the challenges of integrating containers into a multi-kernel environment.
- By deploying our framework on up to 1,024 Intel® Xeon Phi™ Knights Landing nodes, we show that containers impose zero runtime overhead even at scale; and by taking advantage of our integrated framework, we demonstrate that users transparently benefit from lightweight multi-kernels, attaining identical speedups to the native multi-kernel execution.

The rest of this paper is organized as follows. We begin by providing background information on containers and multi-kernels in Section 2. We then describe our framework in Section 3. We provide experimental evaluation in Section 4, and further discussion in Section 5. Section 6 surveys related work, and Section 7 concludes the paper.

2 BACKGROUND

This section provides a brief overview of Linux container technologies. We first discuss their foundations in the Linux kernel and then survey some of the existing implementations. We also present a basic overview of multi-kernels.

2.1 Application Containers

From an HPC perspective, containers provide three features. First, they allow packaging of applications, i.e., the bundling of libraries, configuration files, etc. together with the application code itself, which makes deployment easier and helps with runtime reproducibility. Second, containers enable node resource management (e.g., management of CPU cores, memory, etc.) and accounting. While this is not the most used feature of containers in a typical HPC environment, there are projects that focus on these properties. Third, though not a topic of this paper, containers are being used to isolate a classical HPC application from analytics or deep learning applications running on the same set of nodes to reduce data movement, while maintaining good performance and scalability for the HPC application. Linux provides the following features relevant to containers.

2.1.1 Namespaces. A Linux namespace is a scoped view of kernel resources, a form of lightweight virtualization in the Linux kernel. Namespaces make system resources appear to the processes within the namespace as if the processes had their own isolated instance of the particular resource. Changes to the resource are only visible to processes that are part of the same namespace. There are a few exceptions such as the propagation of mount points. Linux currently supports seven namespaces:

- **mnt:** Directory hierarchy, mount points
- **pid:** Process ID number space
- **net:** Network devices, network stacks, ports, etc.
- **ipc:** System V IPC, POSIX message queues
- **uts:** Hostname and NIS domain name
- **user:** User and group IDs (relatively new, not all Linux distributions support it)
- **cgroup:** Cgroup root directory

By default, processes run in the system-wide global namespace, but there are APIs that enable creating new name spaces. Namespace information is exposed in the `/proc` filesystem. For all processes in the system, the folder `/proc/[pid]/ns/` holds one entry for each namespace. In recent Linux versions these files are symbolic links and their value is the namespace identifier in the kernel.

There are a number of system calls that enable manipulating namespaces. The `clone()` system call creates a new thread or process, and if specified, new namespaces can be created as well. The new thread will be placed into the new namespace. `setns()` allows the calling thread to switch to the namespace specified in the argument, where the argument is a file descriptor pointing to one of the `/proc` files mentioned above. Finally, the `unshare()` system call enables creating new namespaces and placing the calling thread into them. One thing that is important from a security point of view is that most of the namespace creation routines (except for user namespaces) require root privilege.

Namespaces are the essence of containers, and although general purpose container implementations make heavy use of most of the available namespaces, containers targeting HPC environments usually rely on the `mnt` namespace. The `mnt` namespace provides the fundamental support for application packaging as it enables processes to have a private view of the filesystem, including libraries, etc. Again, this is the key for specializing an application's user-space components. More information on this topic is provided in Section 2.1.3.

2.1.2 Control Groups (cgroups). Control cgroups (i.e., `cgroups`) are a feature of the Linux kernel that enable processes to be organized into hierarchical groups. Their usage of kernel resources can then be controlled and accounted for. The `cgroups` infrastructure allows controlling resource usage, such as CPU time and affinity, memory limitations, access to devices, network usage and traffic shaping, I/O control of block devices and throttling of I/O streams, etc. The Linux kernel's cgroup API is provided through a pseudo-filesystem called `cgroupfs`.

There are two versions of cgroups. Each version provides multiple subsystems so that specific resources can be controlled. Currently, from an HPC perspective, CPU and memory limitations are the most relevant resource to be controlled, which can be accomplished via the `cpuset` subsystem. In the future, with the increasing importance of multi-tenant deployment, other subsystems may also gain relevance. However, as described in the related work section, the granularity of control for these resources is coarse grained

Table 1: Overview of Linux container technologies.

Project / Attribute	Docker [22]	rkt [6]	Singularity [15]	Shifter [24]
Supported namespaces	all	except user	user, mount, pid, ipc	mount
Uses cgroups	yes	yes	no	no
Image format	OCI	appc	img	UDI
Standardized image	yes	yes	no	no
Daemon process required	yes	no	no	no
Network isolation	yes	yes	no	no
Direct device access	possible	possible	yes	yes
Root filesystem	`pivot_root()` and `chroot()`	`pivot_root()` and `chroot()`	`chroot()`	`chroot()`
Implementation language	Go	Go	C, python, sh	C, sh

limiting the applicability of `cgroups` for HPC scenarios [28]. As described in Section 2.1.3, `cgroups` remain mostly unused in current HPC targeted container implementations.

2.1.3 Container Implementations. We investigated four container implementations and summarize our findings in Table 1. Docker is the most widely used container engine providing a rich set of features [22]. It can utilize all namespaces of the Linux kernel (including the user namespace). It also is capable of dealing with `cgroups` for resource restrictions and monitoring. Docker relies on the image format defined by the Open Containers Initiative (OCI), which in turn can be executed by the `runC` runtime. Docker relies on a daemon process, running with root privileges, to configure and spawn containers. The daemon provides additional features, such as file system layering and the management of container images and instances. Note, however, that the `runC` protocol does not strictly require this. In fact, there are independent implementations of `runC` from Docker. Docker's often mentioned weakness is the security implications of this daemon based structure.

Another well-known container engine for commercial/cloud deployments is `rkt` from `CoreOS` [6]. `rkt`'s main innovation was the elimination of the daemon process, which leads to more straightforward resource accounting. `rkt` is also capable of dealing with most of the Linux namespaces as well as with `cgroups`. Additionally, it provides the concept of *pods*, which is a unit of execution that allows bundling multiple containers together. `rkt` uses an image format called Application Container Image (ACI), whose actual format is defined in the App Container Specification (appc), a similar standardization effort to Docker's OCI.

On the HPC side of the container spectrum, there are two efforts that have emerged. Singularity from LBNL [15] is a simple container implementation with the primary motivation of enabling users to control their application environments through packaging. Singularity utilizes only a subset of namespaces for this purpose (i.e., **mnt**, **ipc**, **pid** and **user**) and it does not support `cgroups`. It uses its own image format, but provides tools to convert from other container images, for example from Docker. Singularity indicates it runs as an unprivileged user. However, its main container execution tool occasionally escalates to root privileges (for loop mounting images, etc.), and thus it runs as a setuid binary owned by root.

Shifter, from NERSC, is another HPC-oriented container engine [24]. Although Shifter primarily focuses on the distribution of container images to compute nodes, it also provides an execution runtime. Shifter's container runtime is simpler than Singularity. It supports only the **mnt** namespace. Although Shifter indicates that it can run Docker containers, in reality it converts Docker images to its own User Defined Image (UDI) format, which is then loop mounted during execution.

What makes both Singularity and Shifter appealing for HPC is their natural integration with MPI. This partially comes from their simple execution model (i.e., just setting up namespaces and `clone()` the application) and the direct access to network devices.

2.2 Multi-kernel Operating Systems

Lightweight multi-kernels leverage many-core CPUs and run multiple OS kernels, typically a lightweight kernel and a Linux kernel, simultaneously. The LWK provides high performance and scalability while Linux provides compatibility for Linux APIs and the Linux environment. To accomplish this, the LWK component of a multi-kernel usually implements performance sensitive kernel services and Linux is relied upon for the rest of the system calls through an offloading mechanism.

We consider two multi-kernels in this paper, and provide a short overview of each. IHK/McKernel is a lightweight multi-kernel developed at RIKEN. It consists of two main components: A low-level software infrastructure called Interface for Heterogeneous Kernels (IHK) [26] and an LWK called McKernel [8]. IHK is a general framework that provides capabilities for partitioning resources in a many-core environment (e.g., CPU cores and physical memory) and it enables management of lightweight kernels. We emphasize that IHK can reserve and release host resources dynamically and no reboot of the host machine is required when altering configuration.

McKernel is specialized for HPC workloads. It boots from IHK, and it requires the presence of Linux for running actual applications. For each process running on McKernel there exists a process on the Linux side, which we call the *proxy-process*. The proxy process' central role is to facilitate system call offloading. Primarily, it provides execution context on behalf of the application so that offloaded calls can be directly invoked in Linux, but it also enables Linux to maintain certain state information that would have to be otherwise kept track of in the LWK. McKernel for instance has no notion of

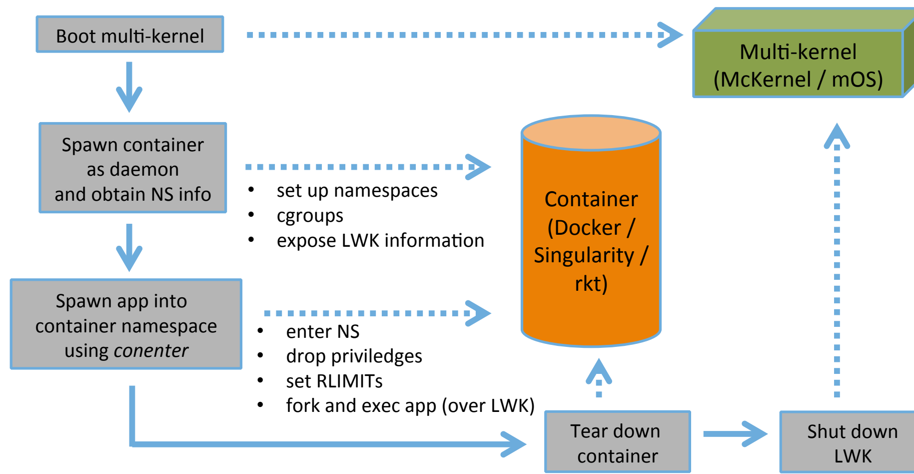


Figure 1: Container execution framework architectural overview.

file descriptors, but all file I/O operations go through the proxy. As we will see later, this is crucial with respect to containers, since the illusion of application packaging is achieved through interaction with the file system.

mOS is a lightweight multi-kernel developed at Intel. While mOS and McKernel share common goals, mOS follows a different design than IHK/McKernel. It compiles the LWK code base directly into Linux. One implication of this design decision is that mOS' system call offloading mechanism is different than that of the McKernel proxy approach. Instead of running a proxy process on Linux, mOS retains Linux kernel compatibility at the level of kernel data structures; e.g., `task_struct`, enabling mOS to, for example, move threads directly into Linux. Therefore, system call offloading is implemented via migrating the issuer thread into Linux, executing the system call, and migrating the thread back to the LWK component.

3 DESIGN AND IMPLEMENTATION

Our goal was to build a framework that would execute containers on top of lightweight multi-kernel operating systems transparently to users. We set the following conditions for our system:

- Seamless integration with MPI.
- Container engine transparency.
- Multi-kernel transparency.

Seamless integration with MPI implies that we should be able to pass a containerized binary to `mpirun` in a straightforward manner, and at the same time the framework should allow running multiple MPI ranks inside a node. Only HPC targeted container engines provide this feature by default. For example, when running MPI with Docker, an often mentioned solution is to spawn a container on each compute node with a dedicated IP address first and execute `mpirun` separately using the new IP addresses as host list. This behavior is very different than what regular MPI users expect. Container engine transparency means that we should be able to execute different container types in a unified fashion. Note that this does not imply the framework internally understands the image format of the container, but rather it interacts with the corresponding container runtime.

As an example of our framework, to execute the following MPI invocation:

```
mpirun -n <N> -env OMP_NUM_THREADS=4 -hostfile ~/hosts \
-ppn 64 /miniapps/miniFE/miniFE.x nx=660 ny=660 nz=660
```

The following command can be used:

```
mpirun -n <N> -env OMP_NUM_THREADS=4 -hostfile ~/hosts \
-ppn 64 conexec --lwk mckernel \
docker://ubuntu:miniapps /miniapps/miniFE/miniFE.x \
nx=660 ny=660 nz=660
```

As shown, the executable passed to `mpirun` is `conexec`. An architectural overview of `conexec` is shown in Figure 1. As `mpirun` may spawn multiple `conexec` processes, i.e., when multiple MPI ranks are executed inside a node, we use file locks and reference counters internally to determine when booting the multi-kernel and/or spawning the container is necessary. Specifically, the main execution steps of the framework are as follows. The first `conexec` process will set up or initialize the multi-kernel. Depending on the degree of flexibility the multi-kernel offers, this may involve offlining resources and booting an LWK, or just simply verifying that the multi-kernel is ready. The framework then spawns the container by examining its type (Docker in the example above) and calling the corresponding container APIs. Note that we spawn an empty daemonized container first, and then obtain its namespace identification. Information about the multi-kernel also needs to be exposed in the container, i.e., the necessary device files and tools (e.g., `mcexec` for McKernel and `yod` for mOS) that are used to instruct the LWK to run an application. These steps are performed by `conexec` automatically. Subsequent `conexec` processes will simply obtain the namespace information.

Another component of our framework, denoted by `conenter` in Figure 1 utilizes the `nsenter()` system call to spawn the actual application in the container identified by the namespace information in the previous step. The framework then waits until the application

returns. Relying on reference counting, the last conexec process that exits tears down the container and shuts down the LWK.

As we mentioned above, both multi-kernels utilize system call offloading for file operations. This provides us with a very convenient framework to enforce that the LWK application's view of Linux resources corresponds to that of the container's namespaces. In case of McKernel, what we need to do is to run the proxy process in the container and the LWK process will automatically inherit the right attributes. Similarly, for mOS we need to run the tool that spawns the LWK process in the container. This is shown conceptually in Figure 2.

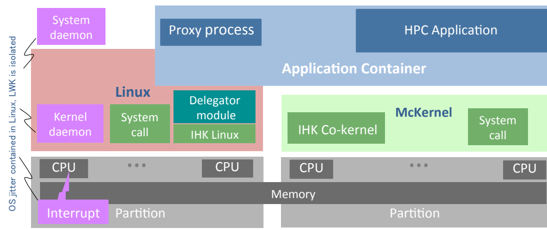


Figure 2: Containerized execution on top of IHK/McKernel.

There are a number of implementation issues we encountered to make this work. IHK's Linux kernel module makes heavy use of process IDs to identify internal data structures when system call offloading is performed. These needed to be all namespace aware to make sure we find the correct data.

Another issue worth mentioning was the Singularity container's lack of support for spawning daemonized containers as well as the missing APIs for obtaining namespace information. For this purpose, we modified the Singularity execution runtime to support those. Note, that this does not have any impact on compatibility with actual Singularity container images. Our framework currently supports Docker and Singularity, while rkt support is still work in progress.

4 EVALUATION

This section describes the experiments we performed and discusses their results.

4.1 Experimental Environment

Our experiments were performed on two platforms. A small 32-node Xeon cluster comprising Intel® Xeon™ CPU E5-2670 v2 nodes interconnected by Mellanox Infiniband MT27600 Connect-IB and on Oakforest-PACS (OFP), a Fujitsu built, 25 peta-flops supercomputer installed recently at The University of Tokyo [7]. OFP is comprised of eight-thousand compute nodes that are interconnected by Intel's Omni Path network. Each node is equipped with an Intel® Xeon Phi™ 7250 Knights Landing (KNL) processor, which consists of 68 CPU cores, accommodating 4 hardware threads per core. The processor provides 16 GB of integrated, high-bandwidth MCDRAM and is accompanied by 96 GB of DDR4 RAM. For all experiments, we configured the KNL processor in SNC-4 flat mode; i.e., MCDRAM and DDR4 RAM are addressable at different physical memory locations and both are split into four NUMA nodes. Altogether, the operating system sees 272 logical CPUs organized around eight NUMA domains. We emphasize that our experiments utilized the high-performance interconnects on both platforms.

The software environment we used is as follows. The Xeon cluster runs CentOS 7.2 with Linux kernel 3.10.0-327.4.5. The compute nodes on OFP run XPPSL 1.4.1 with Linux kernel version 3.10.0-327.22.2. XPPSL is a CentOS based distribution with various Intel provided kernel level improvements specifically supporting the KNL processor.

For the miniapp experiments on OFP, we dedicated 64 CPU cores to the application and 4 CPU cores for OS activities. For the Linux runs we used the Fujitsu's HPC optimized production environment. For the McKernel measurements we deployed IHK and McKernel, commit hash 7b3872ed and c32b1ada, respectively. We deployed Docker 1.11 and Singularity 2 on the Xeon cluster, and on OFP we used only Singularity.

4.2 Results

We highlight that our experiments aim at demonstrating the followings: First, that the proposed container execution framework works with a variety of container engines and that it can also handle multi-kernels. Second, that containers impose no overhead on execution even at large scale. And finally, that by executing the same applications in a container on top of specialized LWKs we can transparently gain the same benefits as if we ran the codes natively on the LWK.

To highlight the contrast between the host's CentOS and the OS in the container, we deployed an Ubuntu 14.04 installation both in Docker and in Singularity. Besides the application executable we added the necessary shared libraries, i.e., the network drivers and Intel MPI.

The first measurement was performed on the smaller Xeon cluster and we used the Intel MPI Benchmark (IMB) to assess the overhead that containers impose on ping-pong communication. Figure 3 summarizes the results. We ran six scenarios: native execution on Linux, native execution on McKernel, Docker on Linux, Docker on McKernel, Singularity on Linux and Singularity on McKernel. Note that we used the exact same application binary and the same libraries in the container as on the host. The key observations from these results are that indeed containers impose no overhead on communication and that we see the same latency improvement for the containerized execution on top an LWK as the native LWK case. This was expected as high-performance interconnects (such as Infiniband) are driven from user-space, bypassing the OS kernel.

In our second experiment we used up to 1,024 nodes of the OFP machine. We deployed our container execution framework as well as Singularity and used the same Ubuntu image as previously. We only needed to add the necessary shared libraries to support the Omni Path network.

The mini applications we used are as follows. GeoFEM is a conjugate gradient code developed at The University of Tokyo [23]. CCS-QCD is lattice Quantum Chromodynamics (QCD) simulation from The University of Hiroshima that is part of the Fiber mini application suite, available at https://github.com/fiber-miniapp/ccs-qcd. Finally, miniFE is an proxy application for unstructured implicit finite element codes from the CORAL benchmark suite, available

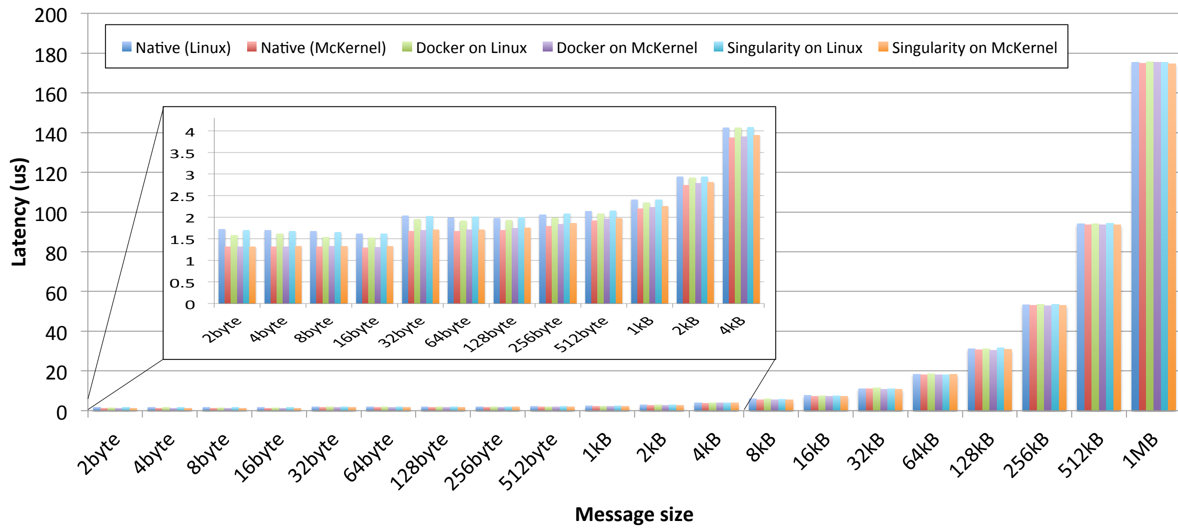
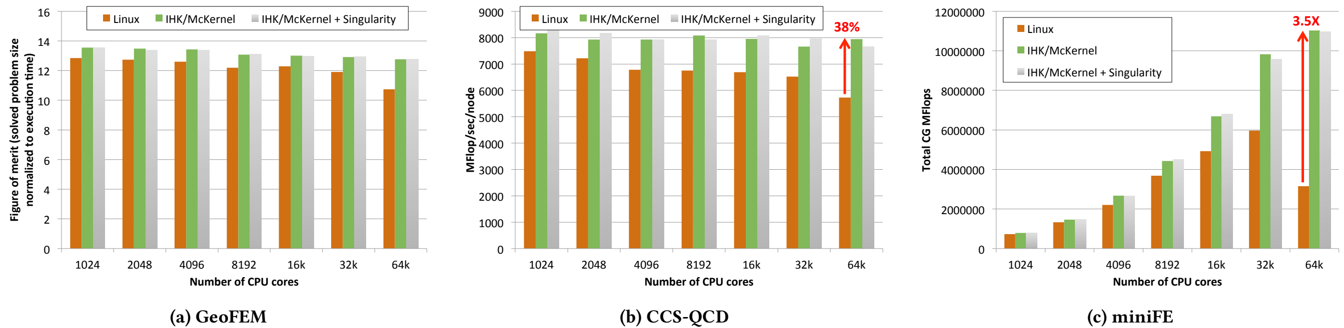


Figure 3: Intel MPI Benchmark Ping-Pong Results.



(a) GeoFEM (b) CCS-QCD (c) miniFE

Figure 4: Mini applications scalability results.

at https://asc.llnl.gov/CORAL-benchmarks/Throughput/MiniFE_ref_2.0-rc3.tar.gz.

We compare the execution of these application on Linux, on McKernel and running them in a container on top of McKernel. McKernel outperforms Linux at scale on all of these benchmarks. The detailed analysis of where these speedups derive from is outside the scope of this paper and is published elsewhere [11], however, we note that one of our main findings was that most of the performance improvement was due to the LWK's ability to be easily modified to suite unconventional requirements, which corresponds to the specialization argument of this paper. Again, what we seek to confirm here is whether or not running these benchmarks in a container on top of McKernel results in identical performance improvements as the native McKernel execution. Figure 4 depicts the results.

Note that both the native Linux and McKernel results are average of multiple (up to five) runs, but due to the limited availability for exclusive usage of the OFP machine, we obtained only one execution for the containerized scenario. Unfortunately, we had no opportunity to test our hypothesis on mOS, but we did observe similar improvements for the native mOS case [11]. As one can see the results for running the benchmarks in a Singularity container on top of McKernel look almost identical to the native McKernel execution. Specifically, we observed up to 18%, 38% and 3.5x speedups on GeoFEM, CCS-QCD and miniFE, respectively. Note that the measurements were not run directly after each other and we could not ensure to have the exact same 1,024 nodes in all cases. We attribute the slight performance differences to that.

In summary, our framework enabled us to deploy an approximately 500 MB Ubuntu image, which contained all necessary libraries, and by running it on top of a lightweight multi-kernel we could transparently benefit from kernel level improvements.

5 DISCUSSION

This section provides further discussion on the limitations and challenges we identified while running containers in HPC environments. Although containers are often depicted as the enablers of full control over user-space execution environments, they do

impose certain limitations. As user-space libraries often have dependencies with respect to the OS kernel's device drivers, the libraries deployed in the container need to comply with the host kernel. This is particularly true in the HPC context, where high performance networks and/or accelerators require very specific libraries to match the hardware's performance profile[1].

Similarly, application binaries that leverage specific hardware features (e.g., rely on specific CPU instructions) may not be deployed on platforms that do not support those features.

The second issue we found is with regard to MPI, particularly how MPI jobs are spawned. As MPI processes are created by executables that reside on the host machine, i.e., the Process Management Interface (PMI) proxy process, the MPI library in the container needs to adhere to the host's PMI protocol. Unfortunately, while there have been efforts to establish an ABI level PMI protocol standard that would span across all MPI implementations, this is currently not possible.

6 RELATED WORK

We survey a number of related studies covering operating systems for multi-cores, multi-kernels in HPC as well as containers.

The K42 [14] research project took scalability as the primary concern in OS design. Similarly how mOS and IHK/McKernel selectively implement a set of performance sensitive system calls on the LWK side, K42 enabled application to bypass the Linux APIs and call directly into native K42 interfaces. However, it involved a significant entanglement with Linux which made it cumbersome to keep up with the latest Linux modifications. While mOS and McKernel also rely on Linux, one of their primary design criteria was to minimize the effort required to keep up-to-date with the rapidly moving Linux kernel.

Multi-kernels in general purpose computing have also been studied. Tessellation [20] and Multikernel [2] are driven by the observation that modern computers have similar architectural attributes to networked system and so the OS should also be modeled as a distributed system. The Tessellation project [20] proposed Space-Time Partitions, an approach that partitions CPU cores into groups called cells. Each cell hosts specific system services or a particular application. Because applications and system services can be assigned to distinct cells, Tessellation's structure is similar to both mOS and IHK/McKernel, where HPC applications are assigned to LWK cores while system daemons reside on CPU cores managed by Linux.

Multikernel [2] runs a small kernel on each CPU core and OS services are built as a set of cooperating processes. Each process is running on one of the multi-kernels and communicates using message passing. Similarly to Multikernel, the IHK/McKernel model relies on a message passing facility that allows communication between the two types of kernels, and consequently between the application and its Linux proxy process.

The idea of multi-kernels in the HPC context has also been studied for a number of years. FusedOS [25] was the first system to combine Linux with an LWK. FusedOS' primary objective was to address core heterogeneity between system and application cores and at the same time to provide a standard operating environment. Contrary to mOS and McKernel, FusedOS runs the LWK at user level. The kernel code on application CPU cores is simply a stub that offloads all 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 this CL process that runs on Linux. Consequently, FusedOS 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 to avoid interference with the HPC application running on the LWK CPUs. This property has been one of the main driver for both mOS and McKernel.

Hobbes [4] was one of the projects in DOE's Operating System and Runtime (OS/R) framework for extreme-scale systems. The central theme of the Hobbes design is to support application composition, which is emerging as a key approach to address scalability and power concerns anticipated in future extreme-scale architectures. Hobbes utilizes virtualization technologies to provide the flexibility to support requirements of application components for different node-level operating systems and runtimes. The Kitten [16] LWK forms the base layer of Hobbes, and Palacios [18], running on top of Kitten, serves as a virtual machine monitor.

Argo [17] is another DOE OS/R project targeted at applications with complex workflows. While Argo originally also targeted a multi-kernel based software architecture, it recently turned toward primarily relying on container technologies. Currently, it investigates how to enhance the Linux kernel's container framework so that it can meet HPC requirements [28].

The applicability of Linux containers in high-performance computing has received considerable attention in recent years. An early evaluation by Xavier et al found that containers impose near zero overhead on HPC workloads, although their experiments were performed on very small scale using hardware that is not typically used in the supercomputing context [27]. Jacobsen et al demonstrated containers' advantages from a storage perspective. In particular, they showed how containers can alleviate pressure on parallel file systems by keeping metadata operations local when a large number of shared libraries are utilized by the application [13]. Hale et al. compared various container technologies deployed on a Cray XC30 system in [12]. One of their key findings is that optimized images can occasionally outperform native user installations as the container environment can be suitably specialized by the developers of a library. While all this work focuses on containers in an HPC context, it only considers running on Linux, as opposed to our efforts of specializing kernel space as well.

As we briefly mentioned in Section 1, an approach toward addressing cloud computing as a platform for elastically scaling services are Unikernels, such as MirageOS [21] and IncludeOS [3]. Unikernels are specialized, single-address-space machine images constructed by using library operating systems. Note that the concept of library operating systems is not new, Libra [1] proposed a similar system for JVM. Some of the benefits Unikernels provide are small footprints and more opportunity for kernel level specialization. Unikernels, however, usually run in virtualized environments and require the application to be recompiled. On the other hand, lightweight multi-kernels can provide a Linux compatible environment via service offloading.

[1]Note that our multi-kernels can take advantage of Linux device drivers [9].

A recent effort, HermitCore [19], investigates Unikernels' applicability to HPC. HermitCore has a very similar goal to multi-kernels, however, we see two main differences between multi-kernels and HermitCore. First, HermitCore does not provide system call offloading, which implies that any OS service it supports must be implemented in the Unikernel itself. With HPC applications becoming increasingly complex, we are not convinced that this is the right approach. Additionally, while HermitCore's co-kernel management is very similar to IHK, e.g., they both leverage the Linux kernel's offlining features, HermitCore requires modifications to the Linux kernel, which makes its deployment more cumbersome.

7 CONCLUSION AND FUTURE WORK

The increasing hardware complexity and the growing software diversity in HPC environments encourage software specialization to address these challenges. While containers enable greater control over user-space components, lightweight multi-kernel operating systems allow software specialization of kernel space.

In this paper, we proposed a unified framework that enables the deployment of containers on top of lightweight multi-kernels, thereby extending software specialization across the entire software stack. Through a large-scale deployment of our framework on 1,024 Xeon Phi nodes we have demonstrated that identical speedups to the native multi-kernel deployment can be achieved from containerized environments. We believe that specialization of HPC software will play an increasingly important role as we move towards exascale and beyond.

In the future, we will further extend our framework to support a wider range of container engines. We also intend to investigate how the incompatibility between containerized libraries and the host kernel may be addressed.

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.

REFERENCES

[1] Glenn Ammons, Jonathan Appavoo, Maria Butrico, Dilma Da Silva, David Grove, Kiyokuni Kawachiya, Orran Krieger, Bryan Rosenburg, Eric Van Hensbergen, and Robert W. Wisniewski. 2007. Libra: A Library Operating System for a JVM in a Virtualized Execution Environment. In *Proceedings of the 3rd International Conference on Virtual Execution Environments (VEE '07)*. ACM, New York, NY, USA, 44–54.

[2] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. 2009. The multikernel: a new OS architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles (SOSP '09)*. 29–44.

[3] A. Bratterud, A. A. Walla, H. Haugerud, P. E. Engelstad, and K. Begnum. 2015. IncludeOS: A Minimal, Resource Efficient Unikernel for Cloud Services. In *2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom)*. 250–257.

[4] Ron Brightwell, Ron Oldfield, Arthur B. Maccabe, and David E. Bernholdt. 2013. Hobbes: Composition and Virtualization As the Foundations of an Extreme-scale OS/R. In *Proceedings of the 3rd International Workshop on Runtime and Operating Systems for Supercomputers (ROSS '13)*. Article 2, 8 pages.

[5] BDEC Committee. 2017. *The BDEC "Pathways to Convergence" Report.* Technical Report.

[6] CoreOS. 2017. rkt: A security-minded, standards-based container engine. https://coreos.com/rkt. (April 2017).

[7] Joint Center for Advanced HPC (JCAHPC). 2017. Basic Specification of Oakforest-PACS. http://jcahpc.jp/files/OFP-basic.pdf. (March 2017).

[8] Balazs Gerofi, Akio Shimada, Atsushi Hori, and Yutaka Ishikawa. 2013. 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.*

[9] B. Gerofi, M. Takagi, A. Hori, G. Nakamura, T. Shirasawa, and Y. Ishikawa. 2016. On the Scalability, Performance Isolation and Device Driver Transparency of the IHK/McKernel Hybrid Lightweight Kernel. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 1041–1050. DOI:https://doi.org/10.1109/IPDPS.2016.80

[10] Balazs Gerofi, Masamichi Takagi, Yutaka Ishikawa, Rolf Riesen, Evan Powers, and Robert W. Wisniewski. 2015. Exploring the Design Space of Combining Linux with Lightweight Kernels for Extreme Scale Computing. In *Proceedings of ROSS'15*. ACM, Article 5, 8 pages.

[11] Balazs Gerofi, Masamichi Takagi, Rolf Riesen, Robert W. Wisniewski, Kengo Nakajima, Taisuke Boku, and Yutaka Ishikawa. 2017. Performance and Scalability of Lightweight Multi-Kernel based Operating Systems. In *Submission (SC '17)*.

[12] Jack S. Hale, Lizao Li, Chris N. Richardson, and Garth N. Wells. 2016. Containers for portable, productive and performant scientific computing. *CoRR* abs/1608.07573 (2016).

[13] Douglas M. Jacobsen and Richard Shane Canon. 2015. Contain This, Unleashing Docker for HPC.

[14] Orran Krieger, Marc Auslander, Bryan Rosenburg, Robert W. Wisniewski, Jimi Xenidis, Dilma Da Silva, Michal Ostrowski, Jonathan Appavoo, Maria Butrico, Mark Mergen, Amos Waterland, and Volkmar Uhlig. 2006. K42: Building a Complete Operating System. *SIGOPS Oper. Syst. Rev.* 40, 4 (April 2006), 133–145.

[15] Gregory M. Kurtzer. 2016. Singularity 2.1.2 - Linux application and environment containers for science. http://dx.doi.org/10.5281/zenodo.60736. (Aug. 2016).

[16] Sandia National Laboratories. 2017. Kitten: A Lightweight Operating System for Ultrascale Supercomputers. https://software.sandia.gov/trac/kitten. (Jan. 2017).

[17] Argonne National Laboratory. 2017. Argo: An Exascale Operating System. http://www.mcs.anl.gov/project/argo-exascale-operating-system. (March 2017).

[18] J. Lange, K. Pedretti, T. Hudson, P. Dinda, Zheng Cui, Lei Xia, P. Bridges, A. Gocke, S. Jaconette, M. Levenhagen, and R. Brightwell. 2010. Palacios and Kitten: New high performance operating systems for scalable virtualized and native supercomputing. In *IEEE International Symposium on Parallel Distributed Processing (IPDPS)*. DOI:https://doi.org/10.1109/IPDPS.2010.5470482

[19] Stefan Lankes, Simon Pickartz, and Jens Breitbart. 2016. HermitCore: A Unikernel for Extreme Scale Computing. In *Proceedings of the 6th International Workshop on Runtime and Operating Systems for Supercomputers (ROSS '16)*. ACM, New York, NY, USA, Article 4, 8 pages. DOI:https://doi.org/10.1145/2931088.2931093

[20] Rose Liu, Kevin Klues, Sarah Bird, Steven Hofmeyr, Krste Asanović, and John Kubiatowicz. 2009. Tessellation: Space-time Partitioning in a Manycore Client OS. In *Proceedings of the First USENIX Conference on Hot Topics in Parallelism (HotPar'09)*. 1. http://dl.acm.org/citation.cfm?id=1855591.1855601

[21] Anil Madhavapeddy and David J. Scott. 2013. Unikernels: Rise of the Virtual Library Operating System. *Queue* 11, 11, Article 30 (Dec. 2013), 15 pages.

[22] Dirk Merkel. 2014. Docker: Lightweight Linux Containers for Consistent Development and Deployment. *Linux J.* 2014, 239, Article 2 (March 2014). http://dl.acm.org/citation.cfm?id=2600239.2600241

[23] K. Nakajima. 2003. Parallel Iterative Solvers of GeoFEM with Selective Blocking Preconditioning for Nonlinear Contact Problems on the Earth Simulator. In *Supercomputing, 2003 ACM/IEEE Conference*. 13–13. DOI:https://doi.org/10.1145/1048935.1050164

[24] NERSC. 2017. Shifter - Linux Containers for HPC. https://github.com/NERSC/shifter. (March 2017).

[25] Yoonho Park, E. Van Hensbergen, M. Hillenbrand, T. Inglett, B. Rosenburg, Kyung Dong Ryu, and R.W. Wisniewski. 2012. 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.* 211–218.

[26] Taku Shimosawa, Balazs Gerofi, Masamichi Takagi, Gou Nakamura, Tomoki Shirasawa, Yuji Saeki, Masaaki Shimizu, Atsushi Hori, and Yutaka Ishikawa. Interface for Heterogeneous Kernels: A Framework to Enable Hybrid OS Designs targeting High Performance Computing on Manycore Architectures.

[27] M. G. Xavier, M. V. Neves, F. D. Rossi, T. C. Ferreto, T. Lange, and C. A. F. De Rose. 2013. Performance Evaluation of Container-Based Virtualization for High Performance Computing Environments. In *2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*. 233–240.

[28] J. A. Zounmevo, S. Perarnau, K. Iskra, K. Yoshii, R. Gioiosa, B. C. V. Essen, M. B. Gokhale, and E. A. Leon. 2015. A Container-Based Approach to OS Specialization for Exascale Computing. In *2015 IEEE International Conference on Cloud Engineering*. 359–364.