Contain This, Unleashing Docker for HPC

Douglas M. Jacobsen
Computational Systems Group
NERSC, Lawrence Berkeley National Laboratory
Berkeley, USA
Email: dmjacobsen@lbl.gov

Richard Shane Canon
Technology Integration Group
NERSC, Lawrence Berkeley National Laboratory
Berkeley, USA
Email: scanon@lbl.gov

***Abstract*—Containers are a lightweight virtualization method for running multiple isolated Linux systems under a common host operating system. Container-based computing is revolutionizing the way applications are developed and deployed. A new ecosystem has emerged around the Docker platform to enable container based computing. However, this revolution has yet to reach the HPC community. In this paper, we provide background on Linux Containers and Docker, and how they can be of value to the scientific and HPC community. We will explain some of the use cases that motivate the need for user defined images and the uses of Docker. We will describe early work in deploying and integrating Docker into an HPC environment, and some of the pitfalls and challenges we encountered. We will discuss some of the security implications of using Docker and how we have addressed those for a shared user system typical of HPC centers. We will also provide performance measurements to illustrate the low overhead of containers. While our early work has been on cluster-based/CS-series systems, we will describe some preliminary assessment of supporting Docker on Cray XC series supercomputers, and a potential partnership with Cray to explore the feasibility and approaches to using Docker on large systems.**

***Keywords*-Docker; User Defined Images; containers; HPC systems**

I. Introduction

The use of Linux containers to accelerate development and ease distribution and deployment of applications has recently exploded. This revolution is being led by technologies such as Docker [1] and its emerging ecosystem, but also includes developments by other players, such as Project Atomic, CoreOS, and others. While this transition is already having an impact on the enterprise and web space, its implications for scientific and technical computing including HPC are still unlcear. We believe that container-based computing has the potential to dramatically impact scientific computing and we have done early prototyping to investigate how container concepts can be integrated into HPC environments and benchmarked some of the alternative implementations. In this paper, we will provide some background on container computing including Docker and why we believe it has value to the scientific and HPC community. We will then describe our prototype implementation including our rationale for certain design choices and some of the constraints imposed by the existing architectures. Next, we will present some benchmarks that compare the performance of our implementation with some of the other implementation options. We will close with some discussion of this prototype and conclusions.

II. Background

Exploiting Linux Containers to support flexible, scalable computing has gained rapid adoption in the past two years. While Linux has possessed the basic features to support containers (cgroups and namespaces) and basic tools (i.e. LXC [2], OpenVZ) to manage them have existed for nearly a decade, the rapid adoption has been driven by the appearance of a few key technologies that have simplified using containers and led to the rapid emergence of an ecosystem around containers. This adoption is partly being driven by the emergence of technologies like Docker which provide a framework for managing container instances coupled with a powerful image management system backed by a growing collection of images. Docker is not alone as other competitors are emerging to offer similar capabilities, and new layers of capabilities such as orchestration are starting to build on the container infrastructure.

Linux containers address many of the requirements handled by full virtual machines (VMs), such as customization and isolation. However, containers rely on capabilities in the kernel level to implement these features. This has a few consequences. One is processes running in containers on a system run in a common linux kernel. This means that if a process or user manages to disrupt the kernel (i.e. due to a bug) it can impact other containers running on that systems. In contrast, full virtual machines typically rely on hardware features to provide full isolation. Consequently, full VMs typically offer greater protection between applications running on a shared system. This is compounded by the fact that the Linux kernel offers varying levels of isolation depending on the subsystem. For example, containers can have fairly strong levels of isolation for processors and memory, but offer weaker isolation for I/O.

The main advantage of containers over virtual machines is they are typically much more light-weight. For example, containers can require significantly less memory since a container will often only run the specific application or process that is needed. For example, a web application may only need to run apache. Other supporting services (i.e. name services, management, etc) are provided by the host system and typically shared across containers. This frees up the

memory that would be required in a VM environment, where each VM needs to have those services running. Additionally, a container based system only needs to run a single copy of the kernel where as for VMs, each VM has its own instance of the kernel. These can lead to significant overheads, especially for relatively light-weight processes that may only require tens of megabytes. These extras processes and kernel space also impact start up time. Since starting a VM requires initializing a virtual machine and booting an entire kernel, booting can take several minutes. In contrast, starting a container is essentially just starting a process so it typically requires fractions of second. This fast startup can be very useful for highly dynamic workloads that may need to quickly shift resources between different components based on demand.

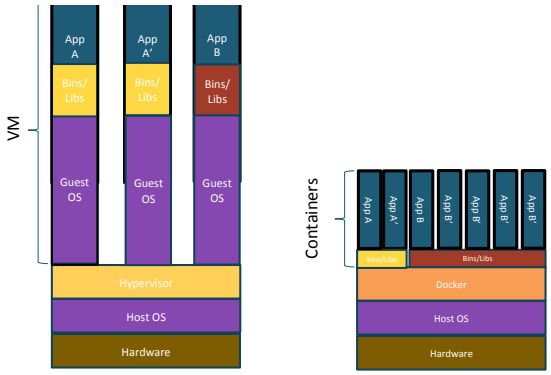

Fig. 1. Comparison of VMs versus Docker containers. Each VM requires the full overhead of a OS. Docker containers share a kernel and require much less memory and disk space.

Accessing shared resources is also different between the two models. Most applications need to have access to data stored in a global high-performance or use it to share state between steps in a workflow. In a VM environment, each VM must run an instance of the file system client. If many VMs are running on each node, this can lead to a dramatic increase in the number of file system clients the file system must serve. This also adds complexity to manage and configure the VMs to access the file systems, as well as raises security concerns if the user has any elevated privileges in the VM environment. In contrast, a container model can typically map through a file system from the host system into individual containers. This means there is only one instance of the file system client per host system. This can typically be done more securely depending on the implementation. We will discuss this further under in Section IV.

A study by IBM found that the lower overhead of containers can significantly impact subsystem and application performance [3]. According to the study, "In general, Docker equals or exceeds KVM performance in every case we tested." This study found that I/O transaction and latency sensitive benchmarks were particularly better performing on Docker. This makes sense, since full VMs typically require I/O operations to traverse multiple layers before being serviced which adds overhead. For example, a MySQL benchmark achieved over 2x more transactions per second and latencies that were as much as 10x lower for certain transaction rates.

III. Motivation

Centers like NERSC are increasingly struggling to keep pace with the rapid expansion in applications, libraries, and tools demanded by the user community, especially new data-intensive communities. This growth is driven by a number of factors. In some cases, large communities are developing software to serve their specific scientific community. In other cases, users may be interested in specific tools that are difficult to install, have a long list of dependencies, and are difficult to port. In some cases, this software may be specifically targeted at an OS environment that is common for their domain but may conflict with the requirements from another community. For example, the biology and genomics community may adopt Ubuntu as their base OS with a specific version of Perl and Python. Meanwhile, the High-Energy Physics community may use Scientific Linux as their platform of choice with very specific requirements for certain libraries, compilers, and scripting tools. While porting these tools to other OS versions may be possible, the overhead to do the port and validate it may be too high for a community. Modules [4] can be used to support different versions of libraries, scripting tools, etc, but building a robust, well tested stack with the exact combination of dependencies can be tedious and challenging.

In many cases, what users desire is the ability to easily execute their scientific applications and workflows in the same environment used for development or adopted by their community. In some cases, this can include being able to seamlessly go from their desktop to the HPC environment. Some communities have turned to the cloud because it promises to provide this flexibility. However, using a cloud environment can be challenging as users have to typically address all of the components that would normally be provided by a managed cluster or HPC center. For example, the users need to solve how they handle workload management, file systems, and basic provisioning. The overhead to address these requirements solely to gain flexibility over the software stack is typical too large to be feasible. In our experience, what users desire is the ability to easily define their own images and, then, easily insatiate these environments on large scale HPC systems at centers like NERSC. NERSC has termed this capability as User Defined Images (UDI).

Containers promise to offer the flexibility of cloud-type systems coupled with the performance of bare-metal systems. Furthermore, containers have the potential to be more easily integrated into traditional HPC environments which means that users can obtain the benefits of flexibility with out the added burden of managing other layers of the system (i.e. batch systems, file systems, etc).

IV. Implementation Alternatives

There are several approaches to supporting User Defined Images (UDI). We will briefly describe some of the options and provide the rationale for the approach we chose.

CHOS NERSC has experience providing similar functionality via CHOS [5]. CHOS was developed at NERSC to allow a shared cluster to concurrently support multiple environments, even within a single node. This was driven by the need to support several large HEP projects which had their own approved software stacks. CHOS uses a custom kernel module that provides a process specific symbolic link. This symbolic link can point to different directories depending on which process is accessing it. CHOS has worked well for supporting a small number of managed environments, but isn't designed to allow users to define their own custom environment. Using CHOS on a Cray system would require porting the kernel module to the Cray OS which may not be trivial. But the bigger issue is CHOS's lack of support for user-defined images. Finally, CHOS lacks broad community support. NERSC felt it was important to be able to tap into the Docker image repository since we anticipate this will become a rich ecosystem for specialized images and applications. For these reason, we focused on options that could integrate with Docker.

MyDock Recently, NERSC deveoped a custom wrapper for Docker (MyDock) which allows a user to run specific docker commands and restricts the execution of docker containers to run: as themselves (i.e. non-root), using the native networking interface, and with specific paths mapped into the container. This is specifically targeted towards allowing users to bring in their own images, but doesn't allow them to use all of the features of Docker. In our experience, these restrictions were well suited for the ways scientific users need to execute their applications. For example, MyDock was used by the Dark Energy Survey to demonstrate how they could use Docker to create a portable execution environment. The choice of Docker was suggested by NERSC staff as an alternative to virtual machine/cloud-style options and the use of Docker has since gained acceptance within the project. MyDock has proven useful, but it presents several challenges in porting this approach to the Cray systems. For example, Docker relies heavily on local disk to function. While it would be possible to work around this requirement, it could have performance implications. Furthermore, MyDock requires the Docker daemon to be running on each compute node. This would add complexity and introduces the risk that processes may not get properly cleaned up.

CRAY_ROOTFS Discussions with Cray Engineers led us to investigate options to leverage support in ALPS to "chroot" into system-defined directories during the execution of an $aprun$ using the $CRAY\_ROOTFS$ environment variable. This helps solve parts of the problem but still requires integration. In addition, there remains the question of where the image should reside and in what format it should be stored. The $CRAY\_ROOTFS$ directive is typically used for Dynamic Shared Library (DSL) support and the image is accessed via Cray's Data Virtualization Service (DVS). While we could have used a similar approach here, we were interested in exploring other alternatives including storing the images as unpacked trees in the Lustre scratch file system and storing the images as loopback mountable image files. After some benchmarking tests (described later), we determined that storing the images in loopback mounted files provided the best overall performance.

Other Options There are other options we considered but ruled out early for various reasons. For example, a local disk could have been emulated via an iSCSI mount. This was ruled out due to the added complexity. Another option was to use a logical volume accessed through a loop back file. This was also ruled out for complexity reasons.

TABLE I
SUMMARY OF POTENTIAL IMPLEMENTATION APPROACHES.

Approach	Pros	Cons
CHOS - ChrootOS	Integrated with the batch and login systems	Not designed for user generated images
MyDock - Docker Wrapper	Uses Docker directly	Requires running a Docker daemon on each compute node
CRAY_ROOTFS		
- DVS	Mimics current DSL approach	DVS servers can be a bottleneck
- Lustre (unpacked)	Scales with Lustre File System	Metadata server could become a bottleneck
- Lustre (packed)	Scales with Lustre File System, Metadata is more localized	Additional complexity to pack images

V. PROTOTYPE IMPLEMENTATION

NERSC has created an initial implementation of the UDI system capable of running efficiently on a Cray XC or XE class system called "Shifter". We had several goals in mind when designing Shifter to be an HPC-enabled UDI solution including:

- Scalability and Performance of running applications
- Scalability and Performance of setting up the container/image
- Accessibility of Shared Resources, Parallel Filesystems, Interconnect
- Compatibility with batch system resource management
- Compatibility with Docker as well as other container/image formats
- Ability to fully leverage existing Docker images and Docker push/pull functionality
- Robust, Secure Implementation

Considering these different factors, combined with some of the potential deployment issues for running Docker on Cray compute nodes, for Shifter we have opted not to directly use Docker in the CLE environment, but rather to automatically extract images from native formats and convert to a common format on an external gateway node, and then provide software within Shifter to setup the environment on the compute node(s) using that common image format. This methodology allows the Shifter solution to interoperate between different upstream image/container providers, and it provides the opportunity to customize and tune the image for use in the HPC environment.

A. Implementation Description

The major functions that Shifter needs to perform are:

- Allow user to select or download an image
- Convert and transfer images
- Integrate UDI request mechanism into Workload Manager (e.g., qsub, sbatch)
- Mechanism to setup and customize image on compute nodes
- Mechanism to completely deconstruct any image and return compute nodes to normal state
- Provide mechanism for generic internode communication, even if libraries for native access to the HSN are not present in image

To achieve all these functions, Shifter is decomposed into four major components: an Image Gateway, command-line utilities, "udiRoot", and Workload Manager (WLM) Integration components. The command-line utilities serve to allow a user to interactively manage and select images as well as ease environmental translations between the host-system environment and the target UDI. The Image Gateway is responsible for managing the images, keeping a data-store of presently loaded images, and transferring images to the computational platforms. In the case of Docker images, the Image Gateway actually communicates with a local docker daemon to pull down an image from DockerHub or a private registry, and then extract the image and any needed metadata. The "udiRoot" component contains all the scripts and configurations that run on a compute node to actually make the UDI available, and to deconstruct it at the end of a job. Workload Manager integration is critical because the WLM is directly responsible for determining which nodes are to be used, and thus job prologue and epilogues are used to setup the UDI in the job. Furthermore, the WLM must provide some mechanism for accessing the UDI in the batch job.

B. User Experience

Figure 2 shows a basic flow diagram for how the Shifter system operates and integrates into the workflow of a user. From the user's perspective, it is intended to be relatively straightforward to use. First, the user needs to either select or create a Docker container image with their target application(s) and dependencies and push that image into DockerHub. Next, the user logs into the Shifter-enabled computational resource and issues a command like "docker pull X" where "X" represents the tagged container revision. That will cause the image to be retrieved from DockerHub, or a designated private registry, and prepare it for use by Shifter. Once the image is ready, the user simply submits a batch job requesting the target image. Any aprun/srun will automatically be run within the image, and so some care needs to be taken to set the environment variables up appropriately within the batch script.

When submitting a Shifter job, the user has the option to designate volume mappings, very similar to the volume mappings supported by regular Docker. This enables a user to map, for example, /scratch/user/path to /output within the context of the batch job, thus enabling /output to be used for writing in the job, supposing of course that "/output" was defined as a volume target in the original Docker container image. Another option the user has is to designate that the container should be "run", this means that if the original Docker container has a defined entry-point and environmental variables, then that entry point can be automatically used without having to write an explicit batch script. Some Workload Managers may require a simple wrapper script to be written, however.

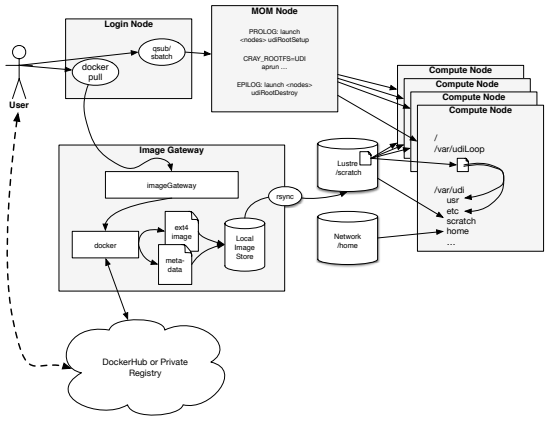

Fig. 2. Diagram of the various components of the Shifter prototype for User Defined Images.

At the conclusion of a Shifter job, the user receives output and exit status just as they would for any other job, and, on the surface does not represent a major departure from any normal HPC batch job.

C. Restrictions on Images Used in Shifter

Owing to the way Shifter is implemented, effectively relying on chroot and the batch system, there are some minimal requirements that must be true about any image in order to be used within Shifter many of these are similar to normal Docker limitations. Among others, the image must be based on a relatively "normal" GNU/Linux distribution, for example having standard canonical paths (/bin, /usr, /etc, ...) [6], based on glibc or compatible variants, and contain at minimum /bin/sh. The application packaged within the image should not make use of features of Linux kernels newer than the one loaded in the base system. In practice this rarely a problem in our experience, as most applications only make use of fairly traditional system calls. Because there is only one running Linux kernel operating the host system and all UDI jobs, it is not possible to load kernel modules for the particular or exclusive use of a UDI. Doing so could represent a major security risk if allowed. For security reasons, Shifter attempts to detect potentially dangerous path names in regions where Shifter must interact with the image. If any dangerous path names are detected, the image is rejected. Therefore, images for use in Shifter should not make use of path names (especially in /, /etc, or /var) that do not confirm to regular

Linux "norms".

D. Image Gateway

Standard Docker enables a user to iteratively modify an image and then store it using the "docker push" functionality. One of the primary goals of the Shifter project is to enable access to the rich ecosystem of available Docker images in DockerHub, as well as to leverage any containers stored in private registries by allowing users to "pull" previously created images. Owing to the complexities of deploying Docker in an HPC environment, we elected to only run the docker daemon on a physically discrete system, dubbed the Image Gateway. At present, there is a simple command line interface provided to users to login, pull, and list images on the gateway using standard Docker commands. All of these commands communicate with the Image Gateway using a simple TCP-based conversational protocol.

When a connection is initiated to the Image Gateway, xinetd launches a python script (imageGateway), which evaluates the request and then interacts with dockerd to satisfy that request. When a new image or a new version of an image is available, imageGateway pulls the image to a local docker graph and then extracts the image from it using the "docker save" functionality. The needed size of the ext4 image is evaluated based on the uncompressed size of the resulting tarball, and then the layers are extracted in the proper order into the ext4 image file. In addition, special care is taken to extract the entryepoint, working directory, and environment variables of the final layer to enable a user to later "run" the container.

Once the ext4 image file and metadata are generated, imageGateway finally transfers the resulting images to the target computational platform. The user does not have any direct control over this process and the image is rsync'd as a special-purpose unprivileged user onto the Lustre /scratch filesystem or other more appropriate image store. The images are named based on a unique identifier that can be used to discriminate between unique versions of an image of the same name (e.g., if the "latest" tag of a Docker container repo changes). This capability is leveraged by the workload manager integration to disambiguate versions of images to ensure the same version used at job run time as was submitted.

E. udiRoot – compute node management

UDI setup and teardown is managed by the udiRoot component of Shifter. The udiRoot scripts are typically called by the prologue and epilogue capabilities of a workload manager, but under some circumstances may also be engaged interactively for non-scheduled interactive access to a UDI image. udiRoot supports a number of important capabilities which include managing needed kernel modules for Shifter, setting up the image for consumption in a job, customizing the image to enable parallel filesystem access and install needed configuration files, configure and operate private sshd to enable internode communication, enable user-specified filesystem mappings and, of course, tear down the UDI at job completion.

At the beginning of job the "setupRoot.sh" script is called with a number of options specifying the needed UDI as well as features required. First, the script attempts to validate the requested image, typically by verifying that it exists in the proper location of the filesystem, however different image types have different validation procedures. Second, assuming the image is an image file (e.g., ext4), then the loop block device drivers are loaded and the image is mounted readonly and setuid-incapable, typically on /var/udiLoop. To enable customization of the image, however, a new rootfs (tmpfs) is mounted read-write and setuid-incapable, typically on /var/udi. The /var/udi mount point is the target location for setting up UDI. All the paths in the base of the loopback-mounted image are bind-mounted into /var/udi. If any paths appear to be named dangerously (i.e., could confuse the software) then those paths are skipped and are bind-mounted in /var/udi. Special care is taken to setup etc, var, and opt within /var/udi since a variety of custom items need to be placed into these locations to support site configurations. Similarly, any site-configured parallel filesystems can be bind-mounted into the image at this point. All the bind mounts made by setupRoot.sh are "nosuid". Most also use "nodev" except for /dev, of course. These precautions help to ensure that users cannot subvert the security of the system and escalate their privileges.

An important use-case for the Shifter functionality is to enable multi-node calculations to function, even if the UDI does not contain the necessary software to access the HSN. In that case, the application can still communicate using TCP/IP, but to support common application needs, we have constructed a purpose-built ssh daemon that can be more-or-less safely run within user-defined containers. This ssh daemon is statically linked against LibreSSL and musl (a small, static glibc-compatible libc used for micro or embedded Linux systems). It is important to use a statically linked sshd from within the Shifter container since it may be dangerous to run processes as root in an unknown and untrusted environment. The ssh daemon is started chroot'd in /var/udi and functions in many ways similar to Cray's CCM functionality [7]. Finally, a nodes list is written to /var/nodelist within the UDI container (e.g., analogous to $PBS_NODEFILE) to allow a user to easily discover which nodes are part of their job when running in the UDI.

At the termination of the job the "unsetupRoot.sh" script is called by the Workload Manager job epilogue. This script is relatively simpler than the setupRoot.sh script and is designed to be robust enough to correctly cleanup the node regardless of the complexity of the requested image. unsetupRoot.sh first terminates the running sshd, if there is one. Next it iteratively unmounts all the paths under /var/udi (in a reverse-sorted manner). Finally it unmounts the /var/udiLoop mount and removes any kernel modules that were loaded to support Shifter. A Cray nodehealth check plugin is also included with udiRoot to ensure that nodes are cleaned up properly. If the nodehealth check fails, the node should be marked admindown to prevent more jobs from scheduling on it.

F. Workload Manager Integration

The workload manager is a critical component to the function and user interface of Shifter. The workload manager is used to request a Shifter image and specify options like volume mapping. WLM prologue/epilogue functionality is used to setup/teardown the Shifter image by calling the appropriate udiRoot scripts on all the relevant compute nodes. The WLM is actually what puts the user "into" the UDI at job start time this is done by performing a chroot() into /var/udi just prior to exec'ing the user process. Different WLM systems have different mechanisms for achieving this. The final area that is influenced by the WLM is resource management. By default, docker creates its own cgroups to limit resources and provide management capabilities. This, however, is at odds with the traditional model for HPC management of systems. One of the advantages of the Shifter system is that the built-in process management/tracking functionalities of the WLM should work natively, even if the WLM uses cgroups. This is because Shifter does not try to impose any, but does expose all the cgroups mounted in the base OS to enable the WLM to make use of them.

Three WLM configurations have been used with Shifter: Torque/Moab with ALPS, SLURM with ALPS, and Native SLURM [8]. The torque-based setup uses environment variables passed with the job to determine which UDI should be used, as well as any options. The SLURM implementations rely on a purpose-build plugin which adds the "–image" and "–imagevolume" options to sbatch and salloc. During job submission for either, the equivalent of a "docker pull" is performed to ensure that the latest version of an image is downloaded to the system. Once the image is present, the qsub/sbatch/salloc will unblock and the job will be submitted. At job submission time a prolog runs which calls the udiRoot scripts. On an ALPS-based system, this is performed on the MOM node and the nodehealth check software is engaged to remotely call setupRoot.sh on all the compute nodes (xtxqtcmd). On a native SLURM system, a simpler prolog is called on all the nodes separately and in parallel.

To execute a job on the Shifter-setup compute nodes, the key functionality is to get the batch script or other user process chroot's into /var/udi. This is done in ALPS by specifying CRAY_ROOTFS=UDI in the environment of any aprun. In native SLURM, the plugin automatically chroots the slurmstepd immediately prior to execing the job script.

From the perspective of the WLM, tearing down the UDI is done in a similar way as the setup. In the case of ALPS, the job epilogue will make use of the nodehealth check system to call the udiRoot scripts on all the relevant compute nodes, whereas in native SLURM the epilogue runs directly on the compute nodes.

VI. Benchmarking and Comparisons

In Section IV we described some of the potential options for how to implement Shifter. Here we compare some of those approaches using the Pynamic benchmark [9] from Lawrence Livermore National Laboratory. This benchmark can be used to generate a set of test python modules with each module having a random number of methods. The number of modules and average number of methods can be specified during the generation phase. Once the modules have been generated, a special driver script is executed to measure the time required to load the modules. The benchmark reports both the import time and "visit" time. In general, the visit time will be similar across the different options, but the import time can vary greatly. Pynamic is known to stress the metadata performance of a file system since Python must access each module to build up the name space. In some cases, the load time for python modules can take minutes to hours on a heavily loaded system running at scale. This benchmark was designed to measure this behavior.

Figure 3 shows a plot of the Pynamic benchmark across several different storage options. The setup was generated with $so\_generator.py$ 495 1850. This corresponds to 495 modules (python library files) with an average of 1,850 methods per module. Most results include both a first access timing and a cached timing. In most cases, the test were performed on *Alva*, an Cray XC-30 test and development system. Since it is a test system, it has a small Lustre file system. However, we believe that similar results would be obtained on larger scale file systems since the main bottleneck for this benchmark is the metadata performance.

We will briefly describe the different storage options. Shifter is the NERSC implementation which has an Ext4 image stored in a Lustre file system and mounted via a loop mount. The Flash (DataWarp) data point is an unpacked image stored in a Flash-based file system running on an early version of Cray's DataWarp. The underlying file system for the Flash storage is accessed over DVS. The GPFS-Native data point was run on a login node which runs the native GPFS client and used an unpacked image stored in a GPFS file system. Note that it is currently not feasible to run the native GPFS client on a Cray compute node. The Lustre data point is an unpacked tree stored in a Lustre file system. It was executed on a compute node. GPFS-DVS represents the performance of accessing an unpacked image stored in a GPFS file system via DVS from a compute node. The final two data points use Docker running on a commodity cluster node with a local disk. These both use Docker's thin provisioning Logical Volume driver. In the first case, the underlying Logical Volume storage resides on a GPFS file system as a file in GPFS. The final data points is a standard Docker installation using a local disk.

The main observation with these results is that the Shifter approach performs very well. Cases that must heavily interact with the parallel file system's metadata service typically perform worse (GPFS - Native, Lustre, GPFS - DVS). DVS performs reasonably well after caching, but first access can be slow. Finally, native Docker typically performs very well. One noticeable data point is the GPFS - Native cached performance. GPFS's client-side cache can perform very well if the metadata can fit in the limited cache space. However, once this cache is exceeded, the performance drops off significantly. In this case, we suspect the entire Pynamic benchmark fit in the

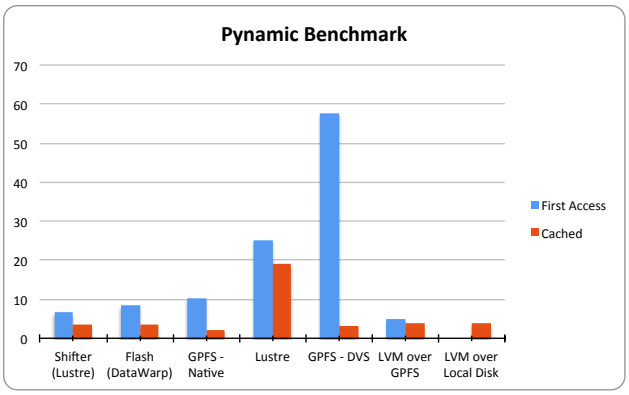

Fig. 3. Comparison of Pynamic execution times on different storage options.

GPFS metadata cache.

While none of these benchmarks test scaling, in general, we expect the Shifter approach to scale very well, since metadata can be more effectively cached on the compute node and metadata I/O operations occur at the Lustre object storage target (OSTs) level, not in the Lustre metadata target. This allows Shifter to leverage the more scalable component of the Lustre file system (the OSTs) and avoid contention on the Lustre metadata server. This is possible because the metadata is not globally shared and each image is read-only mounted. Even with the advent of Lustre's Distributed Namespace Environment (DNE), we still expect the Shifter approach will provide better performance for cases where metadata doesn't need to be globally maintained (i.e. synchronized).

We also conducted other metadata intensive benchmarks which we do not report and found similar results. So we do not believe these general trends are specific to Pynamic but would apply to other metadata intensive cases. It is also noteworthy that since the parallel file systems and interconnect are accessed natively, there is no additional overhead and we would expect to see native performance to the parallel file systems and over the interconnect. This is in sharp contrast to virtual machine based approaches which typically see large impacts due to the added overhead. We have not conducted any benchmarks to demonstrate this, but the previously cited results from IBM support this assertion [3].

VII. Discussion

A. Implications

While Shifter was designed to support User Defined Images, we have since realized it could help address other system related bottlenecks. For example, we have observed startup time delays on applications that use dynamically shared libraries. This issue is particularly noticeable at large scales. This is primarily due to the overhead for the application to traverse the file system to locate the required libraries (similar to the Pynamic use case). Shifter could be used to create custom images that contain all of the required libraries in the image in well defined locations. Based on the benchmarking results, this could significantly reduce the time needed to load the libraries and start the applications.

Shifter can also facilitate reproducibility. This is particularly important for scientific work, where scientists may need to verify results years later or duplicate an analysis pipeline but with updated data. Docker is already being considered as a tool to address this challenge [10]. Since images can be saved and tagged with Docker, users can easily bring back an image that was used in the past. There are limitations to this. For example, if the image has external dependencies or has strict dependencies on the system (i.e. MPI libraries and interconnect firmware levels) the image may no longer function. However, this provides a level of reproducibility that is difficult to achieve today. In addition, since Shifter can integrate into Docker's image repository, Docker Hub, this makes it easy for collaborators to develop and share images, even across institutions. Docker Hub was largely inspired by GitHub as way to facilitate the sharing of images and it is poised to become the de facto destination for distributing images. Already scientists are starting to build and package images via Docker Hub to improve collaboration and simplify reproducing results. This can help scientists be more productive, but also help them be better scientists.

B. Security

One driving concern in the design of Shifter was security. While at first glance the introduction of User Defined Images would seem to dramatically increase security risks, we believe that the approach used with Shifter largely mitigates any additional risks. We view the container as basically an extension of the application which the user can already control today. So Shifter largely simplifies and streamlines the ability to create and invoke environments, but doesn't provide any capabilities beyond what a regular user already possesses. Where some additional control is provided, Shifter is careful to limit how that capability can be used to prevent introducing additional security risks. For example, since the processes are all executed as the user (i.e. non-root), the user doesn't have any elevated privileges when running in their defined environment. We believe the most critical risk is that a user could elevate their privileges using something inside the image. However, the image and all mounted file systems are mounted with setuid and device support disabled. This addresses the most likely mode of escalation. Furthermore, privileged services (i.e root executed services) are not run inside the images.

Another concern is that the images may contain software that has (known) flaws. This is largely true already, since users can typically install their own software and may not maintain it. So this risk exists already and Shifter doesn't dramatically change this risk. Image repositories do introduce a new vector for attackers to exploit. However, users can already download and execute malicious code today, so this would be just a new example of an old theme. In addition, the Docker developer community is planning to add the ability sign and certify images which should help address some of the risk.

Since the images for Shifter are stored and packed on a Gateway server, this does provide an centralized place to audit and scan images for known vulnerabilities. NERSC may explore how we can leverage this option. For example, we could alert users that a selected image has a defect and warn them to avoid using the image. However, the authors still believe that these risks should be weighed carefully and that Shifter's approach does a good job of minimizing risks while greatly increasing productivity.

VIII. Future Directions

Moving the Shifter implementation to production-ready quality is our primary goal. To this end, there are a number of important areas that we would like to consider. First, we are investigating 3rd party solutions to the Image Gateway, such as the Openstack Glance software. We are also considering methods of running multiple Shifter UDI jobs on a single node - for example in NERSC's edison serial queue. Expanding on the native-format-agnostic capability of Shifter, we would like to extend support to qcow2 images, as well as other formats. Finally, we are also working on building a platform at NERSC to automatically generate and build optimized images focused on supporting MPI applications.

IX. Conclusion

Container-based computing is driving a revolution in computing. It is likely that container computing coupled with a growing repository of images will dominate how applications are developed and delivered in the coming years. While this revolution has yet to impact the scientific and HPC community, we believe that the flexibility and productivity gains it enables will drive its adoption in this space. While we consider this initial implementation of Shifter as a prototype, we believe it will serve as an early entry-point for our users to start exploring and leveraging this new model.

Acknowledgment

This work was supported by the Director, Office of Science, Office of Advanced Scientific Computing Research of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

The authors also wish to acknowledge the technical input and support from Cray staff, especially Dave Henseler, Dean Roe, Martha Dumler, Kitrick Sheets, and Dani Connor.

References

[1] "Docker," https://www.docker.com/.

[2] M. Helsley, "Lxc: Linux container tools," *IBM devloperWorks Technical Library*, 2009.

[3] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, "An updated performance comparison of virtual machines and linux containers," *technology*, vol. 28, p. 32, 2014.

[4] "Modules: Environment modules," http://modules.sourceforge.net/.

[5] S. Canon and C. Whitney, "Chos, a method for concurrently supporting multiple operating system," *Computing in High Energy Physics and Nuclear Physics*, 2004.

[6] "Linux standard base," http://www.linuxfoundation.org/collaborate/workgroups/lsb.

[7] T. Fly, D. Henseler, and J. Navitsky, "Case studies in deploying cluster compatibility mode," *Proceedings of the Cray User Group*, 2012.

[8] D. Auble, "Slurm native workload management on cray systems," *Proceedings of the Cray User Group*, 2014.

[9] "Pynamic: The python dynamic benchmark," https://codesign.llnl.gov/pynamic.php.

[10] C. Boettiger, "An introduction to docker for reproducible research, with examples from the R environment," *CoRR*, vol. abs/1410.0846, 2014. [Online]. Available: http://arxiv.org/abs/1410.0846