Custom Execution Environments with Containers in Pegasus-enabled Scientific Workflows

Karan Vahi*, Mats Rynge*, George Papadimitriou*, Duncan A. Brown†, Rajiv Mayani*
Rafael Ferreira da Silva*, Ewa Deelman*, Anirban Mandal‡, Eric Lyons§, Michael Zink§
*Information Sciences Institute, University of Southern California, Marina Del Rey, CA, USA
†Department of Physics, Syracuse University, NY, USA
‡RENCI - University of North Carolina, NC, USA
§Department of Electrical and Computer Engineering, University of Massachusetts at Amherst
{vahi,rynge,georgpap,mayani,rafsilva,deelman}@isi.edu,
dabrown@syr.edu,anirban@renci.org,elyons@engin.umass.edu, zink@ecs.umass.edu

Abstract—**Science reproducibility is a cornerstone feature in scientific workflows. In most cases, this has been implemented as a way to exactly reproduce the computational steps taken to reach the final results. While these steps are often completely described, including the input parameters, datasets, and codes, the environment in which these steps are executed is only described at a higher level with endpoints and operating system name and versions. Though this may be sufficient for reproducibility in the short term, systems evolve and are replaced over time, breaking the underlying workflow reproducibility. A natural solution to this problem is containers, as they are well defined, have a lifetime independent of the underlying system, and can be user-controlled so that they can provide custom environments if needed. This paper highlights some unique challenges that may arise when using containers in distributed scientific workflows. Further, this paper explores how the Pegasus Workflow Management System implements container support to address such challenges.**

I. Introduction and Motivation

Container technologies have become ubiquitous in all areas of scientific computing [1], [2]. While Docker [3] has been a leader in areas such as microservices, containers did not become common in high throughput computing (HTC) or in high performance computing (HPC) environments until Singularity [4] was introduced. Scientific computing specialists now have a range of container tools available to them, in addition to Docker and Singularity. There are also HPC-specific solutions like Shifter [5], open frameworks such as Podman [6] (which can be used to develop, manage, and run application containers), and orchestration tools like Kubernetes [7] (which can be used for automated deployment, scaling, and management of containerized applications).

In the context of scientific workflows, container technologies are especially interesting for two reasons: (1) they supply a way to foster reproducibility by providing a fully defined and reproducible environment; and (2) they are able to provide a flexible, custom, user-controlled environment in a manner that the underlying centrally managed compute cluster cannot (due to the fact that administrators have as main goal of providing a stable, slow-moving, multi-user environment). Science reproducibility in scientific workflows is often implemented as a way to reproduce the exact computational steps taken to reach the final results. While these steps are completely described, including the input parameters, datasets, and codes, the environment in which these steps are executed is only described at a higher level with endpoints and operating system name and versions. While this is enough to attain reproducibility in the short term, systems evolve and are replaced over time, breaking the workflow reproducibility. Containers are well defined and have whatever lifetime the owner desires, meaning that containers outlast whatever compute environment the workflow was first executed in. Similarly, containers also enable the workflow to be seamlessly transferred to completely different compute environments.

Applications are increasingly relying on a diverse set of underlying technologies and libraries to optimize the use of evolving computing hardware. As a result, requests for custom software environments are becoming more and more common. Users' software stack requirements may conflict with a system-provided stack or even with other users. These requirements are often impossible to satisfy for a stable, multi-user compute cluster environment. A typical example nowadays is TensorFlow [8], a popular machine learning toolkit. TensorFlow is Python-based and requires a set of very recent Python libraries. For example, this environment is not impossible to satisfy under a RHEL 7 based compute cluster. However, TensorFlow is a non-trivial stack to build and provide support for. Many HPC centers now solve this problem by providing Singularity images of such tools.

While containers increase reproducibility and enable custom environments, using containers in a distributed scientific workflow introduces some unique challenges. Most notable is the challenge of distributing the associated container images and making them available to the compute jobs. Pegasus [9] workflows regularly contain thousands or millions of jobs, simultaneously running across a set of different compute environments. To distribute the image at such a large scale and make the image available on the node where a job executes, special care has to be taken. Additionally, the container technologies are fragmented, so a one-size-fits-all approach may not be ideal while trying to support workflow execution in varied execution environments.

With the above goals in mind we have added new capabilities to Pegasus WMS to support variety of Container technologies in different execution environments.

This paper is organized as follows. We start by describing the requirements and design considerations that we identified in the process of supporting containers in Pegasus. Our approach on containers support is described in Section III. In Section IV, we present results from our experiments on a real world workflow and we quantify some of the overheads that result from using containers in Pegasus workflows. Section V describes two application workflows that are now using containers for their production runs. In Section V, we also report our experiences with and lessons resulting from executing these workflows with containers in a production environment. SectionVI describes the related work and finally, Section VII concludes with a brief summary of results and a discussion of potential future research.

II. Requirements and Design Considerations

Pegasus allows scientists to describe their computational pipelines as a directed, acyclic graph of tasks. The Pegasus input format (called the DAX) is a high-level, portable description that is agnostic of the underlying computing environment and refers to data and user codes using logical identifiers. Pegasus consumes this description and generates an executable workflow that is tailored for users that target computing environments such as local desktops, campus clusters, computational grids, and cloud environments. During this process, Pegasus automatically identifies the necessary input data and adds data management tasks to the user workflow that are responsible for fetching the data required for the workflow, stages-out the generated outputs to a user specified location, and optionally removes data products that are no longer needed as the workflow executes. The separation between the high-level user description of the pipeline in the DAX and the actual executable workflow that is executed on the computing resources has enabled our user community to keep abreast with infrastructure improvements and migrate their pipelines from original HPC focused environments (such as local campus clusters) to more distributed computing environments (such as Open Science Grid and clouds).

Running an application or service using containers is a well known and straightforward process. However, incorporating this process into scientific workflow systems, presents a unique set of challenges. Thus, we identified some over-arching requirements for architecting support for containers in Pegasus. There requirements are listed below:

1) *Support for different container technologies*: Early on it was clear to us that a one-size-fits-all approach would not suffice when picking a container technology for widespread use. For example, Docker, while popular in traditional corporate computing environments and in local captive computing resources, is not supported on most shared computing infrastructures as the Docker agents and jobs run as root. Singularity, however, is a preferred container technology that allows containerized jobs to run in user space in HPC environments. Some HPC centers such as the National Energy Research Scientific Computing Center (NERSC) have introduced their own specific container technologies. Shifter, enables users to securely run Docker images on NERSC's systems at scale. The goal of Pegasus is to allow users to optimize computing resources at their disposal and leverage technologies that are supported on those resources.
2) *Work in Distributed Environments*: Irrespective of the container technology supported, it was important to ensure that Pegasus' support enabled users to utilize containerized jobs in distributed environments. In such environments, users often don't know *a priori* which node or cluster their job might land on. The task of fetching and deploying a container that a job requires on a node and any associated setup required for the container (such as loading an image in the local container registry, etc.) should be handled by the workflow management system and not in users' scripts.
3) *Easy Configuration and Representation*: It should be easy for users to configure which container and type of container is required by their jobs, allowing different jobs in the same workflow to use different containers and technologies. The underlying representation to describe containers used for a workflow should be compact and prevent duplication.
4) *Support for Public Registries*: Today, a lot of popular container images are available to users in public registries (such as Docker Hub and Singularity Hub). Our solution should support retrieval of images from these registries for use in a workflow and should also scale-up for large workflows whereby the registries are not accessed repeatedly for the same image. Private images can be loaded directly from an image file.

III. Approach

We based our overall approach to incorporating support for containers by making containers first class citizens in our model and treating them as an input dependency for a job. Keeping in mind that we wanted our solution to scale-up for large workflows and access public registries without overloading these registries during a workflow run, we decided to represent the container dependency as a data dependency, and leverage Pegasus data management capabilities to manage distribution of containers required for a workflow. Support for containers in Pegasus was first introduced in Pegasus 4.8.0, which was released in September 2017 with support for Singularity and Docker Containers. The current version of Pegasus, version 4.9.1, also has support for Shifter.

A. Container Execution Model

Executing a job via a container on a remote node usually requires some setup and cleanup actions before and after the job has run. For example, before launching a job, the associated container image might need to be retrieved and loaded in the local container registry. After job completion, the container image might need to be unloaded/removed. We decided to incorporate the container setup for a job into PegasusLite [10], a light-weight Pegasus remote execution engine which wraps the user task on the remote worker node when a job is scheduled to the node. PegasusLite is responsible for figuring out the appropriate job directory in which the job executes, staging-in datasets that a job requires, launching the job, staging-out data, and cleaning up the job directory.

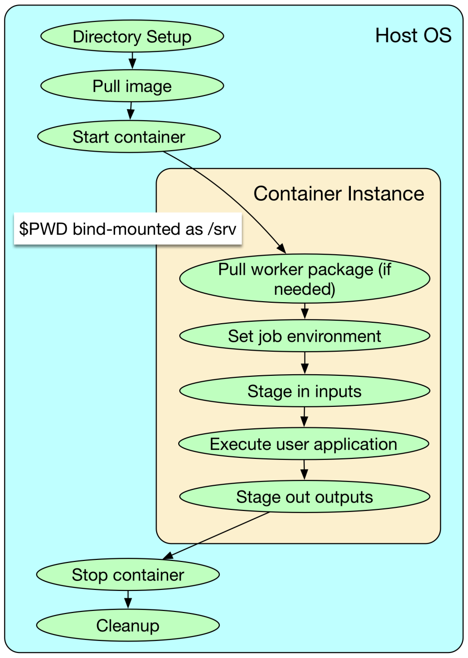

Fig. 1. The PegasusLite steps taken on the bare-metal host and inside the container instance.

The updated PegasusLite flow for handling a containerized job when starting on a remote worker node is outlined below:

1) Sets up a directory in which it will run a user job.
2) Pulls or links the container image to that directory.
3) Mounts the job directory into the container as /scratch for Docker containers, while as /srv for Singularity containers.
4) Container then runs a job specific script created by PegasusLite that does the following:
    a) Figures the appropriate Pegasus worker to use in the container if not already installed.
    b) Sets up the job environment to use (including transfer and setup of any credentials transferred as part of PegasusLite).
    c) Pulls in all the relevant input data and executables required by the job.
    d) Launches the user application using pegasus-kickstart.
    e) Ships the output data to the staging site.
5) Optionally, shuts down the container (only applicable to Docker containers).
6) Cleans up the directory on the worker node.

The Pegasus planner configures PegasusLite for each job, with container specific directives using a pluggable interface. This interface allows us to easily incorporate newer container technologies. Below, we describe the supported Container technologies and how PegasusLite is configured for each of them. For the purposes of this section, we assume the associated container image file is present on the node where the job is executed. Section III-B describes how the container image required for the job gets to the worker node.

1) Docker: PegasusLite first loads the container from the container image file into the local node registry. It then identifies the user (the user on the host OS) that the job is being launched as and creates the same user in the container if it does not exist. Running the job inside the container with the same UID/GID as the host OS ensures data access to the host working directory and guarantees any outputs created by the job are as the same user as on the host OS. The job directory as determined by PegasusLite gets mounted as /scratch into Docker containers. The job is then setup and executed in the container as described in Step 4.

2) Singularity: Comparatively, Singularity setup is more straightforward as it is designed to be executed in user space and the image file can be invoked as any other Linux executable using singularity exec command. In case of Singularity, the job directory (as determined by PegasusLite) is mounted as /srv in the container, and then the job is setup and executed in the container as described in Step 4. Using /srv under Singularity comes from Singularity not using overlay-fs in some cases, and hence having to rely on existing mount points in the image when bind mounting.

3) Shifter: Shifter is a technology developed by NERSC and optimized for executing on HPC machines. Shifter containers are different from Docker and Singularity because Shifter containers cannot be exported to a container image file that can reside on a filesystem. Additionally, the containers are expected to be available locally on the compute sites in the local Shifter registry. In case of Shifter, the job directory (as determined by PegasusLite) is mounted as /scratch in the container and the job is then setup and executed in the container as described in Step 4.

B. Data Management for Containers

Pegasus treats containers as an input data dependency for a job that needs to be staged to a compute node if it is not already present there. If a container is described in the Transformation Catalog as residing in a container registry (such as Docker Hub or Singularity Hub), we first export the image as a container image file as part of the executable workflow. To achieve this, we modified our data transfer tool *pegasus-transfer* to pull the images from Docker or Singularity Hub and export them as container image files. Treating containers as data allows us to:

- Execute wokflows in distributed environments, where the container, along with job input data, gets deployed at runtime on a remote compute node when a job starts.

- Optimize transfer of containers for large workflows in a manner similar to how Pegasus does for datasets [10]. This is of particular importance when workflows refer to containers in public container registries. For a workflow, Pegasus will retrieve a particular container image from a public registry once per data staging site irrespective of the number of jobs in the workflow. This is significant as the access pattern for container images from a workflow execution can appear as a Denial of Service attack for the operators of the registries.
- Export the image to a file format from a Hub, which also allows us to stage-in the image to the compute nodes via a staging server in the instance that the actual compute nodes may not have direct access to the public internet.
- Symlink against a container image file if a shared filesystem is available on the compute nodes of a target execution resource. In this case, the data staging node added by Pegasus in the executable workflow will place the container image on a directory on the shared filesystem and all the jobs will then symlink the container into their job directories. This is particularly useful when jobs refer to large container images. This also minimizes saturation of network links in a compute site. In Section IV, we highlight the benefit of this optimization.

By default, Pegasus only mounts the job directory determined by PegasusLite into the application container. However, in the Transformation Catalog, users can specify additional directories that need to be mounted into the container and made available to the job. This allows jobs in the workflow to symlink against pre-existing input data sets that may be available on the node. Many computing sites now distribute datasets using CVMFS [11] (e.g., the Gravitational-wave Open Science Center [12]), which Pegasus can mount into the container automatically when a user job starts.

Overall, treating containers as an explicit data dependency for jobs has given us the flexibility to leverage containers in a variety of execution environments while simultaneously optimizing the access pattern based on the individual compute environments.

C. Representation

Pegasus users describe their applications as transformations in a Transformation Catalog. The Transformation Catalog maps logical transformations to physical executables on a particular system. It also provides additional information about the transformation such as what system they are compiled for, what profiles or environment variables need to be set when the transformation is invoked, and so on. Users have an option of marking the executables as installed on a particular system or as stageable, in which case Pegasus will transfer the executable along with the job. We updated the Transformation Catalog to allow users to refer to a container that is required for execution. A sample representation is illustrated below

```
- transformations:
  - namespace: "example"
    name: "keg"
    version: 1.0
    site:
    - name: "isi"
      arch: "x86"
      os: "linux"
      container: "centos-pegasus"
      pfn: "/shared/pegasus/bin/pegasus-keg"

      # INSTALLED means pfn refers to path in the container.
      # STAGEABLE means the executable can be staged into
      # the container
      type: "INSTALLED"

- cont:
  - name: "centos-pegasus"

    # URL to image in a docker|singularity hub|shitfer
    # repo url OR URL to an existing docker image
    # exported as a tar file or singularity image
    image: "docker:///rynge/montage:latest"

    # can be either docker or singularity or shifter
    type: "docker"

    # mount information to mount host directories into
    # container  format src-dir:dest-dir[:options]
    mount:
    - "/Volumes/Work/lfs1:/shared-data/:ro"

    # environment to be set when the job is run in the
    # container only env profiles are supported
    profile:
    - env:
        JAVA_HOME: "/bin/java.1.6"
```

The container itself is defined using a separate “cont” entry. Multiple transformations can refer to the same container. Users can choose whether to use either the same container for all their jobs in the workflow or different containers for different types of jobs. We also support the notion of allowing users to stage their executables into a container at runtime. This is useful when a user maybe using a standard base container image from a public repository but wants to user their own executables.

We briefly describe the attributes supported for describing a container

- cont - A container identifier.
- image - URL to image in a Docker—Singularity hub—shifter repo URL or URL to an existing Docker image exported as a tar file or an existing Singularity image. An example of a Docker hub URL is docker:///rynge/montage:latest and an example of a singularity is shub://singularity-hub.org/pegasus-isi/fedora-montage. Shifter images can only be referred to by a shifter URL scheme that indicates that the image is available in the local shifter repository on the compute site. An example of this is shifter:///papajim/namd_image:latest.
- mount - mount information to mount host directories into container of format src-dir:dest-dir[:options]. This is used to mount directories from the shared filesystem on a compute site in the container, for symlinking against pre-existing inputs a job may require.
- profiles - One or many profiles can be attached to a transformation for all sites or to a transformation on a particular site. For containers, only env profiles are supported.

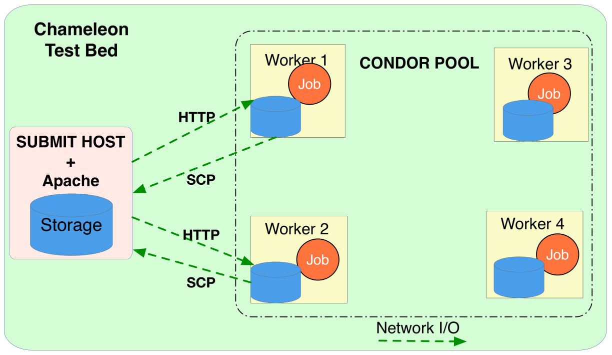

Fig. 2. Non Shared Fileystem Setup

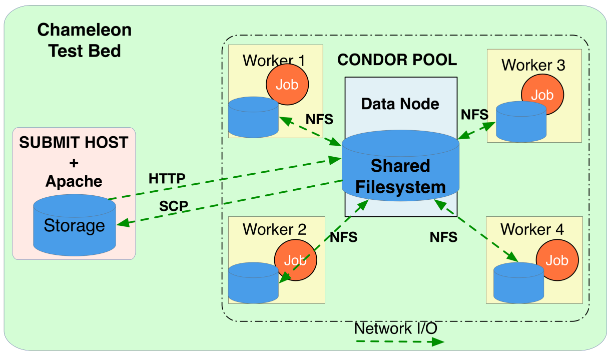

Fig. 3. Shared Filesystem Setup

IV. Experiments

In order to obtain a better understanding of the performance overheads when adding container transfers and instance management, a set of experiments were conducted. The Chameleon testbed [13] was used as our testing infrastructure. Our setup consisted of one workflow submit node, one NFS server node, and four worker nodes, located in Texas Advanced Computing Center (TACC). All of the nodes were bare metal nodes with 24 physical cores, 128GB of RAM, and 10Gbps network connection. However, because we wanted to simulate lower network speeds on the submit node, we capped its network link to 1Gbps. Additionally, the submit node didn't have access to the storage server via NFS. The shared filesystem was only shared across the worker nodes. Our software stack on the submit node apart from HTCondor and Pegasus, also included an *http* server, in order to allow the workers to fetch input data to the jobs.

As a test workflow, we used the CASA application workflow described in SectionV-B, which is normally configured to use Docker containers. The workflow was configured to use *http* for staging-in data to the jobs and *scp* to store data back to the data staging site. The CASA workflow we used in our experiment had 63 compute tasks in the abstract workflow and 73 jobs in the executable workflow (when there is no task clustering). The 10 additional tasks are the data transfer and auxiliary tasks added by Pegasus during the planning of the workflow. Due to the real-time nature of this workflow, each compute task is designed to complete execution within a few seconds as it processes incoming data every minute.

We devised three experiments. All experiments executed the workflow using PegasusLite [10] mode. In this mode, each job is wrapped in a lightweight PegasusLite instance that determines the directory in which the user task will be run, pulls in the input data for the job from data staging site, launches the task, and stages out the generated outputs back to the data staging site before cleaning up. The first experiment which served as a base, involved executing the CASA workflow without any containers, with input data (application data) staging occurring via *http* from the submit host, as shown in Figure 2. The second experiment involved executing workflows with containers (Docker and Singularity both), with input data (container and application data) staging occurring via *http* from the submit host, as shown in Figure 2, and the stage-out of data from workers to submit host via *scp*. In the third experiment, we staged the input data to the NFS (as shown in Figure 3) and then had the compute jobs symlink against the input data on the NFS instead of storing a true copy to the local filesystem. For all the experiments we ran workflows 10 times in each configuration, and present average results over these 10 runs where applicable. Additionally, while running these experiments, in order to collect network and block device statistics, on all nodes we were recording system activity every second via *sar* [14]. The goals of these experiments were the following:

- Demonstrate the increase in walltime due to the staging of application containers and how job clustering can help mitigate the overhead.
- Show that the staging of application containers for a workflow for each task can saturate both the network links and disk IO.

The first graph in Figure 4 shows an increase in end to end workflow walltime from 172.2 seconds to 681.7 and 321.6 when Docker containers and Singularity containers are used respectively and when there is no job clustering (Cluster size 1 - purple bar and lines in the plots). We also noticed that the Docker workflow walltime is much longer than Singularity. This is primarily because of the difference in the size of the Docker and Singularity image files (488MB and 153MB respectively) even though the underlying recipe files are functionally equivalent. Clustering the tasks together (so that one job executes 12 tasks; green bars in the plot) helps reduce the workflow walltime as job clustering leads the image being transferred only once per 12 tasks, and to fewer jobs in the final executable workflow; thereby reducing the number of times the container image is transferred from the submit host to the worker nodes.

Figure 5 shows the network link usage on the submit host, which is also the data staging site for the Docker case in non shared filesystem setup, as shown in Figure 2 (with and without job clustering). The top subgraph (no job clustering) shows sustained period of network saturation on the link because of the associated data transfers of the containers per

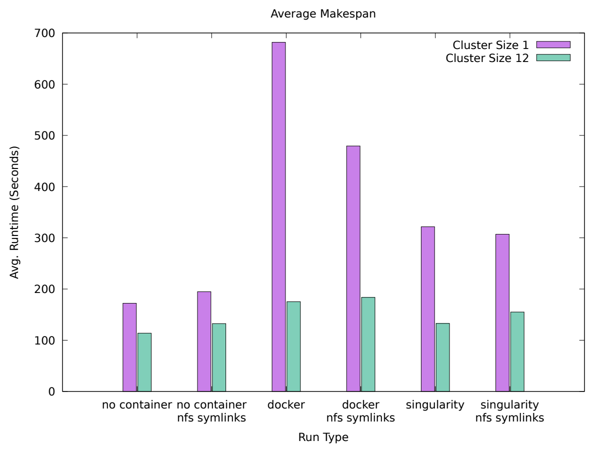

Fig. 4. Average workflow makespan per execution environment setup

job. On the other hand, in the bottom subgraph (clustering 12 tasks per job) the network saturation is not sustained.

Figure 6 shows average service time of I/O requests on one of the workers over a workflow run for executions with and without Docker containers; and with and without job clustering. In the case of no containers, the effect on the average service time is negligible. Introducing containers leads to a significant increase in average disk wait times, even though we are symlinking against the Docker image file on the NFS instead of doing a true copy to the local disk. The reason for this is that the Docker image image file still needs to be untarred internally on local disk by Docker before being loaded in the local node registry. A proposed optimization for smarter Docker loading in the PegasusLite script is explained in Section VII, which would lower the loading overhead when multiple jobs land on the same compute nodes. The same average service time plot for Singularity shown in Figure 7 does not present a similar increase. This can be attributed both to Singularity images being read directly, as well as being comparatively much smaller in size.

An interesting observation is that the use of NFS doesn't improve the workflow makespan except in the case of Docker, and that too without any job clustering. The goal of the NFS in the experimental setup is to minimize or alleviate the prolonged network saturation on the Submit node when transferring a lot of input data at the same time and in this case specifically, container images. In the case of no clustering, compared to Docker, Singularity doesn't saturate the network link on the Submit node, which can be attributed to its significantly smaller image size, and as a result NFS doesn't improve the performance as much. In the case of clustering 12 tasks together, experiments using either Docker or Singularity aren't benefiting by the use of NFS, since clustering tasks together reduces the amount of the transferred input data. Finally, in the cases where network saturation isn't the bottleneck, using the NFS adds extra overhead because of the additional stage-in and stage-out steps that need to be executed by Pegasus.

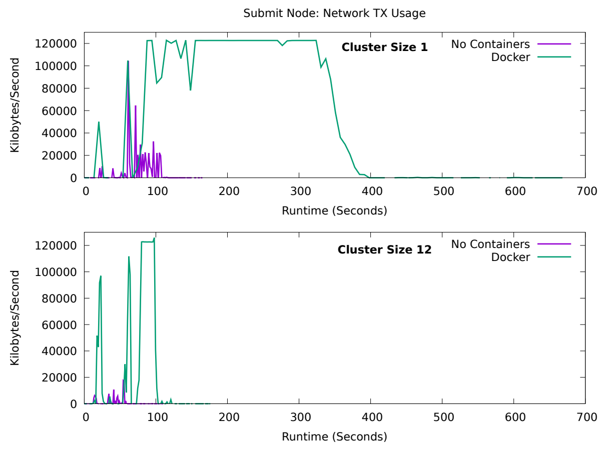

Fig. 5. Egress network traffic on the submit node, without the use of containers and using Docker. NFS is **not** used.

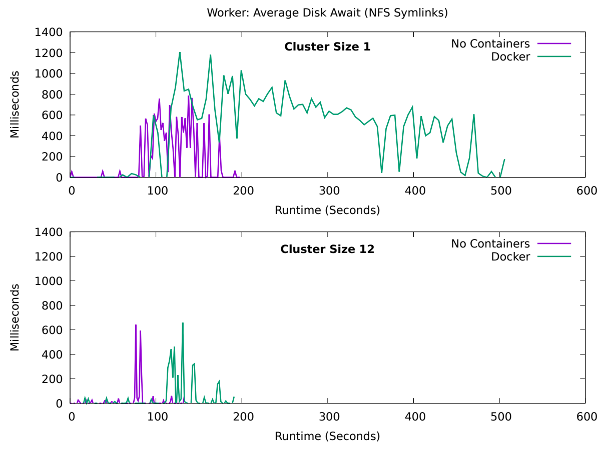

Fig. 6. Average service time of I/O requests on worker 4 using Docker containers with NFS symlinking.

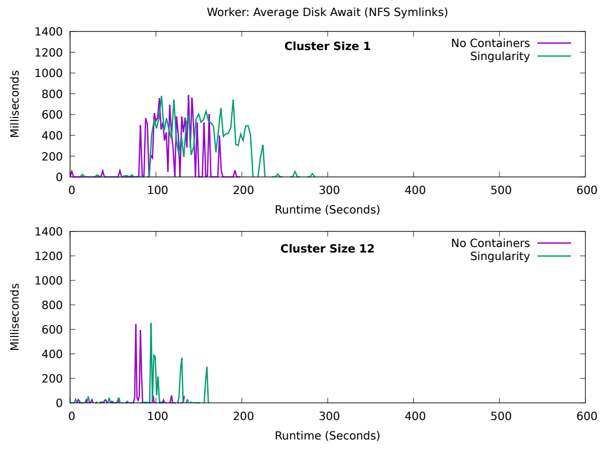

Fig. 7. Average service time of I/O requests on worker 4 using Singularity containers with NFS symlinking.

V. Case Studies

Since container support was first added to Pegasus, our user base has been migrating their existing Pegasus pipelines to use containers. In this section, we describe two such applications, one using Singularity and the other using Docker.

A. PyCBC

PyCBC is a python-based software package used to explore astrophysical sources of gravitational waves [15]. It contains algorithms that can detect coalescing compact binaries [16]–[20] and measure the astrophysical parameters of detected sources [21]. PyCBC was used in the discovery of gravitational waves from binary black holes [22] and binary neutron stars [23], in the ongoing analysis of gravitational-wave data by the LIGO and Virgo Scientific Collaborations [24], and in the analysis of these data by independent groups [25].

PyCBC executables are Python scripts that call functions from PyCBC-provided, system, and third-party Python libraries, as well as compiled code from shared-object libraries. Unlike previous generations of gravitational-wave search executables that were provided as statically-linked C code with minimal dependencies [26], PyCBC programs have a complex run-time environment that must be carried with each executable. A standard PyCBC installation requires that the install directory is available at runtime so that Python can find the package's libraries. It also requires that the build and runtime environments are compatible (e.g., compatible versions of glibc, gcc, and Python). Some PyCBC executables (e.g., `pycbc_inspiral`) require runtime compilation of code using SciPy weave [27], so the execution environment must have a full installation of gcc and the Python development libraries.

On clusters where the user community has control over the Python interpreter and software installation, the complexities of the run-time environment have been managed by using a standard software installation and the Python virtual environment tool `virtualenv` to create known environments into which all the software is installed. However, when running on OSG or XSEDE clusters for example, these requirements may not be satisfied. Although CVMFS can provide access to the PyCBC libraries at runtime on the OSG, many OSG execute machines do not have the required environment to weave-compile code at runtime. The initial solution for these environments was to build bundled executables using PyInstaller [28]. These bundles contain all of the Python and user-space C libraries required, as well as a Python interpreter to run the code. This bundle must also contain pre-compiled objects for all of the weave code that is needed at runtime. A build script run under Travis CI as part of PyCBC's build and test system complied the necessary code into the bundle.

The use of PyInstaller allowed the construction of more self-contained executables that can be deployed on the OSG. However, this approach had several complexities: PyInstaller bundles are not completely static and still require a dynamically linked version of glibc at runtime. Since Linux systems are backwards but not forwards compatible, the bundle needed to be built on the lowest-common denominator operating system for the execution platform (e.g. RHEL6 for OSG). The bundle building script itself was quite complicated and the insertion of weave-compiled objects into the bundle required running the executable, extracting the complied object code from the weave cache, and inserting it into the bundle as part of the build script. If a user added new code that was not exercised as part of this build, then the compiled objects would be missing from the bundled executable and attempts to use that feature would fail at run time.

Containerization has allowed us to mitigate these problems and create a more robust run-time environment. Since PyCBC's Travis CI build script was already configured to build a Docker container, this container is be converted into a Singulariy image using the OSG-provided `cvmfs-singularity-sync` tool [29]. This tool pulls the latest PyCBC container (as well as released versions) from Docker Hub, converts the Docker container to a Singularity image, and copies the resulting image to the OSG's CVMFS origin server. Since the PyCBC maintainers have complete control over the PyCBC Docker container environment, any necessary run-time software (including software needed for runtime compilation of code) can be installed into the container.

The implementation of containers in Pegasus meant that PyCBC's workflow generation script did not need to be modified as deployment of the container is managed by the Pegasus WMS. If the user wants Pegasus to run a PyCBC program inside a container, then Pegasus profiles are set to specify the container type, the path to the container image, and directives to mount CVMFS inside the container for access to data needed at runtime. The PyCBC program that invokes the Pegasus planner can set Pegasus profiles based on its command-line arguments, so no code changes were needed in the PyCBC code itself. During testing, we discovered that two changes were needed in Pegasus for optimal use of containers in PyCBC: better handling for containers distributed via CVMFS and better handling of data staging for containerized jobs. We describe these changes in Sec. V-C below. A containerized PyCBC workflow was used to run the analysis for the First Open Gravitational-Wave Catalog [25] on the OSG.

B. CASA

CASA is an NSF Engineering Research Center with a focus on low atmosphere sensing, particularly with the use of networks of Doppler weather radars for the purpose of improving severe weather warning systems, emergency response, and situational awareness. Radar systems produce voluminous data, requiring substantial computing and networking infrastructure to process the data in a timely manner. Radar data is fused together to create derived products on which chains of post-processing occur, including image creation, GIS style analysis, and notification mechanisms for decision support. One of these chains, Nowcasting, was selected as a representative example of a domain user workflow that was well tailored for this approach to processing. Nowcasting is a form of short range forecasting that primarily uses observed radar data over time to produce a non-linear advection model that estimates future positions of radar echoes. Every minute, the Nowcasting algorithm produces 31 grids, representing the projected radar reflectivity every minute from 0-30 minutes. As part of the workflow, CASA creates images of each of these grids and extracts contours representing projected areas of meteorological impact based on predefined thresholds. Contours are represented as GIS style polygons, which are effective ways of describing geographic areas of weather risk to end users. However, contouring into well-ordered non-convex polygons is a function of the weather contained and the grid size, which is a CPU intensive process approaching O(n^2) complexity. Because timeliness is a key concern for effective end user response, innovative approaches to processing, including the use of the academic cloud, are necessary to keep up with the one minute update rate of nowcasting.

C. Experiences/Lessons Learnt

Based on our users' feedback from using containers in their scientific workflows, we have introduced optimizations and made changes to our approach. Here we describe a few of these changes.

1) Direct Access to Singularity Images via CVMFS: The PyCBC pipeline usually runs on a HTCondor-based computing infrastructure (e.g., on the Open Science Grid, on clusters at Syracuse University and AEI-Hannover, or on the LIGO Data Grid). On this infrastructure, the singularity image files are distributed using CVMFS, a scalable, reliable, and low-maintenance software distribution service that is available on all the nodes. In the most general case, Pegasus opts to pull a container image once to the data staging site and then lets the jobs pull the image to the compute nodes filesystem along with the other input data. However, this approach did not take advantage of the out-of-band caching and distribution of the Singularity images provided by CVMFS. In order for PyCBC workflows to directly use Singularity images stored in CVMFS, we updated Pegasus to allow for bypassing of container images files to the staging site and to enable symlinking to pre-existing images on the compute sites. These improvements allowed PyCBC workflows to use containers without introducing any new data transfer overhead associated with container data-staging.

2) Moved transfer data staging into the container rather than the host OS: In Pegasus 4.8.x and 4.9.0, the PegasusLite script (the lightweight job wrapper that is used to launch a job on the compute node) executing on the host OS was responsible for pulling in the job inputs from the staging site into the directory where the job runs, and then mounted that directory into the application container at runtime. The user application then was launched in the container and executed. The generated outputs were then staged back to the staging site by PegasusLite. This approach had the advantage of doing all the data transfers outside of the container relying on the tools provided by the computing infrastructure provider. However, this left the user with no control over using their own preferred choice of data transfer protocols when staging the data products to the worker nodes. With Pegasus 4.9.1, we moved data staging to occur within the application container so that users can install and leverage additional data staging tools specific to their infrastructure. We ran into this issue while executing PyCBC workflows on local cluster at Syracuse, where preferred grid transfer tools such as *gfal-copy* are not pre-installed.

3) Loading multiple Docker image tar files: Currently, Pegasus doesn't have any optimization in place to avoid loading an image that already exists in the local Docker images, or an image that is already being loaded by a another PegasusLite script. This can result in multiple "docker load -i" commands to be dispatched within a short period of time, severely impacting the performance of the local disk. In Figure 6, we presented the effect of loading multiple Docker images during a CASA workflow run where we observed the average wait time for an I/O service request times to be as high as 1.2 seconds. As a result, a very powerful node (24 physical cores with 128GB RAM) becomes almost unresponsive. In a scenario that the workers are shared among projects, this can also affect others if their jobs get scheduled to the affected nodes. Preliminary results we have from machines using SSDs present a much more subtle effect, however we are planning to address this issue on the upcoming Pegasus release.

VI. Related Work

Over the past decade, container systems such as Docker and Singularity have emerged to facilitate the sharing and migration of software by defining immutable, reusable execution environments. Their rapidly adoption by the scientific community have already supported a number of scientific progresses [1], [2]. Several systems have been leveraged to improve the way containers are stored, shared, and indexed. For instance, the OSG has leveraged the CVMFS filesystem for storing and sharing Singularity images for their users [11]. This solutions provides a consistent and efficient environment across all OSG computing sites. On the deployment management aspect, Kubernetes [7], an open source cluster manager for Docker containers, decouples application containers from the details of the system they run. Both technologies have significantly improved the way containers are built, stored, and delivered, and they have been leveraged by several workflow management systems as discussed below. Kubernetes has been widely used by EGI for managing containerized workloads and services. Through EGI Cloud Container Compute [30] users can start a cluster of virtual machines and create a Kubernetes cluster to run Docker containers, which can be spawned to their high throughput computing infrastructure.

In the context of scientific workflows, new systems have been designed for specifically running workflows on cloud environments with extended support for Linux containers [31]–[36], while well-established workflow systems have evolved to provide seamless support for running containerized applications [37], [38]. Skyport [31], [32] utilizes Docker containers to solve software deployment issues via software isolation. It targets automated deployment of Docker containers on cloud environments. Once built, container images are stored in the Shock data management system [31], in which data is represented as an object with a unique identifier (the Shock node ID) and metadata describing computational and scientific provenance information. At runtime, input files and Docker images are automatically deployed into cloud instances to perform computation. In Pegasus, we can emulate such behavior by running workflows on OSG where application container images are stored in the CVMFS filesystem. Additionally, we provide mechanisms to properly configure the execution environment within the container as defined by the user in the workflow description (as described in Section III).

Airflow [33] is a workflow system for running DAG-based workflows on cloud platforms. Recently, they have enabled support for Kubernetes to orchestrate the execution of Docker-enabled application containers in commercial clouds. Similarly, CWL-Airflow [34] provides a lightweight abstraction layer for running workflows described with the Common Workflow Language (CWL) [39] interface. Pachyderm [35] is data-driven pipeline execution system natively built for running workflows on Docker and Kubernetes. In Pachyderm, workflows are organized into Git repositories and describe operations to be performed in data files stored in such repositories, which fosters reproducibility through version control. Nextflow [36] is a workflow management system that uses Docker and Singularity for multi-scale handling of containerized computation. More recently, they have also enabled support for Kubernetes in a similar manner as for Pachyderm. Most of these systems target the orchestration of application containers on cloud platforms. In contrast, our approach is platform agnostic, i.e. application containers can be deployed in a range of computing platforms including standard laptops, campus clusters, and HPC and HTC systems. To the best of our knowledge, Pegasus is the first workflow system providing such versatility while preserving the fundamental concept of scientific workflow portability, i.e. framework and platform agnostic descriptions of workflows.

Makeflow, a well-established workflow management system, has evolved to enable support for application container execution on distributed computing resources [37], [38]. In order to capacitate Makeflow with containers, they have conducted a study [37] on how to best integrate container technology into workflow systems by analyzing different methods for orchestrating and running workflow tasks (e.g., individual tasks per container, multiple compatible tasks per container, etc.). Our proposed and implemented approach complies with the outcomes of this study, and extends it by providing mechanisms for automated data management for containers.

VII. Conclusion and Future Work

In this paper, we have described our overall approach to incorporating a variety of container technologies in Pegasus WMS, enabling our users to use them in varied execution environments. Since first releasing support for containers in Pegasus 4.8.0, our user base has slowly and surely started migrating their production workflows to use containers. Our approach focused on efficiently managing automatic container deployment on remote nodes on which workflow jobs are executed. This deployment went hand in hand with ensuring that requisite job input data was made available to the user application in the container while simultaneously preserving data access optimizations (such as mounting directories hosting input data automatically). By working with different user communities, we now realize that the question of whether to execute data transfers required for a job from within the container or from the host OS is not yet answered. We have seen compelling arguments for both approaches. We now feel that, instead of preferring one over the other, we should present the options to the users and Pegasus should support both options regardless.

Thus far, we have allowed users to refer to a pre-built container hosted either in a public container repository or exported as a tar image on a file server. In the future, we plan to allow users to specify the container build file (such as DockerFile in Pegasus catalogs) instead of using a pre-built image. Pegasus will then build the image automatically as part of the executable workflow and deploy the image to remote compute nodes where jobs execute. This makes the user workflow more self contained as the entire environment is described in a Docker file alone, enabling reproducibility and ease of sharing application user workflows. Additionally, we plan to support private container registries by implementing the necessary credential support in our data transfer tool *pegasus-transfer*. We would also like to explore use of Docker alternatives like Podman [6] which, unlike Docker, is a daemonless container engine used to manage OCI containers and can run containers in root or rootless mode by utilizing Linux namespaces. Use of Podman would enable us to overcome drawbacks of running Docker in HPC environments and would bridge the gap between Docker and Singularity.

Lastly, the development of Unikernels [40] is of interest, as they are essentially applications compiled along with a library kernel into a micro VM. Packaging in this manner provides smaller image sizes and faster boot times than containers, and at the same time isolation characteristics of a VM.

Acknowledgments

This work is funded by NSF contract #1664162, “SI2-SSI: Pegasus: Automating Compute and Data Intensive Science”; and NSF contract #1826997, “CC* Integration: Delivering a Dynamic Network-Centric Platform for Data-Driven Science (DyNamo)”. Development of containerization in PyCBC was supported by NSF contract #1443047, “CIF21 DIBBs: Domain-Aware Management of Heterogeneous Workflows: Active Data Management for Gravitational-Wave Science Workflows.” DAB was supported in part by NSF contract #1748958 to the Kavli Institute for Theoretical Physics.

References

[1] C. Boettiger, “An introduction to docker for reproducible research,” *ACM SIGOPS Operating Systems Review*, vol. 49, no. 1, pp. 71–79, 2015.

[2] P. Di Tommaso, E. Palumbo, M. Chatzou, P. Prieto, M. L. Heuer, and C. Notredame, "The impact of docker containers on the performance of genomic pipelines," *PeerJ*, vol. 3, p. e1273, 2015.

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

[4] G. M. Kurtzer, V. Sochat, and M. W. Bauer, "Singularity: Scientific containers for mobility of compute," *PloS one*, vol. 12, no. 5, p. e0177459, 2017.

[5] "Shifter: User defined images," https://www.nersc.gov/research-and-development/user-defined-images.

[6] "Podman," https://podman.io.

[7] "Kubernetes," https://kubernetes.io.

[8] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "Tensorflow: A system for large-scale machine learning," in *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, 2016, pp. 265–283.

[9] E. Deelman, K. Vahi, G. Juve, M. Rynge, S. Callaghan, P. J. Maechling, R. Mayani, W. Chen, R. Ferreira da Silva, M. Livny, and K. Wenger, "Pegasus: a workflow management system for science automation," *Future Generation Computer Systems*, vol. 46, pp. 17–35, 2015.

[10] K. Vahi, M. Rynge, G. Juve, R. Mayani, and E. Deelman, "Rethinking data management for big data scientific workflows," in *Workshop on Big Data and Science: Infrastructure and Services*, 2013, funding Acknowledgments: OCI SDCI program grant #0722019 and OCI SI2-SSI program grant #1148515. [Online]. Available: http://pegasus.isi.edu/publications/2013/Vahi-PegasusLite-IEEE-BigData-2013.pdf

[11] "OSG Helpdesk: Docker and Singularity Containers," https://support.opensciencegrid.org/support/solutions/articles/12000024676-docker-and-singularity-containers.

[12] M. Vallisneri, J. Kanner, R. Williams, A. Weinstein, and B. Stephens, "The LIGO Open Science Center," *J. Phys. Conf. Ser.*, vol. 610, no. 1, p. 012021, 2015.

[13] "Chameleon cloud testbed," https://www.chameleoncloud.org.

[14] "Sysstat utilities page," http://sebastien.godard.pagesperso-orange.fr/man_sar.html.

[15] A. H. Nitz, I. W. Harry, J. L. Willis, C. M. Biwer, D. A. Brown, L. P. Pekowsky, T. Dal Canton, A. R. Williamson, T. Dent, C. D. Capano, T. J. Massinger, A. K. Lenon, A. B. Nielsen, and M. Cabero, "PyCBC Software," https://github.com/gwastro/pycbc, 2018.

[16] B. Allen, W. G. Anderson, P. R. Brady, D. A. Brown, and J. D. E. Creighton, "FINDCHIRP: An Algorithm for detection of gravitational waves from inspiraling compact binaries," *Phys. Rev.*, vol. D85, p. 122006, 2012.

[17] B. Allen, "χ^2 time-frequency discriminator for gravitational wave detection," *Phys. Rev.*, vol. D71, p. 062001, 2005.

[18] S. A. Usman *et al.*, "The PyCBC search for gravitational waves from compact binary coalescence," *Class. Quant. Grav.*, vol. 33, no. 21, p. 215004, 2016.

[19] A. H. Nitz, T. Dent, T. Dal Canton, S. Fairhurst, and D. A. Brown, "Detecting binary compact-object mergers with gravitational waves: Understanding and Improving the sensitivity of the PyCBC search," *Astrophys. J.*, vol. 849, no. 2, p. 118, 2017.

[20] A. H. Nitz, T. Dal Canton, D. Davis, and S. Reyes, "Rapid detection of gravitational waves from compact binary mergers with PyCBC Live," *Phys. Rev.*, vol. D98, no. 2, p. 024050, 2018.

[21] C. M. Biwer, C. D. Capano, S. De, M. Cabero, D. A. Brown, A. H. Nitz, and V. Raymond, "PyCBC Inference: A Python-based parameter estimation toolkit for compact binary coalescence signals," *Publ. Astron. Soc. Pac.*, vol. 131, no. 996, p. 024503, 2019.

[22] B. P. Abbott *et al.*, "Observation of Gravitational Waves from a Binary Black Hole Merger," *Phys. Rev. Lett.*, vol. 116, no. 6, p. 061102, 2016.

[23] ——, "GW170817: Observation of Gravitational Waves from a Binary Neutron Star Inspiral," *Phys. Rev. Lett.*, vol. 119, no. 16, p. 161101, 2017.

[24] ——, "GWTC-1: A Gravitational-Wave Transient Catalog of Compact Binary Mergers Observed by LIGO and Virgo during the First and Second Observing Runs," 2018.

[25] A. H. Nitz, C. Capano, A. B. Nielsen, S. Reyes, R. White, D. A. Brown, and B. Krishnan, "1-OGC: The first open gravitational-wave catalog of binary mergers from analysis of public Advanced LIGO data," *Astrophys. J.*, vol. 872, no. 2, p. 195, 2019.

[26] D. A. Brown, "Using the INSPIRAL program to search for gravitational waves from low-mass binary inspiral," *Class. Quant. Grav.*, vol. 22, pp. S1097–S1108, 2005.

[27] E. Jones, T. Oliphant, P. Peterson *et al.*, "SciPy: Open source scientific tools for Python," http://www.scipy.org/, 2001–.

[28] H. Goebel *et al.*, "PyInstaller," https://www.pyinstaller.org/.

[29] B. Bockelman, M. Rynge *et al.*, "OSG CVMFS Singularity sync," https://github.com/opensciencegrid/cvmfs-singularity-sync/.

[30] "EGI Cloud Container Compute," https://www.egi.eu/services/cloud-container.

[31] W. Gerlach, W. Tang, K. Keegan, T. Harrison, A. Wilke, J. Bischof, M. D'Souza, S. Devoid, D. Murphy-Olson, N. Desai *et al.*, "Skyport: container-based execution environment management for multi-cloud scientific workflows," in *Proceedings of the 5th International Workshop on Data-Intensive Computing in the Clouds*. IEEE Press, 2014, pp. 25–32.

[32] W. Gerlach, W. Tang, A. Wilke, D. Olson, and F. Meyer, "Container orchestration for scientific workflows," in *2015 IEEE International conference on cloud engineering*. IEEE, 2015, pp. 377–378.

[33] "Apache Airflow," https://airflow.apache.org.

[34] M. Kotliar, A. Kartashov, and A. Barski, "Cwl-airflow: a lightweight pipeline manager supporting common workflow language," *bioRxiv*, p. 249243, 2018.

[35] J. A. Novella, P. Emami Khoonsari, S. Herman, D. Whitenack, M. Capuccini, J. Burman, K. Kultima, and O. Spjuth, "Container-based bioinformatics with pachyderm," *Bioinformatics*, vol. 35, no. 5, pp. 839–846, 2018.

[36] P. Di Tommaso, M. Chatzou, E. W. Floden, P. P. Barja, E. Palumbo, and C. Notredame, "Nextflow enables reproducible computational workflows," *Nature biotechnology*, vol. 35, no. 4, p. 316, 2017.

[37] C. Zheng and D. Thain, "Integrating containers into workflows: A case study using makeflow, work queue, and docker," in *Proceedings of the 8th International Workshop on Virtualization Technologies in Distributed Computing*. ACM, 2015, pp. 31–38.

[38] C. Zheng, B. Tovar, and D. Thain, "Deploying high throughput scientific workflows on container schedulers with makeflow and mesos," in *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. IEEE, 2017, pp. 130–139.

[39] P. Amstutz, M. R. Crusoe, N. Tijanić, B. Chapman, J. Chilton, M. Heuer, A. Kartashov, D. Leehr, H. Ménager, M. Nedeljkovich *et al.*, "Common workflow language, v1. 0," 2016.

[40] A. Madhavapeddy, R. Mortier, C. Rotsos, D. Scott, B. Singh, T. Gazagnaire, S. Smith, S. Hand, and J. Crowcroft, "Unikernels: Library operating systems for the cloud," *SIGPLAN Not.*, vol. 48, no. 4, pp. 461–472, Mar. 2013. [Online]. Available: http://doi.acm.org/10.1145/2499368.2451167