Solving the Container Explosion Problem for Distributed High Throughput Computing

Tim Shaffer
University of Notre Dame
tshaffe1@nd.edu

Nicholas Hazekamp
University of Notre Dame
nhazekam@nd.edu

Jakob Blomer
European Laboratory for Particle Physics (CERN)
jblomer@cern.ch

Douglas Thain
University of Notre Dame
dthain@nd.edu

***Abstract*—Container technologies are seeing wider use at advanced computing facilities for managing highly complex applications that must execute at multiple sites. However, in a distributed high throughput computing setting, the unrestricted use of containers can result in the *container explosion problem*. If a new container image is generated for each variation of a job dispatched to a site, shared storage is soon exceeded. On the other hand, if a single large container image is used to meet multiple needs, the size of that container may become a problem for storage and transport. To address this problem, we observe that many containers have an internal structure generated by a structured package manager, and this information could be used to strategically combine and share container images. We develop LANDLORD to exploit this property and evaluate its performance through a combination of simulation studies and empirical measurement of high energy physics applications.**

I. INTRODUCTION

Large scale science depends upon high-throughput computing across multiple facilities in order to meet the demands of simulation and data analysis. Modern high-throughput applications are rarely the single statically-linked executables of the past, but now consist of thousands of software components, including scripts, modules, libraries, and configuration files written in many different languages. In the past, deploying such complex applications at computing sites required either a globally-deployed filesystem (such as CVMFS [1]) or manual effort to install each piece. Today, container technologies such as Docker [2] and Singularity [3] are increasingly used to deploy complex applications at computing facilities, without requiring each site to manually install every software package.

However, the combination of distributed high-throughput computing and container technologies leads to the **container explosion problem**. High-throughput jobs are often generated automatically by submission systems on behalf of multiple users. For each type of job, a container must be generated with the necessary dependencies for a set of jobs. Over time, containers multiply: as a user's work evolves, different jobs need different software, and new containers are generated. Each computing site has a different set of users and projects that may change dynamically with offered load and resource availability. Often, containers are replicated across sites and to many individual nodes. Because related containers share many elements, a significant amount of storage may be wasted due to logical duplication.

To address this problem, we observe that containers often have an internal structure, making use of a standard package manager, such as RPM for standard OS packages, Pip or Conda for Python-related packages, and Modules or Spack for HPC software. Rather than treat each container as a black box of arbitrary files, we can consider it as a *set of packages* drawn from a software repository. From this perspective, all that is needed is a declarative statement of dependencies, and the necessary container can be materialized or destroyed as needed. Further, applications with common needs could potentially be served by shared containers that have the union of dependencies present, thus reducing the total storage needed.

LANDLORD is a system designed to address the container explosion problem for high-throughput computing. It first observes or infers the package dependencies of submitted applications, then generates the execution environment needed by each application. As required, it creates, merges, splits, or deletes container images in order to balance the total storage consumed by containers against the size of individual containers. A significant design consideration lies in the choice of a threshold (α) for when to combine similar images. If α is too high, then too many containers are combined, and the individual images become impractically large. If α is too low, then too many individual container images are created, and the total storage consumption becomes impractically large.

Our prototype of LANDLORD is designed for use with Singularity containers generated by high energy physics applications dependent on large software repositories managed by CVMFS. Using example applications from the ATLAS, CMS, and LHCb experiments, we evaluate the degree of sharing between different applications, and the performance of container analysis and generation. Using trace-driven simulation, we explore different strategies for managing a limited container cache area, varying the α parameter. We observe that, first, our techniques are most effective when the dependency structures are hierarchical, resulting in a compact distribution of common packages. Second, we observe that each configuration has a wide "operational zone" where a moderate value of α (0.65 to 0.95) results in a useful balance between total storage consumed and container size.

1530-2075/20/$31.00 ©2020 IEEE
DOI 10.1109/IPDPS47924.2020.00048

II. Background

A. Containers for Scientific Computing

Modern scientific applications depend upon complex software repositories that consist of terabytes of data and millions of files. Unlike "traditional" high performance applications that are single, statically-linked executables, modern applications are agglomerations of software that consist of high level scripts, multiple executables written in different languages, support libraries that may be dynamically linked, and configuration and data files loaded at runtime. Given the large number of users, software packages, and versions in use even at a single site, management of external software and dependencies has become a difficult problem in its own right.

Users and site administrators are embracing containers as a more flexible method of defining environments for scientific computing, giving users complete freedom to assemble their own software stacks. Containers are designed to be isolated from the host system, so users do not need to consider the configuration of execution nodes. This also helps to reduce the burden on administrators maintaining shared software collections. Container images being (by design) completely self-contained greatly simplifies management and deployment, but at the same time limits opportunities for sharing common components. Much in the same way that a statically linked executable contains a full copy of each library used, container images necessarily include a complete set of dependencies.

Docker [2] first popularized the use of container technologies in industry, but proved to be a difficult fit for scientific computing facilities, because it required the deployment of new privileged services and required the use of node-local storage for running container images. Several competing technologies emerged for use in large facilities, such as Singularity [3], Shifter [4], and CharlieCloud [5]. These technologies make direct use of the underlying operating system facilities to mount container images and run them directly, without an intervening service. As a result, they more easily integrate with the shared parallel filesystems in use at large facilities.

Containers are typically defined by a *recipe* which is used to construct the image programatically. For example, a Dockerfile describes a sequence of Unix commands, networking changes, and package installs that are used to generate a layer of a Docker image. However, it is important to note that a recipe describes a *sequence* of changes that would have a different effect if the order was changed. Each addition to the sequence of events adds a new *layer* which increases the size of the container, even if the event removed data.

An alternate way of specifying a container is to state a *set of dependencies* that must be present within the container. Binder [6], for instance, uses declarative requirement files in a Python code repository to generate a container for the application. Unlike a recipe, a set of dependencies has no order, and so one may combine or break apart sets without starting over. We use this property to significant effect in the design of LANDLORD.

B. Distributed HTC

High Throughput Computing (HTC) is an operating regime where the user's objective is to complete the greatest amount of work possible over a long period of time. HTC applications are often scientific simulations or data analysis applications that must be run on a large number of independent configurations or input files. HTC typically involves a pool of work that is much larger than the available computing resources. HTC applications are thus well-poised to take advantage of multiple execution sites (such as the Open Science Grid (OSG) or the Extreme Science and Engineering Discovery Environment (XSEDE)) in order to increase the total application throughput. Some applications also choose their execution environments strategically, e.g. running most jobs on campus or grid resources, but submitting certain high-priority/critical jobs to a public cloud at additional cost.

In this work, we will consider the Large Hadron Collider (LHC) experiments as an example of distributed HTC applications. The LHC experiments at CERN require a tremendous amount of computational power to simulate high energy physics (HEP) processes, process the torrent of data produced by the detectors (88 petabytes of new data in 2018 [7]). Modelling and statistical analysis jobs rely on a large number of derived datasets, resulting in a global data volume of approximately 1 EB [8]. At present, most of this computing power is provided by the Worldwide LHC Computing Grid (WLCG), which consists of more than 170 computing centres in 42 countries. During normal operation, the WLCG carries out over 400k CPU-hours of computation and transfers more than 80 PB of data per month. Over the course of a year this adds up to around 700 million compute jobs or 5 billion CPU-hours/year [9].

HPC resources are an appealing source of computing power to supplement the WLCG in meeting the computational demands following the LHC's high-luminosity upgrade. Unfortunately, HPC sites often impose restrictions on network activity and system configuration, preventing WLCG jobs from running directly on HPC resources. Containers offer a potential solution for importing software environments as a guest user at a site. WLCG jobs, however, are external to the execution site and must be automatically imported and managed.

The simplest way to handle dependencies for external jobs is to prepare local images as part of job submission. This step must be automatic, as it can be difficult or impossible to tailor distributed jobs to each execution site. With WLCG jobs, for example, each experiment's full software repository is assumed to be available via a specialized transfer mechanism (CMVFS [1], the CernVM File System). Researchers do not design WLCG applications with container support in mind, so any preparation for a particular site must be transparent to the application. Container creation can also become expensive relative to job execution, so it would be ill advised to construct a complete container image for every job. Management systems such as Docker [2] and Shifter [4] reflect this reality in using persistent image stores.

III. The Container Explosion Problem

We define the **container explosion problem** in HTC as follows: given a large (and probably growing) number of jobs that require many overlapping software dependencies, simply creating a container to fulfill each job's requirements will lead to a combinatorial explosion in the number of images stored.

In industry, this proliferation of container images to the point of management difficulty is referred to as "container sprawl", just like the related phenomenon for VMs called "image sprawl" [10]. For larger multi-container applications, it is not feasible to manually manage all component containers and ensure stability and compatibility between all versions. Container orchestration systems such as Kubernetes [11] allow users to declaratively define the high-level services/components of applications, while the orchestration layer manages the concrete resources (persistent storage, container instances, etc.) in the public cloud. To aid in managing software environments in containers, Kubernetes package managers such as Helm [12] can instantiate specific versions of each software component and clean up outdated containers.

In HTC settings, however, there is a key difference: applications are expressed as a stream of discrete *jobs* rather than as *services* that can be provisioned and then cleanly torn down. This means that there is not a clearly-defined lifetime for an application and the containers it requires; jobs from older applications may be run again at any point in the future, and individual jobs from many different versions of an application may run concurrently.

As mentioned previously, container images do not allow for sharing components as is possible with local installations, site-wide modules, or copy-on-write filesystems. Instead, each container carries complete copies of all components. In the naïve case, each variation in job requirements results in the creation of a whole new container. In this scenario many identical copies of common base components and dependencies are stored across a set of prepared container images. Since each job-specific variation exists as a completely separate container, caching does little to alleviate this duplication. Only jobs with identical requirements can reuse existing containers. In the case of the LHC, each experiment maintains different software environments and individual jobs vary frequently in the components and versions they use. In this situation, each variation and permutation of job requirements creates a completely distinct container image, leading to very poor cache utilization (Section VI quantifies this effect). Our goal then is to maximize the throughput of jobs that can be run using some fixed amount of cache space for container images.

We can observe the container explosion problem by examining the initial use of containers by the LHC experiments, which as global-scale computational projects provide insight into the behavior and challenges of new approaches in scientific computing. CVMFS recently added experimental support for efficiently serving the contents of container images [13]. The ATLAS and CMS experiments uploaded several dozen software environments as part of their evaluations of container

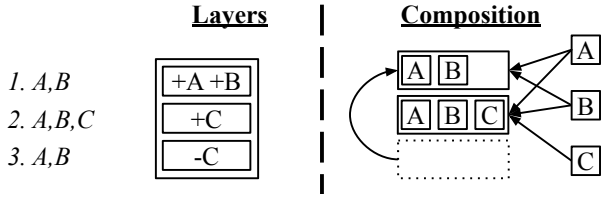


Fig. 1: Refining via layers vs. Composition.

technologies. At present CVMFS stores nearly 150 versions of these base environments, with around 1,000 distinct layers. CVMFS retains all historical versions to ensure reproducibility and backwards compatibility, making simple garbage collection impossible. Since transferring the entire container repository for every job quickly becomes prohibitively expensive, it is necessary to create tailored images based on a required subset of the full software repo. There are a number of potential approaches to work around the combinatorial explosion in job-specific images, but none are satisfactory.

Imperfect Solution: Full-repo Images. Rather considering the precise requirements for each job, the simplest way to reduce the number of containers in use is to place an *entire* software repository into a single image, which can then support a large number of jobs. In the case of the LHC experiments, each full CVMFS repo is measured in TB (see Figure 2 for exact sizes). Likewise, a complete copy of the Python Package Index (PyPI) would be roughly 190,000 packages consuming over 6 TB (at the time of writing). Unfortunately, this approach is likely to exceed a number of practical limits on container size. Individual worker nodes may have limited local disk space and be unable to store large container images. Even if the large container fits, it is likely that a given job does not need *all* of the repository simultaneously, so it is wasteful to transfer unneeded data. This concept is a driving influence on projects like Slacker [14].

It also becomes prohibitively expensive to update and transfer such large container images. The US collaboration of the ATLAS, ALICE, and CMS projects are currently experimenting with CVMFS applications on computing resources at various supercomputers in the United States including Cori at NERSC [15]. When full-repo images were built and scaled out onto a large number of nodes inside the NERSC infrastructure, the process took around 24 hours, making it difficult to deploy up-to-date versions of the software on a regular basis. In addition, the process requires administrative involvement in image creation, deployment, and cleanup. As additional projects want to take advantage of the resources at NERSC, the administrative burden of managing multiple CVMFS images on multiple software versions increases accordingly. Taking this approach negates the flexibility and hands-off administration that containers were intended to provide.

Imperfect Solution: Layering. Docker allows container environments to be composed from reusable image layers. Docker can take advantage of modern filesystems like BTRFS [16] that provide efficient snapshots and transparent

sharing of files and directories between different revisions. As a practical matter, Docker is generally not available in HPC environments for administrative and security reasons. Likewise, guest users at large sites do not generally have the ability to directly manipulate file system snapshots or export/load local filesystem volumes. More conceptually, layering images addresses a different problem than the issue at hand. With Docker, base images can be extended and refined over time by appending layers. When preparing to run external HTC jobs, however, we must compose a set of largely independent pieces to fit specific job requirements, without any particular ordering relationship to previous images. It is therefore difficult to map this set of semi-independent pieces into a linear sequence of refinements that will fit future job requests.

Figure 1 compares layering and composition after processing several jobs. Although item C is hidden in the lower layer, it still exists in a previous layer and must be transferred and stored. Since changes to layered images are strictly additive, old content can be masked but not removed. Also note that the first and third jobs in Figure 1 have identical requirements and the corresponding layers are functionally equivalent, yet this is not apparent to Docker based on their contents. With composition, on the other hand, it is immediately clear when images are equivalent and can be reused.

Imperfect Solution: Block Deduplication. Another potential avenue to deal with the container explosion problem is data deduplication for disk images. The virtualization community has developed a number of solutions for efficiently deduplicating disk images [17] and running virtual machines with many incremental changes [18]. There has also been extensive research on deduplication [19], [20] of filesystem data [21], [22] and disk blocks [23], [24]. It is not difficult to identify duplicated files or blocks within container images. However, we lack a means to combine the extraneous copies; each container image by design contains complete copies of all data, and sharing of data across images is not possible for users of the system. When supporting multiple users with a potentially large number of container images, utilization of site-wide storage becomes a concern. Simply adding storage capacity to accommodate each user or application is not a sustainable solution over the life of a system [25]. Rather, we would prefer to make better use of what site storage is available by reducing unnecessary duplication among container images.

IV. Key Insight: Specification-level Management

Having discussed the methods for defining container images, we arrive at a key insight of this work: **container specifications offer more opportunities for management and optimization than containers themselves**. While a build script gives a sequence of steps to produce a final container image, it does not give information about the desired properties of the resulting image. Likewise when examining a previously built container, it is difficult to determine how it can be used (whether it provides a particular version of a package), since recipes allow for arbitrary modifications to an image. It is not generally safe to substitute one image with another, even if the two have some subset of their contents in common. Conversely, two container images may be functionally identical despite having different contents if the build process is not strictly deterministic. Note that almost all build systems will produce variations in timestamps, logs, configuration files, etc. that make direct comparison of images difficult.

	Running Time	Prep. Time	Minimal Image	Full Repo
alice-gen-sim	131s	59s	6.0G	450GB
atlas-gen	600s	37s	2.7G	4.8TB
atlas-sim	5340s	115s	7.6G	4.8TB
cms-digi	629s	62s	8.4G	8.8TB
cms-gen-sim	2360s	71s	6.1G	8.8TB
cms-reco	961s	78s	7.3G	8.8TB
lhcb-gen-sim	1010s	67s	3.7G	1.0TB

Fig. 2: Benchmark applications for LHC experiments [26].

Rather than trying to recover information from previously built images, the specifications used to construct them offer higher-level information about their functional characteristics. These specifications give minimal requirements that an image must fulfill without specifying anything about the exact image contents. If a specification requires a subset of packages in a previously built image, we should be able to use the latter to satisfy the former specification; the concrete image meets the specified requirements and includes some additional (unrequested) packages. Using this approach, we could perform more sophisticated image caching based on container capabilities rather than strictly comparing file contents. Using higher-level information from specifications, the system has additional flexibility in how it fulfils specifications that allows for reduced storage in a very narrow case (strict subset).

Specifications afford another opportunity to a container management system: unlike build scripts or recipes, it is possible to automatically manipulate or combine specifications. A composite specification can be formed as the union of requirements from two or more specifications. This kind of composite image could be used in place of any of its constituent specifications, since it meets the minimum requirements given in each. Note that in some cases, incompatibilities among requirements make combination impossible.

While caching and merging specifications gives a mechanism to reduce duplication among stored container images, we still do not know which specifications to merge. Choosing randomly or by order of job submission, for example, is liable to join specifications with little in common. This would increase the sizes of images to be transferred among worker nodes, while doing little for de-duplication. Instead, we want to merge specifications with many common components. We now introduce a simple metric for similarity between specifications and an algorithm for automatically managing an image store, with a tunable parameter controlling how aggressively to reduce duplication and increase storage utilization.

V. Online Management

When working on campus computing resources or cloud environments that support software-defined infrastructure, it might be possible to make system-level adjustments to mount global filesystems or otherwise meet the needs of a particular user or application. In national-scale HPC settings or across a large number of sites, however, we would prefer to leverage the capabilities available to guest users of computing sites to mitigate the container explosion problem. Rather than employing techniques like block deduplication in reaction to existing images, we propose proactively managing the collection of container images before they are created.

We therefore define LANDLORD, a system for making a decision online to efficiently satisfy the dependency requirements for submitted jobs. This method must be suitable for inclusion either in individual researchers' workflows or as part of the setup for batch job or pilot job scheduling, and must be efficient in terms of both computation time and storage space. We suppose that some local storage is available (e.g. a few terabytes of scratch space attached to a head node) for caching exported repository contents and a cache of generated container images. We also suppose that each compute node has scratch space available for storing container images locally, but that the total repository contents or the collection of all container images may be too large to store on every worker node. We would thus like to make good use of the available storage, acknowledging that there must be *some* repeated work and waste. For each user-submitted job LANDLORD must automatically prepare a suitable container image, and (if possible) avoid creating a new one for each job request.

LANDLORD Deployment. We designed LANDLORD to operate as an automated step during job submission. Since this work supposes that scientists may need to run jobs across many sites and will need to work with the privileges of a guest user, our prototype of LANDLORD is implemented as a lightweight job wrapper. To use LANDLORD when submitting a job, a scientist must first prepare an image specification for each job. Simple specifications may be hand-written; we also developed several simple analysis tools to automatically generate specifications by scanning for Python `import` statements, `module load` directives, or logs from previous jobs. Researchers would also set up their particular submission systems to wrap invoked jobs. Then on job submission, LANDLORD first scans its configured cache directory for existing images that are "close" to the job's specification, creates/updates images in the cache as necessary, and finally launches the job inside the prepared container.

While our user-level prototype is a good fit for a single unprivileged user, administrators may wish to employ LANDLORD for site-wide container management. The same core functionality of LANDLORD could easily be adapted into a plugin for a site's batch system. For the LHC experiments, CVMFS data is normally public and shareable, making a LANDLORD plugin particularly simple to implement. A more general-purpose plugin would need to take into account data security and privacy, which we leave as future research. In addition to batch systems, there are other situations where this plugin approach is applicable. When using a pilot job system, for example, scientists are effectively operating a "user-level scheduler". Scientists have the option of using this same plugin approach to connect LANDLORD to a pilot job system, allowing LANDLORD to transparently optimize container storage without requiring application changes.

Similarity Metric. In order to make decisions about how to optimize the storage for a collection of container images, we need a means to quickly identify containers that are "close" (for some definition of close) as candidates for optimization. Rather than examining the containers themselves, we will compare the specifications used to generate them. We are less concerned here with the particular choice of metric than with choosing a simple, adequate, and non-controversial metric. We chose the Jaccard distance under appropriate choice of set elements as it gives a straightforward way to group sets with similar contents and is very well used and studied. When working with software repositories, each package is usually assigned a name/version string that is defined to be unique within the repo. We can thus use the Jaccard distance to determine the similarity between sets of package requirements.

For two sets A and B, the Jaccard distance d_j is:

$$d_j(A,B) = 1 - \frac{|A \cap B|}{|A \cup B|} = \frac{|A \cup B| - |A \cap B|}{|A \cup B|}$$

The Jaccard distance captures several desirable properties when dealing with specifications. In the case of two specifications that differ only by one element, the Jaccard distance will be small. For a pair of specifications with nothing in common, the distance will be high. In addition, a constant-time approximation of the Jaccard metric (MinHash [27]) is available for making an efficient first pass at selecting similar images when the number of packages or components is large. This approximation is important in practice due to the sizes of the data involved; metadata listings alone for full-repository CVMFS images consumed multiple gigabytes of storage, so it is desirable to robustly support very large specifications.

A limitation of using the Jaccard distance this way is that it does not capture conflicts between components. Public software repositories generally support explicit version constraints, so two specifications may include constraints that cannot be simultaneously satisfied. This compatibility checking is dependent upon the specific package manager or system in use. For LHC applications this is a non-issue, since CVMFS is normally append-only and all previous versions remain available. If conflicts between specifications are possible, this additional checking step can be performed after using the Jaccard distance to prioritize the set of candidate specifications.

The fundamental operation for LANDLORD's storage optimization strategy is merging container specifications that are "close enough". Using the Jaccard distance metric, we can quickly identify cached specifications that are similar to a new request. To decide if two specifications are "close enough" to optimize, we define the parameter α as the maximal Jaccard

distance between closely related specifications. Since Jaccard distance is by definition between zero and one, α must be in the same range. This α parameter is something like the "globbiness" of the system. Using this α parameter, we can define a very simple caching algorithm for managing and optimizing a central image store.

Given: Cached container image collection I
Input: Container specification s
Result: Suitable container image satisfying specification s

```
for i ∈ I do
    if s ⊆ i then
        // An existing image satisfies s
        return i;
    end
end
for j ∈ I such that d_j(s, j) < α do
    // Selection can be sorted by d_j()
    // Attempt to merge
    if s and j do not conflict then
        Replace j in the cache with merge(s, j);
        return j;
    end
end
// Couldn't re-use or merge
Insert new image s in the cache;
return s;
```

Algorithm 1: Container cache management

Choosing α near zero requires that specifications are extremely similar before considering them for merging. In the extreme case with $\alpha = 0$, only *identical* images will be considered close, so no images will be merged. This results in a larger number of independent images. Choosing α to be larger makes it more likely for images to be considered similar and merged. This results in more augmented images that serve multiple tasks. In the extreme case of $\alpha = 1$, *every* pair of images is considered close and merged if possible. This results in large container images that have accumulated many specifications. Using the α parameter, it is possible to continuously vary between these two extreme behaviors.

A potential issue with this automatic merging strategy is "bloated" images that accumulate infrequently used dependencies and increase overhead indefinitely for future tasks. The Jaccard distance gives a natural way to capture and address this effect. As an image becomes bloated due to repeated merges, its distance from any individual request increases. After sufficient growth, the image will become too far from any request to even be considered. Without regular use, the bloated image will eventually be evicted from the cache. Choice of α therefore places an upper limit on the amount of undesirable bloat in images. Later, we examine the effect of the α parameter by simulating image management over a large number of application requests.

VI. Case Study: LHC Experiments

We now explore the container explosion problem and LANDLORD more thoroughly in the context of the LHC experiments at CERN. Figure 2 shows several LHC benchmark applications run under Shrinkwrap, a tool developed as part of this work for efficiently building container images from CVMFS. Here `gen` (generation), `sim` (simulation), `digi` (digitization), and `reco` (reconstruction) are phases of the experiment pipelines. Running Time gives the average run time for a single instance of the application. These benchmark applications were run using local container images, where preparation time is the amount of time required to create such an image by downloading the contents via Shrinkwrap and compressing the resulting data into an image file. Note that this time does not include any tracing or dependency analysis. If static analysis or manually-defined specifications are not available, runtime tracing (possibly over multiple runs to try to capture all behaviors) becomes necessary, significantly increasing the preparation time. Minimal Image indicates the size of the tailored container image.

Characterizing Package Dependencies. Our first step in evaluating a storage optimization strategy for a global software repository is to characterize its contents, which will inform our simulations later. In the SFT CVMFS repository, where many of the core research packages at CERN are hosted, dependency information is included in build metadata associated with each package. This metadata gives a convenient way to construct a dependency tree of the entire repository. We constructed such a dependency tree for the SFT repository to use in the simulations discussed later. Figure 3 gives an indication of the structure and sizes of software components in the repository. For each fixed specification size (on the x axis), we selected a random sample of packages. The lower line shows the storage size of only that selection. Recursively including the package dependencies is shown in the upper two curves. We repeated this procedure 100 times for each specification size, taking the median. The on-disk size of the selections appears to increase proportionally to selection size. Recursively including the dependencies of these same selections, however, results in a significant increase in number of packages and size on disk. For small selections (less than 100 packages), the complete images with dependencies might contain 5x as many packages as requested, with storage increasing proportionally. With larger selections, however, this increase becomes less pronounced. This non-linear change is the result of the tree structure of the software dependencies. There are a number of core components that are transitive dependencies of a large number of packages. These common dependencies are only included once, so subsequent selections that depend on them do not add them again. As the number of selections increases, this curve will approach the total repository size.

Simulating HTC Jobs. We tested two schemes for generating image requests for HTC jobs, taking into account the properties of the software collections and generating images via a uniform random scheme for comparison. We consulted

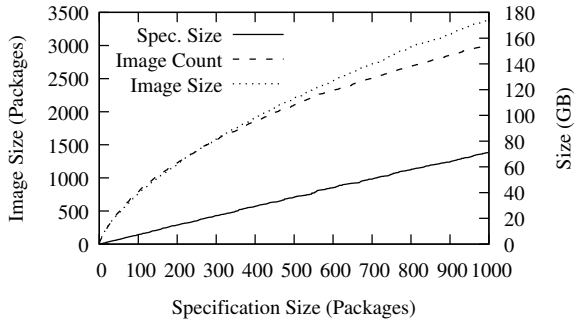


Fig. 3: Image size vs. selection size

with the developers of CVMFS, as well as HEP researchers at our university collaborating with CERN to determine how current users interact with CVMFS, with the benchmarks in Figure 2 as examples. We expect significant variability in files accessed and total size among different users and experiments. We nonetheless observed that certain core components are used near-universally. While multiple versions and variations might exist, these components have a very high likelihood of appearing in every container image. These components correspond to the base frameworks, setup scripts, calibration data, etc. needed for most jobs. For the purposes of simulation, we would like to see these components in a large proportion of jobs across all simulated jobs from different users and experiments. There is also a large set of components that must be available and are used in some applications, but which are very rarely used overall. It is important to make these "long tail" components available for researchers that need to use them, but it would be wasteful to include them universally when they are rarely used.

For the purpose of simulation we assume that the requirements of a job are given as a set of packages. While the Shrinkwrap utility can operate at the granularity of individual files, allowing for partial packages tends to produce unreliable container images. The previously-described dependency tree extracted from the SFT repository consists of 9660 packages, where a program or library typically provides packages for multiple versions, platforms, and configurations. When building a simulated image, we recursively include dependencies of requested software. This approximates the structures of actual container images, while still allowing for variation in package requests. For each simulated request, we chose a random selection of packages and then added the closure of the package dependencies. This image simulation scheme captures the structure inherent in the software collection, in that packages in addition to those requested are automatically included so as to ensure a functional image. The initial selection of packages, however, is simply uniformly random.

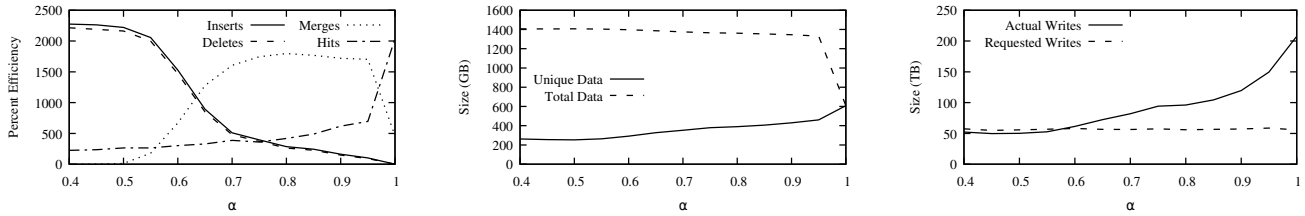
To generate an image for a simulated job request, we randomly made an initial selection of up to 100 packages. We then used one of the two schemes (dependency tree-based or random) to expand the initial selection into a full image. Repeating this procedure, we can create streams of container specifications for simulated jobs.

Simulated Container Caching. Figure 5 shows a single simulation of LANDLORD with $\alpha = 0.75$ and cache size of 1.4 TB processing 500 unique job specifications, each one repeated five times. First, we note that most of the operations are merges. This is to be expected, as this simulation ran with a fairly high α value. The total bytes written also closely tracks merges, indicating that merging is the dominant source of I/O. We still see inserts over the course of the simulation. At more extreme α values, we expect to see one of these operations almost exclusively. As the data in cache continues to rise, we eventually see the cache limit, after which the delete count increases. Over the course of the simulation, inserts and deletes are filling and emptying the cache such that it remains close to its storage limit. We also observe the number of cache hits continue to rise despite deletions. As we will see, merging allows for a greater proportion of hits even if the amount of data remains constant. This is because frequently used data can be merged, reducing duplication. The cache limit then ensures that infrequently used parts are eventually removed.

To evaluate the viability of our approach in automatically optimizing a container image store, we simulated the behavior of the system over time for a range of α values. Choice of α is important here to ensure that common components can be detected and merged in container images, and that old images or poor decisions are eventually removed from the system. Choosing α too far in either direction will result in excessive storage overhead due to duplicated components, or excessive compute and network overhead repeatedly merging largely unrelated images. Our goal in this evaluation is therefore to choose α so as to minimize the storage and compute costs associated with maintaining a collection of images.

Since our simulation uses random simulated requests, there is noticeable variability between individual simulations. Thus for a given choice of cache size, job count, etc. we repeated the simulation 20 times and reported the median behavior over the runs. At each choice of α (in steps of 0.05) we performed a set of 20 simulated runs, allowing us to plot various measurements of the system versus α.

Sweeping over the range of α values in this way, we can immediately see differences in the frequency of simulated operations. Figure 4a shows the upper range of α values where behavior differences appear. From the lower values on the left, the insert and delete counts are the primary (or only) operations, with number of hits relatively constant. This corresponds to a simple LRU-based cache. The insert count is higher due to cache filling, but the two tend to increase in lockstep. As α increases, image merges become more frequent. The merge count steadily increases throughout most of the upper range, while inserts and deletes decrease. This suggests that at high α values, the cache space would be more efficiently used, with some of the duplication merged out. In the extreme case with $\alpha = 1$, every request is merged into a single image in cache, hence the sudden increase in hit rate and decrease in total merges at the far right of Figure 4a. As merging increases,



(a) Total Cache Operations (b) Duplication of Data in Cache (c) Cumulative I/O Overhead

Fig. 4: Cache behavior over a range of α values

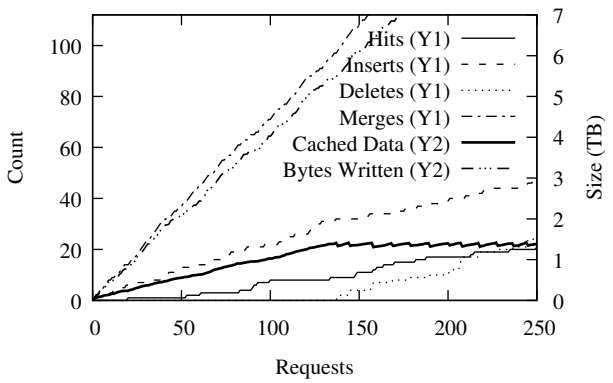


Fig. 5: Behavior of a single simulation

the proportion of hits generally also increases. This indicates more of the requests can be satisfied by previously merged images in the cache. From the perspective of the system, this is beneficial as there is no extra time or compute cost for handling a larger proportion of images.

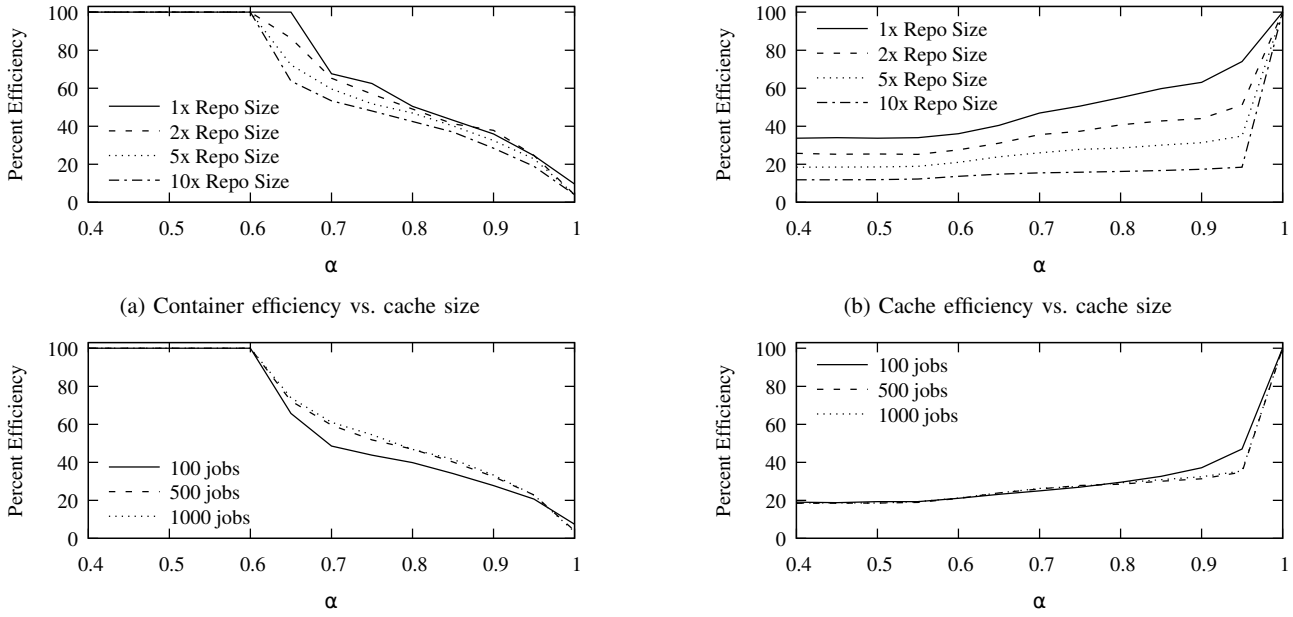
Metrics for Cache Utilization When sweeping over the range of α values, there are a number of metrics available to summarize each simulation run. Many, however, are highly coupled with the particular system configuration and difficult to compare as we vary the parameters of the simulations. We therefore chose to define two metrics, cache efficiency and container efficiency, to indicate the relative utilization of the system. We defined cache efficiency as the ratio of unique data to total data in the cache. In our case, this is equivalent to the ratio of the size of the unique packages to the total cache size. If many images contain copies of the same packages, the cache efficiency decreases. This metric captures duplication within the cache across all images. With no merging there is a high degree of duplication, so the cache efficiency is low. On the other end of the spectrum, maintaining a single, large image containing all data results in cache efficiency of 100%, as the entire cache resides in that single image.

We defined container efficiency as the ratio of the size of the requested container (a set of requested packages plus all dependencies) to the size of the container the system actually used for the job. In the absence of merging, these two are equal so the container efficiency is 100%; jobs are run with exactly what was requested. By merging to allow for image reuse, we include additional, unrequested data in container images. The container efficiency measures this difference between requests and containers. In the extreme case of $\alpha = 1$ with a single large image, for example, the container efficiency is poor because the entire repository is used for every request, regardless of size. These two extreme cases, no merging among many images and a single merged image, can both be useful in some situations. Rather than defining where these limits fall, we discuss choosing limits in Section VI.

Figure 4b shows the actual data sizes used to calculate the cache efficiency. Without merging, the unique data makes up a small proportion of the cache. With increased merging, the amount of unique data increases. For sufficiently high α, merges occur more frequently than cache inserts, resulting a drop in total storage size. On the far right of Figure 4b, the entire cache contents are merged into a single large image, so that the unique and total data sizes become equal. The cache efficiency metric is affected both by the increase in unique data and by the decrease in total cache size.

Overhead of LANDLORD. Under LANDLORD's approach, we use compute and I/O capacity during job submission in order to improve utilization of storage space. At some point, however, this additional I/O cost could become prohibitively expensive. To quantify this computational and I/O overhead, we can measure the cumulative amount of data written over the course of simulated cache operation. We observed that almost all of the additional overhead/latency of LANDLORD was due to disk I/O in creating merged images, while LANDLORD spent very little time performing computation. We thus use cumulative write size as a metric for overhead/latency that is independent of specific hardware or disk performance. Figure 4c shows the amount of data written during simulations over a range of α values. "Requested Writes" gives the total amount of data actually requested by each job over the course of the simulations. This value is on average constant since the same procedure was used to generate all simulated job requirements. "Actual Writes" gives the total amount of data written to cache over the course of simulations. If, for example, an image were evicted and then re-inserted later in a simulation, then the cost of generating and writing the image would be added again.

Without merging, the actual I/O and compute costs in the system closely follow the requests. At low α the actual I/O is slightly smaller than the requested amount due to caching:



(a) Container efficiency vs. cache size

(b) Cache efficiency vs. cache size

(c) Container efficiency vs. unique job count

(d) Cache efficiency vs. unique job count

Fig. 6: Effects of Simulation Parameters on System Efficiency

the system can sometimes reuse images without performing any additional I/O. As α increases, the cost of updating and merging images comes to dominate the total I/O cost. Each time a merge occurs, the resulting image must be written out *in its entirety*. Thus when merges are frequent at high α, some data will be written and re-written many times to satisfy new job requests. Thus while extremely high α makes better use of available storage space, LANDLORD introduces a significant amount of latency/overhead in the form of additional I/O operations. Choice of α thus gives administrators a way to balance between storage utilization and I/O overhead using LANDLORD, e.g. allowing at most a twofold increase in the compute and I/O time compared to directly creating the requested images. (This would correspond to the upper compute limit shown on Figure 8, discussed later in Section VI.)

Sensitivity Analysis. In Figure 6, we plot efficiency curves for a range of simulation conditions. The left column shows container efficiency, while the right column shows cache efficiency. In the first row, the number of jobs and the amount of repetition are constant while the cache size is varied. In the second row, the number of unique requests is varied with the other parameters constant.

The size of the cache has an inverse relationship with both the container and cache efficiency. As seen in Figure 6a and Figure 6b, a larger cache can of course hold a larger number of images, but since each image contains significant duplicated portions, increasing cache size tends to decrease cache efficiency. Conversely, small caches more quickly evict images so that ineffective merges and similar images tend not to remain in cache too long. A larger cache also allows for more opportunities to merge images, leading to decreased container efficiency. When deciding how to handle a request, a large cache full of images is much more likely to contain an image suitable for merging. With a small cache, opportunities to merge are much more dependent on the order of requests.

The effect of varying the number of unique jobs is less pronounced than the effect of cache size. As seen in Figure 6c and Figure 6d, streams of 500 and 1000 unique jobs show nearly indistinguishable behavior, indicating that by 500 jobs the system has reached a steady state. Continuing with an arbitrarily long stream should not result in significant performance changes. However, 100 unique jobs were not sufficient to fill the cache and reach a steady state. In this case the container efficiency is slightly decreased over α, suggesting that some ineffective merges had not made their way out of the cache. Cache efficiency in this case is slightly increased. This would suggest that before reaching a steady state, the cache contents are more assorted and some unnecessary data remains cached.

Effects of Package Dependencies. To evaluate the effects of container contents, we also generated simulated images consisting of packages chosen in a uniform random way. To ensure that total size (or at least total number of packages) is comparable to images generated by the previous method, for this approach we started with an image request generated via the previous scheme (uniform random core selection with dependencies added). We considered only the total number of software packages in the resulting image, and then chose the same number of packages uniformly randomly from the entire repository, ignoring usage information and package dependencies. While images generated in this way are not particularly

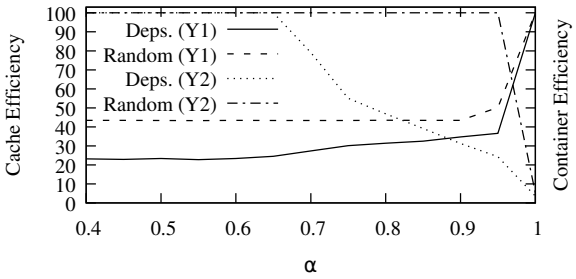


Fig. 7: Impact of dependencies on duplication

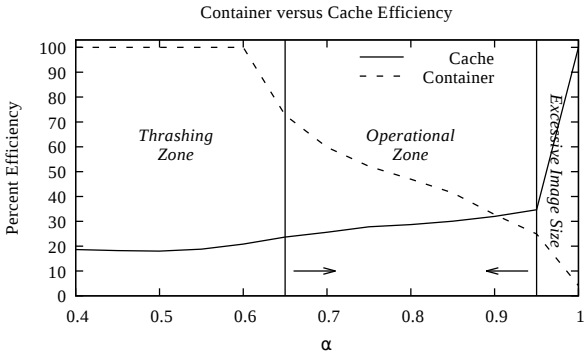


Fig. 8: Limits on efficiency

realistic, they are useful for evaluating image management schemes. By comparing results with random images to those with the previous image generation scheme, we can compare the general case of containers as collections of arbitrary data to the specific focus of this work, i.e. containers with selections of experiment software with dependency relationships.

Figure 7 shows a representative simulation with both synthetic image types included. In the purely random case, there is no correlation between different images. Thus, it is much more difficult to find images similar enough to merge until the α value is very lax. This would indicate that our merging strategy is not applicable to arbitrary collections of data. Random images show little to no effect for most α values. Our merging strategy, which takes advantage of duplicated content included as a result of dependencies in software, would be ill advised for situations that are not known to follow similar patterns of duplication. Even with non-realistic job requests, the tree structure of package dependencies is sufficient to produce pronounced duplication in the cache, which gives an opportunity to apply our storage optimization strategy.

Limits on Cache Utilization. For any choice of container management scheme, there is *some* non-trivial management cost. This could be in the form of time and manual effort on the part of individual users, or a portion of the system's compute and storage space used in the background. In the case of an extremely well-provisioned system, the best solution might simply be to retain *everything*. In terms of our simulated management scheme this corresponds to the extreme cases, holding a large number of single-purpose images at $\alpha = 0$ or building a single all-purpose image at $\alpha = 1$. In the case of high-throughput computing, we expect the total size of applications and data to be much larger than any individual job or even larger than the capacity of the system. In this setting, it becomes necessary to balance the management cost against system constraints on compute and storage. Under the simple management algorithm presented here, choosing an intermediate α value gives researchers or administrators a tunable way to trade some computing power and IO activity for improved storage utilization.

If the requirements of all jobs are known in advance and worker storage and bandwidth are sufficient, maintaining a single, large image for all jobs on all workers is an useful strategy. After the initial (costly) image creation and transfer, there are no continuing costs or delays. When resources are limited or requirements change regularly, however, this approach becomes prohibitively expensive. The high compute and bandwidth overhead apply for every image update, which in the worst case could be every job.

The other extreme situation, simply caching requests with no merging, can also be viable. This approach is simpler and does not come with extreme per-job overhead. In addition, frequent changes to the jobs and requirements do not degrade the efficiency of each job. At large scale, however, the overall system efficiency suffers. Due to duplication among images, larger caches contain less and less unique data. To support a given repository, it becomes necessary to provision a cache much larger than the size of the repository. With CVMFS repositories consuming several terabytes of storage each, the amount of cache space required grows quickly.

These two limiting factors, the compute and transfer cost in the highly merged case and the cache efficiency in the unmerged case, serve as limits on the viable range of α values for a system and its users/applications. Figure 8 shows two vertical lines indicating these limits on a plot of efficiencies. The left line shows a lower limit of around 30% on the cache efficiency in this simulated configuration. Choosing α too low results in cache efficiency reaching below 20%. On the right, the line gives an upper limit on the likelihood of merging. As shown in Figure 4a, the amount of I/O and compute to update images becomes much larger if α is set too high. These two limits define a range of viable values for α. There is no general rule for the placement of these limits, which depends strongly on the performance characteristics of the execution environment, as well as the priorities of the administrators in optimizing the system.

Tuning LANDLORD. Using a simulated workload based on applications from HEP researchers, we instead aimed to illustrate the viability of a straightforward approach to automatic storage optimization when managing containers from multiple users. Considerations at each site such as the amount of scratch storage available for caching container images and upper bounds on the computational cost to prepare each container ultimately dictate the viability of any particular approach.

LANDLORD provides a good deal of flexibility to match the properties of a given execution site and workload(s).

In our simulations, we found that the choice of α was not particularly important, as long as it falls within a wide "operational zone" (0.65 to 0.95). Figure 8 shows that choosing extreme values of α results in a large number of overlapping container images or excessive overhead creating and updating massive images. These extremes correspond to the naïve approaches discussed previously, i.e. many single-use containers or a single all-purpose container, respectively. Choosing α anywhere within the operational zone strikes a reasonable balance between storage utilization and overhead. A new application employing LANDLORD should choose a moderate α (e.g. 0.8) to start, with finer tuning possible to meet specific application or site requirements. A moderate choice of α allows LANDLORD to avoid extremely poor behavior in either direction, without attempting to attain "optimal" performance. LANDLORD thus offers a lightweight mechanism to avoid cases of pathologically poor performance.

VII. CONCLUSION

To address the container explosion problem for HTC applications, we developed LANDLORD, a prototype system that generates an execution environment for containerized jobs, exploiting the hierarchical package dependency structures to better manage a limited cache area. LANDLORD is tunable to meet application and site-specific requirements, but based on trace-driven simulation of LHC applications we observed a wide operational zone that achieves reliably acceptable results. We developed tools for integrating LANDLORD into HEP applications based on CVMFS, but the underlying concepts are easily applicable to other systems. As container-based applications and multi-site computing continue to increase in prevalence, employing higher-level knowledge such as package specifications will be critical in automatically and intelligently managing available resources.

REFERENCES

[1] J. Blomer, P. Buncic, R. Meusel, G. Ganis, I. Sfiligoi, and D. Thain, "The Evolution of Global Scale Filesystems for Scientific Software Distribution," *IEEE/AIP Computing in Science and Engineering*, vol. 17, no. 6, pp. 61–71, 2015.

[2] D. Merkel, "Docker: Lightweight linux containers for consistent development and deployment," *Linux J.*, vol. 2014, no. 239, Mar. 2014. [Online]. Available: http://dl.acm.org/citation.cfm?id=2600239.2600241

[3] G. M. Kurtzer, V. Sochat, and M. W. Bauer, "Singularity: Scientific containers for mobility of compute," *PLOS ONE*, vol. 12, no. 5, pp. 1–20, 05 2017. [Online]. Available: https://doi.org/10.1371/journal.pone.0177459

[4] D. M. Jacobsen and R. S. Canon, "Contain this, unleashing docker for hpc," *Proceedings of the Cray User Group*, 2015.

[5] R. Priedhorsky and T. Randles, "Charliecloud: Unprivileged containers for user-defined software stacks in hpc," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '17. New York, NY, USA: ACM, 2017, pp. 36:1–36:10. [Online]. Available: http://doi.acm.org/10.1145/3126908.3126925

[6] Project Jupyter, Matthias Bussonnier, Jessica Forde, Jeremy Freeman, Brian Granger, Tim Head, Chris Holdgraf, Kyle Kelley, Gladys Nalvarte, Andrew Osheroff, M. Pacer, Yuvi Panda, Fernando Perez, Benjamin Ragan Kelley, and Carol Willing, "Binder 2.0 - Reproducible, interactive, sharable environments for science at scale," in *Proceedings of the 17th Python in Science Conference*, Fatih Akici, David Lippa, Dillon Niederhut, and M. Pacer, Eds., 2018, pp. 113 – 120.

[7] https://information-technology.web.cern.ch/sites/information-technology.web.cern.ch/files/CERNDataCentre_KeyInformation_December2019V1.pdf.

[8] The HEP Software Foundation, J. Albrecht, A. A. Alves, G. Amadio, and et al., "A roadmap for hep software and computing r&d for the 2020s," *Computing and Software for Big Science*, vol. 3, no. 1, p. 7, Mar 2019. [Online]. Available: https://doi.org/10.1007/s41781-018-0018-8

[9] https://www.egi.eu/use-cases/research-infrastructures/wlcg/.

[10] D. Reimer, A. Thomas, G. Ammons, T. Mummert, B. Alpern, and V. Bala, "Opening black boxes: Using semantic information to combat virtual machine image sprawl," in *Proceedings of the Fourth ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, ser. VEE '08. New York, NY, USA: Association for Computing Machinery, 2008, p. 111–120. [Online]. Available: https://doi.org/10.1145/1346256.1346272

[11] https://kubernetes.io/.

[12] https://helm.sh/.

[13] https://indico.cern.ch/event/708041/contributions/3276224/.

[14] T. Harter, B. Salmon, R. Liu, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Slacker: Fast distribution with lazy docker containers," in *14th USENIX Conference on File and Storage Technologies (FAST 16)*. Santa Clara, CA: USENIX Association, 2016, pp. 181–195. [Online]. Available: https://www.usenix.org/conference/fast16/technical-sessions/presentation/harter

[15] L. Gerhardt, W. Bhimji, S. Canon, M. Fasel, D. Jacobsen, M. Mustafa, J. Porter, and V. Tsulaia, "Shifter: Containers for HPC," *J. Phys. Conf. Ser.*, vol. 898, no. 8, p. 082021, 2017.

[16] O. Rodeh, J. Bacik, and C. Mason, "Btrfs: The linux b-tree filesystem," *ACM Trans. Storage*, vol. 9, no. 3, Aug. 2013. [Online]. Available: https://doi.org/10.1145/2501620.2501623

[17] K. Jin and E. L. Miller, "The effectiveness of deduplication on virtual machine disk images," in *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference*, ser. SYSTOR '09. New York, NY, USA: ACM, 2009, pp. 7:1–7:12. [Online]. Available: http://doi.acm.org/10.1145/1534530.1534540

[18] H. A. Lagar-Cavilla, J. A. Whitney, A. M. Scannell, P. Patchin, S. M. Rumble, E. de Lara, M. Brudno, and M. Satyanarayanan, "Snowflock: Rapid virtual machine cloning for cloud computing," in *Proceedings of the 4th ACM European Conference on Computer Systems*, ser. EuroSys '09. New York, NY, USA: ACM, 2009, pp. 1–12. [Online]. Available: http://doi.acm.org/10.1145/1519065.1519067

[19] N. Mandagere, P. Zhou, M. A. Smith, and S. Uttamchandani, "Demystifying data deduplication," in *Proceedings of the ACM/IFIP/USENIX Middleware'08 Conference Companion*. ACM, 2008, pp. 12–17.

[20] P. Kulkarni, F. Douglis, J. D. LaVoie, and J. M. Tracey, "Redundancy elimination within large collections of files." in *USENIX Annual Technical Conference, General Track*, 2004, pp. 59–72.

[21] B. Zhu, K. Li, and R. H. Patterson, "Avoiding the disk bottleneck in the data domain deduplication file system." in *Fast*, vol. 8, 2008, pp. 1–14.

[22] C. Policroniades and I. Pratt, "Alternatives for detecting redundancy in storage systems data." in *USENIX Annual Technical Conference, General Track*, 2004, pp. 73–86.

[23] P. Nath, M. A. Kozuch, D. R. O'hallaron, J. Harkes, M. Satyanarayanan, N. Tolia, and M. Toups, "Design tradeoffs in applying content addressable storage to enterprise-scale systems based on virtual machines," *management*, vol. 7, no. 5, p. 20, 2006.

[24] K. Jin and E. L. Miller, "The effectiveness of deduplication on virtual machine disk images," in *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference*. ACM, 2009, p. 7.

[25] J. Bent, G. Grider, B. Kettering, A. Manzanares, M. McClelland, A. Torres, and A. Torrez, "Storage challenges at los alamos national lab," in *012 IEEE 28th Symposium on Mass Storage Systems and Technologies (MSST)*, April 2012, pp. 1–5.

[26] https://gitlab.cern.ch/hep-benchmarks/hep-workloads.

[27] A. Z. Broder, "On the resemblance and containment of documents," in *Proceedings. Compression and Complexity of SEQUENCES 1997 (Cat. No.97TB100171)*, June 1997, pp. 21–29.