**Noname manuscript No.**
(will be inserted by the editor)

# ENTICE VM image analysis and optimised fragmentation

**Akos Hajnal · Gabor Kecskemeti · Attila Csaba Marosi · Jozsef Kovacs · Peter Kacsuk · Robert Lovas**

Received: date / Accepted: date

**Abstract** Virtual machine (VM) images (VMIs) often share common parts of significant size as they are stored individually. Using existing de-duplication techniques for such images are non-trivial, impose serious technical challenges, and requires direct access to clouds' proprietary image storages, which is not always feasible. We propose an alternative approach to split images into shared parts, called fragments, which are stored only once. Our solution requires a reasonably small set of base images available in the cloud, and additionally only the increments will be stored without the contents of base images, providing significant storage space savings. Composite images consisting of a base image and one or more fragments are assembled on-demand at VM deployment. Our technique can be used in conjunction with practically any popular cloud solution, and the storage of fragments is independent of the proprietary image storage of the cloud provider.

## 1 Introduction

As cloud computing users and providers face the ever changing demands of a wide range of highly connected and autonomous applications, the backbones of clouds are frequently tested to their limits. One of the demanding requirements of recent scenarios like Internet of things to cloud integration is the rapid construction and destruction of computing infrastructure components

This research work has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 644179 (ENTICE).

Akos Hajnal · Attila Csaba Marosi · Jozsef Kovacs · Peter Kacsuk · Robert Lovas
Institute for Computer Science and Control, Hungarian Academy of Sciences E-mail: {hajnal.akos, marosi.attila, kovacs.jozsef, peter.kacsuk, robert.lovas}@sztaki.mta.hu

Gabor Kecskemeti
Department of Computer Science, Liverpool John Moores University E-mail: g.kecskemeti@ljmu.ac.uk

often hosted in a federation of Infrastructure as a Service (IaaS) clouds. These operations depend on the cloud provider's specialised storage area networks that aim at serving, replicating, storing and distributing the disk volumes for the VMs used within the customer's computing infrastructures. The most stressful operation for these dynamic scenarios is the creation of new VMs, which requires the duplication of the substantially sized virtual machine disk contents (called Virtual Machine Images - or VMIs). This operation is required as VMs would not be in full control of the contents of their disks.

In a typical cloud infrastructure, the number of such VMIs may vary from tens to thousands. For example Amazon Web Services currently offers for its Elastic Compute Cloud (EC2) [1] service 31 different community VMIs, additionally from the AWS Marketplace, 2101 images for Software Infrastructure, 552 Images as Developer Tools and 1362 Images as Business Software are available (there are overlaps between the categories). VMI sizes also vary widely, from a few hundred of megabytes to several gigabytes. Thus, storing them as separate images implies significant storage space requirements at the provider side, which is often reflected in the costs of VMI storage at the customers. Consequently, both parties are interested in the reduction of the overall and individual storage footprint of the VMIs. To fight the excessive storage needs, providers either limit the choice (e.g., do not allow custom VMIs or only allow VMIs to be derived from their own) or apply techniques that reduce the stored data on the provider side but maintain the images as if they were stored from the user point of view. The most notable techniques here are copy-on-write (which does not duplicate VMIs on VM creation, instead only keeps track of the written parts of the VMIs for the user's VMs – [21]) and de-duplication (which analyses VMIs stored in the system for common blocks and only stores the unique ones – [18, 22]).

Unfortunately, these solutions only offer remedy to the providers, and even for them it offers limited capabilities. Customers on the other hand not only have the issue of storing images at a single provider, they often could face situations when a VMI stored at one provider should be instantiated at a completely different one. As cross-provider VMIs are problematic to create, instead customers often resolve this issue by utilising DevOps techniques (e.g., using chef recipes – [10]) to automate the creation of the VMI that meets the different provider's requirements. In this article, we propose a customer side space reduction technique, which identifies self-contained and often semantically meaningful and reusable fragments by instrumenting the customer applied automated image construction technique.

We present our technique through an architecture based on the assumption that we can store VMIs in parts and assemble them on-demand. We have contributed the following architectural components: (*i*) a provider independent image catalog; (*ii*) a third-party partial image storage system (which associates searchable metadata with previously defined VMI fragments); (*iii*) the image decomposition system, which analyses the file operations accomplished with the customer's DevOps tools and identifies those files that could be meaningfully separated into their own fragment; (*iv*) a virtual image composer which can reverse the decomposition process and reproduce the customer's originally intended image which previously required only a fraction of its storage space; and finally, ($v$) the VMI launcher which allows customers to directly instantiate decomposed VMIs in a cloud. Our solution, besides the advantage of expected reduced storage requirements, can be used as a complementary tool without altering the proprietary image repository backing up (due to external fragment storage) or affecting the operation of the cloud. As a result, with the help of the multi objective fragment storage optimisation tool of ENTICE [20], even multi-cloud fragmented VMI launch is possible. As a disadvantage, when applied with our VMI launcher, our solution imposes increase in VM instantiation time as the image fragments need to be assembled on the site of the new VM.

In order to analyse the behaviour of our architecture, we have executed two experiments. We have collected several widely used VMI definitions and investigated their impact on the cloud VMI storage. To evaluate the success of our customer side solution, we have provided the metric of storage space reduction ratio. Using this metric, we have evaluated 4 storage scenarios and concluded that our architecture could potentially reduce customer storage costs to $\frac{1}{5} - \frac{1}{6}$ of its original levels.

The rest of the paper is organized as follows. In Section 2, we give a brief introduction about the ENTICE project. In Section 3, we overview the main approaches to split VM images into individual and common parts and merge them back on-demand. In Section 4, we present the users' view of the proposed system that transparently handles image fragmentation and assembly. Section 5 describes the design decisions and technical solutions we applied to implement the image fragmentation concept. Section 6 reports on preliminary experimental results. In Section 7, we discuss the related work. Finally, Section 8 concludes the paper and outlines future works.

## 2 The ENTICE project

The ENTICE project [19,11] is a multidisciplinary team of computer scientists, application developers, cloud providers and operators with the aim to research a ubiquitous repository-based technology for VMI (and container) management called ENTICE environment. This environment proves a universal backbone for IaaS image management operations, which accommodate the needs for different use cases with dynamic resource (e.g., requiring resources for minutes or just for a few seconds) and other Quality of Service (QoS) requirements.

The developed ENTICE environment is completely decoupled from the applications and their specific runtime environments, but continuously supports them through optimised VMI creation, assembly, migration and storage. It is designed to receive unmodified and functionally complete VM images from users, and transparently tailor and optimise them for specific Cloud infrastructures with respect to their size, configuration, and geographical distribution,

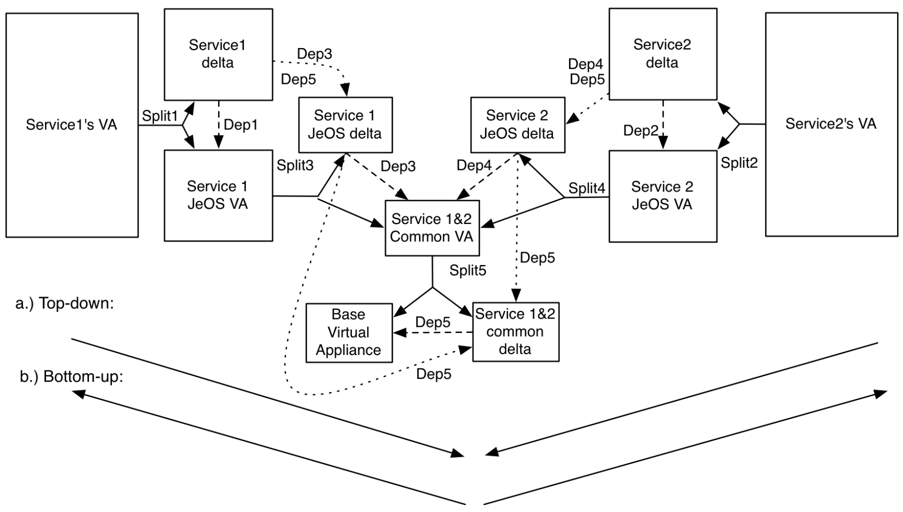

**Fig. 1** ENTICE image fragmentation approaches: a.) top-down; and b.) bottom-up

such that they are loaded, delivered (across cloud boundaries), and executed faster and with improved QoS compared to their current behaviour. ENTICE gradually stores information about the VMI and fragments in a knowledge base that will be used for interoperability, integration, reasoning and optimisation purposes (e.g. repositories should decide at which other repositories one needs replicas of a heavily requested image and at which time such an image is replicated).

## 3 Approaches to Reducing Space Requirements of VM Image Storage

Fig 1 depicts the two main approaches to decomposing images. First the top-down approach (see (a) in Fig. 1) starts from an existing set of monolithic VM images available in the cloud's image repository (see *Service 1's VA* in Fig. 1). After analysing the content of different images, it extracts common parts (see *Service 1&2 common delta*). Identical content is removed from the original images and stored separately. Simultaneously, a *fragment index record* is created that lists all constituting fragments from which the original image can be restored. This approach can be either file or block based. In the former case common files are extracted, in the latter case images are split into fixed-size blocks (e.g., of size 4kB). These are stored individually, and only the identifiers of the different blocks are registered in the fragment index record. The less the block size is the more likely that different images share common blocks, i.e., more space saving can be achieved, but on the other hand, the size required to maintain fragment index records can drastically increase.

Second is the bottom-up approach (see (b) in Fig. 1) that starts from a small set of initial *base images* with limited functionality and of size as little as possible (e.g., minimal operating system images), and every subsequent image is built upon these base images. In the case of new images, only the increments are calculated and stored compared to previous, base image contents, representing layers on top of base images. This approach avoids storing base image contents redundantly in the different images extending the same base images.

Both approaches have advantages and disadvantages. First, both approaches allow to create a Just enough OS (JeOS) [15] VA that will act as a minimal VA that allows the application or service to run. For the top-down approach (see (a) in Fig. 1), considering variable-size blocks, the method to determine an optimal set of common fragments is non-trivial (and even finding one, adding new images might corrupt it); furthermore, in the case of large image set, the number of required comparisons can be very high and compute-intensive. Also, fragments identified in this way typically lack of any semantics (that is, how fragments can be related to specific software components or operating system-related contents). On the other hand, this approach can be well automated and can achieve significant space savings. For the bottom-up approach (see (b) in Fig. 1), the calculation of the difference (increment) between base images (parent) and images extending the same base image (children) is much easier, and the pair of images to compare is inherent. Storage space reduction is achieved by not replicating base image contents in child images. The drawback of this approach is that it cannot decompose automatically existing (legacy) images.

Image comparison can be done either based on byte-level, block-level, or file-level. The finer the granularity is, the more processing power and time are required. The higher space savings may result in higher computation or network cost at decomposing and restoring virtual machine images. Finding a trade-off between reduced storage and acceptable VM deployment time can be challenging [17]. Image assembly can also happen off-line or on-the-fly. In the off-line case, the image of the VM to be deployed is assembled first (on a separate host), then this image is transferred to the host of the VM in a second step. In the case of on-the-fly assembly, a bootable VM is launched, which assembles the underlying disk contents by transferring and merging the corresponding fragments at boot time. In the first case, transferring the assembled image to the host implies additional overhead; in the latter case, it is important not to interfere the boot process with simultaneous disk changes.

This paper proposes an implementation of approach (b), which uses file-level granularity and assemble VM disk images on-the-fly. The reason for choosing that approach was due to the specific requirement of the ENTICE project that is fragments are also related to semantical units (such software packages). From these the system should be able to build new images corresponding to different configurations according to user's needs. By using block-level granularity for units this would not be possible.

## 4 User's Perspective

Ideally, the mechanism that splits VM images to fragments and reassembles them is completely transparent for the end-user, and end-users must also be able to easily compose new images having a set of pre-defined functionalities (e.g., based on a list of required software packages) using a graphical user interface. This section presents a user scenario, which illustrates how they can exploit these features from their point of view.

For the end-users, in the ENTICE environment, the *image catalog* is presented, which lists all the previously created VM appliances (including base and composite images). It provides search functionality as well. Each image entry contains a name, description as well as a list of labels called as *tags*. These tags refer to functionalities that the image is capable of, which may include the name of the underlying operating system (e.g., Ubuntu, CentOS ArchLinux), software names (e.g., tomcat, mysql-server, redmine), or any other custom labels relevant to that image, respectively. Image entries can be filtered by typing in a list of tags. When the end-user finds an appropriate image in the catalog with all the demanded tags, a virtual machine can immediately be run from this image, which will have all the functionalities that the user needs.

If the user does not find an appropriate image, a list of *similar images* – having the most tag matches – is offered for extension. After selecting one of these images, the user can choose one or more additional functionality among the a set of pre-made *installers*. Installers have name, tags, description, and version number (e.g. "mysql-server", "Installs MySQL Community Server version", "v5.6"). After having selected the proper installers, the system will automatically build and add the new image to the catalog by running these installers on the original *source image*. The new image built automatically gets all tags that the installers imply. If no installer found providing the demanded, the user can submit custom shell scripts, Ansible playbooks, etc. to be run on the source image.

When the installation cannot be done automatically with installers, advanced users may choose to create a new image manually by starting a VM from an existing image and do the installation/configurations on its own. When done, the disk image of the VM can be saved (snapshot) and uploaded, which will however not be stored as a whole, but the increment fragment will automatically be computed. The user can specify arbitrary tags, name and description for the new image.

Legacy images can also be added/uploaded to the image catalog with arbitrary name, tags, and description, however, these images, similarly to base images, will not be split into fragments. Still, new image extensions built on top of them will then use increments.

## 5 Design Concepts

This section presents how an image fragmentation system fulfilling the previously described requirements could be designed, how it was be decomposed into separate functional components and how they interact with each other during the different user scenarios. Some implementation details are also provided. The designed system architecture is shown in Fig. 2. In the following subsections, each component is described in more detail except for the third-party *Fragment Storage* component, which is assumed to be a simple data storage (e.g., S3 object storage), the *Image Storage*, which is the proprietary image storage of the related cloud, and the GUI, which is an integrated one for the whole ENTICE environment and thus, beyond the scope of this paper. Note, that the ENTICE environment contains further multi-objective optimisations [20] on these storage components (e.g., it can decide where to store particular base images, where to replicate fragments). These optimisations are out of scope of this paper, but they enable the ENTICE architecture to handle situations such as loss of fragments (i.e., because of hardware failure), or I/O bottlenecks for specific fragments. In this paper, we solely focus on how to identify the smallest fragment set that can still represent the VM images under the control of the ENTICE environment.

The main processes that realize the high-level functionalities of the image fragmentation system are as follows:

– *Base image registration.* The image fragmentation system allows of registering *base images*. Base images correspond to some operating system images, which are not subject of image fragmentation but are stored in clouds' proprietary image repository as a whole. VMs thus can be deployed from base images directly in the related clouds. In this process only the *Virtual Image Manager* component is involved, which maintains the catalog of base images and their details (such as the proprietary image id in the cloud's image repository).
– *Creation of a new virtual image.* The image fragmentation system allows of creating new *virtual* (or composite) images that are composed of a base image and one or more image fragments assembled on-demand. At creating a new virtual image the image fragmentation system automatically computes the related fragment. Any new virtual image either extends a base image or an already existing virtual image with additional functionalities (e.g. new software packages). The extension can be done automatically using pre-made *installers*, or manually, respectively, in which case, an already prepared image file is provided to the fragmentation system. This process involves the interaction of the *Virtual Image Manager* component, which provides details about the base image and the fragments of the source (parent) image, the *Virtual Image Decomposer* component, which calculates the difference between the source and the target image (temporary whole image of the new virtual image), the *Virtual Image Composer* component, which provides the assembly script to build the source image, the

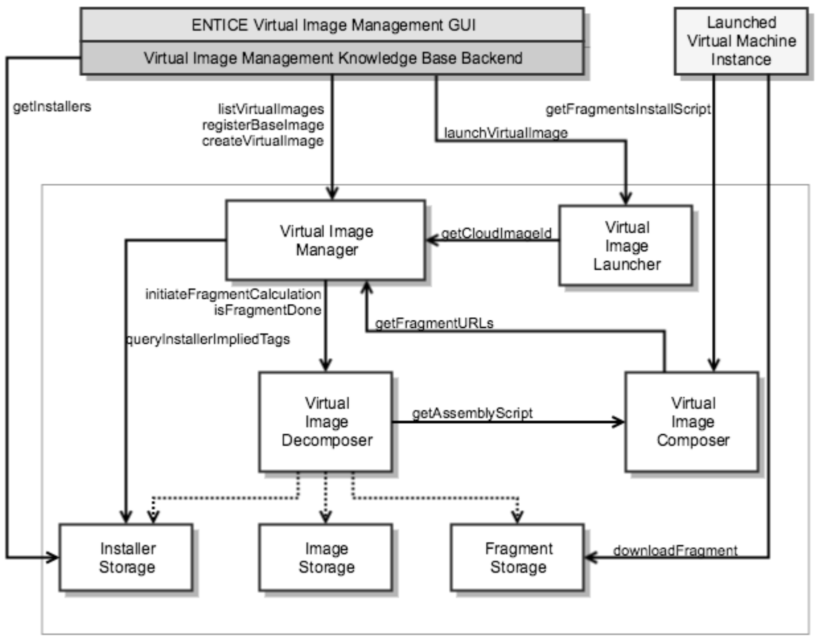

**Fig. 2** System architecture of ENTICE image analysis and optimised fragmentation

*Image Storage*, which stores the image file of the original base image, the *Installer Storage* component, which provides the installer scripts to build the target image, the *Fragment Storage* component, which is used to store the fragments.

– *Virtual image launching.* The image fragmentation system allows of launching VMs from *virtual images.* The system starts up with launching the base image in the related cloud and assembles its underlying disk contents from fragments. This process involves the *Virtual Image Launcher* component, which orchestrates VM launch and image assembly process by interacting with the *Virtual Image Manager* component, which provides the proprietary image id from which the initial VM is started, the *Virtual Image Composer* component, which provides the assembly script that merges all fragments of the virtual image, and the *Fragment Storage* component, which stores the fragments.

Infrastructure level concerns and decisions are considered out of scope for this paper. For example at the infrastructure level there should be multiple guarantees that a certain degree hardware failure does not result in data loss. In this context losing a fragment is the same as a customer losing a VMI due to a hardware failure in the storage subsystem of any cloud. However indeed losing a fragment would affect multiple VMIs (and possibly multiple customers). We also assume that the storage subsystem properly distributes load, caches or replicates to avoid hot spots caused by concurrently accessing fragments required for multiple VMIs in large-scale scenarios. Our services augment cloud systems. Images may be registered in a cloud repository (and the underlying infrastructure takes care of failure tolerance and replication), and/or stored on an object storage service like Amazon S3, where fragments can be stored as well (and the underlying infrastructure takes care of replication and failure tolerance).

## 5.1 Virtual Image Manager

Virtual Image Manager is the central component of the image fragmentation system, which maintains the catalog of base and composite images. Base images typically correspond to some small-size, official images of Linux OS distributions (Ubuntu, Debian, CentOS, etc.), which are not split into fragments; each stored in the proprietary image repository of the clouds. Composite images are assembled at VM deployment-time, which are also referred to as *virtual images* as they are never stored as a whole. The image catalog contains meta-information about each image (identifier, names, descriptions, tags, etc.) and relations. Extension and registration of new images can be initiated through this component; it is responsible for tracking image building and fragmentation processes done in the background and also assisting at VM launch for proper contextualization. Virtual Image Manager provides high-level functionality (list, create, delete virtual images) through a REST API, which serves as a backend for the graphical user interface. We note that this component does not store any image or fragment contents, merely holds references to contents stored in other storages.

Base and composite images form a set of trees in the image catalog, as illustrated in Fig. 3, where root nodes correspond to base images, whereas other nodes correspond to composite images. (Directed) edges represent fragments, which, when merged to parent images result in the children images. A composite image corresponding to a particular node in this tree can thus be obtained merging all fragments along the path from the root node to that node, in the proper order.

To register a new base image we only have to give some simple parameters such as name, description and tags, and id of the image in the related proprietary cloud image repository (repositories) (used to at VM launching), no fragments are computed in this case. When introducing a new composite image, the users have either the option to use pre-made installers ($I_2 \rightarrow I_4$ in Fig. 3), custom install scripts ($I_4 \rightarrow I_7$ in Fig. 3), or provide a snapshot obtained by making modifications manually ($I_5 \rightarrow I_9$ in Fig. 3). In all these cases, the manager initiates the fragment computation between the parent and the new composite image, via the Virtual Image Decomposer component (see Fig. 2).

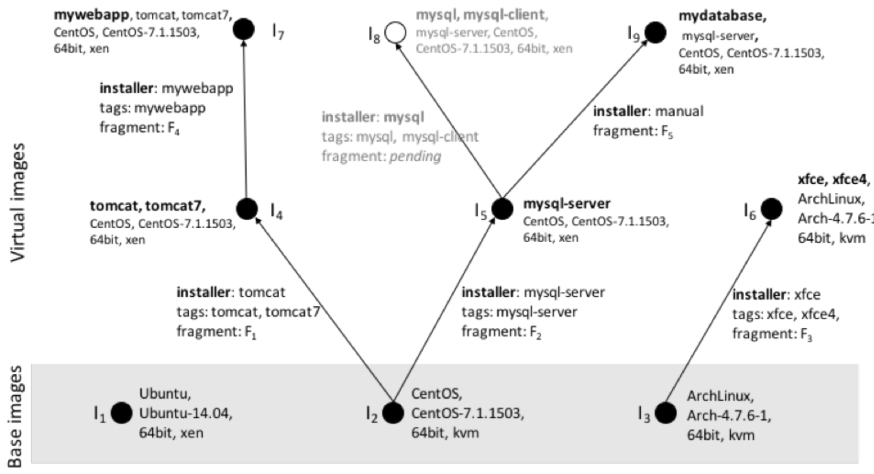
Virtual images
Base images
mywebapp, tomcat, tomcat7, CentOS, CentOS-7.1.1503, 64bit, xen
I7
installer: mywebapp
tags: mywebapp
fragment: F4
tomcat, tomcat7, CentOS, CentOS-7.1.1503, 64bit, xen
I4
installer: tomcat
tags: tomcat, tomcat7
fragment: F1
mysql, mysql-client, mysql-server, CentOS, CentOS-7.1.1503, 64bit, xen
I8
installer: mysql
tags: mysql, mysql-client
fragment: pending
mydatabase, mysql-server, CentOS, CentOS-7.1.1503, 64bit, xen
I9
installer: manual
fragment: F5
mysql-server CentOS, CentOS-7.1.1503, 64bit, xen
I5
installer: mysql-server
tags: mysql-server
fragment: F2
xfce, xfce4, ArchLinux, Arch-4.7.6-1, 64bit, kvm
I6
installer: xfce
tags: xfce, xfce4,
fragment: F3
I1
Ubuntu, Ubuntu-14.04, 64bit, xen
I2
CentOS, CentOS-7.1.1503, 64bit, kvm
I3
ArchLinux, Arch-4.7.6-1, 64bit, kvm

**Fig. 3** Tree of base and composite images

To each node a set of tags is associated in a way that successor nodes contain an aggregation of all tags associated with parent(s) (recursively). Furthermore, when applying installers to extended an image, the new image automatically gets all tags of the related installer(s), which information is provided by the *Installer Storage* component.

The manager component is capable of providing the information from which fragments a composite image can be assembled, that in turn is used by the Virtual Image Composer component (see Fig. 2).

## 5.2 Installer Storage

Installer Storage is the repository, where the so called *installers* are stored. The purpose of using installers is to assist users adding new software components (such as software packages, databases, etc.) to virtual machine images. Installer Storage maintains the catalog of installers offered to the user when they need to extend existing images with missing functionality. Installers have tags associated, which help in browsing/searching among installers.

Each installer package consists of the following components: First, *meta-information* contains name, description, version, and tag information about the installer. Second, *installer code* is used by the Virtual Image Composer component (see Fig. 2) to prepare the image having the required package installed. This code can be e.g., a shell script, a packer receipt or an Ansible playbook. Third, *dependences* are list of tags that the image must own in order to successfully perform the installation, for example, an installer could contain a dependency "ubuntu" meaning that the installer works on images having the tag "ubuntu". Dependencies may also refer to tags implied by other installers (e.g. mysql-server is required prior to installing a certain web application).

When selecting multiple installers, the system will try to automatically reorder the list of requested installers to fulfill all potential dependences, and this order will be used to produce the new image.

Fourth, the optional *post-assembly script* is used by the Virtual Image Composer component at assembling the image of a newly deployed VM. Since fragment merging merely modifies the disk contents, services, daemons will not be started automatically. To avoid the necessity of rebooting the launched VM, this script serves as starting the related services. For example, after the fragment containing mysql-service files, the post-assembly script starts the MySQL database management system (i.e., with the *service mysql start* command). Finally, *pre-assembly script* is for special, rare cases, when prior to merging a fragment, certain actions are required to perform, such as stopping processes, that could conflict with the successful fragment application (e.g., because of file locks, pids, etc.).

## 5.3 Virtual Image Decomposer

This component performs the actual computation of the fragment between two VM images, called *source* and *target* images, when a new virtual image is to be created and added to the image catalog. The steps of this procedure are as follows:

1. *Source image composition or download.* Depending on whether a base or a previous composite image is extended, the source image is either already given and can be downloaded from the related cloud image repository, or the source image has to be composed in the same way as at launching VMs, as it will be described in section 5.4, first by downloading the image corresponding to the base image of the composite image then merging fragments onto it.
2. *Target image creation or download.* In the case of manual extension, a snapshot corresponding to the target image is provided by the user. When using installers, the the target image has to be built which is made on the clone of source image (for later difference computation, both the source and target images must be present). In the current implementation, first the clone of the source image is mounted, then installers are run one-by-one in a *chroot* environment. Depending on the that whether the installer is a script, recipe or playbook, its execution alters correspondingly. As a result of the installation the target image will be available.
3. *Difference calculation.* We used the *rsync* tool with the proper parameters to calculate the difference between the source and target images by mounting them onto different directories *source-dir/* and *target-dir/*. It is done in two steps. In the first step, we determine all files that are new in the *target-dir/* or it has been changed in any way, respectively, identified by hashing or size and last change or modification date. All additions and updates (with whole file contents and attributes) are copied to a new folder named *diff/*. In a second step, we determine those files that exist

in *source-dir/* but missing from *target-dir/* and create a special file called delta-deletion in *diff/*, which contains the paths of all such deleted file entries. We note that rsync is configured to precisely consider all attributes: ACLs, permissions, symbolic links, hard links, group and owner, times of access, change and modification.

4. *Fragment creation.* If we use installers to build the target image, before creating the fragment package, we copy pre- and post-assembly scripts of the installers to the directory of the fragment. Also, at preparing the delta package, a special file */var/lib/cloud/vvmi.id* containing the identifier of the new composite image (maintained by the Virtual Image Manager component) is added as a "watermark". This id helps in automatically finding out the original, predecessor image, when an image is extended manually, and a snapshot is given. Then a compressed archive (*tar.gz*) of the fragment contents (*diff/*) is created.
5. *Storing the fragment.* The fragment is then uploaded to the Fragment Storage, and the corresponding reference to the delta package (download URL) is returned to the Virtual Image Manager. We note that it is also possible to calculate a hash code for the entire fragment contents, and store only fragments whose hash code differs from all previous hash codes, to avoid storing the same delta packages redundantly. In the case of identical hash codes, the same fragment reference is used in the Virtual Image Manager component.

## 5.4 Virtual Image Composer

Virtual Image Composer component (see Fig. 2) is responsible for assembling a composite image from fragments. More specifically, it provides a shell script that contains all the necessary commands required by merging *delta packages* and starting services, as needed.

To generate the *assembly script*, the Virtual Image Composer only requires giving the image id (node) of a composite image to be assembled, which must exist in the image catalog of the Virtual Image Manager. The Virtual Image Composer can then query the Virtual Image Manager for the sequence of fragments, more specifically, the list of URLs from where the fragment packages can be downloaded (stored in edges along the path between the corresponding base image and the composite image). The *assembly script* contains the following functionalities:

1. Download the fragment package from the provided URL.
2. Extract and execute the optional *pre-assembly script* provided by the fragment package.
3. Assemble using fragment package contents and by removing not-needed parts specified by *.delta-delete.*
4. Execute optional *post-assembly script.*
5. Clean-up: remove fragment package, pre- and post-assembly scripts and deletions.

Executing the *assembly script* on the base image of the composite image will reproduce the accurate disk contents of the composite image composed of a series of fragments, also start all services given in post-assembly script. Virtual Image Composer is used by the Virtual Image Decomposer component to assemble composite source images, and also by Virtual Image Launcher to assemble the composite images at deploying a virtual machines, see Section 5.5.

### 5.5 Virtual Image Launcher

This component is used to launch VMs in clouds whose disk images correspond to composite images, and so they cannot be launched in conventional ways provided by the cloud provider API/GUI as their images are not present in the proprietary repository of in the cloud as a whole.

After giving the id of the composite image, the necessary credentials (access key, secret key) and optional parameters (e.g., instance type), the Virtual Image Launcher queries the Virtual Image Manager then launches a VM with disk corresponding to the base image of composite image (root node) using the appropriate cloud API interface (e.g., EC2). As the base image is registered in the cloud, the VM can be booted up like any other images.

In contrast to other VMs in the cloud, this VM is contextualized in a way that after booting up, it downloads and executes the assembly script from the Virtual Image Composer corresponding to the composite image id given at launching the VM. As a result, the launched VM will automatically assemble the disk contents from fragments. In the current implementation, we used *cloud-init* [7] cloud configuration file to perform such contextualization.

## 6 Experimental Results

To evaluate our proposed techniques, we have set up 2 scenarios each with increasing complexity (in terms of the number of VMIs involved in decomposition process). In our first scenario, we have analysed the integrated behaviour of all architectural components from Fig. 2. We have evaluated this scenario on a real life deployment of our system. For the later scenario, we have implemented a simulation environment which analyses online software package repositories (e.g., ones offered by the maintainers of the Ubuntu and Debian Linux distributions) and deduces decomposition options as well as expected fragment sizes based on metadata acquired from these repositories. Thus, in our second scenario, we have collected the recipes for several frequently built Ubuntu Linux based VMIs and analysed the expected behaviour of our virtual image decomposer with them.

Our system uses different auxiliary components that may seem to affect the results: dependency trees for fragments are represented as simple database entries. Scripts are negligible in size compared to fragments or base images. Installers could be larger, but as a general recommendation (for any type of installer), that it should source components (e.g., release tarball of a specific software) either from official sources or use operating system packages thus, only the script part needs to be stored by our system. Based on this we assume the size of the different installers is also negligible.

## 6.1 Small scale experiments

The ENTICE infrastructure utilizes different clouds provided by its academic and industrial partners. These are based on the open source OpenNebula [28] (3 sites), the commercial Flexiant Cloud Orchestrator [13] and VMware ESXi. For the small scale experiments we chose OpenNebula as we are running it in production for years. The familiarity with this middleware allow us to effectively identify and track down possible idiosyncrasies when performing our experiments. For this we chose the LTM stack, meaning Linux, MySQL and Tomcat7, see Fig. 4 and Section 6.2 for more details. As base image we used an image based on the cloud image provided by Canonical based on Ubuntu 16.04 LTS [8]. On top of the LTM stack we deployed the Data Avenue [16] application. Table 2 presents the results.

*Image content size* in Table 2 show the actual disk usage within the image, while *image size* refers to the actual size of the image. We use the compressed QCOW2 format that can cause non-linear differences between content and image size. Also we denoted two image sizes: first is with image maintenance. This means the following: first the package manager caches are cleaned. Second the free space on the image is written over with zero data, this allows achieving better compression results for the QCOW2 format. However the image must be duplicated as QCOW2 will not automatically make use of this. This reduces the final image size by approx. 23.3% at the cost of additional time to produce the final image.

Each step and additional fragment produces an image that contains a functionality that can be used on its own. Let's consider the storage requirements for all images versus the base image and the fragments and calculate the Storage Space Reduction Ratio (SSRR) using the following formula:

$$SSRR = 1 - \frac{size(Base\ Images + Fragments)}{size(Base\ Images + Virtual\ Images)} = 0.7866 \tag{1}$$

This simply means that storing only the fragments and the base image compared to storing the intermediate (virtual) images with the base image requires 78.66% less storage space.

In order to evaluate the overhead of on-the-fly disk content composition, we carried out several experiments, in which we measured the time required to re-assemble VM images from fragments at VM start-up. These data are then compared to the time required to launch the composite image (all the components were set up in advance, stored as whole in the cloud image repository). We used an OpenNebula cloud (version 5.2.0), and two instance types

| **Instance type** | **Composite image service start-up time** | **Fragmented image service start-up time** | **Slow-down** |
|---|---|---|---|
| t2.medium | 59 seconds | 151 seconds | 92 seconds |
| t2.large | 53 seconds | 122 seconds | 69 seconds |

**Table 1** Service start-up times using composite and fragment VM images

t2.medium (2VCPU, 4GB RAM) and t2.large (2VCPU, 8GB RAM). In these experiments, the Fragment Storage component (see Fig. 2) was a simple file system-based storage (fragments are stored in the file system), implemented using a simple tomcat web application deployed in a separate VM.

We measured the time elapsed between VM start time and the time when the service (Data Avenue) is up and running, in the case of both composite and fragmented images, on the two instance types. These data are shown in Table 1 (average values of ten measurements). From these data we can see that launching VMs using fragmented images are indeed slower as expected by 69 and 92 seconds, respectively. On return, however, we gain almost 80% storage space savings. The average fragment assembly times of all the four fragments were 64 and 81 seconds, respectively, which include fragment download, extraction, and service startup times (mysql, tomcat).

We also measured how much time it would require to install these packages manually (or using an automated configuration management tool like Chef or Ansible). The installation of mysql-server and tomcat7 took 96 seconds from console, which is slower than re-assembling these packages from fragments. In the latter measurement we excluded the installation of Data Avenue, which requires manual configuration.

We note that the cloud used in these experiments provides very fast image transfer to the host machine from the image repository. In other clouds, transferring composite images of increased size might be slower. In the case of fragmented image assembly, the base image is required to be transferred, which is of less size, whereas fragments are downloaded in compressed form afterwards. We also note that download and extraction of fragments are done sequentially one after the other. Parallelization of the extraction of the previous fragment and download of the next fragment might speed up assembly.

### 6.2 Experiments on frequently built VMIs

For our second experiments, we have identified commonly built VMIs that use widely available free software and can be built based on a Ubuntu 14.04 base image. First, we have collected a set of base servers that are frequently used, these were the following: (*i*) MySQL, (*ii*) Nginx, (*iii*) Node.js, (*iv*) MongoDB, (*v*) Redis, (*vi*) RabbitMQ, (*vii*) Apache2 and (*viii*) Tomcat7. Next, we have collected a set of server stacks that are widely considered by DevOps teams: (*i*) LAMP – Linux, Apache, MySQL, PHP; (*ii*) LAPP – Linux, Apache, PostgreSQL, PHP; (*iii*) LEMP – nginx, MySQL, PHP; (*iv*) LLMP

| **Fragment/ Base image** | **Fragment size (tar.gz, Bytes)** | **Image content size (Bytes)** | **Virtual image size with (or without) maintenance (QCOW2, Bytes)** |
|---|---|---|---|
| Ubuntu 16.04 LTS 64bit (base) | - | 971 677 696 | 1 064 370 176 (-) |
| update | 27 775 599 | 978 051 072 | 1 068 537 472 (1 133 117 440) |
| mysql-client, mysql-server | 102 746 591 | 1 225 785 344 | 1 232 142 336 (1 519 583 232) |
| tomcat7 | 119 537 550 | 1 364 975 616 | 1 370 947 584 (1 729 298 432) |
| data avenue | 46 553 088 | 1 529 782 272 | 1 422 721 024 (1 855 651 840) |

**Table 2** Size of fragments and corresponding images to produce the LTM stack with Data Avenue deployed

– Linux, Lighttpd, MySQL, PHP; (*v*) LYME – Linux, Yaws, Mnesia, Erlang; (*vi*) MEAN/MERN base – MongoDB, Node.js (leaves out Express.js and Angular.js/React.js); and, (*vii*) LTM – Linux, MySQL, Tomcat7. Finally, we have also identified a few server based applications that are widely deployed, namely: (*i*) WordPress and (*ii*) Redmine. For all these, we have determined the recipe (in the form of the list of required software packages) to be used to transform our Ubuntu base image into the required VMI.

Next, we have constructed a simulation which can understand and analyse package caches from any Debian based OS (like the one in our Ubuntu base VMI). These caches list the metadata (e.g., direct dependencies, alternatives, installed size etc.) of all the possible software packages ($P$) that can be installed on the OS. First of all, this simulation was capable to list us the required packages to be added for a set of packages in a recipe on top of a given base VMI (i.e., the simulation resolves the dependencies of the packages in the recipes). Second, the simulation also mimicked the behaviour of our decomposition technique on the level of software packages ($\forall f \in F : f \subset P$, where $f$ is an arbitrary fragment, $F$ is all possible fragments, and $P$ is the package set). Using this notation we will use the base VMI as $f_{base} \in F$, and the notation of $f_{recipe} \in F_R \in F$ will depict additional packages needed for a given recipe (e.g., $f_{mongodb} \equiv f_{35}$ represents MongoDB). Here, $F_R$ is the set of all fragments which were acquired from the recipes for the identified VMIs. Note, $\forall f_r \in F_R : f_{base} \cap f_r = \emptyset$. Thus, we show how our decomposition technique creates a new fragment ($f_n \in F$) with the following equation:

$$f_n = f_x \cap f_y \tag{2}$$

Where $f_n$ is the decomposed fragment, and $f_x$ and $f_y$ are the pre-existing fragments. Next, based on the metadata contained in the cache, our simulation would also list the expected installation size of all required packages in a fragment, this is denoted as the function $size : F \to \mathbb{N}$. Finally, starting from the fragments resulting from the recipes, the simulation applied the decomposition technique repeatedly until it was not possible to find any pair of not-yet

**Table 3** List of the selected recipe based fragments and their respective sizes

| Fragment – $f$ | Package count: $\lvert f \rvert$ | Simulated $size(f)$ |
|---|---|---|
| $f_{base}$ | 413 | 915MiB |
| $f_5$ | 38 | 116 MiB |
| $f_8$ | 57 | 132 MiB |
| $f_9$ | 27 | 29 MiB |
| $f_{11}$ | 84 | 185 MiB |
| $f_{12}$ | 25 | 118 MiB |
| $f_{13}$ | 70 | 49 MiB |
| $f_{14}$ | 72 | 129 MiB |
| $f_{18}$ | 6 | 5.7 MiB |
| $f_{20}$ | 45 | 33 MiB |
| $f_{21}$ | 10 | 4.8 MiB |
| $f_{22}$ | 45 | 22 MiB |
| $f_{23}$ | 65 | 129 MiB |
| $f_{24}$ | 62 | 52 MiB |
| $f_{31}$ | 70 | 92 MiB |
| $f_{33}$ | 6 | 11 MiB |
| $f_{35}$ | 23 | 115 MiB |
| $f_{77}$ | 27 | 98 MiB |

combined fragments that would result in a non-empty decomposed fragment (we refer to this as the complete decomposition).

Fig. 4 shows the complete, software package level decomposition of the 16 VMIs ($\forall f \in F_R$) that we have collected recipes for previously. The figure shows members of the $F_R$ fragment-set with red dots, the other combined fragments are shown as black ones. The arrows on the figure show the inclusion relation. For example: $f_{53}$ is included in $f_{39}$ is shown as: $f_{53} \rightarrow f_{39}$ and means $f_{53} \in f_{39}$. Thus, having $f_{39}$ during a VM composition procedure means we would not need to acquire $f_{53}$. On the other hand, storing $f_{53}$ as well as $f_{39}$ in a repository is a possible waste of storage (as $f_{39}$ would replicate the almost 3 MiB data from $f_{53}$).

Storing all selected VMIs without fragmenting (i.e., one would always need to store a fragment with the base like: $f'_{35} = f_{35} \cup f_{base}$) would require 16.5GiB storage space. Storing only the base and the recipe related fragments already significantly reduces the storage requirement to 2.2GiB (this is the base behaviour of our implementation and it is shown in Table 3). Rendering a space reduction ratio of: $SPRR_{OnlyRecipe} = 0.8675$. But as mentioned before this still incurs plenty of storage waste as shown by the other combined fragments of the figure. If one would store all fragments identified in the complete decomposition shown in Fig. 4, then this would require 3.5GiB of storage which would introduce great flexibility of future additions to the fragmented VMI set at the expense of more than 1GiB storage. As a result, we would have an $SSRR_{Compl.Dec} = 0.786$

For the efficient operation of the VM Image composer, it needs to ensure it downloads the least amount of redundant content. Thus, ideally it requires a disjunct set of fragments that would lead to the VMI when merged. This

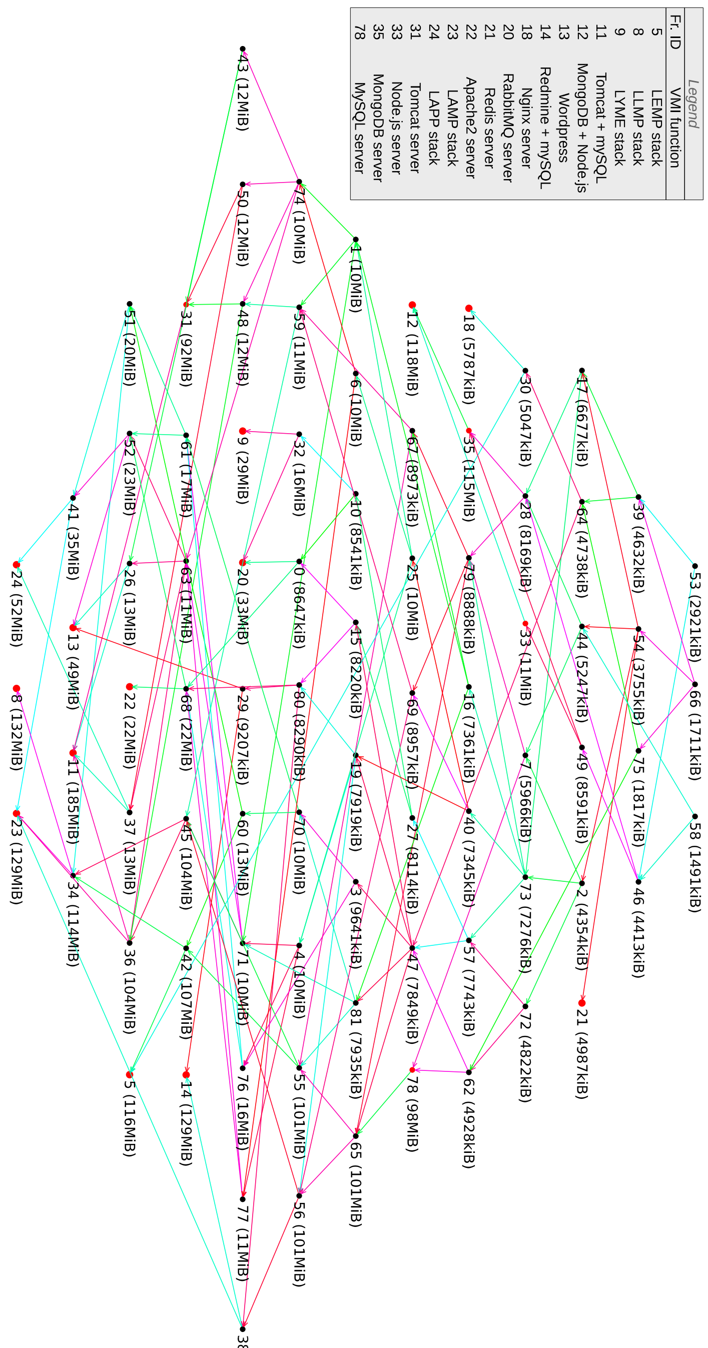

**Fig. 4** Possible fragment relationships of the 16 VMIs in the experiment

requires extension fragments to be defined as follows:

$$f_{x,n}^{ext} = f_x \backslash f_n \tag{3}$$

Where $f_x$ is a fragment which includes $f_n$, and $f_{x,n}^{ext}$ is the complementer of $f_n$ so the union of two results the original $f_x$ fragment. Unfortunately, this leads to even further levels of fragmentation and more storage requirements. Also, it decreases the VM Image composer's performance as it needs to build more parts of the image than it would be necessary (i.e., now it needs to get extension and combined fragments additionally to the base instead of the recipe based fragments initially produced by the decomposer).

To reduce the performance degradation on the composer and while to also reduce our the storage requirements, we follow the following heuristic. We identify combined fragments that could be the foundation for most recipe based fragments. To keep the storage costs low, we only keep combined fragments which would result in extension fragments with a threshold size (i.e., this can be set based on the network properties of the site where the VM Image composer is deployed). This behaviour is governed by the following equations:

$$F_X(f) = \{f_e \in F_R : (f \subset f_e)\} \tag{4}$$

$$\begin{aligned} F_{keep} = \{f_k \in F : (size(f_k) > \mathfrak{T} \wedge |F_X(f_k)| > 1 \wedge \\ (\forall f_o \in F_X(f_k) : (\mathfrak{T} < size(f_{o,k}^{ext}) < size(f_o)/2))\}\} \end{aligned} \tag{5}$$

Here we first define the set $F_X(f)$ that encapsulates all recipe based fragments that derive from the the given fragment $f$ (i.e., they have $f$ as their common root in Fig.4). Later, with the function $F_X(f_k)$, we define the set of all composite fragments which have at least two recipe based fragments derived from them while the expecting their extension fragments to be at least the size of $\mathfrak{T}$ threshold (the upper threshold of extension fragment sizes is set so large duplicates are not considered in the solution). Applying a $\mathfrak{T} = 10MiB$ value results in the following list:

$$F_X = \{f_{78}, f_{38}, f_{34}\} \tag{6}$$

These fragments contain commonly used components like mysql and php. Based on these the simulation estimated several new extension fragments that could replace their original recipe based packages, these are listed in Table 4. This approach significantly reduces the 3.5GiB storage requirement of the complete decomposition. Instead, with the introduction of these three additional fragments, it is reduced even below the base behaviour of just storing all recipe fragments and the base image. The simulated storage requirement is now at 2GiB. Thus we arrived to the space reduction ratio of $SSRR_{WithExtensions} = 0.8786$

For a the complete picture regarding the simulation results, an excerpt of the list of fragments is presented in Table 5, while a full list is available at [23].

**Table 4** List of the extension fragments and their respective sizes

| Fragment – $f$ | Simulated $size(f)$ |
|---|---|
| $f^{ext}_{8,34}$ | 18MiB |
| $f^{ext}_{23,34}$ | 15MiB |
| $f^{ext}_{14,38}$ | 27MiB |
| $f^{ext}_{23,38}$ | 28MiB |
| $f^{ext}_{5,78}$ | 18MiB |
| $f^{ext}_{8,78}$ | 34MiB |
| $f^{ext}_{11,78}$ | 87MiB |
| $f^{ext}_{14,78}$ | 31MiB |
| $f^{ext}_{23,78}$ | 31MiB |

## 7 Related Work

There are several approaches for reducing the storage footprint of VM images (VMIs). First we are looking at how container images are structured and what techniques are there used to reduce their size, and then methods for de-duplicating data segments for VMIs and storage systems in general.

Container technologies for example can be effectively utilized in solutions that allow orchestrating large-scale deployments on top of VMs in clouds [31]. Docker is probably the most well-known container engine [30], others include Rkt [24], LXD [5] or OpenVZ [25]. Docker defines its containers as follows [9]: "Docker containers wrap a piece of software in a complete file system that contains everything needed to run: code, runtime, system tools, system libraries anything that can be installed on a server. This guarantees that the software will always run the same, regardless of its environment.". A container engine can "execute" a container thus, it is analogous to a hypervisor or VM Monitor. A Docker container (image) is built using a special descriptor file called Dockerfile. This file contains a sequential list of commands about (i) which Linux distribution to base the image on; (ii) what software components are required to be installed; (iii) commands to deploy the service; etc. The Dockerfile can be regarded as a recipe to produce the desired container image. This resulting container image is a layered file system, where each command in the Dockerfile produces a new additional layer on top of the previous one. These layers are read-only, and there is a final thin read-write layer provided to write runtime modifications. Docker uses amongst others OverlayFS [9, 2] union file system to mount the layers as a single file system. Docker is designed to run a single process (e.g., Database engine, web server or application) within a container. This means Docker relies on file system related features to achieve the final file system. In case of VMs this is not feasible as VMs can use custom file systems, their file systems can be resized, may contain custom kernels that do not support some of the features required, etc.

On the other hand LXD is a container management tool that utilizes the LXC [4] operating system container, thus LXD runs a complete "Virtual Machine" inside a container. It is image based [6], for example it can use BTRFS [3] for execution, which is a modern copy-on-write file system

**Table 5** Excerpt of list of fragments. For the full list of 81 items see [23].

| Id ($f$) | $size(f)$ | Composes | Extends with packages |
|---|---|---|---|
| 13 | 49MiB | 26,29,52 | tinymce, libphp-snoopy, libphp-phpmailer, libx11-6, libjs-cropper, wordpress-theme-twentyfourteen, libtiff5, libxpm4, libvpx1, php5-gd, libjbig0, wordpress, libgd3 |
| 14 | 129MiB | 29,38 | ruby-blankslate, ruby-hmac, ruby-treetop, ruby-rack-test, ruby-activemodel-3.2, ruby-net-ldap, dbconfig-common, ruby-atomic, rake, ruby-thor, ruby-mysql, ruby-builder, ruby-journey, libyaml-0-2, mysql-client, ruby-railties-3.2, redmine-mysql, ruby-activerecord-3.2, ruby-polyglot, ruby-rack-cache, ruby-thread-safe, ruby-i18n, ruby-rack-openid, ruby-minitest, ruby-activeresource-3.2, ruby-rails-observers, ruby-actionmailer-3.2, ruby-actionpack-3.2, ruby-hike, ruby-arel, ruby-openid, ruby-activesupport-3.2, ruby-sprockets, ruby-tilt, ruby-rack-ssl, ruby-mail, ruby-tzinfo, ruby-multi-json, ruby-coderay, redmine |
| 16 | 7361kiB | 73 | libjson-c2 |
| 18 | 5787kiB | 30 | nginx-light |
| 24 | 52MiB | 37,41 | postgresql-client-9.3, postgresql-9.3, postgresql-common, libpq5, php5-pgsql, libedit2, postgresql-client-common, ssl-cert |
| 30 | 5047kiB | 64 | nginx, nginx-common |
| 32 | 16MiB | 10 | erlang-ssl, erlang-xmerl, erlang-webtool, erlang-asn1, erlang-inets, erlang-crypto, erlang-public-key, erlang-mnesia, erlang-tools, erlang-runtime-tools |
| 41 | 35MiB | 51,52 | libapache2-mod-php5 |
| 42 | 107MiB | 55,60 | php5-mysql |
| 58 | 1491kiB | | libstdc++6, gcc-4.8-base |
| 68 | 22MiB | 0,76,78,80 | apache2, apache2-bin, apache2-data, libapr1, libaprutil1-ldap, libaprutil1 |
| 70 | 10MiB | 3,81 | php5-common, php5-json |
| 77 | 98MiB | 7,62 | libdbi-perl, mysql-common, libmysqlclient18, libdbd-mysql-perl, mysql-server-5.5, mysql-server, mysql-server-core-5.5, libaio1, mysql-client-5.5, libreadline6, libterm-readkey-perl, mysql-client-core-5.5 |

with writeable and read-only snapshots, compression and planned in-band de-duplication. LXD images can be tarballs, either single one called ”unified”; or ”split” where a tarball contains the file system and a second one contains metadata. The file system tarball must contain a full bootable Linux distribution and configured to be able to run in a container (e.g., properly configured network interfaces). The tarball approach would allow to port our solution, however with some limitations (i.e., kernel).

For VM images different de-duplication approaches can be used to reduce their storage footprints: first is block level where each image is treated as a sequence of blocks of fixed size (e.g., 8KB, 16KB, etc.). These blocks are registered (via a hashing function) thus duplicate ones among different images can be filtered. Another approach is to use file based de-duplication. Comparison of these methods is discussed in detail in Section 3. Additionally, compression techniques may be utilized as well [29].

Keren et al. [18] show that with block level de-duplication techniques, one can identify nearly 70% of identical parts in frequently used VMIs. This paper shows several chunk identification techniques, but these chunks never reach over the level of file systems. Finally, the paper addresses the issue of the package management systems and their effect on the efficiency of de-duplication. The different behaviour and dependencies in the widely available package management systems lead to some variance in data chunks on the VMIs. This variance is expected to be unavoidable even if one processes the metadata offered by file systems. They claim that with even simple chunk-level de-duplication methods the size of a single image file can be reduced up to 80%. However the block level handling of the images hides several important inter-relationships between the image parts. Over several hundred VMIs were analysed by Jayaram et al. [17]. The analysis consisted of checking how efficiently 5 de-duplication techniques could work on the images, and also the authors introduced several similarity metrics. The used metrics and techniques were both checked in inter- and intra-VMI contexts. Unfortunately, this paper does not go further than point out the chance of optimizing image storage or delivery. The authors showed in a real world scenario that choosing the right chunk-size for de-duplication is crucial, as the size increases so the de-duplication factor decreases. Unfortunately the analysed techniques were not checked for their capabilities of VM image size optimization.

VMIs are continuously created in datacenters, and duplicated data segments may exist in such VM images, which is a waste of storage resource. The size of VM images is generally large, therefore it is inefficient to load massive VM image fingerprints into memory for a fast comparison to recognize duplicated segments. Xu et al. [26] propose a clustering-based acceleration method called improved k-means clustering to find images having a high chance to contain duplicated segments. In this way, only limited VM image candidate fingerprints are loaded into memory. Their experiments show that it significantly reduces the performance interference to hosting virtual machine with a negligible increase in disk space usage, compared to existing de-duplication methods. Many clouds must manage thousands or more virtual machine images, requiring significant amounts of storage. To provide a good user experience, they must be able to deploy those images quickly. Lin et al. propose a new service for efficiently storing and deploying disk images [22] by exploiting the redundant data found in similar images using de-duplication. It is also integrated with an existing highly-optimized disk image deployment system, called Frisbee, to distribute and install images. They also propose a new chunking algorithm, called AFC, which enables fixed-size chunking for de-duplicating allocated disk sectors. The authors claim that their experiments show that their system reduces storage requirements by up to 3x factor while, imposing only a negligible runtime overhead. Liquid [27] is introduced as a lightweight distributed file system heavily building on ideas from the fields of de-duplication and peer-to-peer computing. The authors consider de-duplicating both running and offline VM images, and they allow rapid cloning mechanisms with the help of copy on read. As Liquid is implemented as a file system it can be rather transparent for the IaaS system, on the other hand the authors require the IaaS systems to completely adopt Liquid in order to achieve the best performance. The authors claim that Liquid provides good I/O performance while doing de-duplication in the background by caching frequently used data blocks and organizing them into chunks to reduce disk operations. P2P technique provides good scalability. On the other hand, an own file system should be used to track VM life-cycle with a metadata server.

## 8 Conclusions and future work

In this paper we presented the ENTICE approach for image analysis, de-duplication and optimised fragmentation. It relies on file-level de-duplication and a bottom-up approach for VM image fragmentation. First we discussed the different approaches available for reducing the storage space requirements for VMIs. Next we presented our proposed techniques and our system implementing these. We evaluated our proposed techniques using two scenarios. In the first scenario we evaluated the integrated behaviour of all architectural components with a real life deployment of our system. In the second scenario we implemented a simulation environment that analyses online software package repositories and deduces decomposition options with expected fragment sizes. We introduced the Storage Space Reduction Ratio (SSRR) that denotes the achieved storage space saving when using fragments compared to the traditional storage. We showed that our technique can achieve a $SSRR = 0.7866$ for the first scenario; and $SSRR_{WithExtensions} = 0.8786$ for the second scenario.

Although the services described here provide a complete functionality for image analysis and fragmentation we see several possibilities to improve their functionalities. Some of these are as follows. We plan to add a component that is responsible for the automated updating of base images (i.e., package and security updates) as well as rebuilding of virtual image trees in the background using the updated images and fragments. We want to use updated usage statistics (the *Knowledge Base* component of ENTICE collects these, see [14] for more details). Based on this we plan to add automated node removals from and creation to the virtual image trees. This would also allow to unify paths and fragments for often used together items, for faster deployment. Additionally as future work we plan to introduce the ENTICE VM image analysis and optimised fragmentation service in the institutional OpenNebula cloud in MTA SZTAKI. This will allow us large scale evaluation to confirm the simulation results.

## Software availability

The ENTICE image analysis and optimised fragmentation software is open source, it is available under the conditions of Apache License version 2 from the ENTICE WP3 GitHub repository [12].

## References

1. Amazon Web Services: Amazon Elastic Compute Cloud (Amazon EC2) (Oct 2017), https://aws.amazon.com/ec2/
2. Brown, N.: Linux kernel storage overlayfs driver (Nov 2016), https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/tree/Documentation/filesystems/overlayfs.txt
3. BTRFS: Wiki (Dec 2016), https://btrfs.wiki.kernel.org/index.php/Main_Page
4. Canonical: Introduction to linux containers (Nov 2016), https://linuxcontainers.org/lxc/introduction/
5. Canonical: Linux containers: What is lxd? (Nov 2016), https://linuxcontainers.org/lxd/
6. Canonical: Lxd 2.0 image management (Apr 2016), https://insights.ubuntu.com/2016/04/01/lxd-2-0-image-management-512/
7. Canonical: cloud-init - the standard for customising cloud instances (Jul 2017), https://cloud-init.io/
8. Canonical: Ubuntu cloud images 16.04 lts daily build (Jul 2017), https://cloud-images.ubuntu.com/xenial/
9. Docker: Storage overlayfs driver (Nov 2016), https://docs.docker.com/engine/userguide/storagedriver/overlayfs-driver/
10. Dyck, A., Penners, R., Lichter, H.: Towards definitions for release engineering and devops. In: Proceedings of the Third International Workshop on Release Engineering. pp. 3–3. IEEE Press (2015)
11. ENTICE: project website. http://www.entice-project.eu/ (Oct 2017)
12. ENTICE: Wp3 github repository (Jul 2017), https://github.com/entice-repository/wp3-image-synthesis
13. Flexiant: Flexiant cloud orchestrator (fco) (Jul 2017), https://www.flexiant.com/flexiant-cloud-orchestrator/
14. Gec, S., Kimovski, D., Prodan, R., Stankovski, V.: Using constraint-based reasoning for multi-objective optimisation of the entice environment. In: 2016 12th International Conference on Semantics, Knowledge and Grids (SKG). pp. 17–24 (Aug 2016)
15. Geer, D.: The os faces a brave new world. Computer 42(10) (2009)
16. Hajnal, Á., Márton, I., Farkas, Z., Kacsuk, P.: Remote storage management in science gateways via data bridging. Concurrency and Computation: Practice and Experience 27(16), 4398–4411 (2015)
17. Jayaram, K., Peng, C., Zhang, Z., Kim, M., Chen, H., Lei, H.: An empirical analysis of similarity in virtual machine images. In: Proceedings of the Middleware 2011 Industry Track Workshop. p. 6. ACM (2011)
18. Jin, K., Miller, E.L.: The effectiveness of deduplication on virtual machine disk images. In: Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference. p. 7. ACM (2009)
19. Kimovski, D., Marosi, A., Gec, S., Saurabh, N., Kertesz, A., Kecskemeti, G., Stankovski, V., Prodan, R.: Distributed environment for efficient virtual machine image management in federated cloud architectures. Concurrency and Computation: Practice and Experience
20. Kimovski, D., Saurabh, N., Stankovski, V., Prodan, R.: Multi-objective middleware for distributed vmi repositories in federated cloud environment. Scalable Computing: Practice and Experience 17(4), 299–312 (2016)
21. Lagar-Cavilla, H.A., Whitney, J.A., Scannell, A.M., Patchin, P., Rumble, S.M., De Lara, E., Brudno, M., Satyanarayanan, M.: Snowflock: rapid virtual machine cloning for cloud computing. In: Proceedings of the 4th ACM European conference on Computer systems. pp. 1–12. ACM (2009)
22. Lin, X., Hibler, M., Eide, E., Ricci, R.: Using deduplicating storage for efficient disk image deployment. EAI Endorsed Trans. Scalable Information Systems 2(6), e1 (2015)
23. Marosi, A.Cs.: List of fragments for "entice vm image analysis and optimised fragmentation" (Jul 2017), https://s3.lpds.sztaki.hu/atisu/papers/entice-fragmentation/fragments.pdf
24. Rkt: The pod-native container engine (Jul 2017), https://github.com/rkt/rkt

25. Virtuozzo: Openvz virtuozzo containers wiki (Nov 2016), https://openvz.org/Virtuozzo
26. Xu, J., Zhang, W., Zhang, Z., Wang, T., Huang, T.: Clustering-based acceleration for virtual machine image deduplication in the cloud environment. Journal of Systems and Software 121, 144–156 (2016)
27. Zhao, X., Zhang, Y., Wu, Y., Chen, K., Jiang, J., Li, K.: Liquid: A scalable deduplication file system for virtual machine images. IEEE Transactions on Parallel and Distributed Systems 25(5), 1257–1266 (2014)
28. Milojicic, D., Llorente, I. M., Montero, R. S.: Opennebula: A cloud management tool. IEEE Internet Computing, 15.2: 11-14. (2011)
29. Hovestadt, M., Kao, O., Kliem, A., Warneke, D. : Adaptive online compression in clouds making informed decisions in virtual machine environments. Journal of Grid Computing, 11(2), 167-186. (2013)
30. Peinl, R., Holzschuher, F., Pfitzer, F. : Docker cluster management for the cloud-survey results and own solution. Journal of Grid Computing, 14(2), 265-282. (2016)
31. Kovacs, J., Kacsuk, P. : Occopus: a Multi-Cloud Orchestrator to Deploy and Manage Complex Scientific Infrastructures. Journal of Grid Computing, 1-19. (2017)