Image Transfer Optimization for Agile Development

Alexei Karve, Andrzej Kochut
IBM T.J. Watson Research Center
1101 Kitchawan Road, Route 134, Yorktown Heights, N.Y. 10598
{karve,akochut}@us.ibm.com

***Abstract*—Cloud computing is becoming a common delivery model for IT services. Development and testing of applications and services is usually conducted on a development cloud environment often within customer premises and deployed in stages to a production cloud. Agile development process integrates development and deployment of IT systems and requires frequent and low cost synchronization between development and deployment cloud environments. This article proposes and evaluates virtual machine transfer algorithm based on image redundancy that allows to reduce bandwidth and time required to transfer specific images from development to production sites. It also explores how a typical image library, including public and private images, evolves over time and what impact it has on potential gain from the proposed algorithm. An analytical model is also proposed that allows to quantify degree of saving from using the algorithm. Evaluation shows up to 80% reduction in terms of transfer time and network bandwidth usage.**

I. Introduction

Recent years brought the emergence of a new IT delivery model called Cloud Computing. It is projected [10] to significantly grow throughout next decade becoming one of the key IT delivery models. Customers utilize various types of services offered by specialized providers [12], [2], [16] ranging from Infrastructure-as-a-Service (IaaS), which offers remote access to computing resources such as virtual machines (VMs) and storage, to Software-as-a-Service (SaaS) which offers fully managed software functionality. Sharing the labor costs along with hardware, software and system management is expected to lead to significant reduction in compute cost both for individual users and enterprises. Compute Cloud also offers possibility to create new generation of IT services that can be easier integrated and delivered. It extensively leverages both virtualization technology [6], [11], [21], [1], [25], [5] and broad scale automation to minimize the delivery costs while keeping high quality of service.

At the same time, adoption of agile or iterative development model (e.g., Agile [23], DevOps [24]) means each service release causes a smaller change but occurs more often. This creates a smooth rate of progressive application changes requiring fast virtual machine image transfer between development and production sites for each release. Therefore, streamlining transfer of VM images between development and production cloud becomes critical to efficiency of the agile development process.

Both our studies [14] and those of other authors [13] show that there is significant degree of similarity across images in virtual machine libraries. It is due to sharing of software packages, such as operating system libraries, software packages, configuration settings, as well as, in many cases, user data that can be replicated in multiple images.

Motivated by the findings mentioned above, this paper proposes an efficient approach for transferring images between development and production sites that significantly reduces amount of network bandwidth consumed and also, as a consequence, reduces the time required to perform a transfer. The approach uses content similarity in image libraries to avoid transferring redundant data. We propose a set of algorithms that identify image similarity and allow efficient computation of the similarity matrices that are later used in deciding which data need to be transferred and which can be reused from other images already present at the target site. We have implemented the system agents as well as transmission manager tracking the state of the system in terms of how the images overlap and which images are present at each of the sites. The system reconstitutes images at production site by copying locally available image blocks and obtaining the remaining data from the development site. Moreover, we have formulated an analytical model that allows to quantify the degree of gain from the proposed algorithm and illustrate how it varies as a function of key system parameters, such as similarity of images within one development phase and across the phases, relative popularity of images (in terms of likelihood of transfer), and also fraction of images transferred in each development phase.

The remainder of this article is organized as follows. Section II introduces the architecture of the system as well as content de-duplication and image transfer algorithms. Section III presents results of the image library evolution study. Section IV formulates, validates, and explores analytical model of the system. Section V presents related work. Finally, Section VI concludes.

II. System Architecture and Algorithms

Fig. 1 shows the system architecture with development data center and the production data center. The development data center shows n images I_1 to I_n and the production data center shows a subset m of these images I_1 to I_m. The *Logical Image Library* represents the complete set of images in development and production data centers. Each distinct image in the library is identified by an uuid. Each data center has a subset of images from the *Logical Image Library* accessible locally. An image from this subset can be quickly instantiated on hypervisors within the data center. Images that need to

978-3-901882-50-0

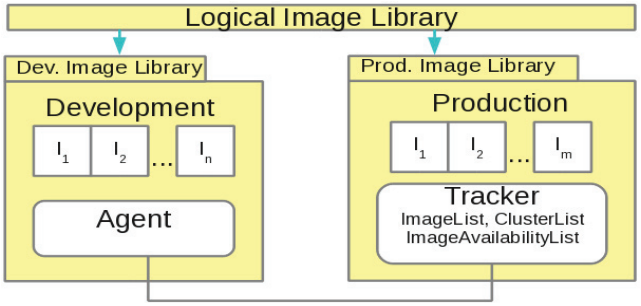


Fig. 1. Virtual machine image transfer system architecture.

Cluster Id	Image index										Cluster size MB
	9	8	7	6	5	4	3	2	1	0	
CL-01	1	0	0	0	0	0	0	0	0	0	238
CL-02	0	1	0	0	0	0	0	0	0	0	314
CL-03	0	0	1	0	0	0	0	0	0	0	182
CL-04	0	0	0	1	0	0	0	0	0	0	256
CL-05	0	0	0	0	1	0	0	0	0	0	476
CL-06	0	0	0	0	0	1	0	0	0	0	458
CL-07	0	0	0	0	0	0	1	0	0	0	317
CL-08	0	0	0	0	0	0	0	1	0	0	358
CL-09	0	0	0	0	0	0	0	0	1	0	70
CL-10	0	0	0	0	0	0	0	0	0	1	303
CL-11	1	0	0	0	1	0	0	0	0	0	419
CL-12	0	0	0	1	0	0	0	0	1	0	141
CL-13	0	0	0	1	0	0	1	0	0	0	174
CL-14	1	1	0	1	0	0	0	0	0	0	187
CL-15	0	0	0	0	1	0	1	0	0	0	319
CL-16	0	0	0	0	0	0	1	0	1	0	294
CL-17	0	1	0	0	0	0	0	0	1	0	440
CL-18	0	0	0	1	0	1	0	0	0	0	84
CL-19	0	0	1	1	0	0	0	0	0	0	116
CL-20	1	0	0	0	0	0	0	0	1	0	260

TABLE I
EXAMPLE IMAGE SIMILARITY MATRIX.

be instantiated in production but are not locally available in production data center need to be copied from development data center(s). The development data center has an *Agent* that is used for communication and transferring blocks of data between development and production data centers. The agent has direct access to the image library subset on it's data center. The Agent in production data center takes on the role of *Tracker*. It maintains the list of images, the overlap of blocks between images (clusters) and the status of images on development and production data centers.

As the transfer between development and production sites progresses, the image status can be: $Available$, $InProgress$, $Complete$, $Verified$ or $Reconstitutable$. The association of images to data centers on which an image is $Available$ is maintained by the $Tracker$. $Available$ status implies the entire image content is present in the data center and can be used to create virtual machine instances and also to reconstitute other images in that data center. $InProgress$, $Complete$ and $Verified$ are intermediate states before an image becomes $Available$. $Reconstitutable$ means that the image can by reconstituted using images present locally in the target data center. Image verification is done by maintaining an xor of non-zero block checksums computed as the blocks are received by the target data center in non-sequential manner. This eliminates the need to wait for the image to become $Available$ to start verifying it.

Cluster Representation

Clusters are meta-data representing sets of blocks that are common across a subset of images where each block is represented by its hash value. Images are decomposed into clusters. We represent clusters with a similarity matrix. TABLE I shows an example of a similarity matrix for a library consisting of 10 images and 20 clusters. The images are numbered *Image-0* to *Image-9*. The similarity matrix has been divided into two sections: the upper part is the set of 10 unique clusters *CL-01* to *CL-10* (each containing blocks present in only one of the images) while the lower part is that for shared clusters. The rightmost column shows cluster sizes (in MB). For example, *Image-9* consists of clusters *CL-01, CL-11, CL-14 and CL-20*. In order to compute the size (in terms of unique blocks) of the image cluster sizes are added (since they are non-overlapping). The total size of unique blocks to reconstitute *Image-9* is $238MB + 419MB + 187MB + 260MB + 444MB = 1548MB$. Actual reconstituted size of the image may be larger because of internal redundancy, i.e. the blocks with same hash occur multiple times within the image.

For simplicity, we refer to each cluster uniquely with a bitset of its constituent images. For example the *CL-14* is referred to as cluster *1101000000* (implementation uses uuid). The least significant bit in the cluster bitset is for *Image-0* on the right and the most significant *Image-9* on the left. The cardinality of this cluster is 3 because three of the bits in this cluster are 1. Thus *Image-6, Image-8*, and *Image-9* share the blocks present in this cluster. The number of times the blocks appear in the image could be different and the block positions may also be different within the images. Each block in the cluster must be present at least once in each of the three images.

When an image is added, the cluster name is extended with the most significant bit on the left. When an image is removed, the higher image indexes are shifted to the right. A cluster does not store or copy the actual data from the image. Instead it only contains the sha1 hashes and references to block numbers within the image.

Each cluster is persisted using a cluster blocks meta file *ClusterFile* consisting of records containing block numbers of distinct sha1s shared by images. Each image may contain one or more blocks with the same sha1 in different positions within the image. Each record can have different number of blocks depending on how many blocks have same sha1 in the image file. We store the total number of blocks, optional image indexes to reference the start position of the blocks and lastly the actual list of block numbers in the corresponding images.

Key Algorithms

The main algorithms for cluster maintenance on *Tracker* are the $clusterize\_image$ and $declusterize\_image$ that update the similarity matrix that is persisted in the $cluster\_list$. The add_image is used to add an image to a data center. When a new image is added, add_image calls the $clusterize\_image$ that updates the clusters. The $remove\_image$ is used to remove an image from a data center. When a particular image

is removed from all data centers, it is declusterized. The $make\_reconstitutable$ creates *Cluster Images* on source data centers for clusters not available on target data center and copies them to the target data center for later reconstitution. The $reconstitute\_image$ is used to reconstitute the target image from locally available images and can be invoked on the data center where the image status is $Reconstitutable$.

Create Content Digest Each data block is identified by its fingerprint computed from a collision resistant hash of the content of the data block. The method $content\_digest$ computes the sha1 hash for each block in the new virtual image and creates the block list sorted in ascending order of sha1 codes. We could use the sha-256 or sha-512 as stronger hash at the expense of higher storage. We create the content digest for every Image added to the *Logical Image Library*. Each record contains the sha1, number of blocks with this sha1 in the image and the list of block numbers. The number of blocks specifies the internal redundancy of a block, i.e. the number of times the same block appears within the image. The block numbers are the block positions in the image. The $create\_cluster$ creates a singleton cluster block file containing all the blocks positions for the single image.

Clusterize Image The method $clusterize\_image$ in Fig. 2 is used to add a new image to the cluster meta-information. It splits existing clusters, adds new block positions to existing clusters to include the image being added and creates a new singleton cluster. We compare the list of sha1s of blocks in image being added (returned by $content\_digest$) with the sha1s in each cluster in current cluster list stored on *Tracker*.

If no blocks from the image are present in a cluster, then we update the cluster to $0Cluster$, where 0 signifies image position in *Cluster BitSet*. Value of 0 means none of the blocks from the $0Cluster$ are present in the Image. If there are 1 or more blocks in common with the cluster, it results in splitting each cluster c into two smaller clusters, first $1Cluster$ is a subset of blocks from the image that are present in the cluster (returned by $intersect\_cluster$) and the second $0Cluster$ remaining blocks in the cluster (returned by $remove\_cluster$ that removed from c the sha1s belonging to the $1Cluster$). When the $1Cluster$ is created, the image id and block numbers from the $content\_digest$ of the image are added to the $1Cluster$ for the sha1 present in the image and cluster c. We add the remaining sha1s from cluster to the $0Cluster$. It is possible that all blocks in cluster are present in the image in which case only the $1Cluster$ is created. We remove the sha1s belonging to the $1Cluster$ from the sha1s of the image being added nc and continue the process until all clusters are handled. We create a new singleton cluster for remaining sha1s in nc.

Declusterize Image The method $declusterize\_image$ in Fig. 3 removes a given image from the library and results in combining clusters. We delete the singleton cluster if present. Then consider three checks for the clusters using the bit position of the Image to be deleted. First step, we look for $cluster\_pair(c0, c1)$ - pairs of clusters containing exactly the same images except the image index being deleted. We combine these two clusters into a single cluster by merging the sha1s using $declusterize\_merge$. We remove the imageId and blocks of the image being deleted from the new Cluster with new bitset with ImageId removed. Second step, if the $cluster\_pair$ is not found, then we look for the clusters containing the image. For every such cluster, we remove the block numbers of the image being deleted and create a new cluster with image index removed using the $declusterize\_image$. Third step is the rest of the clusters that do not contain the image being deleted. For these clusters, only the image index is deleted using $declusterize\_imageindex$, there are no sha1s for blocks to remove because none were present in these clusters. At the end of these three conditions we now have the list of clusters with the *ImageId* removed.

Make an Image Reconstitutable The $make\_reconstitutable$ in Fig. 4 extracts required clusters. It transfers the cluster images to target data center and updates the similarity matrix with the added cluster image information using $clusterize\_image$. The $select\_image$ allows cluster image data to be selected from production data center if available, else from development data center. If there is internal redundancy within the cluster, the blocks are copied only once over the wire.

Reconstitute an image The $reconstitute\_image$ in Fig. 4 generates the target Image file of correct size. The image overlap meta-data is used to determine mappings of blocks from source image to those in the target image being reconstituted. It retrieves the source block data from *Cluster Image* or *Standard Image* and writes the data to the target block number. The reads and writes all happen on the target data center because the $make\_reconstitutable$ has already saved required data to target data center in the form of *Cluster Images*. We have two ways to reconstitute the target image. In one case, it produces a sparse file where we seek and write only the blocks on the target image. In other case we sort the blocks are write them out into a stream filling holes with zeroes.

Empirical Evaluation

We have implemented the algorithms in C++ and Java and performed extensive experiments with transferring images between two data centers in IBM SmartCloud Enterprise [12] (Raleigh, USA acting as production site and Ehningen, Germany serving as development site). Experiments used subsets of images from two commercial libraries: VMWare Marketplace [22] containing 77 images and the IBM SmartCoud Enterprise [12] containing 90 images. The image sizes range from 800MB to 100GB and contain Linux and Windows operating systems and wide range of software packages. In case of IBM these were IBM Rational tools, Information Management tools (e.g., DB2 versions), Websphere Application Server, etc. In case of VMWare Marketplace we have used free open source software stacks from Bitnami [3] and Turnkey [20].

We have performed over 2000 image transfers and reconstitutions across the two data centers and verified the correctness and performance of the algorithms. For the SmartCloud Enterprise Image library with 100 images with size 1.75TB, a block size of 4KB and SHA1 with 20 bytes as the hash code, the

```
let newImg be the Image being clusterized
let oldCL represent the list of clusters for
    the current set of images in the library

function clusterize_image(newImg,oldCL)
  let newCL=∅;
  // Create new cluster representing
  // image being added
  let nc=content_digest(newImg);
  for each c in oldCL do
    // Cluster with intersecting content
    let 1Cluster=intersect_cluster(c,nc);
    newCL.add(1Cluster);

    // Cluster with non-intersecting content
    let 0Cluster=remove_cluster(c,1Cluster);
    newCL.add(0Cluster);

    // Remove processed content
    nc = remove_cluster(nc,1Cluster);
  end for;
  // Add singleton cluster representing
  // content unique to the new image
  newCL.add(nc);
  return newCL;
end function;
```

Fig. 2. Image clusterization algorithm.

```
let image be Image to be declusterized
let oldCL represent the list of clusters for
    the current set of images in the library

function declusterize_image(image,oldCL)
  Delete the singletonCluster for the image;
  let newCL=∅;
  for each cluster_pair(c0,c1) in oldCL do
    oldCL.remove(c1);
    oldCL.remove(c0);
    let newc=declusterize_merge(c0,c1)
    newCL.add(newc);
  end for;
  for each c in oldCL where image ∈ c do
    oldCL.remove(c);
    let newc=declusterize_image(c,image);
    newCL.add(newc);
  end for;

  for each c in oldCL where image ∉ c do
    oldCL.remove(c);
    let newc=declusterize_imageidx(c,image);
    newCL.add(newc);
  end for;
  return newCL;
end function;
```

Fig. 3. Image declusterization algorithm.

```
let t be identifier of the image
let S be Similarity Matrix
let dc be target Data center
procedure make_reconstitutable(t, S, dc)
  for each cluster c in S^t do
    let sourceImage=select_image(dc,c);
    if sourceImage is not present on dc then
      extract ClusterImage from SourceImage;
      transfer ClusterImage to dc;
      clusterize_image(sourceImage);
    end if;
  end for;
end procedure;
procedure reconstitute_image(t, S, dc)
  let f=create(t);
  for each cluster c in S^t do
    let srcImg=select_image(locDC,c);
    cf = open(srcImg);
    for each target block b in c,srcImg do
      cf.seek(b.srcBlockNr);cf.read(blockData);
      f.seek(b.trgtBlockNr);f.write(blockData);
    end for;
    close(cf);
  end for;
  close(f);
end procedure;
```

Fig. 4. Image reconstitution algorithms.

meta-data storage size is 6.45GB. Thus the storage overhead is less than 0.4%. We have measured the overlap across images in terms of unique versus total of 4KB blocks. IBM library has 13% of unique blocks, while VMWare Marketplace library 25% of unique blocks therefore giving significant opportunity for optimization. The measured gain (versus standard rsync-based transfer) in terms of network bandwidth consumption and transfer time is typically on the order of 50% and can be as high as 80%.

III. Image Library Evolution In DevOps Process

In order to perform a comprehensive analysis of potential gain from using the proposed algorithms, and also to guide the development of the analytical model presented in Section IV, we have explored (based on IBM Research Compute Cloud [7]) how the development libraries evolve over time. The library content evolution has significant effect on efficiency of redundancy based optimization. Image libraries evolve over time for two reasons: 1) changes to public (catalog) images, and 2) changes to private (user created) images. The remainder of this section discusses the latter since it is the primary source of image variation in a development environment context.

Fig. 5 shows an example image evolution depicting the iterations of base and development images. We show three base images: Redhat, Suse and AIX each with few iterations where updates were installed on the images. We study evolution of images in IBM Research Compute Cloud [7] over a 2 year period where total of 11,801 images were created. Out of these 670 images were were shared for use on production data center. We found 264 unique paths starting from base (root) images and diverging later. The longest path had 30 iterations. For illustration, the figure shows two paths of development images called App-A showing 16 iterations and App-B showing 24 iterations. Both these paths have 7

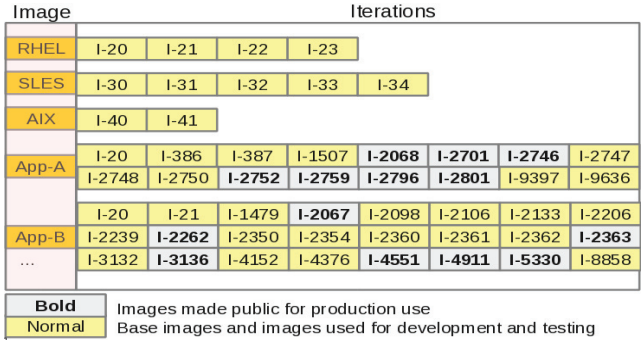


Fig. 5. Image Evolution.

Image	Iter-1	Iter-2	Iter-3	Iter-4	Iter-5	Iter-6	Iter-7
App-A	**I-2068**	**I-2701**	**I-2746**	**I-2752**	**I-2759**	**I-2796**	**I-2801**
Date	06/25 2010	10/08 2010	10/14 2010	10/14 2010	10/15 2010	10/19 2010	10/20 2010
Size GB	4.1	6.6	6.9	7.0	8.2	8.2	8.2
Overlap		93.8	96.3	96.3	96.2	96.3	96.6
New		73.5	8.7	3.7	14.9	3.4	3.7
App-B	**I-2067**	**I-2262**	**I-2363**	**I-3136**	**I-4551**	**I-4911**	**I-5330**
Date	06/25 2010	08/04 2010	08/16 2010	12/02 2010	03/22 2011	04/11 2011	05/03 2011
Size GB	4.1	45.0	46.0	46.0	49.0	50.0	56.0
Overlap		89.3	98.6	97.4	98.4	97.8	97.0
New		1471.4	1.5	3.6	4.5	2.1	4.6

Fig. 6. Image Timeline for App-A and App-B public iterations.

images made public shown in Fig. 6 that also shows the sparse image size in GB. Full size for each image was 72GB. We computed the clusters required to separately represent each path. For App-A image, number of Unique Blocks in clusters = 1,466,251 with 5.59GB distinct image data is sufficient to represent this library requiring 59MB overhead. Top 12 cluster sizes in GB were 1.36, 1.24, 0.54, 0.31, 0.31, 0.31, 0.31, 0.30, 0.29, 0.29, 0.11, 0.01. For App-B image, Number of

Parameter	Description
L	Number of image types in the library
P	Number of phases
T	Number of image transfers in one phase
α_i^p	Image provisioning frequency for image type i in phase p
S^i	Image similarity matrix for image type i
$F_i^{p,t}$	Random variable representing fraction of image available at the production site upon image i transfer assuming it is transfer t in phase p

TABLE II
MODEL'S PARAMETERS.

Unique Blocks in clusters = 6,807,062 with 25.97GB distinct image data sufficient to represent this library requiring 308MB overhead. Top 15 cluster sizes in GB were 14.15, 3.17, 1.38, 0.93, 0.89, 0.82, 0.79, 0.79, 0.68, 0.47, 0.37, 0.29, 0.27, 0.18, 0.11. Fig. 6 shows the date when the image was created, image size in GB, common blocks and new blocks as percentage change for each iteration. This shows greater than 96 percent overlap of clusters with previous iterations. New data results in new clusters that are carried forward in remaining iterations. The App-B also has large internal redundancy when the size grows from 4.1GB to 45GB in Iter-2. Most of this data is represented by the 14.15GB cluster.

IV. ANALYTICAL MODEL

The objective of analytical modeling is to find out what fraction of content required to reconstitute an image at the production site can be obtained locally based on the content of other images present at that site. There are several factors affecting this quantity, but the key one is how rapidly is the set of images evolving in terms of its content similarity. Intuitively, if the library changes very frequently, i.e., new images are being added or existing images are being updated with significant fraction of new content, then the gain from using the proposed transfer scheme will diminish. On the other hand, if the library image set is more stable, and the number of transfers between the development site and the production site is high, the potential gain will increase.

Our prior work [14] focused on modeling potential gain from using image de-duplication while provisioning virtual machine instances. In that case the image library was assumed to be static, therefore the system was in steady-state from the perspective of image content overlap and also provisioning frequencies. Our current analysis extends that model for case of libraries that evolve over time. We use the extended model to quantify the potential gain from redundancy-based synchronization of development and production sites. The key parameters and variables used in the model are presented in TABLE II. Consider a set of virtual machine images in existence throughout a longer development/deployment cycle, for example, 1 year. Denote the number of such images as L. Divide the time into development phases, usually related to a one or two week "sprint" delivering specific functionality. Of course, at any particular time not all of the images are in existence - for example, at the beginning of the development cycle only some initial images are available. The set of available images grow throughout the cycle. We model this behavior as a time-varying image popularity probability vector. Precisely, denote transfer probabilities of an image type $i = 0, \ldots, L$ during phase p by α_i^p. Note that for a given phase p $\sum_{i=1}^{L} \alpha_i^p = 1$. However, as the time evolves, the probabilities change reflecting increased demand for newly added images and updated images, while decreasing demand for the older ones that are no longer used in the development cycle.

Similarly as in our prior model [14], an image i consists of n_i non-overlapping clusters s_k^i for $k = 1, \ldots, n_i$. In addition, for each image i we have a similarity matrix S^i of size n_ixL such that:

$$S_{k,l}^i = \begin{cases} 1 & s_k^i \text{ is part of image } l \\ 0 & \text{otherwise} \end{cases} \tag{1}$$

Denote by $F_i^{p,t} \in [0,1]$ fraction of blocks required to rebuild image i that are available in the production site assuming it is transfer number t in phase p. We are interested in expected value of $F_i^{p,t}$, denoted by $E[F_i^{p,t}]$. Let $A_k^{i,p,t}$ denote the event that upon request to transfer image of type i, assuming it is transfer t in phase p, the production site contains cluster s_k^i. Then:

$$E[F_i^{p,t}] = \sum_{k=1}^{n_i} \frac{|s_k^i| P(A_k^{i,p,t})}{\sum_{l=1}^{n_i} |s_l^i|} \tag{2}$$

with $|s|$ being size of cluster s.

The conditional availability $P(A_k^{i,p,t})$ can be computed as complement of not choosing a given image in all prior transfers (therefore it is not available in production) for images that overlap with image being transferred. Precisely:

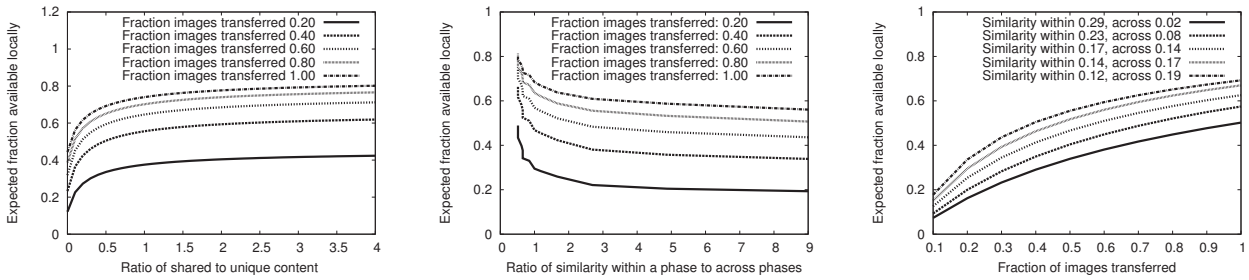
$$P(A_k^{i,p,t}) = 1 - \prod_{\substack{l=1 \\ s.t. \\ S_{k,l}^i=1}}^{L} \left[(1-\alpha_l^p)^t \prod_{j=1}^{p-1} (1-\alpha_l^j)^T \right] \tag{3}$$

Model Validation

We have validated the model by comparing the results with discrete event simulator. The simulator is implemented in C++ and simulates image transfers between development site and the production site. It keeps track of which images have been transferred so far, randomly selects next image to be transferred based on image popularity in the current phase, and outputs fraction of the image overlapping with other images that are already in production site. The parameters tested included varying degrees of similarity across images, number of images transferred within a phase ranging from 1 to the number of images added in this phase, and varying degrees of image popularity evolution.

Empirical Evaluation and Conclusions from the Model

We have examined the effect of key parameters on the fraction of available content in production site when image transfer is requested. Fig. 7a shows effect of ratio of shared to unique content in the library for five different levels of image transfer fractions. Content availability at the production



(a) Effect of unique fraction (b) Effect of in-phase similarity (c) Effect of number of transfers

Fig. 7. Analytical model insights: effect of fraction of unique content (a), effect of similarity among images within a phase versus across phases (b), and effect of number of transfers within a phase (c).

site grows with the degree of content sharing across the library. It also grows with the number of image transfers per development phase. The more the transfers, the more of the content becomes available for reconstitution. Fig. 7b depicts effect of ratio of similarity within a phase to similarity across phases. The more shared across the phases, the more opportunity to find local content. This reflects the difference between the situation when newly added images have a lot or little in common with images created in previous development phases. Content availability is strongly affected by number of images transferred. Finally, Fig. 7c shows effect of fraction of images transferred in each phase (among the ones added in that phase). That has primary impact on the availability of content. Intuitively, frequent transfers reap increased benefits because the odds of finding required content in production increase.

V. Related Work

The important area of related work is redundancy detection allowing to identify parts of virtual machine images that are shared between multiple images in the library. A very good example of such approach is Mirage system [18]. Exploiting image similarity to enable version control for Virtual Machine snapshots is explored in [19]. A very good discussion of the redundancy elimination can be found in [8]. Other interesting sources on this topic are [9], [4]. The proposed approach uses block level content de-duplication and focuses on mechanisms to optimize the representation of image redundancy for efficient transfers. It also focuses on studying the efficiency of the virtual machine transfer rather than de-duplication itself. VMTorrent [17] uses unmodified BitTorrent technology to improve virtual appliance distribution. It allows downloading of the content from peers that already have it therefore reducing download time and also load on the systems of virtual appliance publisher. However, VMTorrent does not leverage similarity across images. Therefore, in common case of significantly redundant virtual images, it provides much smaller gain in terms of both download time and consumed bandwidth than the approach proposed in this paper. LiveDFS [15] is a live deduplication file-system that reduces the storage space by removing redundant data copies.

VI. Conclusions and Future Work

We have presented and evaluated a virtual machine transfer system that uses similarity across virtual machine images to minimize amount of data that has to be transmitted between development and production site during DevOps agile development cycle. The clusterization and declusterization algorithms are validated. The key parameters affecting the performance are degree of similarity among the virtual machines, within and across development phases, the fraction of VMs transferred within each phase, and relative popularity of images (in terms of likelihood of transfer). The system is implemented in a testbed and also an analytical model is formulated, validated and explored. We have also studied image libraries to explore the similarity levels as well as typical time dynamics of how libraries evolve. Plans for future research include exploring the integration of the proposed algorithm with DevOps planning processes to further optimize development productivity.

References

[1] Kernel Virtual Machines. Online. http://sourceforge.net/projects/kvm.
[2] Amazon Inc. Amazon Elastic Compute Cloud. Online, 2009. http://aws.amazon.com/ec2/.
[3] Bitnami. Bitnami. http://bitnami.org/, 2012.
[4] J. Bonwick. ZFS Deduplication. Online, 2009. http://blogs.sun.com/bonwick/entry/zfs_dedup.
[5] Microsoft Corp. Microsoft Virtualization. Online, 2011. http://www.microsoft.com/virtualization/.
[6] R. Creasy. The Origin of the VM/370 Time-Sharing System. *IBM Journal of Research and Development*, 1981.
[7] Jim Doran, Frank Franco, Dilma M. Da Silva, and Alexei Karve et al. Rc2 a living lab for cloud computing. *IBM Research Report*, 2010.
[8] Fred Douglis, Jason Lavoie, John M. Tracey, Purushottam Kulkarni, and Purushottam Kulkarni. Redundancy elimination within large collections of files. In *USENIX Annual Technical Conference, General Track*, 2004.
[9] EMC. Data Domain Replicator Software, Network-efficient replication for backup and archive data. Online, 2011. http://www.datadomain.com/pdf/DataDomain-Rep-Datasheet.pdf.
[10] Gartner Inc. Special Report on Cloud Computing. Online, 2011. http://www.gartner.com/technology/research/cloud-computing/.
[11] R. Goldberg. Survey of Virtual Machine Research. *in IEEE Computer Magazine*, 1974.
[12] IBM. IBM SmartCloud. Online, 2011. http://www.ibm.com/cloud-computing/us/en/.
[13] K. R. Jayaram, Chunyi Peng, Zhe Zhang, Minkyong Kim, Han Chen, and Hui Lei. An empirical analysis of similarity in virtual machine images. *Middleware*, 2011.
[14] A. Kochut and Alexei Karve. Leveraging local image redundancy for efficient virtual machine provisioning. *IEEE Network Operations and Management Symposium*, 2012.

[15] Chun-Ho Ng, Mingcao Ma, Tsz-YeungWong, Patrick P. C. Lee, and John C. S. Lui. Live deduplication storage of virtual machine images in an open-source cloud. *Middleware*, 2011.
[16] Rackspace. Rackspace Cloud. http://www.rackspace.com/cloud/, 2011.
[17] Joshua Reich and Oren Laadan et. al. Vmtorrent: Virtual appliances on-demand. *ACM Sigcomm*, 2010.
[18] 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 Proc. of USENIX Virtual Execution Environments Workshop*, 2008.
[19] Chung Pan Tang, Tsz Yeung Wong, and Patrick P. C. Lee. Cloudvs: Enabling version control for virtual machines in an open-source cloud under commodity settings. *IEEE NOMS*, 2012.
[20] Turnkey. Turnkey. http://www.turnkeylinux.org/, 2012.
[21] VMware. Online. http://www.vmware.com.
[22] VMware Inc. VMware Virtual Appliance Marketplace. Online, 2011. http://www.vmware.com/appliances/.
[23] Wikipedia. Agile software development. http://en.wikipedia.org/wiki/Agile-software-development, 2012.
[24] Wikipedia. DevOps Agile Development Process. http://en.wikipedia.org/wiki/DevOps, 2012.
[25] Xen. Online. http://www.xensource.com.