Minimizing Latency in Serving Requests through Differential Template Caching in a Cloud

Deepak Jeswani* Manish Gupta* Pradipta De* Arpit Malani† Umesh Bellur†
* IBM Research, New Delhi, India
† Indian Institute of Technology, Bombay, India
{djeswani, gmanish, pradipta.de}@in.ibm.com, {malani, umesh}@cse.iitb.ac.in

***Abstract*—In Software-as-a-Service (SaaS) cloud delivery model, a hosting center deploys a Virtual Machine (VM) image template on a server on demand. Image templates are usually maintained in a central repository. With geographically dispersed hosting centers, time to transfer a large, often GigaByte sized, template file from the repository faces high latency due to low Internet bandwidth. An architecture that maintains a template cache, collocated with the hosting centers, can reduce request service latency. Since templates are large in size, caching complete templates is prohibitive in terms of storage space. In order to optimize cache space requirement, as well as, to reduce transfers from the repository, we propose a *differential template caching* technique, called *DiffCache*. A difference file or a patch between two templates, that have common components, is small in size. DiffCache computes an optimal selection of templates and patches based on the frequency of requests for specific templates. A template missing in the cache can be generated if any cached template can be patched with a cached patch file, thereby saving the transfer time from the repository at the cost of relatively small patching time. We show that patch based caching coupled with intelligent population of the cache can lead to a 90% improvement in service request latency when compared with caching only template files.**

Keywords

Cloud Computing; Virtual Appliance; Cache; Block-based differencing and patching;

I. Introduction

A Virtual Appliance (VA) is a virtual machine image file consisting of pre-configured operating system environment, along with application(s) installed, configured and tested in the environment. The use of VA as a basic unit of deployment in cloud introduces a new challenge for cloud service providers. VA image files are large, often tens of GigaBytes in size. When a request for a VA template is received, the entire file must be transferred to the hosting physical server for instantiation. Since hosting centers are geographically dispersed, and image repository may be centralized, therefore, accessing a repository over the Internet to fetch a GigaByte sized file can degrade server provisioning speed significantly. Currently, several vendors, like IBM, Microsoft, Oracle are making their bundled products available as Virtual Appliances, as found in Amazon Machine Images [1]. Although all these images are hosted centrally by Amazon, we envision a scenario where each vendor can maintain its own globally addressable repository, and a cloud hosting center maintains an index to the repository to fetch the file when required. Fig. 1 presents this scenario where different vendors maintain individual image repositories. A delivery location contacts a repository to service a request. However, this also poses the challenge of transferring large machine image files over the Internet.

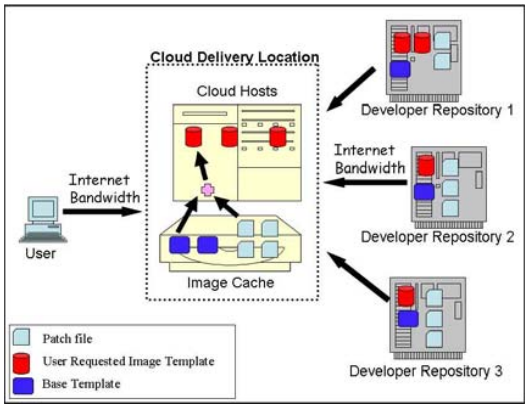

Fig. 1. Cloud Delivery Location hosting the virtual servers are connected to vendor repository sites over the Internet. Image Caching location can act as intermediate storage areas, collocated with the hosting site, that can cache frequently requested machine images.

A well-known practice in content distribution networks is to introduce a cache near the edge servers. Similarly, an *Image Cache* can be collocated with the delivery center to store some of the popularly requested templates. However, storing number of templates can be prohibitive in terms of storage.

Image templates often have high degree of commonality [2], [3]. In this work, we exploit the presence of this commonality among template files to generate difference files or patches between two templates. A patch file can be applied on another template to generate a new template. Instead of caching large templates, we can cache patches and templates, and effectively cater to a larger set of template requests by paying a small cost of patching time, while saving the time to fetch the complete template file from the repository. However, selection of appropriate patches and templates, under given template request history and available cache size, is essential for realizing the benefit of the approach. We propose the *DiffCache* algorithm, that computes a cache composition with the objective of minimizing transfers from repository, thereby reducing request service time.

978-0-7695-4755-8/12 $26.00 © 2012 IEEE
DOI 10.1109/CLOUD.2012.17

A. Motivation

We analyzed a request trace from IBM's Research Compute Cloud (RC2) [4] to assess the merits of our approach. The trace, collected over 1 year, contains 10211 requests for VMs, with 1088 distinct image templates. Only 26% of the requests were serviced from 15 most frequently requested templates. *Thus to register a high cache hit rate, a large number of templates must be cached, leading to high storage cost.*

Although the number of image templates is quite high, and grew exponentially over time, it was observed that there are few original base templates from which they have evolved. Usually a user picks an image template from a catalogue. For example, a template with specific Linux version, then installs additional software components in it, customizes it for a group, and saves it in repository for either public or private use. *Each group can be thought of as a tree starting with a base template and successive levels refining the level above it along different directions, leading to small change in the parent machine image file.* In our trace, we found 138 distinct groups of image templates. The average depth of the trees was 14.55, thereby presenting an opportunity to generate small sized difference files between image templates belonging to the same tree. Average size of a template was 49 GB, whereas patches were in MegaBytes (average size of patch was 350 MB).

We evaluated the efficacy of generating patches between two similar templates. We created a Linux template, and then installed two applications, viz. DB2 and WebSphere Application Server (WAS), separately to generate two new image templates. In another image template we installed both the applications together. Thus we had 4 image templates, each of which were approximately 7 GB in size. We computed the patches between each pair of templates using an open source block-based differencing tool, rdiff [5]. Table I shows the patch sizes which are mostly MegaBytes (MB) in size, as opposed to the original templates which were in GBs. The patch generation process takes time in order of few minutes, and can be performed offline. Table II shows the time to perform the patching operation to recreate the target template back from the basis template. *Time to patch is typically proportional to the size of the patch, and is of the order of few minutes, thereby making it a promising way of caching image templates*.

Base Template	Target Template	Patch Size (in MB)	Patch Generation Time (in sec)
Linux	DB2+Linux	576	798
Linux	WAS+DB2+Linux	1218	1468
Linux	WAS+Linux	713	902
DB2+Linux	Linux	66	217
DB2+Linux	WAS+DB2+Linux	700	1144
DB2+Linux	WAS+Linux	705	1021
WAS+DB2+Linux	Linux	62	210
WAS+DB2+Linux	DB2+Linux	61	206
WAS+DB2+Linux	WAS+Linux	46	180
WAS+Linux	Linux	61	202
WAS+Linux	DB2+Linux	566	899
WAS+Linux	WAS+DB2+Linux	545	796

TABLE I
TABLE SHOWS PATCH GENERATED USING A BASE TEMPLATE AND A TARGET TEMPLATE. THE TARGET TEMPLATE CAN BE RECREATED BY APPLYING THE PATCH ON THE BASE TEMPLATE.

Base Template	Target Template	Patching Time (in sec)
Linux	DB2+Linux	240
Linux	WAS+DB2+Linux	209
Linux	WAS+Linux	177
DB2+Linux	Linux	188
DB2+Linux	WAS+DB2+Linux	269
DB2+Linux	WAS+Linux	224
WAS+DB2+Linux	Linux	176
WAS+DB2+Linux	DB2+Linux	225
WAS+DB2+Linux	WAS+Linux	193
WAS+Linux	Linux	177
WAS+Linux	DB2+Linux	213
WAS+Linux	WAS+DB2+Linux	210

TABLE II
TABLE SHOWS THE TIME TO REGENERATE A TARGET TEMPLATE FROM A BASE TEMPLATE BY APPLYING APPROPRIATE PATCH. IN COMPARISON, ANY TARGET TEMPLATE, WHICH ARE APPROXIMATELY 7GB IN SIZE, WILL TAKE 95 MINS TO FETCH FROM REPOSITORY OVER A 10 MBPS WAN CONNECTION.

B. Contribution

The use of patches to reduce cache space requirement, as well as, increase cache hit ratio has not been studied earlier. We propose the DiffCache algorithm that makes an optimal selection of patches and templates to be stored in the cache that reduces the average request service time by minimizing number of file transfers from repository to compute hosts.

The rest of the paper is organized as follows. In Section II, we formulate the problem of composing cache with patches and templates. In Section III, we present the DiffCache algorithm. We evaluate our approach in Section IV. Section V compares our approach with existing work. Finally, we conclude in Section VI.

II. PROBLEM DESCRIPTION

In this section, we illustrate the problem of caching templates and patches at the hosting location. In this work, we assume that the total cache space is distributed across all the compute hosts. Finally, we present the problem as a linear optimization problem.

A. Problem Illustration

The template and patches to cache are computed at periodic intervals. The procedure uses the request arrival history for the templates to compute relative importance of each template in future. Different mechanisms, like most-popular templates, or most-recently-used templates, can be used to generate an ordering of the templates [6]

Given a set of templates ordered by likelihood of being requested, $I_1, \ldots, I_n$, and a fixed cache space, one can start caching the most important templates, starting from I_1, till the cache space is completely utilized. Let us assume that we dedicate half of the cache space to store templates, and the remaining half to store patches. The caching will then begin by first storing templates from the ordered list, say I_1, I_2, I_3, I_4. Next most important template is I_5. If I_5 can be generated

by patching any of I_1 to I_4, then the corresponding patch is selected for caching. By storing a patch, much smaller in size than the template I_5, we save cache space. Similarly, we can store the patches for the other templates, till the cache space runs out. For a given cache space, this method will be able to effectively service requests for more template types, than it can if only templates are stored. Additional cost is in terms of patching, which is negligible compared to fetching time of complete template from the repository.

However, in this naive scheme, we blindly apportion the cache space equally for template and patches. If we design the utility function for caching carefully, then it is possible to make a better selection of templates and patches, that will maximize the cache hits, thereby minimizing the service time per request. We now present the optimization formulation.

B. Problem Formulation

One of the primary notions used in the formulation is that an image template requested at a node can be generated using another template present on a different node, and a patch on another node. We can represent this notion as, $I_j@H_1 := I_i@H_2 \otimes Patch_{ij}@H_3$, where Template I_j is requested at host H_1, and is generated using Template I_i fetched from host H_2, and patch $Patch_{ij}$ retrieved from host H_3. In case, the necessary files are not cached, the repository is contacted for retrieving the appropriate files.

Let there be N different types of templates that can be requested, and K is the number of hosts. Of these hosts, index 0 is used to represent the repository host, and $(1, \ldots, K)$ denotes the hosts in the cloud hosting center. We use the indices i and j to indicate Templates, and it varies from $(1, \ldots, N)$. Indices p, q denotes the index of the hosts hosting the base template and patch respectively, and vary from $(0, \ldots, K)$. r is also used as host index to denote the host where a new template is requested, and therefore, it varies from $(1, \ldots, K)$. Other notations used are described in Table III.

Our specific problem is to come up with a set of base templates and patches, and their placement on the available hosts. Multiple constraints must be satisfied in order to define the problem precisely.

Unique-Solution Constraint: For a request for a template at a host, multiple ways are there to make the template available. Of these there is exactly one way to ensure the minimum cost. We use a decision variable, S_{ijpqr}, to indicate the method of constructing a template at a specific host. The constraint on the decision variable, which we term as the *unique-solution constraint*, can be expressed as,

$$\sum_i \sum_p \sum_q S_{ijpqr} = 1, \forall j, r \tag{1}$$

Thus we restrict the number of solutions (that is number of 1's in S_{ijpqr}) for a template I_j requested at host H_r to be 1.

Usage Constraint: Note that in order to generate a template I_j at H_r, we can use the template I_i at H_q and $patch_{ij}$ at H_p. If there are several templates in the setup, which uses I_i from H_q and $patch_{ij}$ from H_p, then the *usage count* of these entities must be accounted for such that they are not purged mistakenly when there are active users of them. Our next constraint encodes the idea of *usage count* by accounting for the usage of a template and a patch when a lookup decision is finalized. We call this constraint *Usage constraint*, and is expressed as,

Notation	Arity	Explanation
I_i	$i \in \{1 \ldots N\}$	Denotes the i_{th} image template
$size_{ij}$	$i \in \{1 \ldots N\}$, $j \in \{1 \ldots N\}$	Denotes the size of the patch to construct I_j from I_i
$size_i$	$i \in \{1 \ldots N\}$	Denotes the size of template I_i
H_k	$k \in \{1 \ldots K\}$	Denotes k_{th} host in the cloud hosting center
C_k	$k \in \{1 \ldots K\}$	Denotes space available for caching on k_{th} host
f_{jr}	$j \in \{1 \ldots N\}$, $r \in \{1 \ldots K\}$	The request arrival rate for template I_j at node H_r
S_{ijpqr}	$i \in \{1 \ldots N\}$, $j \in \{1 \ldots N\}$, $p \in \{0 \ldots K\}$, $q \in \{0 \ldots K\}$, $r \in \{1 \ldots K\}$	This is a decision variable, which when set to 1 implies that $I_j@H_r := I_i@H_q \otimes Patch_{ij}@H_p$

TABLE III
TABLE SHOWS DIFFERENT NOTATIONS USED IN THE PROBLEM FORMULATION.

$$S_{iiqqq} = \lceil \frac{\sum_j \sum_p \sum_r S_{ijpqr}}{N.(K+1).K} \rceil, \forall i, q \tag{2}$$

$$patch_{ijp} = \lceil \frac{\sum_q \sum_r S_{ijpqr}}{(K+1).K} \rceil, \forall i, j, p \tag{3}$$

Note that, S_{iiqqq} denotes that Template I_i is on H_q; $patch_{ijp}$ denotes that patch to generate Template j from Template i is on H_p.

The left-hand side of Eqn. 2 denotes whether the template I_i should be placed on host H_q. Now the template will be cached at host H_q (value of variable S_{iiqqq} will be 1), if it is marked to be used at least once. The three summations on the right-hand side determines if template I_i at host H_q is used for any request for template I_j at host H_r. Since the value of the variable S_{iiqqq} should be either 1 or 0, we divide the total count by maximum number of times a cached entity can be used, which is given by the product of number of hosts and number of templates. Thus the expression is bound within 0 and 1. We take ceiling of this expression to satisfy the integrality constraint of S_{iiqqq}. Eqn. 3 can be understood similarly, where $patch_{ijp}$ must be an integer in [0,1].

Capacity Constraint: Capacity constraint ensures that the available space for cache on each node is not exceeded after placing the templates and patches. There can be multiple templates and patches cached on each node, and this constraint ensures that the total space used is less than the cache space.

Capacity constraint is expressed as,

$$\sum_{j} S_{jjrrr}.size_j + \sum_{i}\sum_{j} patch_{ijr}.size_{ij} \leq C_r, \forall r \quad (4)$$

Eqn. 4 finds the cache usage at host H_r. The space is consumed by base template files and patch files. As we saw in Eqn. 2 and Eqn. 3, we get whether base template I_j and $patch_{ijr}$ are getting used by looking at S_{jjrrr} and $patch_{ijr}$ respectively. If they are getting used, it implies that they are cached at host H_r. We sum up the sizes of all templates and patches on host H_r to compute the total space consumed on H_r and this should be less than H_r's capacity for each H_r.

Cost Function: The cost of servicing a request for a template, I_j on a host, H_r, is dependent on the availability of the templates as per the placements made in the previous round of cache computation. There are several ways to register a cache hit: (i) Template I_j is available on H_r; (ii) Template I_j is available on a peer node; (iii) Template I_j can be generated using I_i and $patch_{ij}$, and I_i and $patch_{ij}$ are available on H_r or any of its peer nodes. If all these cases fail, then one may need to either fetch the $patch_{ij}$ from the repository (when the base template I_i is available in one of the peer nodes), or may need to fetch the template I_j from the repository.

Based on how a request will be serviced, given a cache composition, one can compute the latency for servicing a request, or the total bytes transferred. Minimization of either of these two costs can be achieved by the same cache composition. When a request is handled locally on a server, the total time spent is in copying the image file from the local cache, in addition to patching, if required. Let us call this transfer rate *LocalTx*. If a peer server is contacted, then the transfer is over the local area network, and we call the transfer rate *PeerTx*. When repository is involved, then the transfer rate is termed *RepoTx*. Note that *LocalTx* $>$ *PeerTx* $>>$ *RepoTx*. In order to compute the latency of servicing a request, we need the size of the templates and patches used, and the transfer rates. If we are interested in the bytes transferred, only the size is sufficient. We use a cost matrix, γ which stores the value of servicing a request for a template at a host. As we described earlier, the decision variable, S_{ijpqr}, denotes a specific combination to construct a requested template at a node. Therefore, given the cost of the construction, stored in the γ matrix, we can easily compute the total cost for servicing all the requests. We also assume that when request arrival history is available, we can also compute the request load for each template on a node, as denoted by f_{jr}. Finally, the objective function minimizes the total cost of servicing all the predicted requests,

$$min \sum_{j=1}^{N}\sum_{r=1}^{K}\sum_{i=0}^{N}\sum_{p=0}^{K}\sum_{q=0}^{K} f_{jr}.S_{ijpqr}.\gamma_{ijpqr} \quad (5)$$

Eqn. 5 finds the total cost of servicing each request for I_j at host H_r. This computes an aggregate cost since we take frequency of templates being requested on a host, f_{jr}, in consideration. The variable S_{ijpqr} encodes the solution to be used to service a request for template I_j at host H_r and variable γ_{ijprq} gives the cost corresponding to a solution.

Thus, we have a Integer Linear Program to solve to get the optimal placement of the templates and the patches to minimize the total cost of servicing requests.

III. DiffCache Algorithm

In this section, we introduce the algorithm for pre-filling the cache with (complete) base templates, and patches. We first show that the specific problem is an NP-hard problem. Then we present a heuristic for solving the problem.

A. Problem Complexity

Assume that the frequency of requests for each template type on every host is identical. With this assumption, let us minimize the cost function for a given collection of base templates and patches. Let us further assume that there is only 1 host and N template types. The cache space on the available node is C. Now, the problem boils down to packing the maximum number of base templates and patches in the available space of size C. One can observe that this is a specific case of bin-packing problem, or more precisely is the 0-1 Knapsack problem. Bin-packing problem is known to be a NP-hard problem. Thus a special case of our problem, where the frequency is identical, and number of host is set to 1 with fixed space, boils down to a bin-packing problem. Therefore, the generalized version of the cache computation problem is also NP-hard.

B. DiffCache

We present a heuristic to compute the best composition of the cache, given multiple input parameters, like frequency of arrival of requests for each template on a node, cost of servicing a request on a node where the sizes of templates and patches are known.

The algorithm computes the placement over multiple rounds. In each round, the requirement is to *select a template on a host, plus patches along with their hosts*. Each round runs in two steps. In the first step, a template is placed on a node, while satisfying capacity constraint (Line 16). Given this placement, we search for the patches across all nodes which will minimize the total cost ($PickPatches$ at Line 25). We have to ensure that all combination of template and node is considered before finalizing a round ($PickImageAndPatches$ at Line 12). At each round, a new template is picked and the steps are repeated till the cache space is exhausted. The complete algorithm is presented in Algorithm 1.

At the end of each round, we have to update the following data structures ($UpdateDataStructures$)

1) *LookUpTable*: This is a hashtable which returns the templates and patch, along with the hosts, which could be the repository, from which to fetch the respective items. The key to the hashtable is the template id and the host on which it is required. At each round, when new templates and patches are added to cache, the lookup can undergo a change. Use of the hashtable ensures that there is exactly one solution to construct a template on

Algorithm 1 Algorithm to compute cache composition, comprising of templates and patches to be kept at each node

Require: Init Request-Arrival-Frequency-Table from history
Require: Init CostMatrix (γ)
Require: Init Template and Patch sizes,
Require: Init Per-Node-Cache-Space (CS)
Require: Init SolutionLookUpTable(SLT) */* a hashtable that returns how to generate requested template at a node */*

```
1: procedure DiffCache
2: COST := cost of serving all requests from repository
3: spaceAvailable = true
4: while spaceAvailable == true do
5:    /* PickImageAndPatches returns updated data structures */
6:    PickImageAndPatches
7:    if new template or patch added then
8:       /* update COST, CS and SLT based on the return values */
9:       UpdateDataStructures /* Global Data Structures are updated */
10:   end if
11: end while

12: procedure PickImageAndPatches
13: benefit = 0
14: for all Template i do
15:    for all Node k do
16:       if (I_i@H_k == false) && (CacheSpace on H_k ≥ size_i) then
17:          newBenefit = PickPatches(i, k)
18:          if (newBenefit > benefit) then
19:             UpdateDataStructures /* Update Local Data Structures */
20:          end if
21:       end if
22:    end for
23: end for
24: return LocallyUpdatedDataStructures

    /* PickPatches determines which patches to pick, given that I_i on H_k */
25: procedure PickPatches(inti, intk)
26: newCost = total cost after placing I_i on H_k
27: adjust space on H_k due to I_i
28: adjust RefCnt for I_i@H_k
29: store new lookupEntry for I_i@H_k
30: for all Nodes q, q ≠ 0, q ≠ k do
31:    for all Templates j, j ≠ i do
32:       bestCostReduction = 0
33:       for all Nodes r = 0 → K do
34:          /* the costs are looked up from the costMatrix */
35:          costReduction = newCost - oldCost;
36:          if (costReduction > bestCostReduction) then
37:             newSol = save current patch placement
38:          end if
39:       end for
40:       if (newSol is not empty) then
41:          UpdateDataStructures /* Update Local Data Structures */
42:       end if
43:    end for
44: end for
45: return LocallyUpdatedDataStructures
```

a host. In a real-world deployment with large number of templates, one must use a database to store the solutions to speed up the search.

2) *Usage-Count*: For each template on a host, and a patch on a host, we maintain the usage count as the lookup entries are updated. This is essential to ensure that a patch or a template is not removed from the cache assuming it is not in use.
3) *Available-Space*: After each round, the available space on each node for caching is also updated based on the new placements.

IV. Evaluation

In this section, we evaluate the benefits of patch based caching in cloud delivery environments using DiffCache as the cache filling algorithm. The techniques are implemented on a small testbed to demonstrate the feasibility of the approach under typical network conditions.

A. Comparison of Caching using DiffCache

1) Parameters and Initialization: The evaluation is dependent on several parameter values, viz. cache size, average size of template and patch files, workload distribution. In the experiments, cache size is varied from 10 GB to 150 GB per host, with the default value used being 45 GB per host. Number of hosts is varied from 3 to 10, and number of templates from 5 to 50. Unless otherwise stated, default value for number of templates is 10, and number of hosts is 3.

Templates and Patch File Initialization: Template sizes range from 40 GB to 50 GB, and are chosen randomly from a uniform distribution. Given two templates, I_i and I_j, sizes of patch files, $patch_{ij}$ and $patch_{ji}$, depend on the relative sizes of I_i and I_j. If $I_i > I_j$, then it implies that template I_j has been built over template I_i by installing additional components. $patch_{ij}$ is a difference in size of the two template files. On the other hand, we set $patch_{ji}$ to a small size of 10 MB since the patch file needs to maintain only information about sections to be deleted from template I_j to generate I_i. This logic is based on experimental data from Table I.

Initializing Cost Matrix(γ): Cost of making a template I_j available at H_r is dependent on template I_i, hosted at H_q, and a patch, $patch_{ij}$, hosted at H_p. Given the values of the five-tuple $\{i, j, p, q, r\}$, we compute a cost value for making a template available at a node. For our experiments, cost is in terms of time to service a request. Cost is the sum of *time to fetch individual components, viz. template and patch file, in addition to the time to perform patching*, if required. We set the bandwidth values for network links between repository to hosting center to be 10 Mbps, within the hosting center to be 200 Mbps, and patching and copying time within a host translates to a bandwidth of 400 Mbps. The patching time is based on a linear function whose parameters are derived from the experiment data in Table I. Given the sizes of templates and patches, and the bandwidths, we can compute the time to generate a template at a node for different scenarios, when a template and/or a patch is fetched from repository, or peer node, or copied from local cache on the host.

Workload Generation: The workload for the simulation experiments are generated as a random sequence of requests for templates. These requests are then dispatched by a load balancer to the available nodes.

2) Evaluation Results: In our evaluation, we report the average service time to serve a request. DiffCache algorithm makes the selection of entities to be placed in cache for both scenarios, when patches and templates are cached, as well as, when only templates are cached.

Fig. 2 shows the benefit of patch-based caching against non-patch based approach. When patches are not cached, the cost reduces in steps, where each step corresponds to the time when there is sufficient space to cache one more template. But when patches are available for caching, the cost reduces

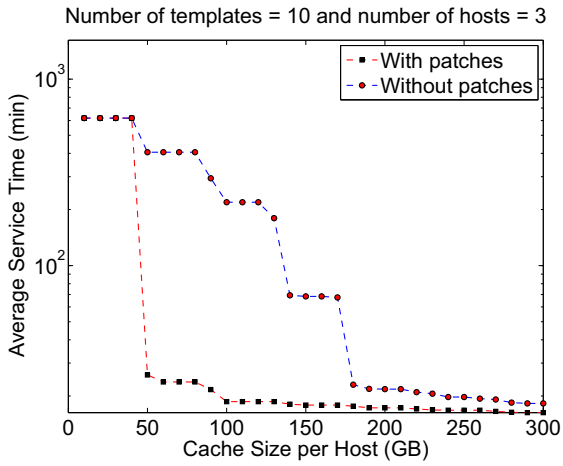

Fig. 2. A comparison of average service time for two caching scenarios. In `without patch caching`, the cache stores complete templates, whereas in `with patch`, the cache stores templates, and patches for generating new templates by patching in-cache templates.

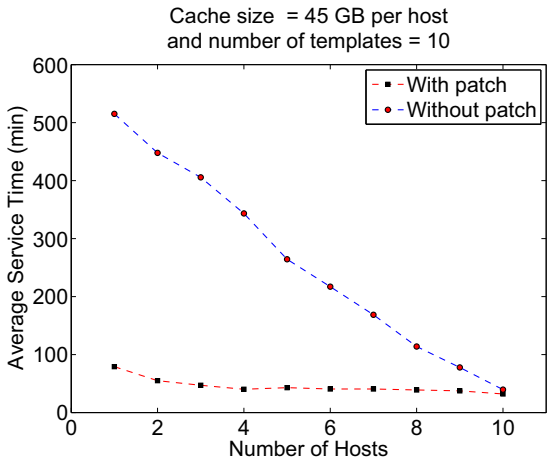

Fig. 4. Comparison of caching scenario with and without storing patches, when the number of hosts is varied. As the number of hosts increase, available caching space increases, thereby helping the "no patch" scenario. With patches, cache utilization is better, which leads to low average service time.

significantly since along with base template, few patches are cached on hosts, thus serving more templates from cache. When the cache size is below 50 GB per node, both approaches perform identically since there is inadequate space in any host to cache even a single template, therefore, caching any patch is useless. At this point, all templates are accessed directly from repository leading to a high service time as large files are fetched over slow links of bandwidth 10 Mbps.

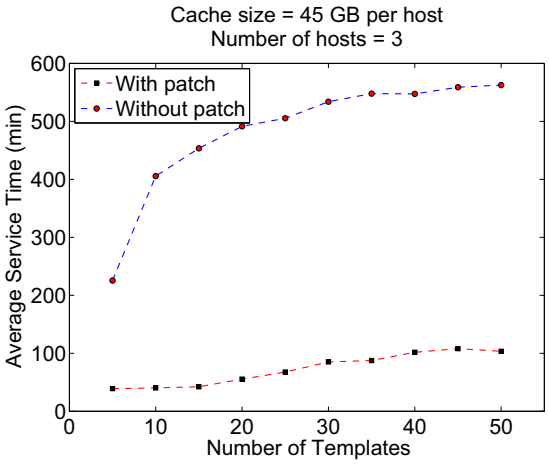

Fig. 3. Comparison of caching scenario with and without storing patches, when the number of templates is varied. As the number of templates increases, patch based caching can scale more effectively than the caching without patches.

We test the scalability of caching with patches with respect to increasing number of templates, as shown in Fig. 3. With fixed cache size, when template count is increased, DiffCache can cache patches to effectively increase the number of cached templates. However, as the template count increases, the cache space is insufficient to store all patches. Therefore, repository is accessed, leading to an increase in average service time.

When the number of nodes is varied, DiffCache shows a consistently good performance, whereas the non-patch based approach matches DiffCache algorithm as the node count equals the number of templates in the system, as shown in Fig. 4. As the number of hosts increases, total cache space also increases. Thus even when caching without patch, there is enough space to store all the templates. However, the benefits of caching with patches is evident when the number of hosts is less, where despite having smaller cache space, the average service time is reduced by caching patches to service requests.

B. Evaluation on Testbed

We present the testbed setup for comparing the caching approaches on a prototype testbed implementation.

1) Testbed Setup: The testbed comprises of 5 hosts: 3 compute hosts, 1 host used as the provisioning engine for dispatching requests, and 1 host as the image repository. Compute hosts are IBM blade servers on the same chassis. Image repository is collocated with the compute hosts, therefore, we programmatically throttle the bandwidth between repository and the compute hosts to stay around 20 Mbps, thereby mimicking a remote host connected over Internet.

On each compute host, we allocate a fixed space on local disk for caching. There are two ways to service a request on a node. If the required template and patch are on peer hosts, it is first copied to a compute host, and then patching is performed to service the request. We term it as *Fetch-n-Patch* mode. However, if all the cache directories across peers are mounted on each other, then it is possible to further improve the performance of caching. In this case, the files are not copied to the compute host from a peer, instead the

patching command is issued on the files residing on mounted directories. If NFS cache is turned on, this technique can benefit from NFS caching. We term it as *On-the-Fly Patch* mode.

There are 5 VM templates, with Linux as base OS, and additional softwares installed on it to create templates with some common components. The base templates are of size 7GB, while the patch sizes vary from 36MB to 1.6GB. Requests for the 5 templates are generated from a uniform random distribution, and dispatched round-robin to the compute hosts. DiffCache algorithm uses the same distribution, but with a different seed, which it assumes to be the request history for evaluating future request arrivals.

In the experiment, we compare caching with-patches and without-patches, where DiffCache selects the cached elements in both cases. The requests are triggered while the cache is being updated with files from the repository. Therefore, first few requests cannot be serviced from cache, and fetches the templates from repository. Subsequently, a request uses the cache lookup table to fetch required files, templates and/or patches, from either a peer or repository, as defined in the lookup table. In production setting, the DiffCache computation will be performed during periods when the workload is light, thereby, causing minimal impact to the service. Unless the workload varies fast, the cache composition will remain stable. Once the necessary files are received on the compute host, patching is performed, if required, to generate the requested template. A VM can be instantiated from the template file generated.

2) Testbed Results: Fig. 5 shows the time taken to service same request stream for three cases: Fetch-n-Patch mode, On-the-Fly patch mode, and without patch caching mode. Request 1, 2, and 3 were dispatched while the cache was being filled. Hence these 3 requests are serviced from the repository, taking around 60 minutes to complete. Note that request 1 took longer to complete for with-patch scenarios. As the cache update was in progress, additional traffic was introduced in the network creating contention. For with-patch case, the number of flows during cache update was 11 (2 templates and 9 patches were being fetched), leading to multiple flows starting simultaneously. For no-patch case, only 2 template files were fetched for caching.

The remaining requests were serviced based on the lookup decision prepared by DiffCache. In no-patch scenario, only two templates could be cached, leading to several requests being serviced from repository. In with-patch case, we are able to service almost all the requests from the distributed cache. For request id 4, service time with patches is longer compared to no-patch case, since the lookup decision decided to pull the template and patch from peer nodes; whereas, in no-patch case, the template was resident in the local cache, and required only a local copy. On-the-Fly patching consistently performs better than Fetch-n-Patch since we are able to avoid sequential copy of the files into the compute host from peers. *Average service time for no-patch, Fetch-n-Patch, and On-the-Fly patch cases are 46.9 mins, 5.02 mins and 3.23 mins respectively when*

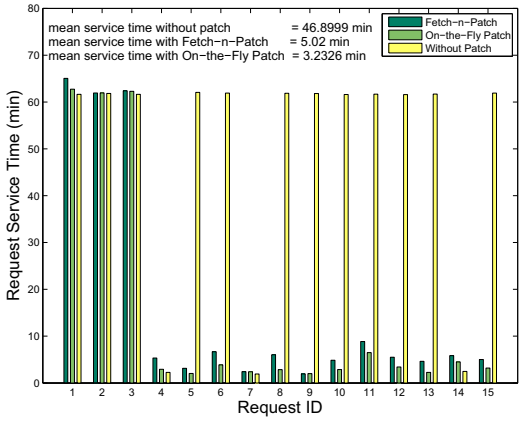

Fig. 5. Comparison of caching with patches and aching without patches on testbed, where DiffCache technique is used to select the cached elements. Mean service time is computed for requests that arrived after cache is completely filled up.

all requests after cache update completion are considered (requests 4 to 15). Overall improvement in servicing a request when patches are used in caching is over 90% more than the case when patches are not used.

We also measure the time for initializing the cache after DiffCache computes the cache composition. Cache space on the 3 hosts were 5GB, 8GB and 8GB. The cache was empty in the beginning. Time taken to fill the cache is around 1 hr, which is dependent on the network bandwidth between the repository and hosts, as well as the size of the files. In practice, cache fill may take less time since some cached files may be reused. The benefit of with-patch case is clear when the utilized cache space is observed. Fig. 6 shows that with no-patch, we are not able to utilize the cache on host-1 due to fragmentation, however, with-patch scenario can utilize more of the available cache space since it stores patch files in cache of host-1.

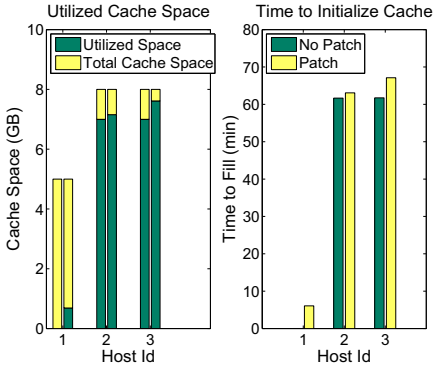

Fig. 6. Comparison of cache space utilization and initialization time. In each block, left-hand bar represents caching without patch, and the right-hand bar denotes caching with-patch.

V. Related Work

The problem of reducing latency while delivering content has been addressed in several contexts, like Web Caching [7],

Content Distribution Networks [8]. Although our work hinges on the idea of caching explored in these domains, the key problem we solve relates to composing the cache content while minimizing network usage cost. The low network transfer overhead translates to reduced latency while serving a request. In surveying the prior art, first we focus on techniques that attempt to mitigate the high network transfer time while delivering large template files in cloud setup. We also characterize the work related to efficient image repository maintenance and explain how it complements our work.

BitTorrent like systems has been one of the most popular techniques to download large files by opening simultaneous connections to retrieve parts of a file. Similar technique is employed by Chen et al. in image file distribution [9]. They describe a virtual machine image file as a BitTorrent seed file, and use upload capacity of peer hosts in a delivery center to accelerate the image file transfer. Although we use the bandwidth between the peer nodes, we avoid transfer of the complete files, and can achieve high effective transfer rate by only transferring a small patch file. A similar approach of leveraging peer-to-peer file transfer is also applied in [10]. In Iceberg, a chunk-based content-aware file partitioning is used to improve image distribution. Our technique is applicable on any file type without maintaining additional information.

Optimizing storage requirement for image repository has been addressed by partitioning large image files into smaller chunks, and maintaining a deduplicated chunk store. While some systems use an unstructured block-based representation [11], [12], others, like Mirage [13], maintain an indexed chunk store. The chunk based image repository can easily support a "delta deployment" strategy during image provisioning, where deltas between images can be used to speed up generation of new images. In this work, we have solved the problem of choosing appropriate deltas and images, and show the benefits of delta deployment in provisioning virtual machine instances in cloud.

VI. Conclusion

Machine image templates that are used to deploy Virtual Appliances on cloud infrastructure are large in size, often ranging in tens of GigaBytes. Fetching image templates stored in centralized repositories incurs long network delay due to slow Internet bandwidth, thereby adversely affecting request service time. Replicating repositories across all hosting centers is prohibitive in terms of storage cost. A common solution to mitigate such latency issue is to maintain a cache collocated with the hosting center.

Image templates show high degree of commonality. This feature has been exploited in optimizing storage requirement of image repository by storing only common blocks across templates. We take this approach a step further to exploit commonality among templates while caching. A patch file between two similar templates is small in size. If the template and the patch file is in cache, then a new template can be generated by using the in-cache template and patch. This can not only get the request serviced from cache, it also saves in terms of cache space requirement. Since patch files are small in size, one can effectively cache more templates by just storing a few templates and multiple patches. Given the request pattern for templates, and the available cache space, it is possible to compute the optimal cache composition that will minimize the number of requests required to contact the repository for servicing. We propose DiffCache algorithm that populates the cache with patch and template files that minimizes the network traffic, and leads to significant gain in reducing service time when compared to standard caching technique of storing template files. Our simulation experiments show significant gains when using patch based caching approach. Patch-based caching using DiffCache shows over 90% improvement when compared to Diffcache generated selection for template only caching in our prototype testbed implementation.

Acknowledgements

We would like to acknowledge the contributions of Jai Kumar Singh and Manoj Soni during initial development and experimentation.

References

[1] "Amazon machine images (amis)." [Online]. Available: http://aws.amazon.com/amis

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

[3] K. R. Jayaram, C. Peng, Z. Zhang, M. Kim, H. Chen, and H. Lei, "An empirical analysis of similarity in virtual machine images," in *Proceedings of the Middleware 2011 Industry Track Workshop*, 2011.

[4] K. D. Ryu, X. Zhang, G. Ammons, V. Bala, S. Berger, D. M. Da Silva, J. Doran, F. Franco, A. Karve, H. Lee, J. A. Lindeman, A. Mohindra, B. Oesterlin, G. Pacifici, D. Pendarakis, D. Reimer, and M. Sabath, "Rc2-a living lab for cloud computing," in *Proceedings of the 24th international conference on Large installation system administration*, ser. LISA, 2010.

[5] "Rdiff." [Online]. Available: http://librsync.sourceforge.net/

[6] P. De, M. Gupta, M. Soni, and A. Thatte, "The collective: a cache-based system management architecture," in *Proceedings of IEEE/IFIP Network Operations and Management Syposium*, ser. NOMS, 2012.

[7] J. Lin, T. Huang, and C. Yang, "Research on web cache prediction recommend mechanism based on usage pattern," in *Proceedings of the First International Workshop on Knowledge Discovery and Data Mining*, 2008.

[8] J. Dilley, B. Maggs, J. Parikh, H. Prokop, and B. Weihl, "Globally distributed content delivery," *IEEE Internet Computing*, 2002.

[9] Z. Chen, Y. Zhao, X. Miao, Y. Chen, and Q. Wang, "Rapid provisioning of cloud infrastructure leveraging peer-to-peer networks." in *ICDCS Workshops*, 2009.

[10] R. Wartel, T. Cass, B. Moreira, E. Roche, M. Guijarro, S. Goasguen, and U. Schwickerath, "Image distribution mechanisms in large scale cloud providers," in *Proceedings of the 2010 IEEE Second International Conference on Cloud Computing Technology and Science*, ser. CLOUDCOM, 2010.

[11] R. Chandra, N. Zeldovich, C. Sapuntzakis, and M. S. Lam, "The collective: a cache-based system management architecture," in *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation - Volume 2*, ser. NSDI, 2005.

[12] S. Tang, Y. Chen, and Z. Zhang, "Machine bank: Own your virtual personal computer," in *IPDPS*, 2007.

[13] G. Ammons, V. Bala, T. Mummert, D. Reimer, and X. Zhang, "Virtual machine images as structured data: The mirage image library," in *Usenix HotCloud*, 2011.