Caching Techniques for Rapid Provisioning of Virtual Servers in Cloud Environment

Pradipta De*, Manish Gupta*, Manoj Soni†, Aditya Thatte*
{pradipta.de, gmanish}@in.ibm.com, manojsoni@gatech.edu, adthatte@in.ibm.com
*IBM Research, New Delhi, India
†Georgia Institute of Technology, Atlanta, GA, USA

***Abstract*—Provisioning a virtual server instance in cloud goes through an elaborate workflow, characterized by user request submission, search for the requested image template in image repository, transfer of the template file to compute hosts, followed by expansion and boot up. Optimizing any step in this workflow is crucial in reducing the service time. In this work, we focus on reducing average service time by masking the template file transfer time from repository, and preparing a VM instance beforehand to service a request instantaneously. We use a strategy of pre-provisioning multiple VM instances from different templates. The instances to be pre-provisioned are determined based on request history. We show the benefits of our method using request trace from an enterprise grade cloud environment. Simulation results show more than 50% improvement in reducing average service time while delivering a server instance.**

I. Introduction

Ability to service a user request quickly is a desired feature in any cloud offering. However, servicing a user's request for a virtual server in cloud follows an elaborate workflow, leading to response time in order of minutes. The workflow begins with user requesting an image template from a catalogue, which is then searched in the template repository and delivered to a compute host for instantiation and booting. The image templates are tens of GigaBytes in size, therefore, depending on the network speed, the transfer time could be significant. The process of booting a virtual server can also consume time.

In this work, we aim at masking the time spent in transfer and booting, thereby reducing the request service time. We propose a technique to prepare virtual server instances from different templates before the request arrives, and maintain the instances in standby state. On request arrival, an instance can be delivered instantly. In order to minimize resource requirement, we analyze the request arrival pattern to predict the future requests. Often requests for virtual servers of the same templates arrive in bursts. To minimize the average service time, we must maintain multiple copies of the server instance from that template. Hence we pre-provision multiple server instances from a template based on the popularity of the template, as well as, by estimating the number of requests for that template.

Caching as a technique for reducing latency is applied in several domains, like OS, Web, CDN. Typically a cache delivers a *copy* of the requested item on request. But in cloud caching we avoid any copy time by delivering an instance, instead of cloning or copying. This is akin to an inventory management problem [1], where the challenge is to replenish the stock efficiently such that maximum number of requests for an item can be serviced. Benefit of typical caching approach in virtual machine provisioning has been explored [2] previously. Other approaches to speed up the template delivery process involves using BitTorrent like streaming [3], or simultaneous transfer of file chunks [4]. Forms of caching by storing snapshots of running virtual machines is employed in [5].

Our key contribution is in designing a pre-provisioning technique beyond typical caching, which is more effective in reducing average service time in cloud.

II. System Model and Assumptions

In this section, we present an overview of cloud architecture, followed by our simulation model to capture flow of a request through the system.

A. Cloud Architecture Overview

A cloud infrastructure maintains a farm of physical nodes, which are used to host virtual servers. A virtual server is instantiated from an image template, as per user request. An image template may comprise of base configurations, like Linux, with additional software components. Templates are maintained by the cloud provider in an image repository. A cloud provisioning engine receives a user request to check for the request type. For provision request, the requested template is searched in the repository, and the template file transfer to a compute host is initiated. Typically templates sizes are of the order of few GigaBytes, therefore, the transfer from image repository to compute node may be time consuming [2]. Once the image is transferred to a compute host, it is expanded, and booted up to create a virtual machine instance from the requested image template. Besides provision request, a user may also request deletion of an instance. Fig. 1 shows the components of the cloud architecture.

B. Simulation Model

We model the cloud provisioning engine as a multi-server queuing system. Each server models an available thread for servicing request in the provision engine. Assuming infinite servers in the model, we can accurately compute the time to service each request, referred to as *service time*, since there is no delay in the queue. In order to model the service time of

978-1-4673-0269-2/12/$31.00 © 2012 IEEE

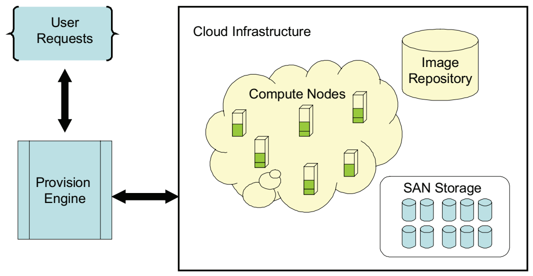

Fig. 1. Cloud architecture overview showing different functional components

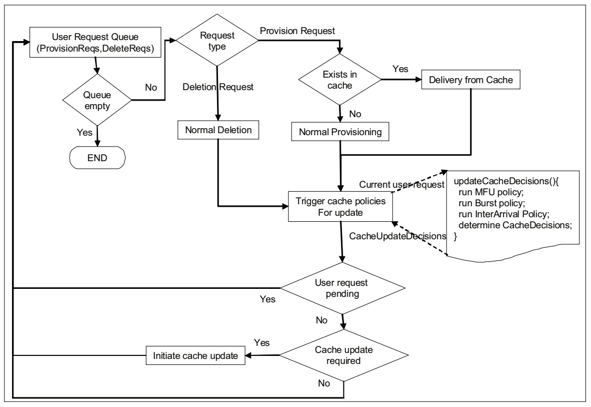

Fig. 2. Flowchart showing the steps in the simulator

each request, we introduce a *start-event* and *finish-event* for each event type.

In order to quantify benefit of pre-provisioning, events denoting *cache pre-filling event* and *cache entry deletion event* are introduced. The space available for pre-provisioning is finite. Therefore, before inserting a new instance, an instance must be deleted from cache if space is insufficient. User requests are never queued, and are serviced as soon as it arrives. Pre-provision requests are issued when there are available threads, however, once a pre-provision request has been triggered, it must complete before the thread can be released. Therefore, if a user request arrives after a pre-provisioning request has been issued, then the user request will wait till a thread is released, which is the only performance penalty in the technique. Fig. 2 shows the flowchart for the simulator.

III. Pre-provisioning Techniques

In this section, we present different approaches for pre-provisioning VM instances. Pre-provisioning strategy selects a set of VM instances for pre-provisioning based on history of requested templates. We assume a storage space (cache) only for a fixed number of VM instances. The cache composition is evaluated periodically, and the cache is replenished by evicting VMs which are unused to free up resource for new VMs to be cached.

Following notations are used to explain the pre-provisioning techniques.

R := number of user requests asking for new image instances within a given time window
N := number of image templates used to create the R requests within a given time window
f_{ij} := 1 if the j^{th} request is for the template i, otherwise 0
C := Maximum number of image instances that can be kept in the pre-provisioned inventory

A. *Most Frequently & Recently Used (MFU/MRU)*

Most-Frequently-Used, or MFU, strategy decides based on the popularity of a template. Within a window of R past requests, it computes the requests for type i as $f_i = \sum_{1 \le j \le R} f_{ij}$. Order of importance of an image is proportional to f_i. Given a cache size of C, the cache is filled up according to the formula:

$$C_i = w_i * C, \tag{1}$$

where w_i is computed as,

$$w_i = \frac{\sum_{1 \le j \le R} f_{ij}}{\sum_{1 \le i \le N} \sum_{1 \le j \le R} f_{ij}} \tag{2}$$

MFU approach implicitly assumes a stationary request distribution within history window. In practice, popularity of an image may fade over an interval. If time elapsed since the last request for an image template is large, then we can assume that the likelihood of a request for that image template is low. MFU is tuned to capture the above temporal variation of request distribution in the Most Recently Used (MRU) technique. We adjust w_i for a template i by attenuating the contribution of instances whose requests are older. A naive approach for attenuation is to reduce the values proportional to the time elapsed since the arrival of an instance request for a specific type. Alternatively, one can assign high importance to recent image types with the assumption that image types go out of fashion very quickly. The attenuated weight, w'_i, factoring in temporal aspect, is expressed as,

$$w'_i = \frac{\sum_{1 \le j \le R} A(j, f_{ij})}{\sum_{1 \le i \le N} \sum_{1 \le j \le R} A(j, f_{ij})} \tag{3}$$

where, $A(x, y)$ is the attenuation function and can be expressed as, $A(x, y) = y * exp(-x)$. The new weights, w'_i, are used in Eqn-1 to compute the number of instances of an image template to be cached.

B. *Burst Adjustment (BA)*

Creation of a new pre-provisioning request entails that a cached VM is purged to make room. Such removal adds inefficiency because the cost of pre-fetching the deleted VM is wasted. Therefore, in contrast to the previous steps, where the entire cache is packed, now the goal is to restrict the number of VMs to be placed in cache to reduce cache deletions. In Burst Adjustment step, *we find the largest burst, B_i, that an image template i has encountered in the past R requests, and then use B_i to limit the number of entries for image template i in cache.* Represented mathematically, the number of entries for image type i in cache, is:

$$Burst\ adjusted\ C_i = min(w'_i * C,\ B_i) \tag{4}$$

Simulation Parameter	Parameter Value
Cache Size	30 (or as mentioned)
History Window	1000 (or as mentioned)
Cache Update Interval	15 mins
Cache Entry Insertion time	10 mins
Cache Entry Deletion time	2 mins
Servicing time on Cache Hit	2 mins
MRU Policy Parameter	10.0
Cluster Size for Burst	11 mins

TABLE I
SIMULATION PARAMETERS

Burst adjustment helps in identifying the number of instances that can be requested in a sufficiently small "time interval". A good estimate of the $timeinterval$ is the average time taken to provision an instance in cache.

C. *Integral Allocation*

The allocation amount, C_i for an image template i, as computed in Eqn-2 or Eqn-4, must be converted to integers before filling the cache. We apply mathematical rounding on each instance count, and start allocation in a descending order of allocation amount, till the entire cache space is utilized. This may lead to overallocation of instances from some templates, while some template instances are not cached. At the end of Burst Adjustment, the policy returns 3 image types to be kept in the cache with allocations [6.7, 2.6, 0.7] respectively. Integral allocation of the ids will be [7, 3, 0] respectively, thereby discarding the third image type.

IV. EXPERIMENTAL RESULTS

In this section, we present simulation results to evaluate the proposed technique.

A. *Simulation Inputs*

Simulation inputs comprise of parameters for modeling the cloud environment, as well as, request logs from an operational cloud environment used for driving the simulation.

1) Simulation Parameters: Table I shows the simulation parameters used in the experiments. Three key parameters are: (a) *cache size* which denotes the total number of VM instances that can be pre-provisioned, (b) *history window* which denotes number of past requests considered, *Cache-update-interval* duration is used to trigger re-computation of cache.

Few other relevant parameters are: (i) *cache entry insertion time* is the time to fetch an image from repository to cache, (ii) *cache entry deletion time* is the time to delete an entry from the cache, (iii) *service time on cache hit* is the time to deliver a cached instance to user. Cache hit service time is non-zero because some user-defined configurations may need to be set up prior to delivering the VM to the user.

2) RC2 Trace Details: We use request logs of 1 year from Research Compute Cloud (RC2), an IBM Research Division cloud computing platform [6]. It serves on an average 200 active users and 800 active VM instances per month. The total of 10200 requests that were requested were for 1088 unique templates. *Less than 10 VMs were requested for 890*

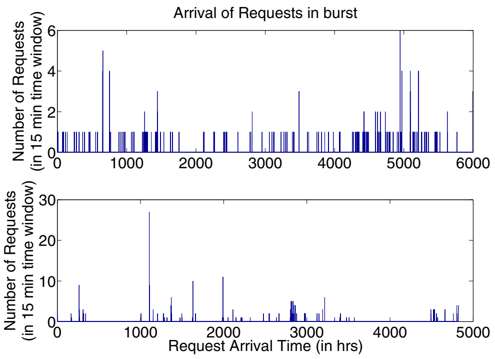

Fig. 3. Multiple VMs from a single template requested within a short interval.

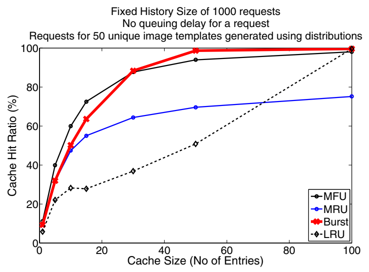

Fig. 4. Comparison of cache hit ratio with varying cache size.

image templates, with 453 templates being requested only once. Requests for the top 15 image types constitute only 26% of the total requests serviced, which implies that merely caching the popular templates may not be sufficient.

Another trend in request arrival is the presence of bursts, as shown in Fig. 3 for two different image types. Often clusters of servers belonging to the same image type are requested leading to a surge for that image. Even if one request from that set takes longer, it will lead the user to wait. Therefore, an efficient caching strategy must try to provision all the instances of a burst, which makes it a hard decision given that burst sizes can go over 20 instances of a type.

3) Synthetic Traces: In synthetic traces, request arrivals follow Poisson distribution, with mean and variance computed from the more popular image templates in RC2 trace. We analyze the trace to find out the total number of unique image templates, the request count for each image template, the statistics for generating the arrival pattern, and the statistics for generating the service time of each request. The image templates with the highest request counts are used to provide the statistics for arrival pattern generation. For the service time generation, we use Gaussian distribution.

B. *Simulation Results*

We show the relative gains of different approaches in comparison with Least Recently Used (LRU) method of cache replacement. The results are shown for synthetic request data, as well as, for RC2 trace data.

1) Results on Synthetic Trace: We consider 50 unique image templates in the cloud environment. We record the hit ratio with varying size of cache and a fixed history window. Fig. 4 shows that Burst Adjustment technique performs the

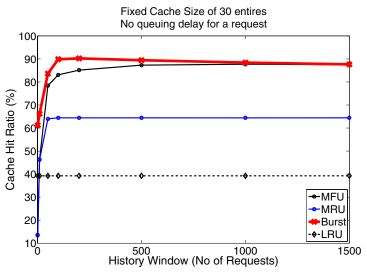

Fig. 5. Comparison of cache hit ratio with varying history window.

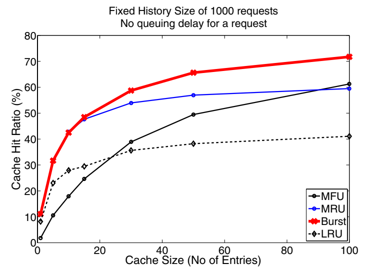

Fig. 6. RC2 trace data is used to compare different techniques.

best with increasing cache size, while LRU performance is the worst. Hit ratio saturates as cache size is increased to 50. With cache size of 50 entries, at least one instance of each image type can be placed, thereby leading to saturation.

When the history window is varied, keeping the size of cache fixed at 30 entries, we observe that the cache hit ratio saturates quickly around history window size of 100 (Fig. 5). A history window of 100 requests is likely to have a request of each type since the templates are uniformly chosen.

2) Results on Trace Data: Unlike the synthetic request data, the trace data is relatively non-uniform in nature. Fig 6 shows the comparison of different techniques for the RC2 trace data. Beyond a cache size of 20, Burst Adjustment outperforms all other techniques. When the cache size is less than 20, then according to Eqn-4, MRU overrides the Burst Adjustment technique; hence the results for both of them are identical. Results for varying history window is similar to Fig 5 using synthetic data. LRU method is not impacted by a varying history window because it always replaces the least recently used entry from the cache without looking at the history. MFU technique may degrade in performance with increasing history window because in the cloud environment, when the history size is increased, several image types which are old are often never requested again. Therefore, giving equal importance to all requests, without taking temporal aspect into account, leads to degraded performance for MFU. The performance improves as soon as MRU is applied on top of MFU. However, MRU also may end up over-allocating instances for an image type. Burst Adjustment reduces the number of instances of an image type to be pre-provisioned, thereby creating room to cache more image types.

Fig. 7 shows the improvement in service time with pre-provisioning. Without pre-provisioning, average service time for a request is 18 minutes. When pre-provisioning is applied, average service time can be reduced to as low as 6 minutes

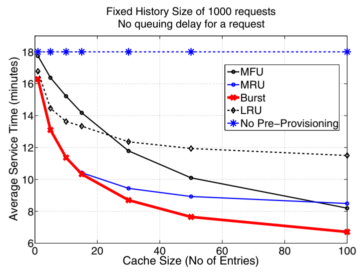

Fig. 7. Plot shows the average service time for provisioning a request.

for some configurations. The best case with a history size of 1000 requests is recorded when the cache size is 100 and burst adjustment policy is applied. The reduction in service time is 62%. With a more realistic cache size of 30 entries, each of average size 30 GB, requiring about 1 TB cache space, there is 51% reduction in service time.

V. Conclusion

Time to provision a virtual machine in cloud is often in order of minutes. In this work, we present techniques to reduce average service time during provisioning. We use the request arrival pattern to pre-populate a cache with virtual machine instances from different templates, which can be delivered instantly on request. We use the popularity of image templates, along with the time of arrival of the requests to determine the expected set of requests. Since requests for an image type often arrive in clusters, we also detect bursts in request arrival pattern to shape the set of instances to be pre-provisioned. We have used request logs of more than one year from an enterprise grade cloud setup to evaluate our techniques. The trace-driven simulation experiments shows more than 60% reduction in average service time.

References

[1] A. Drexl and A. Kimms, "Lot sizing and scheduling – survey and extensions," *European Journal of Operational Research*, vol. 99, no. 2, pp. 221–235, June 1997.

[2] W. Emeneker and D. Stanzione, "Efficient virtual machine caching in dynamic virtual clusters," in *In SRMPDS Workshop, ICAPDS 2007 Conference*, 2007.

[3] Z. Chen, Y. Zhao, X. Miao, Y. Chen, and Q. Wang, "Rapid provisioning of cloud infrastructure leveraging peer-to-peer networks." in *ICDCS Workshops*. IEEE Computer Society, 2009, pp. 324–329.

[4] L. Shi, M. Banikazemi, and Q. B. Wang, "Iceberg: An image streamer for space and time efficient provisioning of virtual machines," in *Proceedings of the 2008 International Conference on Parallel Processing - Workshops*, 2008, pp. 31–38.

[5] 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'05, 2005.

[6] 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'10, 2010, pp. 1–14.