Caching VM Instances for Fast VM Provisioning : A Comparative Evaluation

Pradipta De[1], Manish Gupta[1] Manoj Soni[2], and Aditya Thatte[1]

[1] IBM Research India, New Delhi
{pradipta.de, gmanish, adthatte}@in.ibm.com
[2] Georgia Institute of Technology, Atlanta, GA, USA
manojsoni@gatech.edu

Abstract. One of the key metrics of performance in infrastructure cloud is the speed of provisioning a virtual machine (or a virtual appliance) on request. A VM is instantiated from an image file stored in the image repository. Since the image files are large, often GigaBytes in size, transfer of the file from repository to a compute node running the hypervisor can take time in order of minutes. In addition to it, booting an image file can be a time consuming process if several applications are pre-installed. Use of caching to pre-fetch items that may be requested in future is known to reduce service latency. In order to overcome the delays in transfer and booting time, we prepare a VM a priori, and save it in standby state in a "cache" space collocated with the compute nodes. On receiving a matching request, the VM from cache is instantly served to the user, thereby reducing service time. In this paper, we compare multiple approaches for pre-provisioning and evaluate their benefits. Based on usage data collected from an enterprise cloud, and through simulation, we show that a reduction of 60% in service time is achievable.

1 Introduction

Time to service a request for a new virtual machine in cloud can often run into several minutes. The complete workflow beginning with receiving a request till a new virtual machine is delivered to user, follows a number of steps. First, the requested machine image template from which the VM must be instantiated is looked up in the image repository, then the image template file is copied to a compute host and the VM is then booted up. Image template files are very large in size, often in GigaBytes range. Transferring such large files over the network is time consuming. In addition to it, the boot process can also be slow depending on the number of pre-installed components in the image. Due to these bottlenecks [8, 19], servicing a provision request can take a long time.

In order to speed up virtual server provisioning, there have been approaches to expedite the transfer of the large template files using different streaming techniques [4, 18]. Caching of the template files at the compute nodes to hide the transfer latency has also been explored [8]. To reduce the boot time, one approach is to instantiate a VM, and store it in a standby mode in cache. This saves the time to create an instance from a template and boot the VM. In essence, an inventory of readily deliverable VMs are maintained based on user request patterns.

In this work, we compare 3 techniques for pre-provisioning VM instances. Given a fixed size cache space, each technique comes up with the composition of the inventory. In other words, for a template type, the expected number of VM instances to be requested is calculated and pre-provisioned. If a request matches a pre-provisioned VM instance, it is delivered with minimal delay to the user. The cache or VM inventory space is freed up when a VM instance is delivered to a user. The inventory is replenished periodically with new VM instances.

We have analyzed the server request trace from an enterprise-wide cloud deployment to study the request pattern. We compare three approaches to highlight the benefits of each technique in maintaining the inventory or cache of VM instances for reducing server provisioning time in cloud.

The rest of the paper is organized as follows. In Section 2, we present literature related to caching in different contexts. In Section 3, we present an overview of the cloud architecture, along with our simulation approach. Section 4 presents the pre-provisioning methodologies. In Section 5, we present detailed simulation results showing the benefits of pre-provisioning. Finally we conclude in Section 6.

2 Related Work

Caching as a technique has been extensively researched, specially in the domain of operating system and design of memory hierarchies. Our focus is more on the use of caching techniques in the context of cloud.

Web Caching and CDN: Application of caching to speed up access to content that is accessed repetitively over the Internet has been studied over the last decade [6]. Prefetching web resources by anticipating future trends is a challenging problem. Usage prediction methods, taking help of different techniques like clustering [11], neural networks [13], have proved to be of limited benefit. [6] reports that cache hit rates can rarely cross 40-50%. Instead of tackling the problem of usage prediction using standard methods, in this work gain insight into the peculiarities of a trace to leverage the usage pattern.

In web caching, often the most common cache replacement strategy of replacing the Least-Recently-Used (LRU) item works quite well, as shown in [16]. However, in cloud the user request may arrive in bursts, therefore, the replacement policy may be inefficient if replacement is one item at a time. Rather, in our strategy we predict the most suitable set of image templates, as well as, the number of VM instances of each type.

Content Delivery Networks(CDN) uses caching at Internet scale. The key idea in CDN is to push content closer to user before a request arrives. [12] proposes scheduling algorithms to push content in a timely manner to proxy cache servers, while [1] discusses a cooperative cache management scheme to maximize traffic volume served from cache. These schemes can be useful in pushing cloud images nearer to the user, but we are also interested in maintaining multiple instances of a template.

Caching in Cloud Context: Caching machine image templates has been studied in grid and cluster environments. In [8], Emeneker et al. shows that caching of a virtual image can speed up execution of parallel jobs. However, they do not explore the pros and cons of different caching approaches.

Predictive methods are used more frequently in cloud context in order to manage resource requirement. Tackling the problem of when to scale resources by adding more VMs, or how much resource to apportion to a cloud setup has been addressed by several works. In [2], authors develop a forecasting method to predict resource demand in cloud by using historic data. An approach to auto-scale during flash crowds is presented in [20]. Use of cloning or de-duplication techniques to quickly add new VMs have been addressed in [10]. In our method, we deliver the server instance instantly if there is a cached VM instance matching the request to avoid the overhead of cloning.

Several works identify that fetching an image from a central repository takes up significant proportion of the service time of a request [8, 19]. BitTorrent-like distribution system has been proposed to speed up image delivery in [4]. A similar image streaming approach, by breaking up an image in chunks, has been proposed in [18]. We are complementary to these schemes since we can leverage the speedy distribution of the image once it has been identified by our scheme.

Moka5 provides a solution for desktop virtualization, where the desktops are snapshot at intervals and stored in a central repository [3, 14]. However, it assumes a large storage repository and does not present any intelligent cache mechanism to speed up delivery of images. Similarly, Eucalyptus cloud management system mentions use of caching without providing details [9, 15]. Even, Amazon mentions that frequent users will have the benefit of faster turnaround time which hints at underlying caching. IBM Workload Deployer, previously known as WebSphere CloudBurst Appliance, also mentions the use of caching [5]. However, caching prepared VM instances, as opposed to image templates, distinguishes our work.

Inventory Management: A close look at our technique reveals that it is closest in design to an inventory management system. Whenever a matching request arrives an instance of an image template is used up, and the inventory (cache) must be replenished efficiently at minimum cost. The problem of lot sizing in inventory management [7] deals with picking the appropriate quantities of each item to be provisioned in the inventory. Similar to inventory management, it is necessary to predict the appropriate number of each image template to be kept in cache.

3 System Model and Assumptions

A typical cloud setup maintains a farm of compute nodes. The compute nodes are used to instantiate virtual machines from user-specified image templates, which are stored in an image repository. When a user makes a request for a new server instance, the request is first intercepted by the cloud provisioning engine. A provisioning engine checks for the available image type in the repository, and initiates a transfer of the image to a compute host. Once the image is transferred to a compute host, it is expanded, and booted to create an instance of a virtual machine. The SAN store is used for providing the user data space, similar to Amazon's Elastic Block Storage (EBS). Besides provision request, a user may also request deletion of an instance. Fig. 1 shows the different components of the cloud architecture.

Fig. 1. Cloud architecture overview showing different functional components

3.1 Simulation Model

We model the cloud provisioning engine as a multi-server queuing system. Each server models a request handling thread in the provisioning 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 each request, we introduce a *start-event* and *finish-event* for each event type. For example, provision request is modeled using a start and finish event for that type. Corresponding events for deletion requests are introduced.

Fig. 2. Flowchart showing the steps in the simulator

In order to quantify the benefits of pre-provisioning, events denoting *cache pre-filling event* and *cache entry deletion event* are introduced. We assume that a fixed amount of space is available for pre-provisioning; therefore, we can only pre-provision a fixed number of VM instances. Before inserting a new VM instance, determined by cache update policies, an instance may be deleted from cache. The pre-fetch action never blocks a request already queued in provisioning engine. However, once a pre-provision action is triggered, it must complete before releasing the thread. If a user request arrives while a pre-fetching is in progress, then the user request must wait, thereby increasing its service time. Fig. 2 shows the flowchart for the simulator.

4 Pre-provisioning Techniques

In this section, we present the techniques for selection of cache composition at periodic intervals. First an analytical model is explained to motivate the approaches, and then three approaches are presented.

4.1 Analytical Model for Pre-Provisioning

The provisioning engine is modeled as a single server queue, with an additional cache entity that can store *exactly one* VM instance at a time. Request arrival for image-type-1(I_1), and image-type-2(I_2) follows Poisson distribution, with rates λ_1 and λ_2 respectively. The service time for each server request is image-type dependent, and exponentially distributed with rates, μ_1 and μ_2 respectively. A pre-provisioned instance is fetched and cached only if the cache is empty and there is no pending user request in the queue. If a request for a server instance arrives before the pre-provisioning request is complete, *the pre-provisioning request is canceled, thereby leaving the cache empty*.

The cache entry is purged if there is a cache miss. On cache hit, the request is serviced instantaneously, implying zero service time.

The caching policy can be stated as follows: *Create an instance of I_1 with probability p, otherwise create an instance of I_2 with probability $(1-p)$ whenever the system is detected to be idle. Our aim is to find the value of p such that it minimizes the average end-to-end provisioning time for the requests.*

Theorem 1. *Under the stated assumptions, the optimal policy is to set $p = 1$, otherwise set $p = 0$, if the following condition holds.*
$\frac{\lambda_1}{\lambda_1+\lambda_2+\mu_1} > \frac{\lambda_2}{\lambda_1+\lambda_2+\mu_2}$

Proof. Since the arrival and service processes are memory-less, due to our assumption, therefore, the evolution of the process does not depend on past history. Minimizing expected service time is equivalent to maximizing reduction in service time by using cache. The reduction in expected service time can be represented as:
$$p(\frac{\mu_1}{\lambda_1+\lambda_2+\mu_1})(\frac{\lambda_1}{\lambda_1+\lambda_2})\frac{1}{\mu_1} + (1-p)(\frac{\mu_2}{\lambda_1+\lambda_2+\mu_2})(\frac{\lambda_2}{\lambda_1+\lambda_2})\frac{1}{\mu_2}$$

The first and second terms in the expression corresponds to time reduction when I_1 and I_2 are chosen respectively. Within each block in the expression, $\frac{\mu_1}{\lambda_1+\lambda_2+\mu_1}$ corresponds to the probability that pre-provisioning I_1 completes before next request arrival; $\frac{\lambda_1}{\lambda_1+\lambda_2}$ denotes probability of arrival of I_1 request before I_2 request; and $\frac{1}{\mu_1}$ is the expected savings. The expression has the structure $pA + (1-p)B$ where $A > 0$ and $B > 0$ are constants. Now if $A > B$ then the expression will be maximized by choosing $p = 1$ otherwise by choosing $p = 0$. □

If the service rates for two image types are identical ($\mu_1 = \mu_2$), then I_1 would be the optimal choice for cache if $\frac{\lambda_1}{\lambda_1+\lambda_2} > \frac{\lambda_2}{\lambda_1+\lambda_2}$, and vice versa. Given a window of R requests from history, image type with the highest request count within the window should be cached.

Following notations are used henceforth to explain the 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

4.2 Techniques for Pre-provisioning

Most-Frequently-Used(MFU) strategy leverages the insight of popularity based caching from Section 4.1. 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. Now, given a cache size of C, the cache is completely filled up using the following formula:

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

where w_i is computed as,

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

MFU approach implicitly assumes that the request distribution for an image type is stationary within the history window. In practice, popularity of an image may fade over time, thereby falsifying the stationarity assumption necessary for MFU to perform effectively. 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. Thus, it is required to take into account the time of arrival of a request while selecting the cache composition.

In *Most-Recently-Used(MRU)* approach, we adjust w_i for a template i by attenuating the contribution of instances whose requests are older. Values can be attenuated by applying different functions. For instance, a naive approach 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,

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

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.

In MFU and MRU, selection is based on popularity of an image and available cache size. If the cache size is large, MFU and MRU may populate the cache with a large count of VMs of a template, although in practice the maximum request count for the template is lower than the cached count. This allows the opportunity to fill the cache more judiciously, thereby saving the resource wastage for deleting an unused cache entry during next refresh. *Burst Adjustment(BA)* technique, finds the largest burst, B_i, that an image template i has encountered in the request history of R requests, and then uses B_i to limit the number of VM instances for image template i in cache. Represented mathematically, the number of VM instances of image type i in cache, is:

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

Note that the selection step in Eqn-2 or Eqn-4 computes fractional numbers. During actual provisioning, VM instances occupy integral values, derived by rounding the fractions. This leads to some VM instances, with a low fractional value, being dropped from selection while allocating in decreasing order of the count. A simple example, where cache can store 10 VM instances, illustrates the effect of rounding. At the end of BA technique, VMs of 3 image types are to be cached with instance count [6.7, 2.6, 0.7] respectively. Rounding the values changes the allocation to [7, 3, 0] respectively, thereby discarding the third image type.

5 Experimental Evaluation

In this section, we show the results of evaluating the pre-provisioning approaches using simulation. We explain the simulation parameters, and provide a summary of the RC2 trace data which helps in understanding the results.

5.1 Simulation Parameters

Table 1 shows the simulation parameters used in the experiments. Three key parameters are: (a) *cache size* denotes the total number of instances of VMs that can be pre-provisioned, (b) *history window* denotes the number of past requests that are taken into consideration while computing the cache composition, (c) *pool-size* denotes the number of available threads or resources that can be dedicated for computations, like deletion, pre-provisioning. *Cache-update-interval* parameter is used to trigger the computation of cache composition periodically.

Simulation Parameter	Parameter Value
Cache Size	30 (or as mentioned)
History Window	1000 (or as mentioned)
Pool Size	100 (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 1. Simulation Parameters

Few other parameters relevant for evaluating the caching techniques are: (i) *cache entry insertion time* accounts for the time to fetch an image from repository and place it in the cache, (ii) *cache entry deletion time* accounts for the time to delete an entry from the cache, (iii) *service time on cache hit* accounts for the time to deliver a cached instance to the user request. 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.

MRU technique uses an attenuation function to assign higher importance to the recent requests. A *negative exponential function with a mean of 10.0* is used(refer Eqn-3). In BA technique, we compute the burst size by clustering all requests of an image type that arrive within the *cluster size for burst*.

5.2 RC2 Trace Summary

We collected a 1 year request log from Research Compute Cloud (RC2), which is a cloud computing platform for use by the worldwide IBM Research community [17]. It serves on average 200 active users and 800 VM instances per month, with a user base of 700 users. 10200 requests were logged during the observation. For each request, time of arrival of the provision request and deletion request, as well as, the time taken to provision the request is collected. Provision time is the end-to-end time from the request arrival to the user being notified of successful deployment.

1088 unique image types were requested by 743 different users over the 1-year period. Less than 10 server instances were requested for 890 image templates, with just a single request for 453 image types, making request density for an image template quite sparse. Requests for the top 15 image types constitute only 26% of the total requests serviced. Another trend in request arrival is the presence of requests for an image type arriving in bursts, which could happen when a multi-tier application is being set up with similar servers. Even if one request from this group takes longer, it will force the user to wait. Therefore, an efficient caching strategy must try to provision all the instances of a burst.

5.3 Simulation Results: Using RC2 Trace Data

We compare the cache hit ratio with varying cache size, as shown in Fig. 3. Beyond a cache size of 20, BA outperforms all other techniques. When the cache size is less than 20, then according to Eqn-4, MRU performs better than BA technique; thereby the results for both of them are identical.

Fig. 3. Comparison of different techniques on RC2 trace data, where cache size is varied.

Fig. 4. Comparison of different techniques on RC2 trace data, where history window is varied.

Results for varying history window is shown in Fig. 4 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. BA reduces the number of instances of an image type to be pre-provisioned, thereby creating room to cache more image types.

Fig. 5. Plot shows the average service time for provisioning a request.

Fig. 6. Reduction in misses, due to policy rejecting an image id, as the cache size is increased.

We also report the improvement in service time with caching. *Without pre-provsioning, 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 for some configurations*, as shown in Fig 5. 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%. If we consider a more realistic cache size of 30 entries of an average size of 30 GB, requiring total space of approximately 1 TB, then the reduction of 51% in service time* is still significant.

5.4 Reasons for Cache Misses

Cache misses are due to several reasons, some of which are unavoidable, viz. a request for a template is received for the first time. *Choice of history window size* impacts misses since a larger history window provides a larger set of image types being requested. Third type of *miss occurs due to rounding.*. Since some image types ends up with a zero allocation, therefore, it may lead to a cache miss if a request for the discarded type arrives despite policy correctly inferring the importance the image type. In addition to this, if request inter-arrival time is short, then although the caching decision may be accurate, time for pre-provisioning is insufficient.

Fig. 7. Reduction in misses, due to the absence of an image type in the history window, as history window size is increased

Fig. 8. Cache Hit Ratio computation over a rolling window where rolling window is 1000 entries. Graph also shows the number of unique image types in the window

Fig. 6 shows that as the cache size is increased, it allows more space to accommodate larger number of image types. Thus while performing the integral allocation step in caching; lesser number of image types are rejected, thereby increasing the number of hits. In case of BA, since the policy trims the number of instances to be kept for each image type to the maximum size of burst observed, therefore, it helps in accommodating instances of more image types. Therefore, number of misses due to policy rejecting an image type is lowest for burst adjustment method.

Fig. 7 shows that as the history window size is increased, it allows the cache policy to view more image types, therefore helping the caching policy to take more image types into account while deciding the cache composition. It can be inferred from the figure that with higher history window size the misses will reduce.

Fig. 9. Ratio of number of deletions to number of insertions in cache. Lower ratio indicates that the time wasted for deletion was saved during an insertion.

Fig. 10. Number of misses due to caching in-progress for the RC2 traces having a total of 10210 requests.

We also investigate the evolution of the cache hit ratio over a period of time. In Fig. 8, we consider a sliding window of 1000 requests, to observe the change in cache hit ratio. We also plot the number of unique image types present in the same history window. The graph confirms our observation that *as the number of unique image types increases within a history window, it leads to a drop in the cache hit ratio.*

5.5 Gains Demystified

Deletion of a VM instance from cache implies that the VM instance was provisioned unnecessarily. It wastes the time to pre-provision, as well as, time is spent in deleting it, thereby delaying the insertion of a new VM instance. In BA, since we are conservative in placing a VM instance into the cache, therefore, it reduces the number of deletions. When we compare the number of deletions, with respect to the number of insertions, per policy, we observe that this ratio is lower for BA compared to MRU policy, as shown in Fig 9.

Although MFU policy shows a significantly low deletion to insertion ratio, it still performs worse overall because it suffers due to the choice of image templates. Fig. 6 before shows that MFU policy suffers mainly due to rejection of too many image types while performing the integral allocation step.

Despite accurate prediction of future request arrivals by a policy, it still may not show the result, if the pre-provisioning of the instance does not complete before the next arrival. Often inter-arrival time between requests for an image type is shorter than the time to complete a pre-provision. In our simulation, we assume that if the caching is in progress then it is a cache miss. Fig. 10 shows for each policy the number of requests which recorded a miss although the image instance was being cached.

6 Conclusion

Typically, it takes time in order of minutes to provision a new VM in cloud. Transfer of the large image template file from repository to compute node, and booting are the main causes of delay in the provisioning workflow. We apply caching to alleviate the problem. Using request logs, we determine the image templates which will be high in demand, and also estimate the number of requests for each image type. Thus we can pre-provision VM instances by preparing and storing them in standby mode in cache. On receiving a matching request, a cached VM is readily delivered to the user. We have compared 3 different techniques for selection of the cached VM instances. Under specific configurations, service time to deploy a virtual machine can be reduced by 60% as compared to a no cache enabled scenario.

References

1. Borst, S., Gupta, V., Walid, A.: Distributed caching algorithms for content distribution networks. In: Proceedings of the 29th conference on Information communications. INFOCOM'10 (2010)
2. Caron, E., Desprez, F., Muresan, A.: Forecasting for grid and cloud computing on-demand resources based on pattern matching. In: Proceedings of the 2010 IEEE Second International Conference on Cloud Computing Technology and Science. CLOUDCOM '10 (2010)
3. Chandra, R., Zeldovich, N., Sapuntzakis, C., Lam, M.S.: The collective: a cache-based system management architecture. In: Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation (NSDI) (2005)
4. Chen, Z., Zhao, Y., Miao, X., Chen, Y., Wang, Q.: Rapid provisioning of cloud infrastructure leveraging peer-to-peer networks. In: ICDCS Workshops. pp. 324–329. IEEE Computer Society (2009)

5. IBM Workload Deployer, http://www-01.ibm.com/software/webservers/workload-deployer/
6. Davison, B.D.: A web caching primer. IEEE Internet Computing 5 (July 2001)
7. Drexl, A., Kimms, A.: Lot sizing and scheduling – survey and extensions. European Journal of Operational Research 99(2) (1997)
8. Emeneker, W., Stanzione, D.: Efficient virtual machine caching in dynamic virtual clusters. In: In SRMPDS Workshop, ICAPDS 2007 Conference (2007)
9. Eucalyptus Systems, http://www.eucalyptus.com/
10. Lagar-Cavilla, H.A., Whitney, J.A., Scannell, A.M., Patchin, P., Rumble, S.M., de Lara, E., Brudno, M., Satyanarayanan, M.: Snowflock: rapid virtual machine cloning for cloud computing. In: Proceedings of the 4th ACM European conference on Computer systems (EuroSys) (2009)
11. Lin, J., Huang, T., Yang, C.: 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)
12. Liran, R.C.: Scheduling algorithms for a cache pre-filling content distribution network (2002)
13. Makkar, P., Gulati, P., , Sharma, A.: A novel approach for predicting user behavior for improving web performance. International Journal on Computer Science and Engineering 02(04) (2010)
14. MokaFive Desktop Management Simplified, http://www.moka5.com/
15. Nurmi, D., Wolski, R., Grzegorczyk, C., Obertelli, G., Soman, S., Youseff, L., Zagorodnov, D.: Eucalyptus : A technical report on an elastic utility computing archietcture linking your programs to useful systems. Tech. Rep. 2008-10, UCSB Computer Science Technical Report (October 2008)
16. Podlipnig, S., Böszörmenyi, L.: A survey of web cache replacement strategies. ACM Comput. Surv. (2003)
17. Ryu, K.D., Zhang, X., Ammons, G., Bala, V., Berger, S., Da Silva, D.M., Doran, J., Franco, F., Karve, A., Lee, H., Lindeman, J.A., Mohindra, A., Oesterlin, B., Pacifici, G., Pendarakis, D., Reimer, D., Sabath, M.: Rc2-a living lab for cloud computing. In: Proceedings of the 24th international conference on Large installation system administration. LISA'10 (2010)
18. Shi, L., Banikazemi, M., Wang, Q.B.: 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)
19. Sotomayor, B., Keahey, K., Foster, I.: Combining batch execution and leasing using virtual machines. In: Proceedings of the 17th international symposium on High performance distributed computing (HPDC) (2008)
20. Zhu, J., Jiang, Z., Xiao, Z.: Twinkle: A fast resource provisioning mechanism for internet services. In: INFOCOM. pp. 802–810 (2011)