Optimizing multi-deployment on clouds by means of self-adaptive prefetching

Bogdan Nicolae, Franck Cappello, Gabriel Antoniu

Optimizing multi-deployment on clouds by means of self-adaptive prefetching

Bogdan Nicolae[1], Franck Cappello[1,2], and Gabriel Antoniu[3]

[1] INRIA Saclay, France
bogdan.nicolae@inria.fr
[2] University of Illinois at Urbana Champaign, USA
cappello@illinois.edu
[3] INRIA Rennes Bretagne Atlantique, France
gabriel.antoniu@inria.fr

Abstract. With Infrastructure-as-a-Service (IaaS) cloud economics getting increasingly complex and dynamic, resource costs can vary greatly over short periods of time. Therefore, a critical issue is the ability to deploy, boot and terminate VMs very quickly, which enables cloud users to exploit elasticity to find the optimal trade-off between the computational needs (number of resources, usage time) and budget constraints. This paper proposes an adaptive prefetching mechanism aiming to reduce the time required to simultaneously boot a large number of VM instances on clouds from the same initial VM image (multi-deployment). Our proposal does not require any foreknowledge of the exact access pattern. It dynamically adapts to it at run time, enabling the slower instances to learn from the experience of the faster ones. Since all booting instances typically access only a small part of the virtual image along almost the same pattern, the required data can be pre-fetched in the background. Large scale experiments under concurrency on hundreds of nodes show that introducing such a prefetching mechanism can achieve a speed-up of up to 35% when compared to simple on-demand fetching.

1 Introduction

The Infrastructure-as-a-Service (IaaS) cloud computing model [1, 2] is gaining increasing popularity both in industry [3] and academia [4, 5]. According to this model, users do not buy and maintain their own hardware, but rather rent such resources as virtual machines, paying only for what was consumed by their virtual environments.

One of the common issues in the operation of an IaaS cloud is the need to deploy and fully boot a large number of VMs on many nodes of a data-center at the same time, starting from the same initial VM image (or from a small initial set of VM images) that is customized by the user. This pattern occurs for example when deploying a virtual cluster or a set of environments that support a distributed application: we refer to it as the *multi-deployment pattern*.

Multi-deployments however can incur a significant overhead. Current techniques [6] broadcast the images to the nodes before starting the VM instances, a

process that can take tens of minutes to hours, not counting the time to boot the operating system itself. Such a high overhead can reduce the attractiveness of IaaS offers. Reducing this overhead is even more relevant with the recent introduction of spot instances [7] in the Amazon Elastic Compute Cloud (EC2) [3], where users can bid for idle cloud resources at lower than regular prices, however with the risk of their virtual machines being terminated at any moment without notice. In such dynamic contexts, deployment times in the order of tens of minutes are not acceptable.

As VM instances typically access only a small fraction of the VM image throughout their run-time, one attractive alternative to broadcasting is to fetch only the necessary parts on-demand. Such a "lazy" transfer scheme is gaining increasing popularity [8], however it comes at the price of making the boot process longer, as the necessary parts of the image not available locally need to be fetched remotely from the cloud repository.

In this paper we investigate how to improve on-demand transfer schemes for the multi-deployment pattern. Our proposal relies on the fact that the hypervisors will generate highly similar access patterns to the image during the boot process. Under these circumstances, we exploit small delays between the times when the VM instances access the same chunk (due to jitter in execution time) in order to prefetch the chunk for the slower instances based on the experience of the faster ones. Our approach does not require any foreknowledge of the access pattern and dynamically adapts to it as the instances progress in time. A multi-deployment can thus benefit from our approach even when it is launched for the first time, with subsequent runs fully benefiting from complete access pattern characterization.

We summarize our contributions as follows:

- We introduce an approach that optimizes the multi-deployment pattern by means of adaptive prefetching and show how to integrate this approach in IaaS architectures. (Sections 2.1 and 2.2)
- We propose an implementation of these design principles by enriching the metadata structures of BlobSeer [9, 10], a distributed storage service designed to sustain a high throughput even under concurrency (Section 2.3).
- We experimentally evaluate the benefits of our approach on the Grid'5000 [11] testbed by performing multi-deployments on hundreds of nodes (Section 3).

2 Our approach

In this section we present the design principles behind our proposal, show how to integrate them in cloud architectures and propose a practical implementation.

2.1 Design principles

Stripe VM images in a distributed repository. In most cloud deployments [3–5], the disks locally attached to the compute nodes are not exploited to their full potential: they typically serve to cache VM images and provide temporary storage for the running VM instances. Most of the time, this access pattern utilizes only a small fraction of the total disk size. Therefore, we propose to aggregate the storage space of the local disks in a common pool that is used as a distributed VM image repository. This specialized service stores the images in a striped fashion: VM images are split into small equally-sized chunks that are evenly distributed among the local disks of the compute nodes. When the hypervisor running on a compute node needs to read a region of the VM image that has not been locally cached yet, the corresponding chunks are fetched in parallel from the remote disks storing them. Under concurrency, this scheme effectively enables an even distribution of the read workload, which ultimately improves overall throughput.

Record the access pattern and use it to provide prefetching hints to subsequent remote reads. According to our observations, a multi-deployment generates a read access pattern to the VM image that exhibits two properties: (1) only a small part of the VM image is actually accessed during the boot phase (boot-sector, kernel, configuration files, libraries and daemons, etc.) and (2) read accesses follow a similar pattern on all VM instances, albeit at slightly different moments in time.

For example, Figure 1 shows the read access pattern for a multi-deployment of 100 instances booting a Debian Sid Linux distribution from a 2 GB large virtual raw image striped in chunks of 256 KB. The read access pattern is represented in terms of what chunks are accessed (disk offset) as time progresses. A line corresponds to each chunk and indicates the minimum, average and maximum time since the beginning of the multi-deployment when the chunk was accessed by the instances. It can be noticed that a large part of the disk remains untouched, with significant jitter between the times when the same chunk is accessed.

Based on these observations, we propose to monitor two attributes: the total number of accesses to a chunk and the average access time since the beginning of the multi-deployment. These two attributes help establish the order in which chunks are accessed during the boot phase, with an increasing number of accesses leading to a higher precision. Both attributes are updated in real time for each chunk individually. Using this information, the slower instances to access a chunk can "learn from the experience" of the faster ones: they can query the metadata in order to predict what chunks will probably follow and prefetch them

in the background. As shown on Figure 1, gaps between periods of I/O activity and I/O inactivity are in the order of seconds, large enough enable prefetching of considerable amounts of data.

To minimize the query overhead, we propose to piggyback information about potential chunk candidates for prefetching on top of every remote read operation to the repository. We refer to this extra information as *prefetching hints* from now on. Since remote read operations need to consult the metadata that indicates where the chunks are stored anyway, the extra overhead in order to build prefetching hints is small. However, too many prefetching hints are not needed and only generate unnencessary overhead. Thus, we limit the number of results (and thereby the number of "false positives") by introducing an *access count threshold* that needs to be reached before a chunk is considered as a viable candidate.

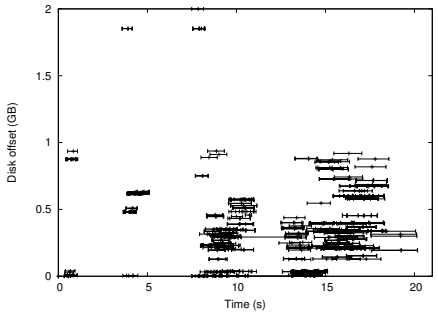


Fig. 1. Accesses to the VM image during a multi-deployment of 100 VM instances

An example for an access threshold of 2 is depicted in Figure 3(a), where 4 instances that are part of the same multi-deployment access the same initial VM image, which is striped into four chunks: A, B, C and D. Initially, all four instances need to fetch chunk A, which does not generate any prefetching hints, as it is the only chunk involved in the requests. Next, the first instance fetches chunk B, followed by instances 2 and 3, both of which fetch chunk C. Finally instance 4 fetches chunk D. Since B is accessed only once, no prefetching hints are generated for instances 2 and 3, while chunk C becomes a prefetching hint for instance 4.

Note that a growing number of chunks that need to be stored in the repository (as a result of adding new VM images) can lead to a high overhead of building prefetching hints, which can even offset the benefits of prefetching. This in turn leads to the need to implement a scalable distributed metadata management scheme (see Section 2.3).

Prefetch chunks in the background using the hints. The prefetching hints returned with each remote read operation can be combined in order to build a prefetching strategy in the background that operates during the periods of I/O inactivity. Note that this scheme is self-adaptive: it can learn on-the-fly

about “unknown” VM images during the first multi-deployment, with no need for pre-staging. After the first run, the whole access pattern can be completely characterized in terms of prefetching hints immediately after the first read occurred, which leaves room to employ optimal prefetching strategies for subsequent multi-deployments.

2.2 Architecture

A simplified IaaS cloud architecture that integrates our approach is depicted in Figure 2. The typical elements of a IaaS architecture are illustrated with a light background, while the elements that are part of our proposal are highlighted by a darker background.

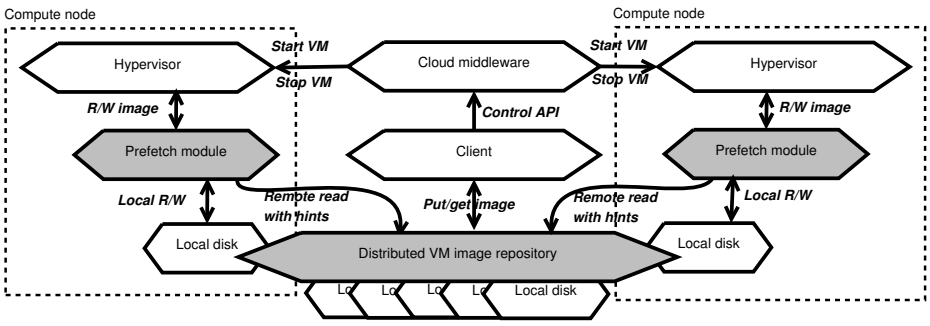


Fig. 2. Cloud architecture that integrates our approach (dark background)

A *distributed storage service* is deployed on all compute nodes. It aggregates the space available on the local disks in a common shared pool that forms the virtual machine image repository. This storage service is responsible to transparently stripe the virtual machine images into chunks. The *cloud client* has direct access to the repository and is allowed to upload and download VM images from it. Furthermore, the cloud client also interacts with the *cloud middleware* through a control API that enables launching and terminating multi-deployments. It is the responsibility of the cloud middleware to initiate the multi-deployment by concurrently launching the hypervisors on the compute nodes.

The *hypervisor* in turn runs the VM instances and issues corresponding reads and writes to the underlying virtual machine images. The reads and writes are intercepted by a *prefetching module*, responsible to implement the design principles proposed in Section 2.1. More specifically, writes are redirected to the local disk (using either mirroring [12] or copy-on-write [13]). Reads are either

served locally, if the involved chunks are already available on the local disk, or transferred first from the repository to the local disk otherwise. Each read brings new prefetching hints that are used to transfer chunks in the background from the repository to the local disk.

2.3 Implementation

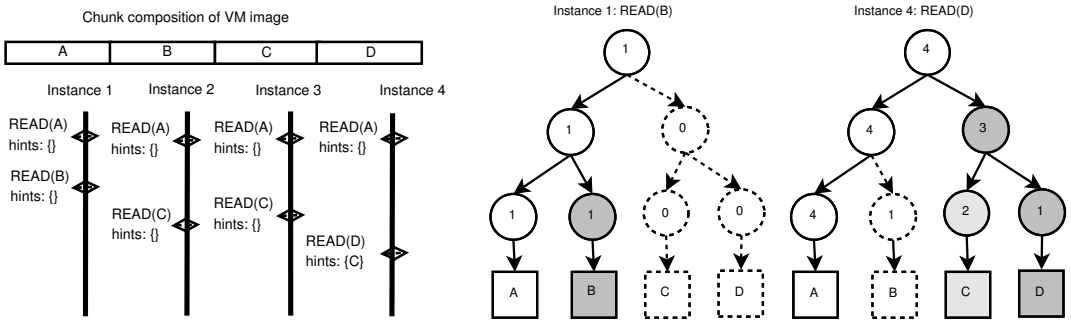
We have chosen to implement the distributed VM image repository on top of *BlobSeer* [9, 10]. This choice was motivated by several factors. First, BlobSeer enables *scalable aggregation of storage space* from the participating nodes with low overhead in order to store *BLOBs* (Binary Large OBjects). BlobSeer handles striping and chunk distribution of BLOBs transparently, which can be directly leveraged in our context: each VM image is stored as a BLOB, effectively eliminating the need to perform explicit chunk management.

Second, BlobSeer uses a distributed metadata management scheme based on *distributed segment trees* [10] that can be easily adapted to efficiently build prefetching hints. More precisely, a distributed segment tree is a binary tree where each tree node covers a region of the BLOB, with the leaves covering individual chunks. The tree root covers the whole BLOB, while the other non-leaf nodes cover the combined range of their left and right children. Reads of regions in the BLOB imply descending in the tree from the root towards the leaves, which ultimately hold information about the chunks that need to be fetched.

In order to minimize the overhead of building prefetching hints, we add additional metadata to each tree node such that it records the total number of accesses to that node. Since a leaf can be reached only by walking down into the tree, the number of accesses to inner nodes is higher than the number of accesses to leaves. Thus, if the access count threshold is not reached, the whole sub-tree can be skipped, greatly limiting the number of chunks that need to be inspected in order to build the prefetching hints.

Furthermore, we designed a metadata caching scheme that avoids unnecessary remote accesses to metadata: each tree node that that has reached the threshold since it was visited the last time, is added to the cache and retrieved from there for any subsequent visits. Cached tree nodes might not always reflect an up-to-date number of accesses, however this does not affect correctness as the number of accesses can only grow higher than the threshold. Obviously, the tree nodes that are on the path towards the required chunks (i.e. those chunks that make up the actual read request) need to be visited even if they haven't reached the threshold yet, so they are added to the cache too.

An example of how this works is presented in Figures 3(b) and 3(c). Each tree node is labeled with the number of accesses that is reflected in the local cache. Figure 3(b) depicts the contents of the cache for Instance 1 at the moment



(a) Evolution of remote fetches in time and the associated hints

(b) Local view of the segment tree for Instance 1 after reading chunk B

(c) Local view of the segment tree for Instance 4 after reading chunk D

Fig. 3. Adaptive prefetching by example: multi-deployment of 4 instances with a prefetch threshold of 2

when it reads chunk B. White nodes were previously added in the cache when Instance 1 accessed chunk A (access count 1 because it was the first to do so). Dark grey nodes are on the path towards chunk B and are therefore added to the local cache during the execution of the read access. Since the access count of the right child of the root is below the threshold, the whole right subtree is skipped (dotted pattern). Similarly, Figure 3(c) depicts the segment tree at the moment when Instance 4 reads chunk D. Again, white nodes on the path towards chunk A are already in the cache. Dark grey nodes are on the path towards chunk D and are about to be added in the cache. Since the access count of the leaf corresponding to chunk C (light grey) has reached the threshold, it is added to the cache as well and C becomes a prefetching hint, while the leaf of chunk B is skipped (dotted pattern).

Using this scheme, each read from the BLOB potentially returns a series of prefetching hints that are used to prefetch chunks in the background. This is done in a separate thread during the periods of I/O inactivity of the hypervisor. If a read is issued that does not find the required chunks locally, the prefetching is stopped and the required chunks are fetched first, after which the prefetching is resumed. We employ a prefetching strategy that gives priority to the most frequently accessed chunk.

3 Experimental evaluation

This section presents a series of experiments that evaluate how well our approach performs under the multi-deployment pattern, when a single initial VM image is used to concurrently instantiate a large number of VM instances.

3.1 Experimental setup

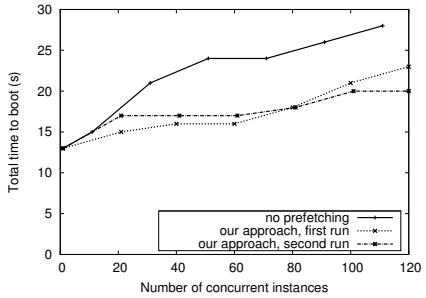
The experiments presented in this work have been performed on Grid'5000 [11], an experimental testbed for distributed computing that federates 9 different sites in France. We have used the clusters located in Nancy. All nodes of Nancy, numbering 120 in total, are outfitted with x86_64 CPUs offering hardware support for virtualization, local disk storage of 250 GB (access speed $\simeq$55 MB/s) and at least 8 GB of RAM. The nodes are interconnected with Gigabit Ethernet (measured: 117.5 MB/s for TCP sockets with MTU = 1500 B with a latency of $\simeq$0.1 ms). The hypervisor running on all compute nodes is KVM 0.12.5, while the operating system is a recent Debian Sid Linux distribution. For all experiments, a 2 GB raw disk image file based on the same Debian Sid distribution was used.

3.2 Performance of multi-deployment

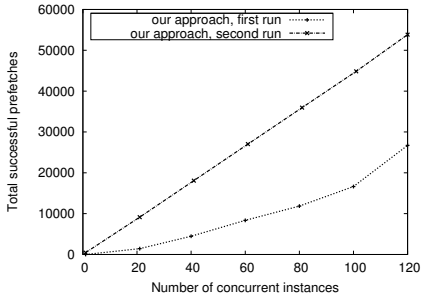
We perform series of experiments that consists in concurrently deploying an increasing number of VMs, one VM on each compute node. For this purpose, we deploy BlobSeer on all of the 120 compute nodes and store the initial 2 GB large image in a striped fashion into it. The chunk size was fixed at 256 KB, large enough to cancel the latency penalty for reading many small chunks, yet small enough to limit the competition for the same chunk. All chunks are distributed using a standard round-robin allocation strategy. Once the VM image was successfully stored, the multi-deployment is launched by synchronizing KVM to start on all the compute nodes simultaneously.

A total of three series of experiments is performed. In the first series, the original implementation with no prefetching is evaluated. In the second series of experiments, we evaluate our approach when a multi-deployment is launched for the first time such that no previous information about the access pattern is available, which essentially forces the system to self-adapt according to the prefetching hints. We have fixed the access count threshold to be 10% of the total number of instances in the multi-deployment. Finally, the third series of experiments evaluates our approach when a multi-deployment was already launched before, such that its access pattern has been recorded. This scenario corresponds to the ideal case when complete information about the access pattern is available from the beginning.

Performance results are depicted in Figure 4. As can be observed, a larger multi-deployment leads to a steady increase in the total time required to boot all VM instances (Figure 4(a)), for all three scenarios. This is both the result of increased read contention to the VM image, as well as increasing jitter in execution time. However, prefetching chunks in the background clearly pays



(a) Total time to boot all VM instances of a multi-deployment



(b) Total number of remote accesses that were avoided for reads issued by the hypervisor as the result of successful prefetches

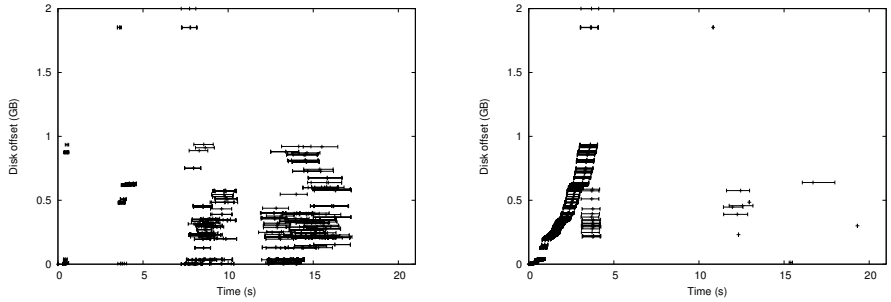
Fig. 4. Performance of self-adaptive prefetching when increasing the number of VM instances in the multi-deployment

off: for 120 instances, our self-adaptation technique lowers the total time to boot by 17% for the first run and almost 35% for subsequent runs, once the access pattern has been learned.

Figure 4(b) shows the number of successful prefetches of our approach as the number of instances in the multi-deployment grows. For the second run, almost all of the $\simeq$450 chunks are successfully prefetched by each instance, for a total of $\simeq$54000 prefetches. As expected, for the first run it can be clearly observed that a higher number of concurrent instances benefits the learning process more, as there are more opportunities to exploit jitter in execution time. For 120 instances, the total number of successful prefetches is about half compared to the second and subsequent runs.

Figures 5(a) and 5(b) show the remote read access pattern for a multi-deployment of 100 instances: for our approach during the first run and the second run respectively. Each line represents the minimum, average and maximum time from the beginning of the deployment when the same chunk (identified by its offset in the image) was accessed by the VM instances. The first run of our approach generates a similar pattern with the case when no prefetches are performed (represented in Figure 1). While jitter is still observable, thanks to our prefetching hints the chunks are accessed earlier, with average access times slightly shifted towards the minimum access times.

Once the access pattern has been learned, the second run of our approach (Figure 5(b)) is able to prefetch the chunks much faster, in less than 25% of the total execution time. This prefetching rush slightly increases both the remote read contention and jitter in the beginning of the execution but with the benefit



(a) Remote accesses during the learning phase of the first-time run

(b) Remote accesses for the second and subsequent runs

Fig. 5. Remote accesses to the VM image during a multi-deployment of 100 VM instances using our approach

of reducing both parameters during the rest of the execution. Thus, jitter accumulates to a lesser extent for a small number of concurrent instances and could be a possible explanation of why the first run is actually slightly faster than the second run for smaller multi-deployments.

4 Related work

Many hypervisors provide native copy-on-write support by defining custom virtual machine image file formats (such as [13]). They rely on a separate read-only template as the backing image file, while storing local modifications in the derived copy-on-write file. Much like our approach, a parallel file system [14–16] can be relied upon to stripe and distribute the read-only image template among multiple storage elements. However, unlike our approach, a parallel file system is not specifically optimized for multi-deployments and thus does not perform prefetching that is aware of the global trend in the access pattern.

Several storage systems such as Amazon S3 [17] (backed by Dynamo [18]) have been specifically designed as highly available key-value repositories for cloud infrastructures. They are leveraged by Amazon to provide elastic block level storage volumes (EBS [8]) that support striping and lazy, on-demand fetching of chunks. Amazon enables the usage of EBS volumes to store VM images, however we are not aware of any particular optimizations for the multi-deployment pattern.

Finally, dynamic analysis of access patterns was proposed in [19] for the purpose of building adaptive prefetching strategies. The proposal uses heuristic functions to predict the next most probable disk access using the recent reference history. The algorithms involved however are designed for centralized

approaches. They typically utilize only a small recent window of the reference history in order to avoid computational overhead associated with prefetching. Thanks to our distributed metadata management scheme, we can maintain a full access history that represents the global trend of the multi-deployment, which can be leveraged to perform an optimal prefetching after the first run.

5 Conclusions

This paper proposed a self-adaptive prefetching mechanism for "lazy" transfer schemes that avoid full broadcast of VM images during multi-deployments on IaaS clouds. We rely on the fact that all VM instances generate a highly similar access pattern, which is slightly shifted in time due to runtime jitter. Our proposal exploits this jitter to enable VM instances to learn from experience of the other concurrently running VM instances in order to speed-up reads not already cached on the local disk by prefetching the necessary parts of the VM image from the repository.

Our scheme is highly adaptive and does not require any past traces of the deployment, bringing a speed-up of up to 17% for the first run when compared to simple, on-demand fetching only. Once the access pattern has been learned, subsequent multi-deployments of the same VM image benefit from the full access history and can perform an optimal prefetching that further increases the speed-up to up to 35% compared to the case when no prefetching is performed.

Thanks to these encouraging results, we plan to further investigate the potential benefits of exploiting the similarity of access pattern to improve multi-deployments. In particular, we see a good potential to reduce the prefetching overhead by means of replication and plan to investigate this issue more closely. Furthermore, an interesting direction to explore is the use of push approaches (rather then pull) using broadcast algorithms once the access pattern has been learned.

Acknowledgments

Experiments presented in this paper were carried out using the Grid'5000 experimental testbed, an initiative from the French Ministry of Research through the ACI GRID incentive action, INRIA, CNRS and RENATER and other contributing partners (see http://www.grid5000.fr/).

References

1. Armbrust, M., Fox, A., Griffith, R., Joseph, A., Katz, R., Konwinski, A., Lee, G., Patterson, D., Rabkin, A., Stoica, I., Zaharia, M.: A view of cloud computing. Commun. ACM **53** (April 2010) 50–58

2. Buyya, R., Yeo, C.S., Venugopal, S.: Market-oriented cloud computing: Vision, hype, and reality for delivering it services as computing utilities. In: HPCC '08: Proceedings of the 2008 10th IEEE International Conference on High Performance Computing and Communications, Washington, DC, USA, IEEE Computer Society (2008) 5–13
3. : Amazon Elastic Compute Cloud (EC2). http://aws.amazon.com/ec2/
4. : Nimbus. http://www.nimbusproject.org/
5. : Opennebula. http://www.opennebula.org/
6. Wartel, R., Cass, T., Moreira, B., Roche, E., Manuel Guijarro, S.G., Schwickerath, U.: Image distribution mechanisms in large scale cloud providers. In: CloudCom '10: Proc. 2nd IEEE International Conference on Cloud Computing Technology and Science, Indianapolis, USA (2010) In press.
7. Andrzejak, A., Kondo, D., Yi, S.: Decision model for cloud computing under sla constraints. In: Proceedings of the 2010 IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems. MASCOTS '10, Washington, DC, USA, IEEE Computer Society (2010) 257–266
8. : Amazon elastic block storage (ebs). http://aws.amazon.com/ebs/
9. Nicolae, B.: BlobSeer: Towards efficient data storage management for large-scale, distributed systems. PhD thesis, University of Rennes 1 (November 2010)
10. Nicolae, B., Antoniu, G., Bougé, L., Moise, D., Carpen-Amarie, A.: Blobseer: Next-generation data management for large scale infrastructures. J. Parallel Distrib. Comput. **71** (February 2011) 169–184
11. Bolze, R., Cappello, F., Caron, E., Daydé, M., Desprez, F., Jeannot, E., Jégou, Y., Lanteri, S., Leduc, J., Melab, N., Mornet, G., Namyst, R., Primet, P., Quetier, B., Richard, O., Talbi, E.G., Touche, I.: Grid'5000: A large scale and highly reconfigurable experimental grid testbed. Int. J. High Perform. Comput. Appl. **20** (November 2006) 481–494
12. Nicolae, B., Bresnahan, J., Keahey, K., Antoniu, G.: Going Back and Forth: Efficient Multi-Deployment and Multi-Snapshotting on Clouds. In: HPDC '11: The 20th International ACM Symposium on High-Performance Parallel and Distributed Computing, San José, CA United States (2011)
13. Gagné, M.: Cooking with linux: still searching for the ultimate linux distro? Linux J. **2007**(161) (2007) 9
14. Carns, P.H., Ligon, W.B., Ross, R.B., Thakur, R.: Pvfs: A parallel file system for Linux clusters. In: Proceedings of the 4th Annual Linux Showcase and Conference, Atlanta, GA, USENIX Association (2000) 317–327
15. Weil, S.A., Brandt, S.A., Miller, E.L., Long, D.D.E., Maltzahn, C.: Ceph: a scalable, high-performance distributed file system. In: Proceedings of the 7th symposium on Operating systems design and implementation. OSDI '06, Berkeley, CA, USA, USENIX Association (2006) 307–320
16. Schmuck, F., Haskin, R.: Gpfs: A shared-disk file system for large computing clusters. In: Proceedings of the 1st USENIX Conference on File and Storage Technologies. FAST '02, Berkeley, CA, USA, USENIX Association (2002)
17. : Amazon Simple Storage Service (S3). http://aws.amazon.com/s3/
18. DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., Vogels, W.: Dynamo: Amazon's highly available key-value store. In: SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles, New York, NY, USA, ACM (2007) 205–220
19. Zhu, Q., Gelenbe, E., Qiao, Y.: Adaptive prefetching algorithm in disk controllers. Perform. Eval. **65** (May 2008) 382–395