On the Benefits of Transparent Compression for Cost-Effective Cloud Data Storage

Bogdan Nicolae

INRIA Saclay, Île-de-France
bogdan.nicolae@inria.fr

Abstract. Infrastructure-as-a-Service (IaaS) cloud computing has revolutionized the way we think of acquiring computational resources: it allows users to deploy virtual machines (VMs) at large scale and pay only for the resources that were actually used throughout the runtime of the VMs. This new model raises new challenges in the design and development of IaaS middleware: excessive storage costs associated with both user data and VM images might make the cloud less attractive, especially for users that need to manipulate huge data sets and a large number of VM images. Storage costs result not only from storage space utilization, but also from bandwidth consumption: in typical deployments, a large number of data transfers between the VMs and the persistent storage are performed, all under high performance requirements. This paper evaluates the trade-off resulting from transparently applying data compression to conserve storage space and bandwidth at the cost of slight computational overhead. We aim at reducing the storage space and bandwidth needs with minimal impact on data access performance. Our solution builds on BlobSeer, a distributed data management service specifically designed to sustain a high throughput for concurrent accesses to huge data sequences that are distributed at large scale. Extensive experiments demonstrate that our approach achieves large reductions (at least 40%) of bandwidth and storage space utilization, while still attaining high performance levels that even surpass the original (no compression) performance levels in several data-intensive scenarios.

1 Introduction

The emerging cloud computing model [1, 2, 28] is gaining serious interest from both industry [31] and academia [16,23,35] for its proposal to view the computation as a utility rather than a capital investment. According to this model, users do not buy and maintain their own hardware, nor have to deal with complex large-scale application deployments and configurations, but rather rent such resources as a service, paying only for the resources their computation has used throughout its lifetime.

There are several ways to abstract resources, with the most popular being Infrastructure-as-a-Service (IaaS). In this context, users rent raw computational resources as virtual machines that they can use to run their own custom applications, paying only for the computational power, network traffic and storage space used by their virtual environment. This is highly attractive for users that cannot afford the hardware to run large-scale, distributed applications or simply need flexible solutions to scale to the size of their problem, which might grow or shrink in time (e.g. use external cloud resources to complement their local resource base [14]).

However, as the scale and variety of data increases at a fast rate [5], the cost of processing and maintaining data remotely on the cloud becomes prohibitively expensive. This cost is the consequence of excessive utilization of two types of resources: *storage space* and *bandwidth*.

Obviously, a large amount of storage space is consumed by the data itself, while a large amount of bandwidth is consumed by the need to access this data. Furthermore, in order to achieve scalable data processing performance, users often employ data intensive paradigms (such as MapReduce [3] or Dryad [10]) that generate massively parallel accesses to the data, and have additional *high aggregated throughput* requirements.

Not so obvious is the additional storage space and bandwidth utilization overhead introduced by operating the virtual machine images that host the user application. A common patten on IaaS clouds is the need to deploy a large number of VMs on many nodes of a data-center at the same time, starting from a set of VM images previously stored in a persistent fashion. Once the application is running, a similar challenge applies to snapshotting the deployment: many VM images that were locally modified need to be concurrently transferred to stable storage with the purpose of capturing the VM state for later use. Both these patterns lead to a high storage space and bandwidth utilization. Furthermore, they have the same high aggregated throughput requirement, which is crucial in order to minimize the overhead associated with virtual machine management.

Therefore, there is a need to optimize three parameters simultaneously: (1) conserve storage space; (2) conserve bandwidth; and (3) deliver a high data-access throughput under heavy access concurrency.

This paper focuses on evaluating the benefits of applying data compression *transparently* on the cloud, with the purpose of achieving a good trade-off for the optimization requirements mentioned above. Our contributions can be summarized as follows:

– We propose a generic sampling-based compression layer that dynamically adapts to heterogeneous data in order to deal with the highly concurrent access patterns generated by the deployment and execution of data-intensive applications. This contribution extends our previous proposal presented in [18]. In particular, we introduce a more generic compression layer that extends the applicability of our proposal to the management of virtual machine images (in addition to the management of application data) and show how to integrate it in the cloud architecture.
– We propose an implementation of the compression layer on top of *BlobSeer* [17, 20], a versioning-oriented distributed storage system specifically designed to deliver high throughputs under heavy access concurrency.

– We perform extensive experimentations on the Grid5000 testbed [12] that demonstrate the benefits of our approach. In particular, in addition to the experiments presented in [18], we highlight the benefits of our approach for virtual machine image storage.

2 Our approach

In this section we present an adaptive, transparent compression layer that aims at reducing the space and bandwidth requirements of cloud storage with minimal impact on I/O throughput when under heavy access concurrency.

2.1 General considerations

Several important factors that relate to the data composition and access patterns need to be taken into consideration when designing a compression layer for the cloud. We enumerate these factors below:

Transparency. In a situation where the user is running the application on private-owned hardware and has direct control over the resources, compression can be explicitly managed at application level. This approach is often used in practice because it has the advantage of enabling the user to tailor compression to the specific needs of the application. However, on clouds, explicit compression management at application level is not always feasible. For example, many cloud providers offer data-intensive computing platforms (such as Elastic MapReduce [32] from Amazon) directly as a service to their customers. These platforms are based on paradigms (such as MapReduce [3]) that abstract data access, forcing the application to be written according to a particular schema which makes explicit compression management difficult.

Furthermore, besides application data, users customize and store virtual machine images on the cloud that are then used to deploy and run their application. However, users are not allowed to directly control the deployment process of virtual machine images and therefore cannot apply custom compression techniques on the images.

For these two reasons, it is important to handle compression *transparently* and offer it as an extra feature to the users, potentially reducing their storage and bandwidth costs with minimal impact on quality-of-service.

Random access support. Most compression algorithms are designed to work with data streams: new data is fed as input to the algorithm, which in turn tries to shrink it by using a dynamic dictionary to replace long patterns that were previously encountered with shorter ones. This is a process that destroys the original layout of the data: a subsequence of the original uncompressed data stream that starts at an arbitrary offset cannot be easily obtained from the compressed data stream without decompressing the whole data.

However, random access to data is very common on clouds. For example, a virtual machine typically does not access the whole contents of the underlying virtual machine image during its execution: only the parts that are needed to boot the virtual machine and run the user application are accessed. Furthermore, application data is typically organized in huge data objects that comprise many small KB-sized records, since it is unfeasible to manage billions of small, separate data objects explicitly [7]. The data is then processed in a distributed fashion: different parts of the same huge data object are read and written concurrently by the virtual machines. Again, this translates to a highly concurrent random read and write access pattern to the data objects.

Therefore, it is important to design a compression layer that is able to overcome the limitations of stream-based compression algorithms and introduce support for *efficient random access* to the underlying data.

Heterogeneity of data. First of all, compression is obviously only useful as long as it shrinks the space required to store data. However, on clouds, the data that needs to be stored is highly heterogeneous in nature.

Application data is mostly in unstructured form. It either consists of *text* (e.g., huge collections of documents, web pages and logs [25]) or *multimedia* (e.g., images, video and sound [26]). While text data is known to be highly compressible, multimedia data is virtually not compressible and in most cases trying to apply any compression method on it actually increases the required storage space and generates unnecessary computational overhead.

Virtual machine images typically represent the contents of the virtual disks attached to the virtual machine. Some parts are claimed by the file system and hold executables, archived logs, etc. These parts are typically not compressible. Other parts are not used by the file system (zero-sequences of bytes) or hold logs, configuration files, etc., which makes them highly compressible.

Thus, the compression layer needs to dynamically adapt to the type of data stored on the cloud, dealing efficiently with both compressible and incompressible data.

Computational overhead. Compression and decompression invariably leads to a computational overhead that diminishes the availability of compute cores for effective application computations. Therefore, this overhead must be taken into account when designing a high-performance compression layer. With modern high-speed networking interfaces, high compression rates might become available only at significant expense of computation time. Since the user is not paying only for storage space and bandwidth, but for the CPU utilization as well, choosing the right trade-off is often difficult.

Memory overhead. Deploying and running virtual machines takes up large amounts of main memory from the nodes of the cloud that host them. Given this context, main memory is a precious resource that has to be carefully managed. It is therefore crucial to design a compression layer that minimizes the extra main memory required for compression and decompression.

2.2 Design principles

In order to deal with the issues presented above, we propose the following set of design principles:

Data striping. A straight-forward way to apply compression is to fully compress the data before sending it remotely in case of a write operation, respectively to wait for the compressed data to arrive and then decompress it in case of a read operation. However, this approach has a major disadvantage: the compression/decompression does not run in parallel with the data transfer, potentially wasting computational power that is idle during the transfer. For this reason, we propose the use of data striping: the piece of data is split into chunks and each chunk is compressed independently. This way, in the case of a write, a successfully compressed chunk can be sent before all other chunks have finished compressing, while in the case of a read, a fully received chunk can be decompressed before all other chunks have been successfully received. Thus, data striping enables overlapping of compression with data transfers, increasing the overall achieved throughput.

At the same time, data striping deals with the random access limitation of many compression algorithms. By compressing each chunk individually, reads and writes and random offsets involve only the chunks that cover the requested range delimited by offset and size. If the data is split at fine granularity, the overhead of random access becomes negligible. However, a chunk size that is too small may limit the potential of compression because fewer repeating patterns appear in smaller chunks. Therefore, it is important to find the right trade-off when choosing the chunk size.

Sampling of chunks. Since the system needs to adapt to both compressible and incompressible data, there is a need to determine efficiently when to compress a chunk and when to leave it in its original form. Obviously, attempting to compress the whole chunk and evaluating the result generates unacceptably high overhead. Therefore, we need a way to predict whether it is useful to apply compression or not. For this reason, we propose to *sample* each chunk, i.e. pick a small random part of the chunk and apply compression on it. Under the assumption that the obtained compression ratio predicts the compression ratio that would have been obtained by compressing the whole chunk itself, the chunk will be compressed only if the compression ratio of the small piece of random data is satisfactory.

The risk of wrong predictions is very low, because the chunk size is much smaller compared to the size of the whole data object, making it likely that the data of the small random part is of the same nature as the data of the whole chunk. Even when a prediction is wrong, given the small size of the chunk, the impact is minimal: either a chunk remains uncompressed when it should have been compressed, which has no negative impact on performance and a minimal impact on saved storage space, or, a chunk is compressed when it shouldn't, which has a minimal computational overhead and no negative impact on storage

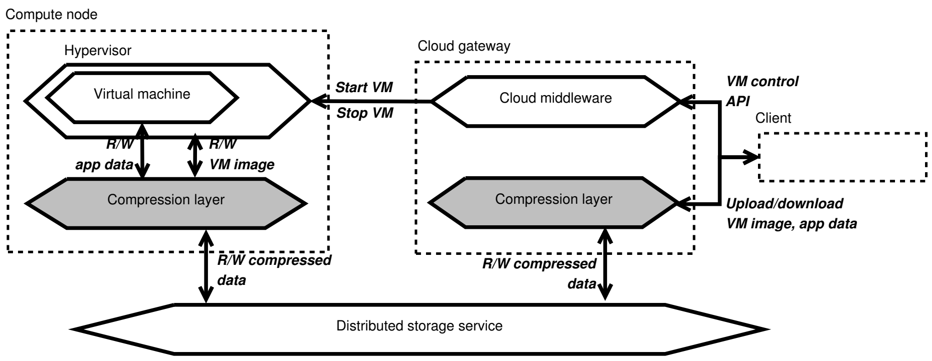


Fig. 1. Cloud architecture that integrates our proposal (dark background)

space (because the result of the compression is discarded and the chunk is stored in its original uncompressed form).

Configurable compression algorithm. Dealing with the computation and memory overhead of compressing and decompressing data is a matter of choosing the right algorithm. A large set of compression algorithms have been proposed in the literature that trade off compression ratio for computation and memory overhead. However, since the compression ratio relates directly to storage space and bandwidth costs, the user should be allowed to configure the algorithm in order to be able to fine-tune this trade-off according to the needs.

2.3 Architecture

Starting from the design principles presented above, we propose a compression layer that integrates in the cloud as shown in Figure 1. The typical elements found in the cloud are illustrated with a light background, while the compression layer is highlighted by a darker background.

The following actors are present:

– **Cloud middleware**: is responsible to manage the physical resources on the cloud: it schedules where new virtual machines are instantiated, it keeps track of consumed resources for each user, it enforces policies, etc. The cloud middleware exposes a control API that enables users to perform a wide range of management tasks: VM deployment and termination, monitoring, etc.
– **Distributed storage service**: is responsible to organize and store the data on the cloud. It acts as a data sharing service that facilitates transparent access to the data within given quality-of-service guarantees (performance, data availability, etc.) that are established by the cloud provider in the service level agreement.
– **Cloud client**: it uses the control API of the cloud middleware in order to interact with the cloud. It also accesses the data storage service in order to manipulate virtual machine images and application data.

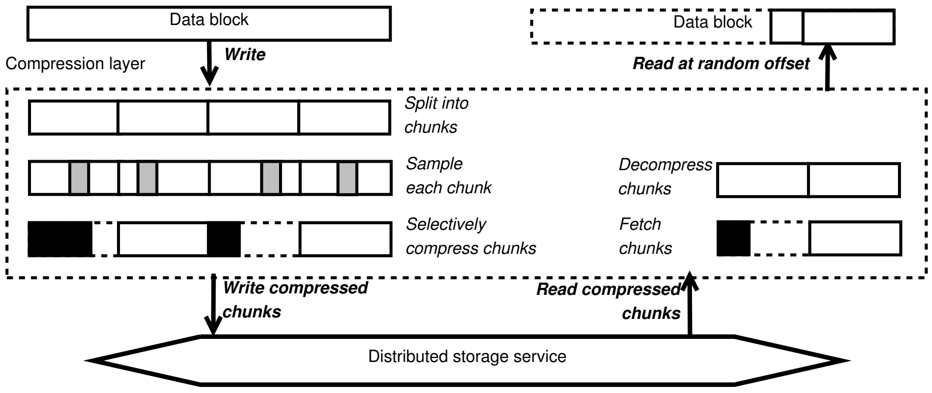


Fig. 2. Zoom on the compression layer

- **Hypervisor**: is the virtualization middleware that leverages the physical resources of the compute nodes to present a virtual operating platform for the virtual machines. In this role, it emulates a virtual file system that is backed up by a virtual machine image, which is a regular file that is accessible from the compute node's host file system.
- **Virtual machine**: represents the virtual environment in which the guest operating system and user applications are running. Virtual machines can communicate with each other and share application data through the distributed storage service.
- **Compression layer**: traps all data accesses (both application data and virtual machine images) and treats them according to the principles presented in Section 2.2. It runs both on the compute nodes and on the cloud gateways, mediating the interactions of the clients, hypervisors and virtual machines with the distributed storage service.

Figure 2 zooms on the compression layer, which is responsible to trap all read and write accesses to the distributed storage service.

In case a write operation is performed, after the data is split into chunks, a small random sample of each chunk is compressed in order to probe whether the chunk is compressible or not. If the achieved compression ratio is higher than a predefined threshold, then the whole chunk is compressed and the result is written to the distributed storage service. If the achieved compression ratio is lower than the threshold, then the chunk is written directly to the distributed storage service without any modification.

In case a read operation is performed, first all chunks that cover the requested range (delimited by offset and size) are determined. These chunks are then fetched from the storage service and decompressed if they were stored in compressed fashion. The uncompressed contents of each chunk is then placed at its relative offset in the local buffer supplied by the application. The read operation succeeds when all chunks have been successfully processed this way, filling the local buffer.

In both cases, the compression layer processes the chunks in a highly parallel fashion, potentially taking advantage of multi-core architectures. This enables overlapping of remote transfers to and from the storage service with the compression and decompression to high degree, minimizing the latency of read and write operations due to compression overhead.

Furthermore, since the compression layer is running on the client-side of the storage service (i.e. directly on the compute nodes and cloud gateways), the burden of compression and decompression is not falling on the storage service itself, which greatly enhances performance under specific access patterns, such as the case when the same chunk is accessed concurrently from multiple nodes.

3 Implementation

In this section we show how to efficiently implement our proposal such that it both achieves the design principles introduced in Section 2.2 and is easy to integrate in the cloud as shown in Section 2.3.

We have chosen to leverage *BlobSeer*, presented in Section 3.1, as the distributed storage service on top of which to implement our approach. This choice was motivated by two factors. First, BlobSeer implements out-of-the-box transparent data striping of large objects and fine-grain access to them, which enables easy implementation of our approach as it eliminates the need for explicit chunk management. Second, BlobSeer offers support for high throughput under concurrency, which enables efficient parallel access the chunks and therefore is crucial to achieving our high performance objective.

3.1 BlobSeer

This section introduces BlobSeer, a distributed data storage service designed to deal with the needs of data-intensive applications: *scalable aggregation of storage space* from the participating nodes with minimal overhead, support to store *huge data objects*, *efficient fine-grain access* to data subsets and ability to sustain a *high throughput under heavy access concurrency.*

Data is abstracted in BlobSeer as long sequences of bytes called BLOBs (Binary Large OBject). These BLOBs are manipulated through a simple access interface that enables creating a blob, reading/writing a range of *size* bytes from/to the BLOB starting at a specified *offset* and appending a sequence of *size* bytes to the BLOB. This access interface is designed to support versioning explicitly: each time a write or append is performed by the client, a new snapshot of the blob is generated rather than overwriting any existing data (but physically stored is only the difference). This snapshot is labeled with an incremental version and the client is allowed to read from any past snapshot of the BLOB by specifying its version.

Architecture. BlobSeer consists of a series of distributed communicating processes. Each BLOB is split into chunks that are distributed among *data providers.*

Clients read, write and append data to/from BLOBs. Metadata is associated to each BLOB and stores information about the chunk composition of the BLOB and where each chunk is stored, facilitating access to any range of any existing snapshot of the BLOB. As data volumes are huge, metadata grows to significant sizes and as such is stored and managed by the *metadata providers* in a decentralized fashion. A *version manager* is responsible to assign versions to snapshots and ensure high-performance concurrency control. Finally, a *provider manager* is responsible to employ a chunk allocation strategy, which decides what chunks are stored on which data providers, when writes and appends are issued by the clients. A *load-balancing* strategy is favored by the provider manager in such way as to ensure an even distribution of chunks among providers.

Key features. BlobSeer relies on *data striping*, *distributed metadata management* and *versioning-based concurrency control* to avoid data-access synchronization and to distribute the I/O workload at large-scale both for data and metadata. This is crucial for achieving a high aggregated throughput under concurrency, as demonstrated by our previous work [15, 19, 20, 22].

3.2 Integration with BlobSeer

Since BlobSeer implicitly performs data striping whenever a data block is written into it, we implemented the compression layer directly on top of the client-side networking layer of BlobSeer, which is responsible for remote communication with the data providers.

Instead of directly reading and writing the chunks from/to BlobSeer, the compression layer acts as a filter that performs the sampling and compresses the chunks when the requested operation is a write, respectively decompresses the chunks if the requested operation is a read. Careful consideration was given to keep the memory footprint to a minimum, relying in the case of incompressible chunks on *zero-copy* techniques, which eliminate the need to copy chunks from one memory region to another when they remain unchanged and are passed to the networking layer.

The compression layer was designed to be highly configurable, such that any compression algorithm can be easily plugged in. For the purpose of this paper we adopted two popular choices: Lempel-Ziv-Oberhumer(LZO) [34], based on the work presented in [30], which focuses on minimizing the memory and computation overhead, and BZIP2 [33], a free and open-source standard compression algorithm, based on several layers of compression techniques stacked on top of each other.

The versioning-based BLOB access API exposed by BlobSeer can be leveraged at application level directly, as it was carefully designed to enable efficient access to user data under heavy access concurrency. However, in order to leverage the compression layer for efficient storage of virtual machine images, we relied on our previous work presented in [21]: a dedicated virtual file system on build on top of BlobSeer, specifically optimized to efficiently handle two recurring virtual machine image access patterns on the cloud: multi-deployment and multi-snapshotting.

4 Experimental evaluation

In this section we evaluate the benefits of our approach by conducting a series of large-scale experiments that target the access patterns typically found on the clouds. In particular, we focus on two settings: (1) read and write access patterns as generated by data-intensive applications (Sections 4.2 and 4.3) and (2) multi-deployments of virtual machines (Section 4.4).

The motivation behind choosing the access patterns for the first setting is the fact that data-intensive computing paradigms are gaining increasing popularity as a solution to cope with growing data sizes. For example, MapReduce [3] has been hailed as a revolutionary new platform for large-scale, massively parallel data access [24]. Applications based on such paradigms continuously acquire massive datasets while performing (in parallel) large-scale computations over these datasets, generating highly concurrent read and write access patterns to user data. We therefore argue that experimenting with such access patterns is a good predictor of the potential benefits achievable in practice.

The motivation behind the choice for the second setting is the fact that multi-deployments are the most frequent pattern encountered on the clouds. Users typically need to deploy a distributed application that requires a large number of virtual machine instances, however, for practical reasons it is difficult to manually customize a separate virtual machine image for each instance. Therefore, users build a single virtual machine image (or a small initial set) that they upload to cloud storage and then use it as a template to initialize a large number of instances from it, which ultimately leads to the multi-deployment pattern.

In both settings we are interested in evaluating both the access performance of our approach (throughput and execution time), as well as the reductions in network traffic and storage space when compared to the original BlobSeer implementation that does not implement a compression layer.

4.1 Experimental setup

We performed our experiments on the Grid'5000 [12] testbed, a highly configurable and controllable experimental Grid platform gathering 9 sites in France. We used nodes belonging to two sites of Grid'5000 for our experiments: Rennes (122 nodes) and Lille (45 nodes). Each node is outfitted with dual-core or quad-core x86_64 CPUs (capable of hardware virtualization support) and have at least 4 GB of RAM. We measured raw buffered reads from the hard drives at an average of about 60MB/s, using the *hdparm* utility. Internode bandwidth is 1 Gbit/s (we measured 117.5 MB/s for TCP end-to-end sockets with MTU of 1500 B) and latency is 0.1 ms.

4.2 Concurrent writes of application data

This scenario corresponds to a typical user data acquisition phase, in which the user application, consisting of many distributed processing elements, gathers application data concurrently from multiple sources (e.g. web crawling or log parsing) and stores it in a huge BLOB for later processing. The scenario generates a write-intensive access pattern in which data is appended concurrently to the same BLOB. We aim at evaluating our approach under such heavy access concurrency circumstances, both in the case when the data to be processed is compressible and in the case when it is not.

In order to perform this evaluation, we use 122 nodes of the Rennes site and deploy BlobSeer on them as follows: 110 data providers are deployed on different nodes, with an additional 10 dedicated nodes reserved to deploy the metadata providers. The version manager and provider manager are deployed on dedicated nodes as well.

Both in the case of compressible and incompressible data, we measure the aggregated throughput achieved when N concurrent clients append 512 MB of data in chunks of 64MB to the same BLOB. Each of the clients runs in its own virtual machine that is co-deployed with a data provider on the same node. Each data provider is configured to use a cache of 512MB, which is deducted from the total available main memory for the client.

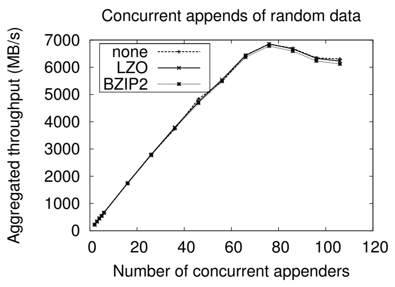
In the first case that corresponds to compressible data, we use the text of books available online. Each client builds the sequence of 512MB by assembling text from those books. In the second case, the sequence of 512MB is simply randomly generated, since random data is the worst case scenario for any compression algorithm.

We perform experiments in each of the cases using our implementation (for both LZO and BZIP2) and compare it to the reference BlobSeer implementation that does not integrate the compression layer. Each experiment is repeated three times for reliability and the results are averaged. The sample size used to decide whether to compress the chunk or not is fixed at 64KB.

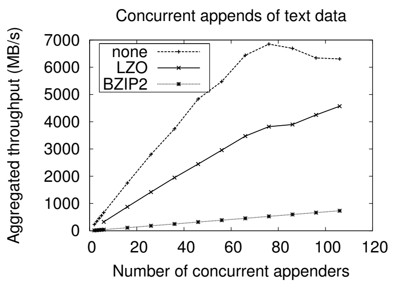
The obtained results are represented in Figure 3. The curves corresponding to random data (Figure 3(a)) are very close, clearly indicating that the *impact of sampling is negligible*, both for LZO and BZIP2. On the other hand, when using compressible text data (Figure 3(a)), the aggregated throughput in the case of LZO, although scaling, is significantly lower than the total aggregated throughput achieved when not compressing data. With less than 1 GB/s maximal aggregated throughput, performance levels in the case of BZIP2 are rather poor.

When transferring uncompressed data, an interesting effect is noticeable: past 80 concurrent appenders, the aggregated throughput does not increase but rather slightly decreases and then stabilizes. This effect is caused by the fact that concurrent transfers of such large amounts of data saturate the physical bandwidth limit of the system, which limits the achievable scalability.

With respect to storage space, gains from storing text data in compressed form are represented in Figure 4(b). With a consistent gain of about 40% of



(a) Writing incompressible random data: high aggregated throughput is ensured by negligible sampling overhead.



(b) Writing compressible text data: high aggregated throughput when using LZO.

Fig. 3. Impact of our approach on aggregated throughput under heavy concurrency. In both cases concurrent clients append each 512 MB of data which is transparently split into 64MB chunks.

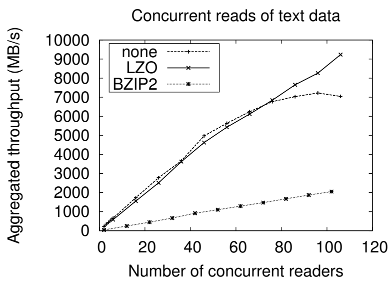
the original size, LZO compression is highly attractive. Although not measured explicitly, the same gain can be inferred for bandwidth utilization too. In the case of BZIP2, the gain reaches well over 60%, which makes up for the poor throughput.

4.3 Concurrent reads of application data

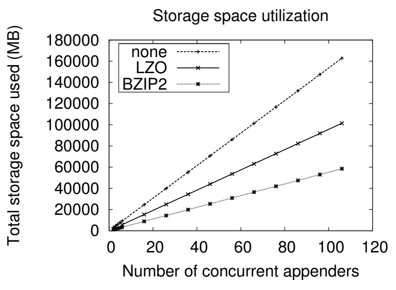
This scenario is complementary to the previous scenario and corresponds to a highly concurrent data processing phase in which the user application concurrently reads and processes different parts of the same BLOB in a distributed fashion.

Since our approach stores incompressible data in its original form, there is no difference between reading incompressible data using our approach and reading the data without any compression layer enabled. For this reason, we evaluate the impact of our approach for compressible data only. Assuming text data was written in compressed form as presented in the previous section, we aim at evaluating the total aggregated throughput that can be sustained by our approach when reading the data back.

We use the same 122 nodes of the Rennes site for our experiments and keep the same deployment settings: 110 data providers are deployed on different nodes, while each of the N clients is co-deployed with a data provider on the same node. One version manager, one provider manager and 10 metadata providers are each deployed on a dedicated node. We measure the aggregated throughput achieved when N concurrent clients read 512 MB of data stored in compressed chunks, each corresponding to 64MB worth of uncompressed data. Each client is configured to read a different region of the BLOB, such that no two clients access the same chunk concurrently, which is the typical case encountered in the data processing phase.



(a) Reading compressed text data: negligible decompression overhead for LZO reaches high aggregated throughput and outperforms raw data transfers.



(b) Counting the totals: BZIP2 saves more than 60% of storage space and bandwidth utilization. LZO reaches 40%.

Fig. 4. Impact of our approach on compressible text data: concurrent clients read 512MB of compressed data saved in chunks of 64MB (left); total bandwidth and storage space conserved (right).

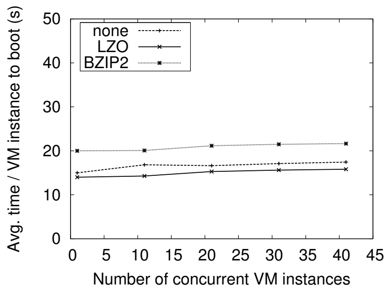
As with the previous setting, we perform three experiments and average the results. All clients of the same experiment read from the region of the BLOB generated by the corresponding append experiment, i.e. the first read experiment reads the data generated by the first append experiment, etc. This ensures that no requested data was previously cached and forces a "cold" run each time.

The results are represented in Figure 4(a). In the case of uncompressed data transfers, the aggregated throughput stabilizes at about 7 GB/s, because large data sizes are transferred by each client, which saturates the networking infrastructure. On the other hand, using LZO compression brings substantial read throughput improvements: the transfer of smaller compressed chunks combined with the fast decompression speed on the client side contribute to a steady increase in aggregated throughput that reaches well over 9 GB/s, which surpasses the bandwidth limits of the networking infrastructure and therefore would have been impossible to reach if data was not compressed. With a maximal aggregated throughput of about 2 GB/s, BZIP2 performs much better at reading data, but the results obtained are still much lower than compared to LZO.

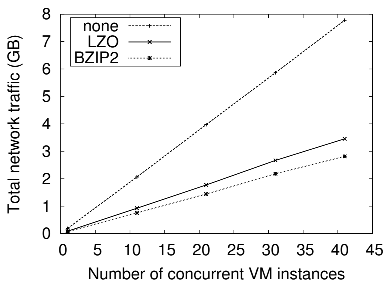
4.4 Concurrent accesses to virtual machine images

Finally, we perform a series of experiments that evaluate the benefits of using compression for virtual machine image storage. In this context, the clients are not the user application that runs inside the virtual machines, but rather the hypervisors that execute the virtual machines and need to access the underlying virtual machine images stored in the cloud.

We assume the following typical scenario: the user has customized a virtual machine image and has uploaded it on the cloud, with the purpose of using it as a template for deploying a large number of virtual machines simultaneously. Once



(a) Average time to boot a virtual machine instance when increasing the number of concurrent instances



(b) Total network traffic generated by the boot process for all instances

Fig. 5. Performance results when concurrently deploying a large number of virtual machines from the same virtual machine image that is stored in compressed fashion using our approach.

the user has asked the cloud middleware to perform this multi-deployment, each hypervisor instantiates its corresponding virtual machine, which in turn boots the guest operating system and runs the application.

In order to implement this scenario in our experiments, we use 45 nodes of the Lille cluster and deploy BlobSeer on it in the following fashion: 40 data providers, 3 metadata providers, one version manager and one provider manager. Each process is deployed on a dedicated node.

Next, we store a 2 GB large virtual machine image (a Debian Sid Linux distribution) in BlobSeer in three configurations: (1) no compression layer (the original BlobSeer implementation); (2) our compression layer with LZO compression, and (3) our compression layer with BZIP2 compression. In all three configurations, the image is split into 2 MB chunks.

We use the same nodes where data providers are deployed as compute nodes. Each of these nodes runs KVM 0.12.4 as the hypervisor. The experiment consists in performing a series of increasing multi-deployments from the virtual machine image that was previously stored in BlobSeer, for all three configurations. We perform two types of measurements: the average time taken by each instance to fully boot the guest operating system and the total amount of network traffic that was generated by the hypervisors as a result of reading the parts of the virtual machine image that were accessed during the boot process.

The obtained results are shown in Figure 5. The average time to boot an instance is depicted in Figure 5(b). As can be observed, in all three configurations the average time to boot an instance increases slightly as more instances are booted concurrently. This effect is due to the fact that more concurrent read accesses to the same data put more pressure on BlobSeer, which lowers throughput slightly. Nevertheless, the curves are almost constant and demonstrate the high scalability of our approach, in all three configurations. The fast decompression time of LZO combined with the lower amounts of data transfers give a constant boost of performance to our approach of more than 20% over the original BlobSeer implementation. On the other hand, due to higher decompression times, BZIP2 performs 30% worse than the original implementation.

Figure 5(b) illustrates the total network traffic incurred in all three configurations. As expected, the growth is linear and is directly proportional to the amount of data that was read by the hypervisors from the virtual machine image. When the image is stored in uncompressed fashion, the total network traffic is close to 8 GB for 40 instances. Using LZO compression lowers the total network traffic by more than 50%, at well below 4 GB. Further reductions, reaching more than 60%, are observed using BZIP2 compression. In this case the total network traffic is about 3 GB, bringing the highest reduction in bandwidth cost of all three configurations.

5 Related work

Data compression is highly popular in widely used data-intensive application frameworks such as Hadoop [8]. In this context, compression is not managed transparently at the level of the storage layer (Hadoop Distributed File System [9]), but rather explicitly at the application level. Besides introducing complexity related to seeking in compressed streams, this approach is also not aware of the I/O performed by the storage layer in the background, which limits the choice of optimizations that would otherwise be possible, if the schedule of the I/O operations was known.

Adaptive compression techniques that apply data compression transparently have been proposed in the literature before.

In [11], an algorithm for transferring large datasets in wide area networks is proposed, that automatically adapts the compression effort to currently available network and processor resources in order to improve communication speed. A similar goal is targeted by ACE [13] (Adaptive Compression Environment), which automatically applies on-the-fly compression at the network stack directly to improve network transfer performance. Other work such as [6] applies on-the-fly compression at higher level, targeting an improve in response time of web-services by compressing the exchanged XML messages. Although these approaches conserve network bandwidth and improve transfer speed under the right circumstances, the focus is end-to-end transfers, rather than total aggregated throughput. Moreover, compression is applied in-transit only, meaning data is not stored remotely in a compressed fashion and therefore requests for the same data generate new compression-decompression cycles over and over again.

Methods to improve the middleware-based exchange of information in interactive or collaborative distributed applications have been proposed in [29]. The proposal combines methods that continuously monitor current network and processor resources and assess compression effectiveness, deciding on the most suitable compression technique. While this approach works well in heterogeneous environments with different link speeds and CPU processing power, in clouds resources are rather uniform and typically feature high-speed links, which shifts the focus towards quickly deciding if to apply compression at all, and, when it is the case, applying fast compression techniques.

Several existing proposals define custom virtual machine image file formats, such as QCOW2 [4] and MIF [27]. These formats are able to hold the image in compressed fashion. However, unlike our approach, compression and decompression is not transparent and must be handled at the level of the hypervisor directly. While this has the advantage of enabling the hypervisor to optimize the data layout in order to achieve better compression rates, our approach has an important benefit in that it is non-intrusive: it handles compression independently of the hypervisor. This greatly improves the portability of the images, compensating for the lack of image format standardization.

6 Conclusions

As cloud computing gains in popularity and data volumes grow continuously to huge sizes, an important challenge of data storage on the cloud is the *conservation of storage space and bandwidth*, as both resources are expensive at large scale and can incur high costs for the end user.

This paper evaluates the benefits of applying *transparent* compression for data storage services running on the cloud, with the purpose of reducing costs associated to storage space and bandwidth, but without sacrificing data access performance for doing so. Unlike work proposed so far that focuses on end-to-end data transfer optimizations, we target to achieve a *high total aggregated throughput*, which is a more relevant metric in the context of clouds.

Our approach integrates with the storage service and adapts to heterogeneous data dynamically, by *sampling small portions of data on-the fly* in order to avoid compression when it is not beneficial. We *overlap compression and decompression with I/O*, by splitting the data into chunks and taking advantage of multi-core architectures, therefore minimizing the impact of compression on total throughput. Finally, we enable *configurable compression algorithm selection*, which enables the user to fine-tune the trade-off between computation time costs and storage and bandwidth costs.

We show a negligible impact on aggregated throughput when using our approach for incompressible data thanks to negligible sampling overhead and a high aggregated throughput both for reading and writing compressible data that brings massive storage space and bandwidth saves ranging between 40% and 60%. Our approach works well both for storing application data, as well as for storing virtual machine images. Using our approach with fast compression and decompression algorithms, such as LZO, higher application data access throughputs and faster virtual machine multi-deployments can be achieved under concurrency, all with the added benefits of lower storage space and bandwidth utilization.

Thanks to our encouraging results, we plan to explore in future work more adaptability approaches that are suitable in the context of data-intensive applications and virtual machine image storage. In particular, so far we used fixed chunk sizes and compression algorithms. An interesting future direction would be to dynamically select the chunk size and the compression algorithm for each chunk individually such as to reduce storage space and bandwidth consumption even further.

Acknowledgments

The experiments presented in this paper were carried out using the Grid'5000/ ALADDIN-G5K 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/ for details).

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, 50–58 (April 2010)
2. Buyya, R., Yeo, C.S., Venugopal, S., Broberg, J., Brandic, I.: Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility. Future Gener. Comput. Syst. 25(6), 599–616 (2009)
3. Dean, J., Ghemawat, S.: MapReduce: simplified data processing on large clusters. Communications of the ACM 51(1), 107–113 (2008)
4. Gagné, M.: Cooking with Linux: still searching for the ultimate linux distro? Linux J. 2007(161), 9 (2007)
5. Gantz, J.F., Chute, C., Manfrediz, A., Minton, S., Reinsel, D., Schlichting, W., Toncheva, A.: The diverse and exploding digital universe: An updated forecast of worldwide information growth through 2011. IDC (2007)
6. Ghandeharizadeh, S., Papadopoulos, C., Pol, P., Zhou, R.: Nam: a network adaptable middleware to enhance response time of web services. In: MASCOTS '03: Proceedings of the 11th IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer Telecommunications Systems. pp. 136 – 145 (12-15 2003)
7. Ghemawat, S., Gobioff, H., Leung, S.T.: The Google file system. SIGOPS - Operating Systems Review 37(5), 29–43 (2003)
8. The Apache Hadoop Project. http://www.hadoop.org
9. HDFS. The Hadoop Distributed File System. http://hadoop.apache.org/common/docs/r0.20.1/hdfs_design.html
10. Isard, M., Budiu, M., Yu, Y., Birrell, A., Fetterly, D.: Dryad: distributed data-parallel programs from sequential building blocks. SIGOPS Oper. Syst. Rev. 41(3), 59–72 (2007)
11. Jeannot, E., Knutsson, B., Björkman, M.: Adaptive online data compression. In: HPDC '02: Proceedings of the 11th IEEE International Symposium on High Performance Distributed Computing. p. 379. IEEE Computer Society, Washington, DC, USA (2002)

12. Jégou, Y., Lantéri, S., Leduc, J., Noredine, M., Mornet, G., Namyst, R., Primet, P., Quetier, B., Richard, O., Talbi, E.G., Iréa, T.: Grid'5000: a large scale and highly reconfigurable experimental grid testbed. International Journal of High Performance Computing Applications 20(4), 481–494 (November 2006)
13. Krintz, C., Sucu, S.: Adaptive on-the-fly compression. IEEE Trans. Parallel Distrib. Syst. 17(1), 15–24 (2006)
14. Marshall, P., Keahey, K., Freeman, T.: Elastic site: Using clouds to elastically extend site resources. In: CCGRID '10: Proceedings of the 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing. pp. 43–52. CCGRID '10, IEEE Computer Society, Washington, DC, USA (2010)
15. Montes, J., Nicolae, B., Antoniu, G., Sánchez, A., Pérez, M.: Using Global Behavior Modeling to Improve QoS in Cloud Data Storage Services. In: CloudCom '10: Proceedings of the 2nd IEEE International Conference on Cloud Computing Technology and Science. pp. 304–311. Indianapolis, USA (2010)
16. Moreno-Vozmediano, R., Montero, R.S., Llorente, I.M.: Elastic management of cluster-based services in the cloud. In: ACDC '09: Proceedings of the 1st Workshop on Automated control for datacenters and clouds. pp. 19–24. ACM, New York, NY, USA (2009)
17. Nicolae, B.: BlobSeer: Towards Efficient Data Storage Management for Large-Scale, Distributed Systems. Ph.D. thesis, University of Rennes 1 (November 2010)
18. Nicolae, B.: High throughput data-compression for cloud storage. In: Globe '10: Proceedings of the 3rd International Conference on Data Management in Grid and P2P Systems. pp. 1–12. Bilbao, Spain (2010)
19. Nicolae, B., Antoniu, G., Bougé, L.: Enabling high data throughput in desktop grids through decentralized data and metadata management: The blobseer approach. In: Euro-Par '09: Proceedings of the 15th International Euro-Par Conference on Parallel Processing. pp. 404–416. Delft, The Netherlands (2009)
20. 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, 169–184 (February 2011)
21. Nicolae, B., Bresnahan, J., Keahey, K., Antoniu, G.: Going Back and Forth: Efficient Multi-Deployment and Multi-Snapshotting on Clouds. In: HPDC '11: Proceedings of the 20th International ACM Symposium on High-Performance Parallel and Distributed Computing. San José, CA, USA (2011), to appear.
22. Nicolae, B., Moise, D., Antoniu, G., Bougé, L., Dorier, M.: Blobseer: Bringing high throughput under heavy concurrency to hadoop map/reduce applications. In: IPDPS '10: Proc. 24th IEEE International Parallel and Distributed Processing Symposium. pp. 1–12. Atlanta, USA (2010)
23. Nurmi, D., Wolski, R., Grzegorczyk, C., Obertelli, G., Soman, S., Youseff, L., Zagorodnov, D.: The Eucalyptus open-source cloud-computing system. In: CCGRID '09: Proceedings of the 9th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing. pp. 124–131. IEEE Computer Society, Los Alamitos, CA, USA (2009)
24. Patterson, D.A.: Technical perspective: the data center is the computer. Commun. ACM 51(1), 105–105 (2008)
25. Pavlo, A., Paulson, E., Rasin, A., Abadi, D.J., DeWitt, D.J., Madden, S., Stonebraker, M.: A comparison of approaches to large-scale data analysis. In: SIGMOD '09: Proceedings of the 35th SIGMOD international conference on Management of data. pp. 165–178. ACM, New York, NY, USA (2009)

26. Raghuveer, A., Jindal, M., Mokbel, M.F., Debnath, B., Du, D.: Towards efficient search on unstructured data: an intelligent-storage approach. In: CIKM '07: Proceedings of the 16th ACM Conference on information and knowledge management. pp. 951–954. ACM, New York, NY, USA (2007)
27. Reimer, D., Thomas, A., Ammons, G., Mummert, T., Alpern, B., Bala, V.: Opening black boxes: using semantic information to combat virtual machine image sprawl. In: VEE '08: Proceedings of the 4th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments. pp. 111–120. ACM, New York, NY, USA (2008)
28. Vaquero, L.M., Rodero-Merino, L., Caceres, J., Lindner, M.: A break in the clouds: towards a cloud definition. SIGCOMM Comput. Commun. Rev. 39(1), 50–55 (2009)
29. Wiseman, Y., Schwan, K., Widener, P.: Efficient end to end data exchange using configurable compression. SIGOPS Oper. Syst. Rev. 39(3), 4–23 (2005)
30. Ziv, J., Lempel, A.: A universal algorithm for sequential data compression. IEEE Transactions on Information Theory 23, 337–343 (1977)
31. Amazon Elastic Compute Cloud (EC2). http://aws.amazon.com/ec2/
32. Amazon Elastic Map Reduce. http://aws.amazon.com/elasticmapreduce/
33. BZIP2. http://bzip.org
34. Lempel-Ziv-Oberhumer. http://www.oberhumer.com/opensource/lzo
35. The Nimbus project. http://www.nimbusproject.org