Using Global Behavior Modeling to Improve QoS in Cloud Data Storage Services

Jesús Montes
Universidad Politécnica de Madrid
Madrid, Spain
jmontes@cesvima.upm.es

Bogdan Nicolae
University of Rennes 1
IRISA
Rennes, France
bogdan.nicolae@irisa.fr

Gabriel Antoniu
INRIA Rennes
Bretagne Atlantique
Rennes, France
gabriel.antoniu@inria.fr

Alberto Sánchez
Universidad Rey Juan Carlos
Madrid, Spain
alberto.sanchez@urjc.es

María S. Pérez
Universidad Politécnica de Madrid
Madrid, Spain
mperez@fi.upm.es

***Abstract*—The cloud computing model aims to make large-scale data-intensive computing affordable even for users with limited financial resources, that cannot invest into expensive infrastructures necesssary to run them. In this context, MapReduce is emerging as a highly scalable programming paradigm that enables high-throughput data-intensive processing as a cloud service. Its performance is highly dependent on the underlying storage service, responsible to efficiently support massively parallel data accesses by guaranteeing a high throughput under heavy access concurrency. In this context, quality of service plays a crucial role: the storage service needs to sustain a stable throughput for each individual accesss, in addition to achieving a high aggregated throughput under concurrency. In this paper we propose a technique to address this problem using component monitoring, application-side feedback and behavior pattern analysis to automatically infer useful knowledge about the causes of poor quality of service and provide an easy way to reasonin about potential improvements. We apply our proposal to BlobSeer, a representative data storage service specifically designed to achieve high aggregated throughputs and show through extensive experimentation substantial improvements in the stability of individual data read accesses under MapReduce workloads.**

I. Introduction

The emerging cloud computing model [1], [**?**] is gaining serious interest from both industry and academia 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 what is needed to solve their problem and nothing more.

In this context, applications that perform complex data processing (data mining, online transaction record management, simulations, etc.), initially reserved to governments and large corporations that could afford the underlying infrastructures to run them, now become accessible to a large public. In order to achieve scalable data processing performance, several paradigms have been proposed, such as MapReduce [2] and Dryad [3]. MapReduce quickly gained popularity and was hailed as a revolutionary new platform for large-scale, massively parallel data processing [4], attracting attention in the role as a service to be offered on the clouds [5].

Since MapReduce applications are mostly data-intensive, a critical component in this context is the underlying storage service, which needs to deliver a *high aggregated throughput* even under a heavy data access concurrency generated when the service runs for a very large number of clients. On clouds however, this alone is not sufficient, as multiple users share the same infrastructure and need *quality-of-service* guarantees. The data storage service must not only be able to sustain a high aggregated throughput under heavy access concurrency, but also guarantee a *stable throughput for each individual data access*.

In this paper we focus on improving storage services running on clouds to deliver a stable throughput for indivdual data transfers while still achieving a high aggregated throughput. A stable throughput guarantee is however an inherently difficult task, because a large number of factors is involved.

First, MapReduce frameworks run on infrastructures comprising thousands of commodity hardware components. In this context, failures are rather the norm than the exception. Since faults cause drops in performance, *fault tolerance* becomes a critical aspect of throughput stability. Second, complex data access patterns are generated that are likely to combine periods of intensive I/O activity with periods of relative less intensive I/O activity throughout runtime. This has a negative impact on throughput stability, thus *adaptation to access pattern* is a crucial issue as well.

The sheer complexity of both the state of the hardware components and the data access pattern makes reasoning about fault tolerance and adaptation to access pattern difficult, since it is not feasible to find non-trivial dependencies manually. Therefore, it is important to automate the process of identifying and characterizing the circumstances that bring the storage service in a state where there are significant fluctuations in the sustained throughput and to take appropriate action to stabilize the system. This paper proposes an innovative approach to this problem. We summarize our contributions as follows:

We define a general methodology based on *component monitoring*, *application-side feedback* and *behavior pattern analysis* in order to discover and characterize the situations that lead to fluctuations of individual data access throughput. Our approach implicitly adapts itself to the hardware configuration of the infrastructure, failure rate and data access pattern of the application. We then apply our proposal to improve

BlobSeer [6], [7], a data management service specifically designed to address the needs of data-intensive applications. BlobSeer is now subject to integration efforts with state-of-the-art cloud toolkits such as Nimbus [8]. Finaly, we perform extensive experimentations in hard conditions: highly-concurrent data access patterns, for long periods of service uptime, while supporting failures of the physical storage components. We validate our approach by demonstrating substantial improvements in individual throughput stability.

II. Proposal

A. Overview

We propose a general approach to automate the process of identifying and characterizing the events that cause significant fluctuations in the sustained throughput of individual I/O transfers. This enables the storage service to take appropriate action in order to stabilize the system. We rely on three key principles:

In-depth monitoring: The storage service is usually deployed on large-scale infrastructures comprising thousands of machines, each of which is prone to failures. Such components are characterized by a large number of parameters, whose values need to be measured in order to describe the state of the system at a specific moment in time. Since data-intensive applications generate complex access-pattern scenarios, it is not feasible to predetermine what parameters are important and should be monitored, as this limits the potential to identify non-obvious bottlenecks. Therefore, it is important to collect as much information as possible about each of the components of the storage service.

Application-side feedback: While extensive monitoring of the system helps to accurately define its state at a specific moment in time, it is not enough to accurately identify a situation where the throughput of individual I/O transfers fluctuates, because the perceived quality of service from the application point of view remains unknown. For example, in the context of MapReduce applications, monitoring the network traffic is not enough to infer the throughput achieved for each task, because the division of each job into tasks remains unknown to the storage service. Therefore, it is crucial to gather dynamically feedback from the upper layers that rely on the storage service in order to decide when the system performs satisfactory and when it does not.

Behavior pattern analysis: Extensive monitoring information gives a complete view of the behavior of the storage service in time. Using the feedback from the upper layers, it is also possible to determine when it performed satisfactory and when it performed poorly. However, the complexity of the behavior makes direct reasoning about the causes of potential bottlenecks not feasible. Intuitively, explaining the behavior is much easier if a set of behavior patterns can be identified, which can be classified either as satisfactory or not. This is so because once a classification is made, taking action to stabilize the system basically means to predict a poor behavior and enforce a policy to avoid it. The challenge however is to describe the behavior patterns in such a way that they provide meaningful insight with respect to throughput stability, thus making healing mechanisms easy to implement.

Starting from these principles, we propose a methodology to stabilize the throughput of the storage service as a series of four steps:

1) Monitor the storage service: A wide range of parameters that describe the state of each of the components of the storage service is periodically collected during a long period of service up-time. This is necessary in order to reliably approximate the data access pattern generated by the data-intensive application. An important aspect in this context is *fault detection*, as information on when and how long individual faults last is crucial in the identification of behavior patterns.

2) Identify behavior patterns: Starting from the monitored parameters, the behavior of the storage service as a whole is classified into behavior patterns. The classification must be performed in such a way that, given the behavior at any moment in time, it unambiguously matches one of the identified patterns. The critical part of this step is to extract meaningful information with respect to throughput stability that describes the behavior pattern. Considering the vast amount of monitoring information, it is not feasible to perform this step manually. Therefore, an automated knowledge discovery method should be applied. For this purpose, we adapted a global behavior modeling technique [9], [10] to identify and describe the behavior patterns in our context. Specific details are provided in Section II-B.

3) Classify behavior patterns according to feedback: In order to identify a relationship between behavior patterns and stability of throughput, the application-side feedback with respect to the perceived quality of service is analyzed. More specifically, for each data transfer a series of performance metrics as observed by the upper layers is gathered. These performance metrics are then aggregated for all data transfers that occurred during the period in which the behavior pattern was exhibited. The result of the aggregation is a score that is associated to the behavior pattern and indicates how desirable the behavior pattern is.

Once the classification has been performed, transitions from desirable states to undesirable states are analyzed in order to find the reason why the storage service does not offer a stable throughput any longer. This step involves user-level interpretation and depends on the quality of the generated behavior model.

4) Predict and prevent undesired behavior patterns: Finally, understanding the reasons for each undesired behavior enables the implementation of prediction and prevention mechanisms accordingly. More specifically, for each undesirable state, the preconditions that trigger a transition to it are determined. Using this information, a corresponding policy is enabled, capable of executing a special set of rules while these preconditions are satisfied. These rules are designed to prevent this transition to occur or, if this is not possible, to mitigate the effects of the undesired state. It is assumed that the application is monitored throughout its execution and the storage service has accessed to the monitoring information.

This makes the service able to predict when an undesirable behavior is about to happen and to activate the corresponding policy.

To illustrate our approach, we have chosen *BlobSeer* [6] as a representative storage service for MapReduce data-intensive applications. Our choice is mainly motivated by the ability of BlobSeer to sustain a *high throughput under heavy access concurrency* significantly better [7] than HDFS [11], the default storage backend for the Hadoop [12] MapReduce framework, largely used on today's cloud platforms. While BlobSeer leverages the underlying networking infrastructure better, this also increases the likelihood of components being stretched to their limits and consequently the likelihood of variations of individual data-access throughput being noticed. BlobSeer is now subjects to integration efforts with the Nimbus [8] cloud toolkit, in cooperation with the Nimbus team.

B. GloBeM: Global Behavior Modeling

In order to identify and describe the behavior patterns of the storage service approach to model the global behavior of large-scale distributed systems [9], [10] (from now on it will be named *GloBeM*). Its main objective is to build an abstract, descriptive model of the global system state. This enables the model to implicitly describe the interactions between entities, which has the potential to unveil non-trivial dependencies significant for the description of the behavior, which otherwise would have gone unnoticed.

GloBeM follows a set of procedures in order to build such a model, starting from monitoring information that corresponds to the observed behavior. These basic monitoring data are then aggregated into *global monitoring parameters*, representative of the global system behavior instead of each single resource separately. This aggregation can be performed in different ways, but it normally consists in calculating global statistic descriptors (mean, standard deviation, skewness, kurtosis, etc.) values of each basic monitoring parameter for all resources present. This ensures that global monitoring metrics are still understandable from a human perspective. This global information undergoes a complex analysis process in order to produce a global behavior representation. This process is strongly based on machine learning and other knowledge discovery techniques, such as virtual representation of information systems [13], [14]. A behavior model presents the following characteristics:

Finite state machine: The model can be expressed as a finite state machine, with specific states and transitions. The number of states is generally small (between 3 and 8).

State characterization based on monitoring parameters: The different system states are expressed in terms of the original monitoring parameters. This ensures that its characteristics can be understood and directly used for management purposes.

Extended statistical information: The model is completed with additional statistic metrics, further expanding the state characterization.

C. BlobSeer

BlobSeer is an efficient distributed data management service specifically designed to deliver a high throughput under heavy access concurrency. Data is abstracted in BlobSeer as huge sequences of bytes called *BLOBs* (*Binary Large OBjects*). Each BLOB is manipulated through a simple versioning access interface that enables fine-grained reads, writes and appends of subsequences of bytes from/to the BLOB.

Three key design factors enable BlobSeer to achieve a high throughput under heavy access concurrency: *data striping*, *distributed metadata management* and *versioning-based concurrency control*.

Data striping: Each BLOB is split into *chunks* that are distributed among *data providers*, which are responsible to store the chunks. To maintain data availability in spite of failures, each chunk is replicated on multiple distinct providers. A configurable chunk distribution strategy is employed when writes and appends are issued in order to optimize chunk placement in such way that accesses to different chunks are as much as possible handled by different machines, effectively distributing the I/O workload.

Metadata decentralization: BlobSeer uses a distributed metadata management scheme to avoid the bottleneck of accessing the same centralized node and prevent a single point of failure.

Versioning-based concurrency control: Writes and appends to a BLOB never modify its contents, but rather generate a new snapshot of it that looks and acts like the original BLOB save for the applied update. Only the difference is physically stored, with unmodified parts shared. This approach enables the concurrency control to isolate updates in their own snapshot, thus avoiding the need for synchronization significantly better than lock-based approaches, which greatly improves achieved throughput.

D. Applying our approach to BlobSeer

We illustrate our approach by implementing the proposed methodology for BlobSeer, as presented in Figure 1. For simplification purposes, we assume the only components of BlobSeer that can cause bottlenecks are the data providers. This is a reasonable assumption, as most of the I/O load falls on the data providers, while metadata is managed in a distributed fashion and incurs a minimal overhead [6]. For this reason, monitoring is performed for the data providers only.

1) Monitor the data providers: We periodically collect a wide range of parameters that describe the state of each data provider of the BlobSeer instance. For this task we use GMonE [15], a monitoring framework for large-scale distributed systems based on the *publish-subscribe* paradigm.

GMonE runs a process called *resource monitor* on every node to be monitored. Each such node publishes monitoring information to one or more *monitoring archives* at regular time intervals. These monitoring archives act as the subscribers and gather the monitoring information in a database, constructing a historical record of the system's evolution.

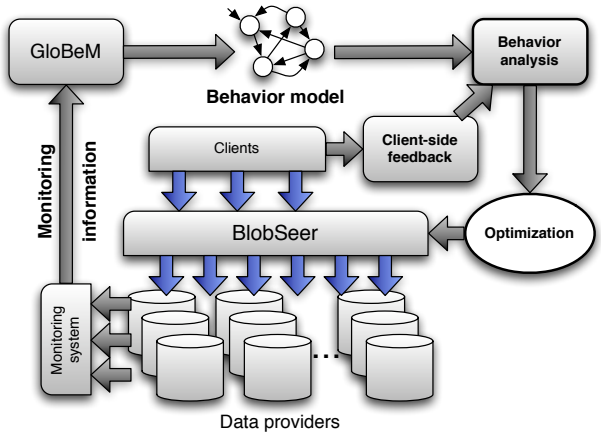


Fig. 1. Our approach applied: Stabilizing throughput in BlobSeer

The *resource monitors* can be customized with *monitoring plugins*, which can be used to adapt the monitoring process to a specific scenario by selecting relevant monitoring information. We developed a plug-in for BlobSeer that is responsible for monitoring each provider and pushing the following parameters into GMonE: number of read operations, number of write operations, free space available, CPU load and memory usage. These parameters represent the state of the provider at any specific moment in time. Every node running a data provider publishes this information each 45 seconds to a single central *monitoring archive* that stores the monitoring information for the whole experiment.

Once the monitoring information is gathered, an aggregation process is undertaken: mean and standard deviation values are calculated for each of the five previous metrics. Additionally, unavailable data providers (due to a failure) are also included as an extra globally monitored parameter. We call the result of the gathering and aggregation the global history record of the behavior of BlobSeer.

2) Identify behavior patterns: The historical data mentioned above is then fed into GloBeM in order to classify the behavior of BlobSeer as a whole into a set of states, each corresponding to a behavior pattern. Thanks to GloBeM, this process is fully automated and we obtain a comprehensive characterization of the states in terms of the most important parameters that contribute to it.

3) Classify behavior patterns according to feedback: For each data transfer, we consider as relevant client-side quality of service indicators (i) the effective observed throughput and (ii) the number of times the operation was interrupted by a fault and had to be restarted. This information is logged for each data transfer, averaged for each state of the behavior model and then used to classify the states into desirable states that offer good performance to the clients and undesirable states that offer poor performance to the clients.

4) Predict and prevent undesired behavior patterns: Finally, the challenge is to improve the behavior of BlobSeer in such way as to avoid undesirable states. This step is completely dependent on the behavioral analysis of the model generated by GloBeM. Since the model is tightly coupled with the application data-access pattern, the infrastructure used to deploy BlobSeer and the corresponding failure model, we detail this step for each of our experiments separately in Section IV.

III. Experimental setup

A. Application scenario: MapReduce data gathering and analysis

We evaluate the effectiveness of our approach for simulated MapReduce workloads that correspond to a typical data-intensive computing scenario on clouds: continuously acquiring (and possibly updating) very large datasets of unstructured data while performing large-scale computations over the data.

For example, a startup might want to invest money into a cloud application that crawls the web in search for new text content such as web pages in order to build aggregated statistics and infer new knowledge about a topic of interest.

In this context, simulating the corresponding MapReduce workloads involves two aspects: i) a write access pattern that corresponds to constant data gathering and maintenance of data in the system and ii) a read access pattern that corresponds to the data processing (in most cases the final result of the data processing are small aggregated values that generate negligible writes; moreover, intermediate data is not stored persistently by MapReduce).

Write access pattern: As explained in [16], managing a very large set of small files is not feasible. Therefore, data is typically gathered in a few files of great size. Moreover, experience with data-intensive applications has shown that these very large files are generated mostly by appending records concurrently and seldom overwriting any record. To reproduce this behavior in BlobSeer, we create a small number of BLOBs and have a set of clients (corresponding to "map" tasks) generate and write random data concurrently to the BLOBs. Each client predominantly appends and occasionally overwrites chunks of 64 MB to a randomly selected BLOB at random time intervals, sleeping meanwhile. The frequency of writes corresponds to an overall constant write pressure of 1MB/s on each of the data provides of BlobSeer throughout the duration of the experiment and corresponds to the maximal rate that a single web crawler process can achieve under normal circumstances.

Read access pattern: In order to model the data processing aspect, we simulate MapReduce applications that scan the whole dataset in parallel and compute some aggregated statistics about it. This translates into a highly concurrent read access pattern to the same BLOB. We implemented clients that perform parallel reads of chunks of 64MB (which is the default chunk size used in Hadoop MapReduce) from the same BLOB version and then simulate a "map phase" on this data by keeping the CPU busy. Globally, we strive to achieve an average I/O time to computation time ratio of 1:7, which is intended to account for the CPU time of both the "map phase" and the "reduce phase".

We execute both the data gathering and data processing concurrently in order to simulate a realistic setting where data is constantly analyzed while updates are processed in the background. We implemented the clients in such a way as to target an overall write to read ratio of 1:10. This comes from the fact that in practice multiple MapReduce passes over the same data are necessary to achieve the final result.

B. Platform description

We performed our experiments on Grid'5000 [17], a highly configurable and controllable experimental platform for grid and cloud research. We used 130 of the nodes of Lille cluster and 275 of the nodes of Orsay cluster. The nodes are outfitted with x86_64 CPUs, and 2 GB of RAM. We measured raw buffered reads from the hard drives at 61.8MB/s on Lille and 53.2 MB/s on Orsay, using the *hdparm* utility. Inter-node 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.

MapReduce-style computing systems are traditionally running on commodity hardware, collocating computation and storage on the same physical box. However, recent proposals advocate the use of converged networks to decouple the computation from storage in order to enable a more flexible and efficient datacenter design [18]. Since both approaches are used by cloud providers, we evaluate the benefits of applying global behavior modeling to BlobSeer in both scenarios. For this purpose, we use the Lille cluster to model collocation of computation and storage node by codeploying a client process with a data provider process on the same node, and the Orsay cluster to model decoupled computation and storage by running the client and the data provider on different nodes. In both scenarios we deploy on each node a GMonE resource monitor that is responsible to collect the monitoring data throughout the experimentation. Further, in each of the clusters we reserve a special node to act as the GMonE monitoring archive that collects the monitoring information from all resource monitors. We will refer from now on to the scenario that models collocation of computation and storage on the Lille cluster simply as *setting A* and to the scenario that models decoupled computation and storage on the Orsay cluster as *setting B*.

C. Simulating node failures

Since real large-scale distributed environments are subject to failures, we implemented a data provider failure-injection framework that models failure patterns observed in real large-scale systems build from commodity hardware that run for long periods of time. We use the multi-state resource availability characterization study described in [19] in order to generate random failure scenarios for our experiments.

IV. Results

We perform a multi-stage experimentation that involves: (i) running an original BlobSeer instance under the data-intensive access pattern and failure scenario described, (ii) applying our approach to analyze the behavior of BlobSeer and identify potential improvements and finally (iii) running an improved BlobSeer instance in the same conditions as the original instance, comparing the results and proving that the improvement hinted by the proposed methodology was indeed successful in raising the performance of BlobSeer. This multi-stage experimentation is performed for both settings A and B described in Section III.

TABLE I
GLOBAL STATES - SETTING A

parameter	State 1	State 2	State 3	State 4
Avg. read ops.	68.9	121.2	60.0	98.7
Read ops stdev.	10.5	15.8	9.9	16.7
Avg. write ops.	43.2	38.4	45.3	38.5
Write ops stdev.	4.9	4.7	5.2	7.4
Free space stdev.	$3.1e7$	$82.1e7$	$84.6e7$	$89.4e7$
Nr. of providers	107.0	102.7	96.4	97.2

TABLE II
GLOBAL STATES - SETTING B

parameter	State 1	State 2	State 3
Avg. read ops.	98.6	202.3	125.5
Read ops stdev.	17.7	27.6	21.9
Avg. write ops.	35.2	27.5	33.1
Write ops stdev.	4.5	3.9	4.5
Free space stdev.	$17.2e6$	$13.0e6$	$15.5e6$
Nr. of providers	129.2	126.2	122.0

1) Running the original BlobSeer instance: We deploy a BlobSeer instance in both settings A and B and monitor it using GMonE. Both experimental settings have a fixed duration of 10 hours. During the experiments, the data-intensive workload accessed a total of $\simeq 11TB$ of data on setting A, out of which $\simeq 1.3TB$ were written and the rest read. Similarly, a total of $\simeq 17TB$ of data was generated on setting B, out of which $\simeq 1.5TB$ were written and the rest read.

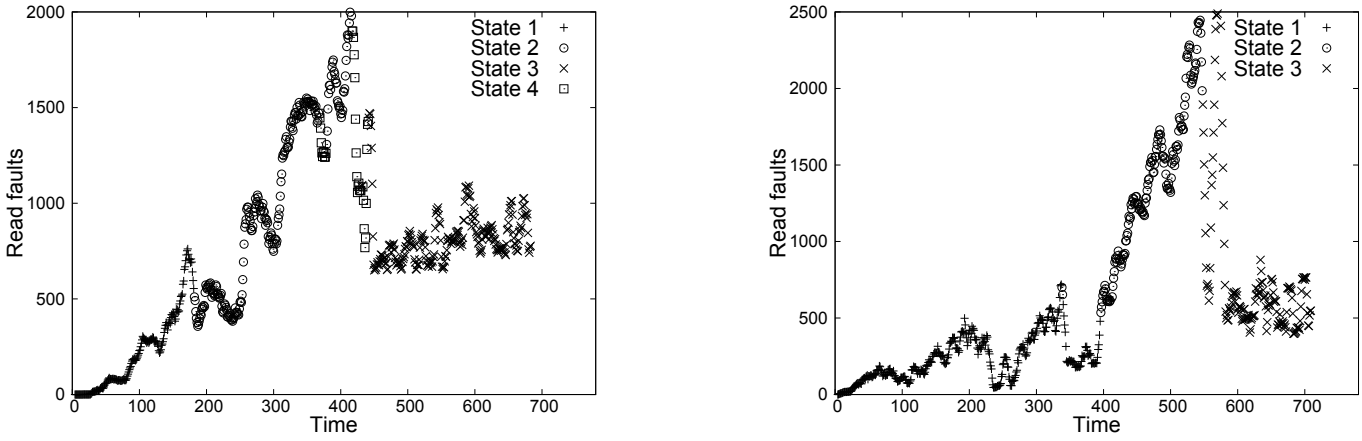
2) Performing the global behavior modeling: We apply GloBeM both for setting A and setting B in order to generate the corresponding global behavior model. Each of the identified states of the model corresponds to a specific behavior pattern and contains the most significant parameters that characterize the state. Tables I and II show the average values for the most representative parameters of each state, both for setting A and setting B respectively. It is important to remember that these are not all the parameters that were monitored, but only the ones selected by GloBeM as *the most representative*. As can be seen, GloBeM identified four possible states in the case of setting A and three in the case of setting B.

The client-side feedback is gathered from the client logs as explained in Section II-D. Average read bandwidths for each of the states are represented in Table III for both settings A and B.

TABLE III
AVERAGE READ BANDWIDTH

Scenario	State 1	State 2	State 3	State 4
Setting A	24.2	20.1	31.5	23.9
Setting B	50.7	35.0	47.0	

units are MB/s



(a) Setting A

(b) Setting B

Fig. 2. Read faults: states are represented with different point styles

Figures 2(a) and 2(b) depict evolution in time of the total number of read faults as observed by the clients for both scenarios. At this point it is important to remember that these are client related data and, therefore, neither read bandwidth nor failure information was available to GloBeM when identifying the states. Nevertheless, the different global patterns identified correspond to clearly different behavior in terms of client metrics, as Table III and Figures 2(a) and 2(b) show.

As previously described, the GloBeM analysis generated two global behavior models, each one corresponding to the behavior of BlobSeer in settings A and B. We performed further analysis using the effective read bandwidth and number of read faults as observed from the client point of view in order to classify the states of the behavior models into *desired* states (where the performance metrics are satisfactory) and undesired states (where the performance metrics can be improved).

In the case of setting A, *State 2* presents the lowest average read bandwidth ($\simeq 20MB/s$). It is also the state where most read faults occur, and where the failure pattern is more erratic. A similar situation occurs with setting B. In this case again *State 2* is the one with the lowest average bandwidth ($\simeq 35MB/s$) and the most erratic read fault behavior. We conclude these states (*State 2* in both settings A and B) to be *undesired*, because the worst quality of service is observed from the client point of view.

Considering now the global state characterization provided by GloBeM for both scenarios (Tables I and II), a distinctive pattern can be identified for these *undesired* states: both have clearly the highest average number of read operations and, in the case of setting B specifically, a high standard deviation for the number of read operations. This indicates a state where the data providers are under heavy read load (hence the high average value) and the read operation completion times are fluctuating (hence the high standard deviation).

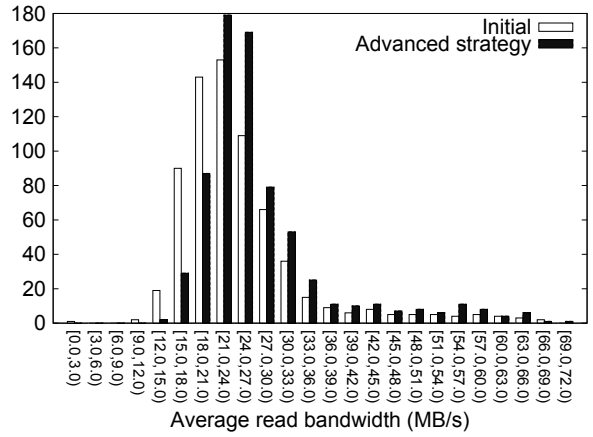
3) Improving BlobSeer: Now that the cause for fluctuations in the stability of the throughput has been identified, our objective is to improve BlobSeer's quality of service by implementing a mechanism that avoids reaching the *undesired* states described above (*State 2* in both settings). Since the system is under constant write load in all states for both settings A and B (Tables I and II) we aim at reducing the total I/O pressure on every data provider by avoiding to allocate providers under heavy read load to store new chunks generated by writers.

This in turn improves the read throughput but at the cost of a slightly less balanced chunk distribution. This eventually affects the throughput of future read operations on the newly written data. For this reason, avoiding writes on providers with heavy read loads is just an emergency measure to prevent reaching an *undesired* state. During normal functioning with non-critically high read loads, the original load-balancing strategy for writes can be used.

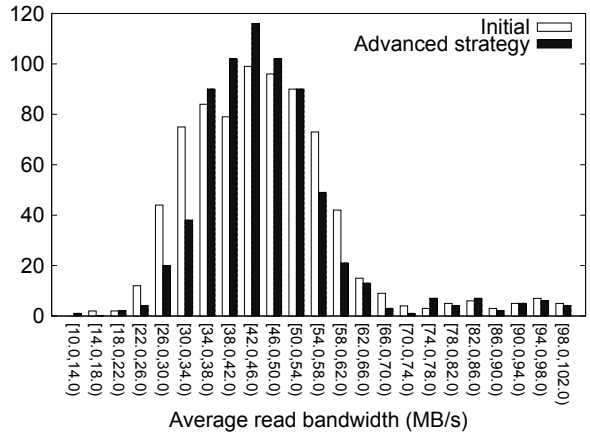
The average read operation characterization provided by GloBeM for *State 2*, which is the *undesired* state (both in settings A and B), is the key threshold to decide when a provider is considered to be under heavy read load and should not store new chunks. We implemented this policy in the chunk allocation strategy of the provider manager. Since data providers report periodically to the provider manager with statistics, we simply avoid choosing providers for which the average number of read operations goes higher than the threshold. We enable choosing those providers again when the number of read operations goes below this threshold.

4) Running the improved BlobSeer instance: The same experiments were again conducted in the exact same conditions, (for both settings A and B), using in this case the improved BlobSeer chunk allocation strategy. As explained, the purpose of this new strategy is to improve the overall quality of service by avoiding the undesirable states identified by GloBeM (*State 2* in both settings A and setting B).

As final measure of the quality of service improvement, a deeper statistical comparison of the average read bandwidth observed by the clients was done. Figures 3(a) and 3(b) show



(a) Setting A



(b) Setting B

Fig. 3. Read bandwidth stability: distribution comparison

the read bandwidth distribution for each experimental scenario. In each case, the values of the original and improved BlobSeer version are compared. Additionally, Table IV shows the average and standard deviation observed in each experimental setting.

TABLE IV
STATISTICAL DESCRIPTORS FOR READ BANDWIDTH (MB/S)

Scenario	mean (MB/s)	standard deviation
Setting A - Initial	24.9	9.6
Setting A - Advanced strategy	27.5	7.3
Setting B - Initial	44.7	10.5
Setting B - Advanced strategy	44.7	8.4

The results seem to indicate a clear improvement (especially in setting A). However, in order to eliminate the possibility of reaching this conclusion simply because of different biases in the monitoring samples, we need further statistical assessment. In order to declare the results obtained using the improved BlobSeer implementation (depicted in Figures 3(a) and 3(b) and Table IV) as statistically meaningful with respect to the original implementation, we need to ensure that the different monitoring samples are in fact obtained from different probability distributions. This would certify that the quality of service improvement observed is real, and not a matter of simple bias.

To this end, we ran the Kolmogorov-Smirnov statistical test [20], on both the initial read bandwidth results and the improved read bandwidth results. This test essentially takes two samples as input and outputs a *p-value*, which must be smaller than 0.01 in order to conclude that they originate from two different probability distributions. The obtained p-values, represented in Table V clearly shows that our results successfully passed the test.

Finally, the results show a clear quality of service improvement in both settings A and B. In setting A, the average read bandwidth shows a 10% increase and, which is more important, the standard deviation was reduced substantially. This indicates a lesser degree of dispersion in the effective read bandwidth observed, and therefore a much more stable bandwidth (for which the difference between the expected bandwidth (the mean value) and the real bandwidth as measured by the client is lower). As it has been said, these read bandwidth mean and standard deviation improvements indicate an significant increase in the overall data access quality of service.

TABLE V
KOLMOGOROV-SMIRNOV TEST RESULTS

Scenario	p-value
Setting A	$2.098e{-}14$
Setting B	0.004529

In setting B, the average read bandwidth remained stable, which is understandable given that, as explained in Section III, we are close to the maximum physical hard drive transfer rate limit of the testbed characteristics and, therefore, achieving a higher value is very difficult. Nevertheless, the read bandwidth standard deviation was again significantly reduced, resulting in a much more stable data access and, therefore, improved data access quality of service.

V. RELATED WORK

Modeling and characterizing the behavior of large scale distributed systems has been approached in several other contexts. The most basic approach is benchmarking [21], which enables manual analysis of the behavior of a system under different workloads. Other approaches describe the system formally using Colored Petri Nets (CPN) [22] or Abstract State Machines (ASM) [23] in order to reason about behavior. Rood and Lewis [24] propose a multi-state model and several analysis techniques in order to forecast the resources availability, aiming at improving scheduler efficiency. Smith et al. [25] analyze the run times of parallel applications from past executions of similar applications. Barham et al. [26] propose Magpie, a toolchain for automatically extracting a system's workload under realistic operating conditions. Finally, in the

same line Pan et al. [27] propose a tool for black-box diagnosis of MapReduce systems, aimed at discovering problems and bottlenecks.

Compared to all this related work, our approach simplifies the analysis using a generic, global system model, therefore enabling an easier further decision-making. It also emphasizes less invasive monitoring and focuses on advanced knowledge discovery techniques that can directly be applied to improve the system. To the best of our knowledge, such an approach has not been investigated so far in the area of distributed storage.

VI. Conclusions

In order to adequately support MapReduce distributed data-intensive applications on the cloud, the underlying data storage backend has to ensure besides a high aggregated throughput under heavy access concurrency also a stable throughput for each individual data transfer, which is a quality of service constraint.

However the sheer complexity of the system's behavior makes reasoning about quality of service a difficult problem, because a lot of distinct factors affect the behavior simultaneously: highly-concurrent data access patterns, long periods of service uptime, failures of physical components, the highly distributed nature of the storage service itself, etc.

This paper proposes a new approach to improve QoS in a distributed storage system, based on component monitoring, application-side feedback and global behavior modeling that can be combined to infer useful knowledge about potential bottlenecks, making reasoning about potential imporvements much easier.

We have successfully applied our approach to improve the individual throughput stability delivered by BlobSeer, a representative distributed storage service which aims at providing a high aggregated throughput under heavy access concurrency and thus is an ideal candidate as a MapReduce backend.

Evaluations on the Grid'5000 testbed revealed substantial improvement in individual read throughput stability, thereby raising the overall quality of service provided by BlobSeer, both for the case of collocated computation tasks with data providers and decoupled computation and storage.

In the near future, we plan to pursue the encouraging results found so far by adopting our approach for real MapReduce deployments (such as Hadoop [12]) which are backed up by real cloud middleware (such as Nimbus [8]). Furthermore, A stable throughput at the level of the storage service makes the cost of the I/O operations more predictable, which in turn has the potential to improve the efficiency of scheduling algorithms and reveal new options for quality of service management at the upper layers. Future work in this direction is planned as well in the long term.

References

[1] L. M. Vaquero, L. Rodero-Merino, J. Caceres, and M. Lindner, "A break in the clouds: towards a cloud definition," *SIGCOMM Comput. Commun. Rev.*, vol. 39, no. 1, pp. 50–55, 2009.

[2] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[3] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: distributed data-parallel programs from sequential building blocks," *SIGOPS Oper. Syst. Rev.*, vol. 41, no. 3, pp. 59–72, 2007.

[4] D. A. Patterson, "Technical perspective: the data center is the computer," *Commun. ACM*, vol. 51, no. 1, pp. 105–105, 2008.

[5] "Amazon Elastic Map Reduce," http://aws.amazon.com/elasticmapreduce/.

[6] B. Nicolae, G. Antoniu, L. Bougé, D. Moise, and A. Carpen-Amarie, "Blobseer: Next generation data management for large scale infrastructures," *Journal of Parallel and Distributed Computing*, 2010, to appear.

[7] B. Nicolae, D. Moise, G. Antoniu, L. Bougé, and M. Dorier, "BlobSeer: Bringing high throughput under heavy concurrency to Hadoop Map/Reduce applications," in *24th IEEE International Parallel and Distributed Processing Symposium (IPDPS '10)*, Atlanta, USA, 2010, pp. 1–11.

[8] "The nimbus project," http://www.nimbusproject.org/.

[9] J. Montes, A. Sánchez, J. J. Valdés, M. S. Pérez, and P. Herrero, "The grid as a single entity: Towards a behavior model of the whole grid," in *OTM Conferences (1)*, ser. Lecture Notes in Computer Science, R. Meersman and Z. Tari, Eds., vol. 5331. Springer, 2008, pp. 886–897.

[10] ——, "Finding order in chaos: a behavior model of the whole grid," *Concurrency and Computation: Practice and Experience*, vol. 22, 2009.

[11] K. Shvachko, H. Huang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *26th IEEE (MSST2010) Symposium on Massive Storage Systems and Technologies*, May 2010.

[12] "The Apache Hadoop Project," http://www.hadoop.org.

[13] J. J. Valdés, "Similarity-based heterogeneous neurons in the context of general observational models," *Neural Network World*, vol. 12, pp. 499–508, 2002.

[14] ——, "Virtual reality representation of information systems and decision rules," *Lecture Notes in Artificial Intelligence*, vol. 2639, pp. 615–618, 2003.

[15] A. Sánchez, "Autonomic high performance storage for grid environments based on long term prediction. Chapter 9.2," Ph.D. dissertation, Universidad Politécnica de Madrid, 2008.

[16] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The google file system," *SIGOPS - Operating Systems Review*, vol. 37, no. 5, pp. 29–43, 2003.

[17] "Aladdin-grid'5000," http://www.grid5000.fr, 2010.

[18] J. Shafer, S. Rixner, and A. L. Cox, "Datacenter storage architecture for mapreduce applications," in *ACLD: Workshop on Architectural Concerns in Large Datacenters*, 2009.

[19] B. Rood and M. J. Lewis, "Multi-state grid resource availability characterization," in *GRID '07: Proceedings of the 8th IEEE/ACM International Conference on Grid Computing*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 42–49.

[20] M. A. Stephens, "Edf statistics for goodness of fit and some comparisons," *Journal of the American Statistical Association*, vol. 69, no. 347, pp. 730–737, 1974.

[21] J. J. Dongarra and W. Gentzsch, Eds., *Computer benchmarks*. Amsterdam, The Netherlands: Elsevier Science Publishers B. V., 1993.

[22] C. Bratosin, W. M. P. van der Aalst, N. Sidorova, and N. Trcka, "A reference model for grid architectures and its analysis," in *OTM Conferences (1)*, 2008, pp. 898–913.

[23] Y. Gurevich, "Evolving Algebras: An Attempt to Discover Semantics," in *EATCS Bulletin*. European Assoc. for Theor. Computer Science, Feb. 1991, vol. 43, pp. 264–284.

[24] B. Rood and M. J. Lewis, "Resource availability prediction for improved grid scheduling," in *Proceedings of the 2008 Fourth IEEE International Conference on eScience (e-Science 2008)*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 711–718.

[25] W. Smith, I. T. Foster, and V. E. Taylor, "Predicting application run times with historical information," *J. Parallel Distrib. Comput.*, vol. 64, no. 9, pp. 1007–1016, 2004.

[26] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier, "Using magpie for request extraction and workload modelling," in *OSDI*, 2004, pp. 259–272.

[27] X. Pan, S. Kavulya, J. Tan, R. Gandhi, and P. Narasimhan, "Ganesha: Black-box diagnosis for mapreduce systems," in *Proceedings of the Second Workshop on Hot Topics in Measurement & Modeling of Computer Systems*, Seattle, WA, USA, 2009.