Co-funded by the European Commission within the Seventh Framework Programme

Project no. 318521

HARNESS

Specific Targeted Research Project
HARDWARE- AND NETWORK-ENHANCED SOFTWARE SYSTEMS FOR CLOUD COMPUTING

Characterisation Report
D5.1

Due date: 30 September 2013
Submission date: 30 October 2013

Start date of project: 1 October 2012

Document type: Deliverable
Activity: RTD
Work package: WP5

Editor: Thorsten Schuett (ZIB)

Contributing partners: ZIB

Reviewers: Gabriel Figueiredo, John McGlone

Dissemination Level

PU	Public	√
PP	Restricted to other programme participants (including the Commission Services)	
RE	Restricted to a group specified by the consortium (including the Commission Services)	
CO	Confidential, only for members of the consortium (including the Commission Services)	

Revision history:

Version	Date	Authors	Institution	Description
0.1	2013/06/24	Christoph Kleineweber	ZIB	Outline
0.2	2013/07/15	Christoph Kleineweber	ZIB	First draft
0.3	2013/08/28	Christoph Kleineweber	ZIB	Revised version
0.4	2013/09/06	Christoph Kleineweber	ZIB	Integrated reviews
1.0	2013/09/29	Alexander Wolf	IMP	Final review and edits by Coordinator

Tasks related to this deliverable:

Task No.	Task description	Partners involved°
T5.1	Define heterogeneous storage resource model	IMP, SAP, ZIB*

°This task list may not be equivalent to the list of partners contributing as authors to the deliverable
*Task leader

Executive Summary

The goal of Deliverable 5.1 is to describe how to characterise storage resources and how to optimise mappings of applications, with examples from the *Hardware- and Network-Enhanced Software Systems for Cloud Computing* (HARNESS) validation use cases.

In this report, we define *quality of service* (QoS) classes for sequential access, random access, cold storage, and best-effort storage and describe the categorisation of the validation use cases. We evaluated the behaviour of sequential and random I/O on spinning *hard disk drive*s (HDDs) and *solid-state drive*s (SSDs) in a multi-tenant scenario. The evaluation has shown that the level of concurrent I/O has a significant impact on sequential read performance using HDDs. The impact of concurrency for other workloads and other devices is less significant. Finally, we present a performance model, derived from our observations.

This deliverable will be followed by D5.2, to be submitted in M24, which will extend the work of this deliverable to cover more details on our framework for modelling the performance of different workloads on different storage devices.

Contents

Acronyms

AWS *Amazon Web Services*. 1

DIR *directory service*. 7, 8

EBS *Amazon Elastic Block Store*. 1

HARNESS *Hardware- and Network-Enhanced Software Systems for Cloud Computing*. i, 1, 4, 11

HDD *hard disk drive*. i, 1, 3, 13–17

HDFS *Hadoop distributed file system*. 4

HPC *high-performance computing*. 4

IMP *Imperial College London*. 5

IOPS *input/output operations per second*. 1, 4, 11, 14, 15, 19

MRC *metadata and replica catalog*. 7–9, 12

NAS *network-attached storage*. 7

OSD *object storage device*. 7–9, 11–15, 19

POSIX *portable operating system interface*. 1, 7

QoS *quality of service*. i, 1, 3, 4, 11, 13, 15

RAID *redundant array of independent disks*. 9

RPC *remote procedure call*. 7

RTM *reverse time migration*. 4

SAN *storage area network*. 7

SSD *solid-state drive*. i, 3, 13–17

SSL *secure socket layer*. 7

UUID *universally unique identifier*. 8

ZIB *Konrad-Zuse-Zentrum für Informationstechnik Berlin*. 1, 4

1 Introduction

Predictable performance for storage-intensive applications requires also a predictable performance of the used storage system. While the characteristics of a single-user storage system are well known, e.g. by modelling *hard disk drive*s (HDDs), the performance behaviour of multi-tenant distributed storage systems is more complex. A cloud user is usually unaware of infrastructure details and the workloads of other users on the same infrastructure. Moreover, performance characteristics of conventional cloud platforms may vary between different providers, different resource instances, or by time. In such an environment, it is not possible to make any assumptions about the performance of an application by knowing only the available resources.

We will implement *quality of service* (QoS) extensions for the *portable operating system interface* (POSIX)-compliant, distributed file system XtreemFS [9]. XtreemFS was mainly developed by *Hardware- and Network-Enhanced Software Systems for Cloud Computing* (HARNESS) project partner *Konrad-Zuse-Zentrum für Informationstechnik Berlin* (ZIB) in the XtreemOS and Contrail FP7 projects. XtreemFS is well suited to cloud platforms on a globally distributed infrastructure and offers basic concepts to be used by multiple participants. The focus of these extensions will be to offer the possibility of an efficient use of heterogeneous hardware to fulfil the performance and capacity requirement of the users.

Current cloud storage providers offer only the possibility to choose a specific capacity, or have very limited QoS models. An *Amazon Elastic Block Store* (EBS) volume in the *Amazon Web Services* (AWS) cloud, for instance, is currently limited to a capacity of 1TB and 4000 *input/output operations per second* (IOPS). Guarantees on a streaming throughput are not possible. These facts suggest that EBS volumes are not implemented by a distributed storage system.

Previous research on QoS for storage systems has limitations. Various publications deal with a QoS-aware request processing on storage systems [6, 7, 11]. The authors describe block-based storage access and not object-based systems like XtreemFS. Huang et al. [8] describe the idea of using Toyoda's algorithm [14] for multi-dimensional resource provisioning in storage systems. Mu et al. [12] apply this approach to object-based storage systems. Both of these papers do not consider different file system access patterns and the volume size is limited to one storage device. We are not aware of any previous work on time-dependent anticipated usage patterns, which is demanded by Requirement R21 of Deliverable D2.1 [2].

To reach our mission to give performance guarantees for file system access, we have to define more flexible quality classes that match the requirements of a wide range of applications, and to understand the behaviour of storage systems for these classes in a multi-tenant usage scenario. We make this performance evaluation empirically for different access patterns and different levels of multi-tenancy.

Requirement R23 of Deliverable D2.1 [2] demands that we develop performance models for storage systems in the HARNESS cloud. We describe the considered workload types in Chapter 2. In Chapter 3, we give an introduction to XtreemFS. Chapter 4 describes the architectural extensions of XtreemFS to support quality of service guarantees and satisfy the storage requirements specified in the HARNESS project. The behaviour of XtreemFS in terms of the described workloads will be analysed in Chapter 5.

2 Workload Characterisation

The performance characterisation of storage systems differs significantly for different access patterns and different storage devices. For example, the sequential access performance of an HDD is higher than its random access performance. Other devices, such as *solid-state drive*s (SSDs), offer a high sequential and random throughput, but have a smaller capacity. In addition, applications show differences in their storage access patterns. To be able to provide storage with optimal performance characteristics and, thus, fulfil QoS requirements, we provide for each application a corresponding QoS *class*. Each of the classes consists of a capacity requirement, while two of our classes contain also a performance requirement. The metric of the performance requirement depends on the access pattern. We distinguish between sequential access, random access, cold storage, and best-effort access. We assume durability for all of the quality classes and, therefore, simply providing in-memory volumes does not fulfil the requirement.

2.1 Sequential Access

The sequential access pattern means an application is reading or writing a large file or a substantial part of the file in a stream. As disk head movements (seeks) are minimised, a rotating HDD achieves its best performance for this access pattern. Also, SSDs can handle sequential writes better that random writes. Because we consider the access pattern on the file-system level, and do not deal with a raw block device, it is important to guarantee that the logical file structure is also reflected in the physical storage device.

Many applications were designed with regard to this behaviour of rotating disks. A typical class of applications with a sequential access pattern are MapReduce applications [1], where potentially large input files are read in the map phase and result files are written in the reduce phase. The AdPredictor demonstrator [3] fits this category.

A sequential access pattern can be characterised as follows: We have a sequence of read or write requests to the same file. Each request r_i consists of an offset o_i and a request length l_i, which are given in bytes. A sequence of requests is a sequential access if $o_{i+1} = o_i + l_i + 1$ holds for all requests of the sequence.

The related QoS class in XtreemFS will guarantee a specified throughput in megabytes per second.

2.2 Random Access

A random access pattern is characterised by small reads or writes to different offsets of a huge file or to different small files. On rotating disks, this workload is dominated by the seek and settling time of the disk head and the throughput is significantly slower than for sequential access. In contrast, SSDs can handle random access better than rotating disks, but still not as fast as sequential access.

Typical applications with a random access pattern are relational databases or email servers, where frequent small writes appear at random positions.

A sequence of random access requests r_i with an offset o_i and a length l_i has the properties that the length of the requests l_i is relatively small (e.g. a few kB) and the offset o_i of the i^{th} request does not depend of the offset of the previous request r_{i-1}.

We quantify the random access performance by the number of IOPS with a size of 4 kB per second.

2.3 Cold Storage

We use the term *cold storage* for data that are written once and read very rarely. This holds, for example, for backups or archiving. For this kind of storage access, spare or leftover capacity can be used. These types of volumes may be used to replace a tape library.

This QoS class is described only by a capacity requirement.

2.4 Best-Effort Storage

The best-effort performance class can be used for any application with storage performance requirements that cannot be classified exactly or applications that do not rely on high-performance storage. A best-effort reservation contains also only a capacity component. Performance is provided as available, but not guaranteed, and thus a best-effort volume can be considered as a volume without an associated QoS.

2.5 Characterisation of the Validation Use Cases

The HARNESS project provides three applications as validation use cases: *reverse time migration* (RTM), Delta Merge, and AdPredictor. RTM is an *high-performance computing* (HPC) application that also displays a sequential access pattern. Delta Merge is part of the SAP HANA in-memory database system whose access pattern is currently unclear. AdPredictor is a machine-learning application that is available in as a MapReduce implementation and, thus, has a sequential access pattern on the used storage system.

2.5.1 Access pattern of AdPredictor

In Year 1 we analysed the access pattern of the storage system of AdPredictor. One of our AdPredictor implementations is based on the Apache Hadoop framework.[1] Hadoop offers an interface to replace *Hadoop distributed file system* (HDFS) by a custom file system. We implemented an XtreemFS adaptor for Hadoop and analysed AdPredictor using this. A MapReduce job executed by Hadoop reads input data in the map phase from the distributed file system. The intermediate results that are produced by the map tasks are stored on the local disks of the compute nodes. Finally, the results, which are produced by the reduce tasks, are again stored in the distributed file system.

Our evaluation has shown that our current implementation of the AdPredictor demonstrator produces intermediate results that are significantly larger than the input and output data. Because of this data flow, the running time of AdPredictor is dominated by storing and transferring intermediate results and so the performance of the distributed file system has only a minor relevance. HARNESS partners ZIB

[1] `http://hadoop.apache.org`

and *Imperial College London* (IMP) are currently working on an advanced version of AdPredictor that circumvents this problem and, thus, becomes more relevant for the XtreemFS evaluation.

3 XtreemFS Architecture

XtreemFS is a distributed and object-based file system consisting of a central *directory service* (DIR), metadata servers called *metadata and replica catalog*s (MRCs), *object storage device*s (OSDs), and clients (Figure 3.1). Separating data and metadata handling has been shown as a scalable alternative to monolithic architectures such as *network-attached storage* (NAS) or *storage area network* (SAN). XtreemFS offers POSIX-compatible semantics and is accessible from Microsoft Windows, Linux, and Mac OS clients. To ensure availability and reliability, replication is supported using a distributed lease negotiation algorithm [10]. Additional features include snapshots, end-to-end checksums, and parallel I/O. The internal communication is implemented by a protocol-buffers-based *remote procedure call* (RPC) protocol. In the remainder of this chapter, we will have a closer look at the XtreemFS components.

Elementary design decisions in XtreemFS were made to fulfil the requirements to be used by cloud providers. With regard to the CAP theorem [5], XtreemFS is designed to ensure consistency and partition tolerance. This choice gives the possibility to use XtreemFS to provide infrastructures for cloud providers, which might be distributed over different data centres. The architecture was also designed with the security demands of a distributed environment in mind and optionally uses certificates for authentication and *secure socket layer* (SSL) encryption. The multi-tenancy aspect of cloud infrastructures was considered by the possibility to use different volumes. A *volume* is an entity with its own file system name space in an XtreemFS cluster. All services use a staged architecture [15].

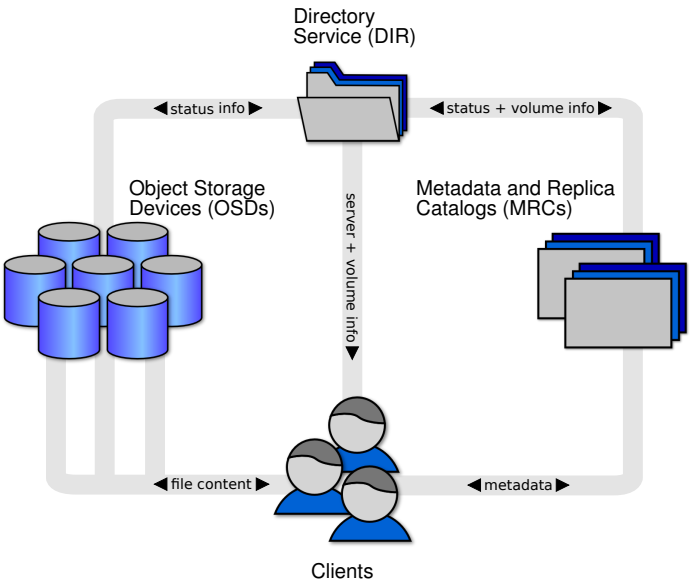

Figure 3.1: XtreemFS architecture.

3.1 Object Storage Device

OSDs store the content in an XtreemFS cluster. A file is split into chunks with a configurable size. The OSD offers an interface to access or modify these chunks. The OSD service is built on a local file system stack to use existing features of the underlying operating system such as block allocation or the page cache. The OSD also implements advanced features such as replication, versioning, scrubbing and check summing. OSDs can be added or removed at run time to an XtreemFS setup to change the capacity on demand.

The XtreemFS OSD supports two kinds of replication: read-only replication and read/write replication. Read-only replication can be used to deploy immutable files to a set of OSDs. The file can be read from each of the OSDs and, thus, the read performance increased. Read/write replication uses one master replica for each replicated file, which serialises all requests. The master replica is selected by a distributed, PAXOS-based lease negotiation [10]. Read/write replication is intended to increase reliability and availability, rather than performance.

3.2 Metadata and Replication Catalog

The Metadata and Replication Catalogs store and manage all metadata of an XtreemFS installation. Metadata is all information about a file or directory, except its content. This includes the file name, attributes such as permissions or time stamps, the directory hierarchy, and the responsible OSDs for a file.

One XtreemFS volume is managed by one MRC. To mitigate potential drawbacks in terms of scalability, it is recommended to partition the data manually to different volumes and deploy them across different metadata servers. In a multi-tenant cloud scenario, for example, one volume per user is best. XtreemFS volumes are lightweight and have a negligible overhead. The available OSDs can be shared between the volumes.

The MRC uses BabuDB [13], a non-relational database that was developed with the objective of metadata storage in file systems and offers a key/value store interface. BabuDB supports database snapshots, which are used to implement the file system snapshots in XtreemFS. Another advanced feature of BabuDB is database replication. Using this feature, a replication of the MRC is implemented. BabuDB offers the possibility to combine multiple write operations into an atomic operation. The MRC implements atomic file system operations, such as a rename, using this atomic database operations.

3.3 Directory Service

The DIR is a global registry for all services and volumes in XtreemFS. MRCs and OSDs initially register at the DIR and report status information periodically. Using the DIR, an MRC can determine the set of available OSDs and a client can determine the responsible MRC for a volume. All entities in an XtreemFS cluster are identified by a *universally unique identifier* (UUID). The DIR resolves these UUIDs to the corresponding servers and provides their configuration. The DIR also uses BabuDB internally.

3.4 XtreemFS Client

Access to XtreemFS can be performed using a library that encapsulates the communication with the different services. The library, called `libxtreemfs`, is available in Java and C++ versions. Based on this library, a Fuse-based client[1] for Linux and Mac OS, a CBFS-based client[2] for Microsoft Windows, and an Apache Hadoop integration are available. The Fuse and CBFS clients offer the ability to mount XtreemFS volumes in the same way as a local file system to the operating system.

Figure 3.2 illustrates file system access using a Fuse/CBFS client. A request is forwarded through the file system interface of the operating system to `libxtreemfs` using Fuse or CBFS. On open, `libxtreemfs` communicates with the MRC to get the file position on the OSDs. In the next step, the file content is directly read from or written to the OSDs. Files can be written to a single OSD or striped over multiple OSDs, similar to a *redundant array of independent disks* (RAID) level zero. Striping cannot, however, be used in combination with read/write replication.

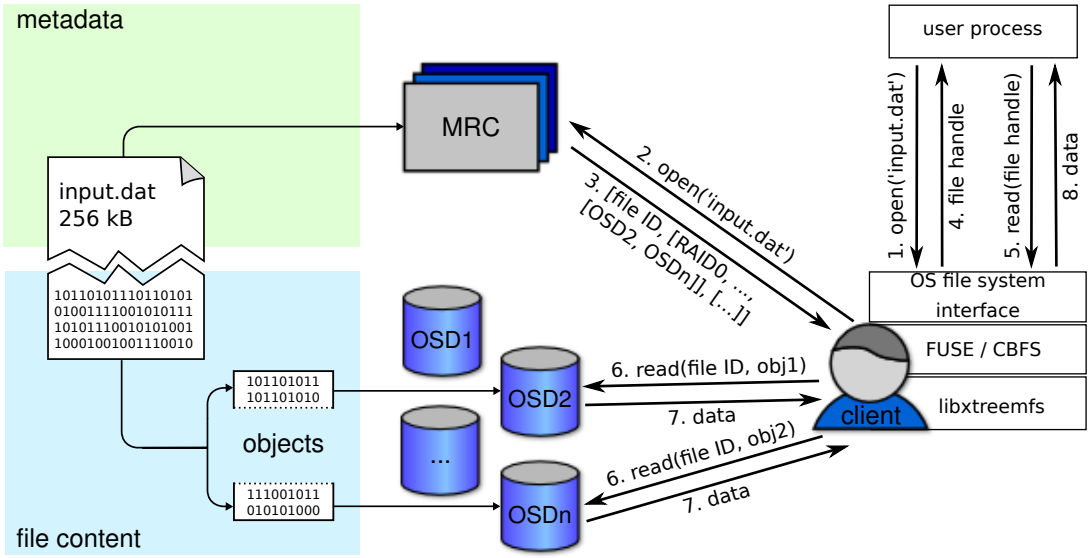

Figure 3.2: File access with XtreemFS.

[1] `http://fuse.sourceforge.net/`
[2] `https://www.eldos.com/cbfs/`

4 Multi-Tenancy Extensions to XtreemFS

To provide a multi-tenant storage system with predictable performance, we introduce QoS guarantees on the volume level. We extend the XtreemFS architecture with two components to provide this functionality to satisfy HARNESS Requirement R20 [2]. The first extension is a reservation scheduler, which guarantees that not more volumes are placed on an OSD than it can handle. The second extension is a queuing stage at the OSD, which ensures that each volume gets the reserved amount of performance in an overload situation.

4.1 Reservation Scheduler

We begin by describing the way XtreemFS is designed to schedule reservations.

4.1.1 Quality of service reservations

A QoS reservation is given by the following components:

- volume capacity;
- sequential performance;
- IOPS; and
- a reservation type

For the remainder of this chapter, we refer to the first three properties as the *resources demands* of a volume, and the combination of sequential performance and IOPS as the *performance requirements* of a volume. The reservation type must be one of the workloads defined in Chapter 2. A capacity must be given for each reservation type. A sequential performance is only allowed for the corresponding workload type and an IOPS value only for a random-access reservation. Giving a sequential performance or an IOPS requirement for a cold storage or best-effort reservation is not allowed. The QoS requirement is defined at volume creation and cannot be changed until the volume is deleted.

Striping files over multiple disks is one way to increase the performance of a storage system. We also use striping to have the possibility to create volumes with a capacity or performance that cannot be fulfilled by a single OSD. We make the assumption that the access for a random access volume is distributed uniformly over all OSD in a stripe. For a sequential access volume, we assume that the whole stripe is accessed by parallel I/O.

We can formalise the volume placement as follows. We have a given set of OSDs $O = \{o_1, ..., o_n\}$. Each OSD o_i has a set of available resources $r_o(o_i, \|V(o_i)\|) \in \mathbb{R}^k$. In our case, the resources are the OSDs capacity, the sequential throughput, and the random IOPS, and thus $k = 3$. The set $V(o_i)$ contains all volumes v_j that are mapped to the OSD o_i. In Chapter 5 we see that r depends on the number of mapped volumes. A volume v_j has resource requirement $r_v(v_j) \in \mathbb{R}^k$. Each volume v_j has an access

pattern $p(v_j) \in \{sequential, random, cold\_storage, best\_effort\}$. Each OSD o_i has a usage type $t(o_i) \in \{sequential, random, best\_effort\}$.

A valid volume placement V must respect the following constraints: First, the access pattern of the volume and the usage type of the OSDs must match. A match is given if the access pattern and the usage type are equal or if the access pattern is $cold\_storage$. Second, we must guarantee that an OSD can provide the resource demands of all mapped volumes: $\Sigma_{v \in V(o_i)} r_v(v) \leq r_o(o_i)$.

4.1.2 Scheduling objectives

The general objective of the reservation scheduler is to maximise the resource usage, while the performance requirements of the mapped volumes are fulfilled. To fulfil the resource requirements of all volumes, the resources of each OSD must be larger then or equal to the requirements of all volumes mapped to the OSD. If a volume is striped over multiple OSDs, the resource demand of the volume on each of the OSDs is the total resource requirement of the volume divided by the stripe width. The schedule must be reported to the MRC, which handles the binding between OSDs and volumes.

We assign dedicated OSDs to each kind of workload. This strategy has various reasons. Mixing up sequential and random workloads on rotating hard disks would cause a significant performance drop for sequential workloads, since additional disk-head movements would appear and no effective prefetching would be possible. We discuss this effect in Section 5.1. Additionally, different performance metrics for sequential and random access make it difficult to estimate performance capacity of an OSD for a mixed workload. Thus, only cold storage reservations may be placed on each OSD, since access to these volumes can be delayed until the OSD is idle.

4.2 Request Processing

Beside the reservation scheduling at the time of the volume creation, it is necessary to control the amount of performance a volume gets at run time. Therefore, the reserved performance should be a lower bound of the available performance of a volume, that is, a volume may get more performance than reserved if available.

Otherwise, if the sum of the requested performance is larger than the performance capabilities of an OSD, it is necessary to ensure that the performance requirement of each volume is fulfilled. To ensure that every volume gets the proportional share of the available OSD performance, we extend the OSD by a weighted fair-share queuing stage. As the OSD still has a staged architecture (see Chapter 3.1), new functionality can be integrated with minimal intrusion.

5 Performance Models

To make a volume placement that fulfils QoS requirements, it is necessary to understand the performance behaviour of the XtreemFS OSDs. In this chapter, we use empirical data to derive a performance model that can then be used by the reservation scheduler. According to the workload characterisation in Chapter 2, we analyse the performance for sequential and random access. We analyse the performance behaviour for a single OSD with a varying number of concurrent readers or writers for both of the workloads. The benchmarks have been executed on both a spinning HDD and an SSD. The hard disk was a Seagate ST9250610NS having a capacity of 250 GB and a rotation speed of 7200 RPM. The SSD was a Samsung 840 Pro with a capacity of 512 GB, a sequential throughput of 540 MB/s for reading and 520 MB/s for writing.

5.1 Sequential I/O Performance

For the evaluation of sequential access we consider the read and write performance by separate benchmarks. To simulate a multi-tenant scenario, we read or write with a varying number of concurrent streams to a file having a size of 10 GB. Each stream reads or writes its own file.

5.1.1 Read access

Figure 5.1 shows the performance of a varying number of sequentially reading threads on a HDD. After a small increase, we see a significant performance drop for an increasing number of parallel readers. This behaviour can be explained by additional disk-head movements, which become necessary when more than one stream is read from a disk. A huge number of parallel readers would result in a nearly random access behaviour.

The performance of concurrent read streams on an SSD is shown in Figure 5.2. The performance drop of an increasing number of concurrent readers is less significant for this type of storage device.

5.1.2 Write access

Figures 5.3 and 5.4 show the same experiment for parallel write access. Unlike for read access, we do not see a performance drop for concurrent write streams. This can be explained by the internal file structure of XtreemFS. The XtreemFS OSD splits an incoming write stream into chunks with a default size of 128 kB and writes each chunk to a separate file. When multiple streams are written concurrently to XtreemFS, the write of the chunks is interleaved, but this does not result in a significant performance drop as only small disk-head movements are necessary for the HDD.

Because our QoS model distinguishes only between different access patterns, but not between read and write access, we must consider the minimum of the read and the write performance for our model. We can conclude that the sequential I/O performance drops with an increasing number of parallel streams.

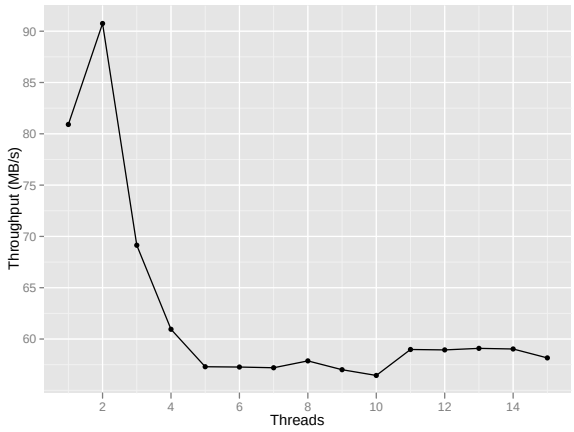

Figure 5.1: Sequential read performance for HDDs.

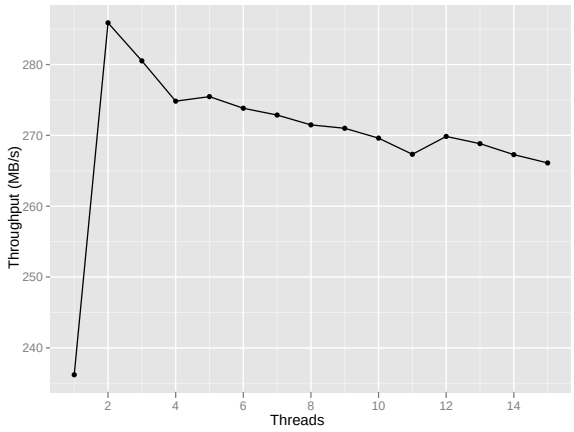

Figure 5.2: Sequential read performance for SSDs.

5.2 Random I/O Performance

To evaluate the random access performance of an XtreemFS OSD, we consider read and write performance again by different benchmarks. The objective of this benchmark is to determine the number of IOPS with a size of 4 kB. To perform this evaluation, we use a file with a size of 10 GB and perform I/O requests at random positions.

5.2.1 Read access

The performance of the random read performance with a varying number of concurrent readers on a HDD is shown in Figure 5.5. The performance drop for the increasing number of concurrent readers can

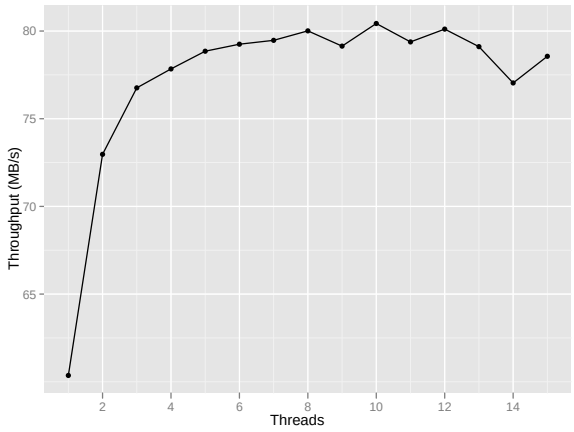

Figure 5.3: Sequential write performance for HDDs.

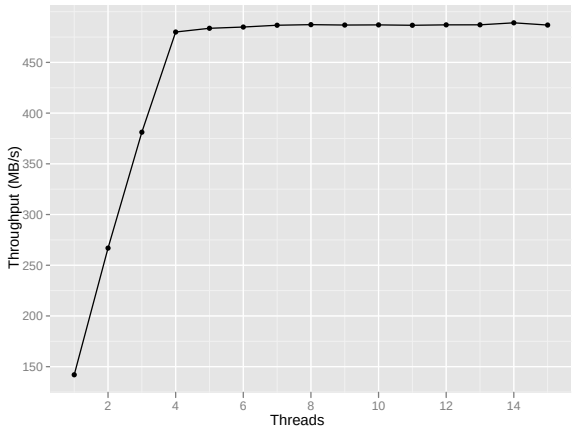

Figure 5.4: Sequential write performance for SSDs.

be explained by the increasing seek distances: the disk head must overcome a growing amount of used capacity. As shown by Figure 5.6, this behaviour for SSDs is less significant.

5.2.2 Write access

Figure 5.7 shows the results of the random write benchmark for HDDs and Figure 5.8 for SSDs. For this experiment, the throughput is similar to the read benchmark.

The QoS model for random access differs from the model for sequential access. The random I/O performance of an OSD with an HDD depends on the amount of used capacity and the on disk file layout rather than the number of concurrent accesses. We can assume that the number of IOPS an OSD can service is constant, so the volume scheduler does not have to take the current number of mapped volumes on an OSD into account. This model is also usable for SSDs.

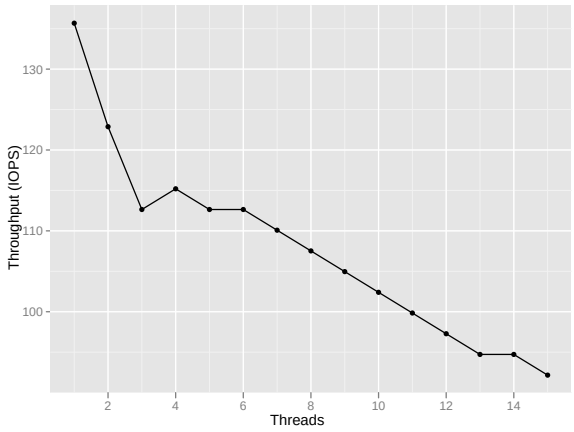

Figure 5.5: Random read performance for HDDs.

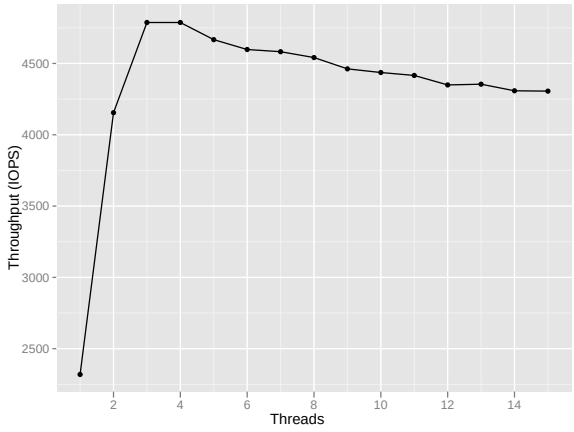

Figure 5.6: Random read performance for SSDs.

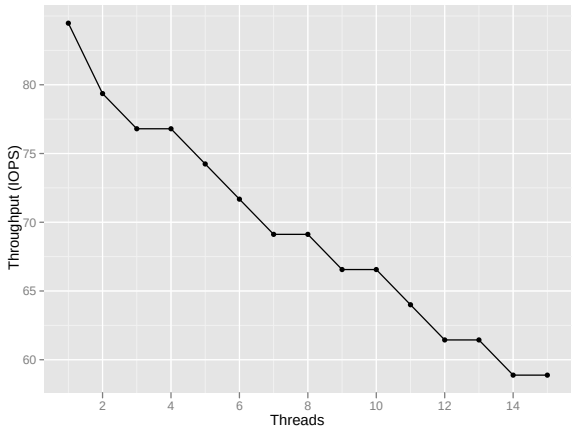

Figure 5.7: Random write performance for HDDs.

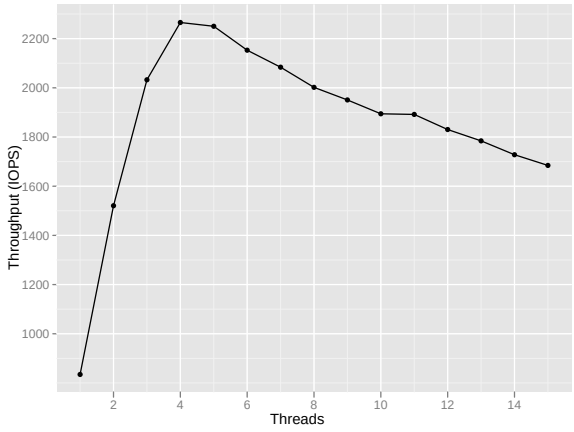

Figure 5.8: Random write performance for SSDs.

6 Conclusion

In this report we have experimented with different types of workloads, which cover a wide range of typical cloud applications. The distributed file system XtreemFS fulfils elementary requirements for a multi-tenant scenario and offers a well-designed architecture to integrate performance guarantees. In Year 1 we extended XtreemFS with two components: a reservation scheduler that maps volumes to OSDs and an overload protection to guarantee the reserved performance on the OSD level.

To find a performance model for the reservation scheduler, we analysed the behaviour of an XtreemFS OSD in a multi-tenant scenario for both sequential and random access. Our evaluation has shown that the total throughput of an OSD decreases with a growing number of parallel reading streams for a sequential access. For parallel write streams, this performance drop is less significant. We have seen that the number of tenants has no influence on the random throughput (i.e. the IOPS) of an OSD. Based on these facts, we present a prototype of our reservation scheduler in Deliverable D5.4.1 [4].

Bibliography

[1] J. Dean and S. Ghemawat. MapReduce: A flexible data processing tool. *Communications of the ACM*, 53(1), 2010.

[2] FP7 HARNESS Consortium. General requirements. Project Deliverable D2.1, 2013.

[3] FP7 HARNESS Consortium. Validation plan. Project Deliverable D2.3, 2013.

[4] FP7 HARNESS Consortium. Data storage component (initial). Project Deliverable D5.4.1, 2013.

[5] S. Gilbert and N. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant Web services. *SIGACT News*, 33(2):51–59, June 2002.

[6] A. Gulati, I. Ahmad, and C. Waldspurger. PARDA: Proportional allocation of resources for distributed storage access. In *FAST*, pages 85–98.

[7] A. Gulati, A. Merchant, and P. J. Varman. mClock: Handling throughput variability for hypervisor I/O scheduling. In *Proceedings of the Symposium on Operating System Design and Implementation*, pages 1–7, 2010.

[8] L. Huang, G. Peng, and T. Chiueh. Multi-dimensional storage virtualization. *SIGMETRICS Performance Evaluation Review*, 32(1):14–24, June 2004.

[9] F. Hupfeld, T. Cortes, B. Kolbeck, J. Stender, E. Focht, M. Hess, J. Malo, J. Marti, and E. Cesario. The XtreemFS architecture—A case for object-based file systems in Grids. *Concurrency and Computation: Practice and Experience*, 20(17):2049–2060, 2008.

[10] B. Kolbeck, M. Hogqvist, J. Stender, and F. Hupfeld. Flease—Lease coordination without a lock server. In *IEEE International Parallel Distributed Processing Symposium*, pages 978–988, 2011.

[11] C. R. Lumb, A. Merchant, and G. A. Alvarez. Facade: Virtual storage devices with performance guarantees. In *FAST*, pages 131–144, 2003.

[12] F. Mu, J. Shu, B. Li, and W. Zheng. Multi-dimensional storage QoS guarantees for an object-based storage system. In *Proceedings of the 6th International Conference on Computational Science*, pages 687–694. Springer-Verlag, 2006.

[13] J. Stender, B. Kolbeck, M. Hogqvist, and F. Hupfeld. BabuDB: Fast and efficient file system metadata storage. In *International Workshop on Storage Network Architecture and Parallel I/Os*, pages 51–58, 2010.

[14] Y. Toyoda. A simplified algorithm for obtaining approximate solutions to zero-one programming problems. *Management Science*, 21(12):1417–1427, 1975.

[15] M. Welsh, D. Culler, and E. Brewer. SEDA: An architecture ofr well-conditioned, scalable Internet services. *ACM SIGOPS Operating Systems Review*, 35(5):230–243, 2001.