Bachelor Thesis

Development of a decision support system for performance tuning in Lustre

submitted by

Julius Plehn

Fakultät für Mathematik, Informatik und Naturwissenschaften
Fachbereich Informatik
Arbeitsbereich Wissenschaftliches Rechnen

Studiengang:	Software-System-Entwicklung
Matrikelnummer:	6535163
Erstgutachter:	Dr. Michael Kuhn
Zweitgutachter:	Anna Fuchs
Betreuer:	Dr. Michael Kuhn, Anna Fuchs

Hamburg, 2018-23-05

Abstract

Performance critical components in HPC require an extensive configuration. This is especially true when a filesystem has to be configured for maximum performance, as it is a good practice to benchmark the system every time configuration changes have been made. Furthermore the correlation between the configuration and performance metrics might not be obvious, especially after several benchmarks. As the configurable components are distributed, the configuration management is an additional challenge. This trial and error process is time consuming and error-prone. In order to be able to make valuable decisions it is important to gain detailed insights into the system and the state of the configurations.

The goal of this thesis is to evaluate the usage of a decision support system to simplify the process of performance optimization. Therefore a detailed insight into the possibilities of performance optimization is provided. Additionally, a Command Line Interface (CLI) is developed to perform automated Lustre client changes and IOR benchmarks. The decision support itself is supported by a web interface, which is used to visualize the configuration changes and the benchmark results.

Contents

1. Introduction **5**
- 1.1. Motivation . . . 6
- 1.2. Thesis Goals . . . 6
- 1.3. Outline . . . 7

2. Background **8**
- 2.1. Lustre . . . 8
  - 2.1.1. Architecture . . . 8
  - 2.1.2. Layers . . . 10
  - 2.1.3. Client Configuration . . . 11
- 2.2. IOR . . . 17

3. Related Work **20**
- 3.1. Manual Benchmarking . . . 20
- 3.2. Intel Manager For Lustre . . . 20
- 3.3. Automatic Configuration within Lustre . . . 21

4. Design and Implementation **22**
- 4.1. CLI . . . 22
  - 4.1.1. Automation JSON . . . 23
- 4.2. Web Interface . . . 24
  - 4.2.1. Diff . . . 24
  - 4.2.2. Benchmark Visualization . . . 25
  - 4.2.3. Recommendations . . . 26
  - 4.2.4. MongoDB . . . 26
  - 4.2.5. Docker . . . 27

5. Evaluation **28**
- 5.1. Test setup . . . 28
- 5.2. Automation configuration . . . 28
- 5.3. Standard tests . . . 29
  - 5.3.1. IOR benchmark with files-per-process and reordered read back configuration . . . 29
  - 5.3.2. IOR benchmark with files-per-process, reordered read back configuration and continuous block size . . . 32
  - 5.3.3. IOR benchmark with random access configuration . . . 35

5.4. Stripe size tests 38
5.4.1. IOR benchmark with 4 MiB transfer size 38
5.4.2. IOR benchmark with 4 MiB transfer size and 4 MiB stripe size . . 39

6. Conclusion **40**

7. Future Work **41**

Bibliography **42**

Appendices **44**

A. Abbreviations **45**

B. Demo **46**

List of Figures **47**

List of Listings **48**

List of Tables **49**

1. Introduction

High performance computing (HPC) aggregates the power of single servers to a unified parallel computing system. While single servers have certain bottlenecks like the maximum amount of Floating Point Operations Per Second (FLOPS), memory capacities, network bandwidth and storage size, a cluster in an HPC environment is able to overcome those limitations. To benefit of these capacities, applications need to be optimized for the optimal utilization of the provided hardware. For example, to use processing cores on different servers a communication protocol is needed. The de facto standard for this purpose is the Message Passing Interface (MPI), which makes it possible to send data across the high speed network to other processes and to synchronize the state of the application. Given the possibility to use independent processes to speed up the application it is obvious that this scenario benefits of the usage of distributed parallel file systems.

A distributed parallel file system is characterized by some objectives and best practices, which can be found in various implementations. In general, a file is split into smaller pieces (often called stripe or chunk) which then get saved to a number of servers. The size of those chunks and the number of servers the file gets distributed to highly depends on the application. Besides the actual data, the meta data is distributed to specific servers as well. This is necessary in order to be able to provide a unified view on the whole file system, while maintaining a single access point to the clients. While this describes the distributed aspect of those file systems, a parallel access pattern is necessary to have a performance benefit. The general approach to archive optimal file system performance is to have independent processes (e.g. by using MPI) which read and write to process specific sections of a shared or exclusive file. Using this approach to vertical scaling, the I/O performance is no longer limited by a single system.

The demand for HPC is reaching new heights as new applications suitable for this area of computing are arising and existing applications are evolving. For example the demand for climate models with a bigger resolution and more time steps is increasing but the computation capabilities are not available at this time. This immediately shows the relevance for exascale computing. An example given in [12, p. 92] describes a scenario where a climate model requires "a calculation at 100x resolution and 10x more time-steps, for a total of 1000x more computation than is possible today".

The computing power and memory capacities are developing much faster than the storage capacities. Therefore it is a big challenge to develop systems where the balance between

those bottlenecks is even. A visualization that shows the different rates of development is shown in Figure 1.1.

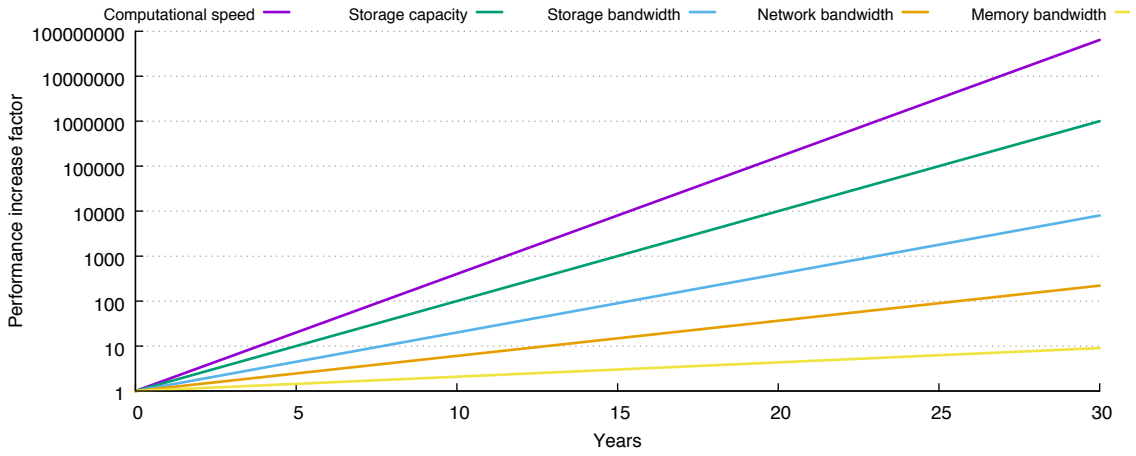

Figure 1.1.: Development of computational speed, storage capacity and storage speed [based on 13, p. 75]

1.1. Motivation

As shown in the section above the demand for HPC is rising and the challenges that go along with are enormous. Efficient, reliable and predictable systems are mandatory for performance critical applications - now and especially when facing new challenges the future might hold. To gather more insights into the behavior of distributed parallel file systems, a good understanding of the configuration options is necessary. Depending on the options to set on the system, several possibilities might need valuable consideration. Paired with specific benchmark requirements, a single option on a Lustre file system can require an amount of manual configuration, manual benchmarking and evaluation that is unsuitable for a production or testing environment. A tool that simplifies and automates this recurring and error-prone process could increase the file systems performance and provide greater insights into the file system in general. Use cases might include integration into the existing Lustre development pipeline, general benchmarking of the file system and training purposes to gain a better understanding of the file system internals.

1.2. Thesis Goals

The purpose of this thesis is to develop and evaluate a decision support system that provides the user with a better understanding of the behavior of the Lustre file system. The system developed during this thesis consists of two independent tools. A CLI handles the Lustre client configuration and manages the benchmark runs, while a web

interface is used to compare the client configuration and to visualize the benchmark results. Additionally, the web interface serves as a storage backend which enables further data evaluation. As the configuration changes performed by the CLI are temporarily, the regular behavior of the file system is not interrupted.

1.3. Outline

The remainder of the thesis is organized as follows:

- In the **Background** chapter, the fundamentals of the Lustre file system and the IOR benchmark are presented. A special focus lies on the Lustre client, as this is the part which is configured and evaluated later on.
- The **Related Work** chapter describes common approaches to evaluate and configure Lustre file systems. A comparison to the attempt described in this thesis is made.
- The **Design and Implementation** chapter puts the focus on the tools that have been developed during this thesis.
- In the **Evaluation** chapter an overview of tested Lustre configurations is given. Additionally, the decision support tool is used to evaluate the performance of the current configuration while giving recommendations for further optimization.
- At the end a **Conclusion** for the usage of the decision support tool is given, followed by ideas for the **Future Work**.

2. Background

This chapter gives an introduction into the fundamental concepts of the Lustre file system and the IOR benchmark.

2.1. Lustre

Lustre is a distributed file system used in over 60% of the supercomputing sites listed on the TOP500.org list [17]. A distributed file system is necessary if an application exceeds the capabilities of a single server. In a vertically scaled environment the limiting factors are networking bandwidth and storage size and speed. Even when using modern NVMe Solid State Drives, the reading speed will not exceed 5000 MB/s while the writing speed is limited to 3000 MB/s [7]. When using Lustre, several servers, each with their individual limited capabilities, are unified to a single POSIX compliant file system. To gain a speedup and to actually benefit from the distributed file system, it is necessary to split files across several servers. This process is called file striping and more details are provided in section 2.1.2. Using this approach Lustre "is best known for powering many of the largest HPC clusters worldwide, with tens of thousands of client systems, petabytes (PB) of storage and hundreds of gigabytes per second (GB/sec) of I/O throughput." [9, p. 27].

2.1.1. Architecture

Management Server

The Management Server (MGS) provides information about the Lustre file system to all the other components. While this server is not involved in the actual I/O, this service is the central entry point for all other servers. Therefore, a fail-over configuration is desired. The configuration itself is saved on a Management Target (MGT).

Metadata Server

Metadata Servers (MDSs) are providing information about the metadata to the clients. MDSs determine the file layout either using default settings or user provided configuration

on a per file or directory base. Once the client received the file layout no further interaction with the MDS is necessary. The metadata is stored on 1-4 Metadata Targets (MDTs). Each Lustre file system has to have at least one MDS. Additional servers can add fail over capabilities. Furthermore in Lustre 2.4 the Distributed Namespace Environment (DNE) was introduced [9, p. 31]. It is now possible to add additional MDSs and MDTs which hold a sub-directory of the file system. Beginning with Lustre 2.8 it is also possible to distribute a single directory across multiple MDTs. Using this workload distribution it is possible to increase the number of meta data operations.

Metadata Target

MDTs store the actual metadata of the file system. This includes the file names, permissions and extended attributes. The extended attributes contain the *layout EA*, which consists of the Object Storage Targets (OSTs) the file is stored on and the actual location of the data chunks on those servers.

Object Storage Server

Object Storage Servers (OSSs) are providing file access and network request handling to their connected OSTs. Each OSS can handle up to 32 OSTs which each hold up to 256 TB on a ZFS file system.

Object Storage Target

OSTs hold the actual data in its own local file system. It is possible to use either ldiskfs, an improved version of the ext4 file system, or ZFS.

Client

Clients are consumers of the file system. They are able to mount the file system by providing the address of the MGS. As Lustre is POSIX compliant there are no further dependencies to write to the file system. However, in order to mount a Lustre file system a kernel module needs to be installed. This module can be obtained as a RPM package or as source code, which provides greater flexibility. An example on how to mount a Lustre file system is given in Listing 2.1. The performance optimization of the client is the main focus of this thesis. Other components provide configuration options as well but the client is the perfect test candidate because the temporary configuration changes have no affect on other users.

```
1 mount -t lustre $(MGS):/lustre /mnt/lustre/client
```

Listing 2.1: Lustre Mounting

In Figure 2.1 a common setup is shown. The DNE is used to distribute the workload of meta data operations. Additionally, the MGSs, MDSs and OSSs are configured for fail-over.

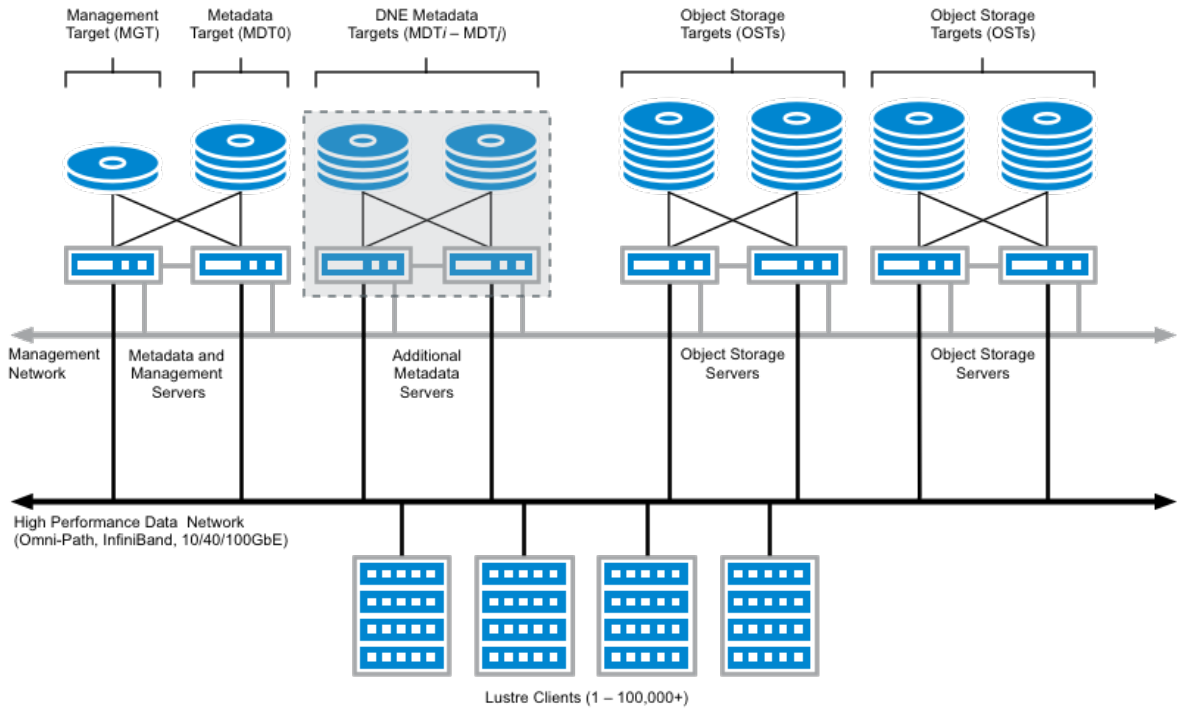

Figure 2.1.: Lustre file system overview [4, p. 2]

2.1.2. Layers

llite

Llite depends on the Virtual File System (VFS) implemented in Linux. A VFS provides an additional layer between a logical operation on the file system and the physical storage of the data. As llite implements a compatible VFS interface, POSIX compliant operations reach the Lustre stack through llite. Therefore llite provides several configuration options to optimize the file system performance.

Object Storage Client

Every connection to an OST is handled by a dedicated Object Storage Client (OSC) that resides on every client that mounted the file system. Requests are encoded in Remote Procedure Calls (RPCs) which have a fixed size of 1 MB by default. This value can be increased to up to 4 MB per RPC. The OSCs are crucial for the file system performance

as they are the low level connection to the actual storage. Several parameters like the amount of parallel RPCs and data caches need to be considered.

Striping and LOV

In order to be able to save data across multiple OSTs and therefore make use of the parallel file system, striping is used. Using this technique it is possible to spread out a file across up to 2000 OSTs [9, p. 36]. Striping can be configured independently for directories and files using the `stripe_count` and `stripe_size` options. The `stripe_count` value specifies the number of OSTs the data should be distributed across. The default value is one, which means the data is only saved on one OST. This works well for small files, as every connection to another OST adds an overhead, which in many use cases is not beneficial. The `stripe_size` option defines the size of the data chunks distributed. The default value is 1 MB.

The logical object volume (LOV) provides access to the OSTs by aggregating the OSCs to a single namespace. Therefore a client sees a "single, coherent, synchronized namespace, and files are presented within that namespace as a single addressable data object, even when striped across multiple OSTs." [16, p. 27].

In Figure 2.2 the whole Lustre client software stack is shown. The decision support system developed in this thesis is applied to the llite and OSC layer.

2.1.3. Client Configuration

In this section the Lustre client configurations that are evaluated later on are reviewed. A special focus lies on their behavior on the file systems performance and their relation to other configurations. The configurations regarding the OSC and llite layers are reviewed on their own, as they do not affect each other.

lctl

The main utility to manage Lustre components is *lctl*. For this thesis, the commands *get_param* and *set_param* are especially important, as the management tool developed uses these to configure the Lustre clients. To select the appropriate settings the syntax *{file-system-name}.{subsystem}.{parameter}* is used. Additionally the wildcard character * allows for filename patterns. Options set by *set_param* are not written to the MGS and therefor are lost after a reboot. To set parameters permanently the command *conf_param* is necessary. Listing 2.2 shows how to get and set an llite configuration.

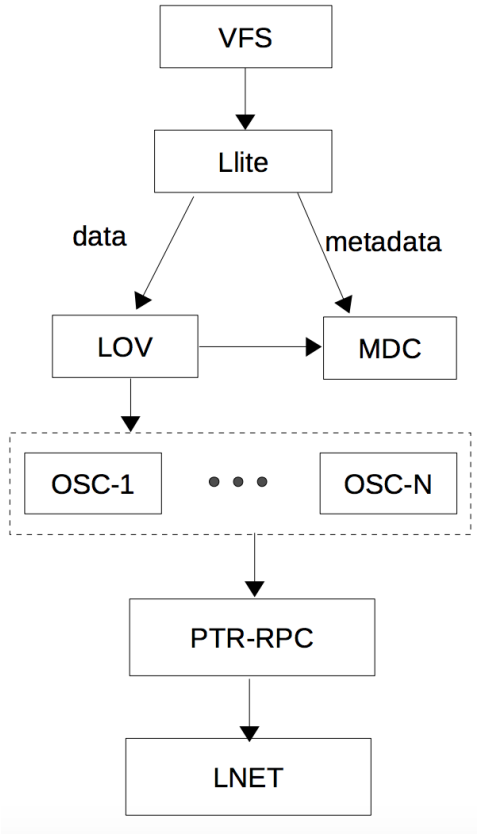

Figure 2.2.: Lustre client layers [20, p.11]

```
1 lctl get_param llite.*.max_read_ahead_mb
2 lctl set_param llite.*.max_read_ahead_mb = 64
```

Listing 2.2: lctl get_param and set_param

OSC

max_rpcs_in_flight
This option sets the number of concurrent operations that will be performed by the client per OST at the same time.

max_pages_per_rpc
This option specifies the number of pages that will be encoded in every RPC sent by an OSC. The size of a single page is defined by the CPU. While modern hardware can support larger page sizes it is common for operating systems to set the page size to 4

KiB[1]. RPCs are handled by the Lustre Network (LNet). In order to be able to handle the concurrency and scheduling of RPCs a credit system is in place. Every RPC claims one credit point while in execution and an additional MiB of data adds another credit point. Therefore when using 256 pages/RPC every 1 MiB transaction costs two credits. However, enabling 1024 pages/RPC results in 4 MiB of data that can be encoded into one RPC. This results in a credit point system shown in Table 2.1. Eventually this means that it is possible to send more data with less RPCs and credits, resulting in more credits left for additional transfers.

Especially in combination with other configurations this setting can have a significant impact on the performance. For example when setting `max_rpcs_in_flight` to 8 and `max_pages_per_rpc` to 256 the theoretical amount of data which can be transferred at the same time is limited to 8 MiB. However, when setting `max_pages_per_rpc` to 1024 this value rises to 32 MiB.

	256 pages/RPC	1024 pages/RPC
1 MB write	1 RPC, 2 Credits	1 RPC, 2 Credits
2 MB write	2 RPC, 4 Credits	1 RPC, 3 Credits
3 MB write	3 RPC, 6 Credits	1 RPC, 4 Credits
4 MB write	4 RPC, 8 Credits	1 RPC, 5 Credits

Table 2.1.: LNet Credits [2, p. 14]

`max_dirty_mb`
This configuration sets the maximal amount of dirty data to be cached in the memory for each OSC. Data that needs to be saved permanently is written to the cache first. From the applications perspective the write process is completed at this stage. Lustre can use this cache to queue up data to create well formed RPCs. This is especially useful for applications that perform small I/O. When experimenting with this option one has to keep in mind that the respective client thread is blocked while a forced cache flush is in process. Additionally, in this phase of the transmission the data is only saved in volatile memory. In case of a system interruption that data is inevitably lost. A rule of thumb is to set `max_dirty_mb` = `max_rpcs_in_flight` $*$ 4.

`checksums`
Checksums can help to detect corrupt data early on within the Lustre stack. Every package sent and received over the network or stored in the memory is validated using a 32-bit checksum. In case of data corruption the data is sent again up to five times. Due to other safety precautions implemented in hardware like cyclic redundancy checks (CRC) in networks and error-correcting code (ECC) in memory one recommendation is to disable the checksums altogether [18].

[1] 1 KiB = 2^{10} bytes = 1024 bytes

`checksum_type`
Lustre provides two different checksum algorithms which are used if the `checksums` option above is activated.

Adler-32 is an algorithm invented by Mark Adler. It has been implemented for the first time in the zlib library in 1995 [1]. A 32-bit checksum is made up by concatenating two 16-bit integers `S1` and `S2`. `S1` is the sum of all bytes plus one whereas `S2` is the sum of all values from the previous `S1` calculation. In the end, the modulo 65521 (largest prime number $< 2^{16}$) is calculated for both integers. An example is given in Table 2.2.

Character	ASCII code	S1	S2
L	76	$1 + 76 = 77$	$0 + 77 = 77$
u	117	$77 + 117 = 194$	$77 + 194 = 271$
s	115	$194 + 115 = 309$	$271 + 309 = 580$
t	116	$309 + 116 = 425$	$580 + 425 = 1005$
r	114	$425 + 114 = 539$	$1005 + 539 = 1544$
e	101	$539 + 101 = 640$	$1544 + 640 = 2184$

Table 2.2.: Adler-32 example

The next step is to convert `S1` and `S2` to their hexadecimal equivalents, which is `0x280` and `0x888` respectively.

$$\texttt{0x888} \ll 16 + \texttt{280} = \texttt{0x08880280} \tag{2.1}$$

2.1.: Final Adler-32 checksum

As shown in Equation (2.1) the final checksum is `0x08880280`. If the receiver calculates the same checksum the chances are high that the data has not been corrupted.

Another possibility is to use a **CRC-32** algorithm. In this case, a message is divided by a polynomial and the remainder of the division is then attached to the original message. The selection of the polynomial is the most important step in implementing a CRC algorithm as it has a severe influence on its error-detecting capabilities. Depending on the degree of the polynomial additional zeros have to be added to the message. For example when sending the message 11011 while using the polynomial $1x^5 + 1x^4 + 1x^2 + x$ (110101) a CRC computation can look like shown in Listing 2.3.

```
1101100000
110101
------
0000110000
    110101
    ------
     00101
```

Listing 2.3: CRC example

In this case the CRC encoded message is the concatenation of the message to send and the remainder of the division: 1101100101. In order to verify this message a receiver has to divide the checksum by the same polynomial which has been used before.

Another possibility is to use a CRC version implemented in hardware called **CRC32C**. Algorithms implemented in hardware usually mean a substantial speedup due to advanced cycle management.

In Figure 2.3 every implementation has been benchmarked. The Adler checksums algorithm is superior when hardware acceleration is not available. If both, the CRC32C and Adler implementations are available, the hardware implementation should be preferred, because otherwise CPU cycles are wasted.

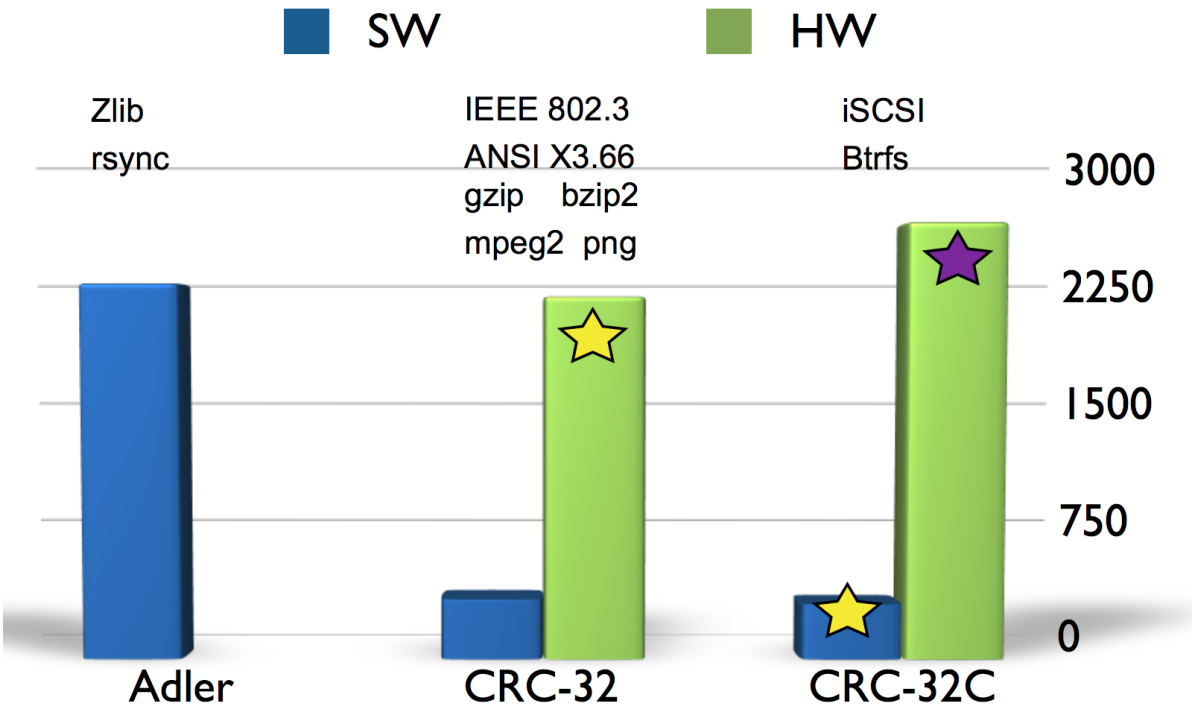

Figure 2.3.: Bulk Checksum Speeds, MB/s [19]

Llite

`max_cached_mb`
This configuration specifies the amount of cached data within the llite layer.

`max_read_ahead_per_file_mb`
This configuration specifies the amount of data that can be read ahead for each file. After a second sequential request that can not be fulfilled by the cache, the next RPC sized chunk is requested automatically. An access to a non-sequential chunk resets the algorithm.

`max_read_ahead_mb`
This configuration specifies the total amount of MB allocated to the read ahead cache.

`max_read_ahead_whole_mb`
This defines the maximal size of a file that can be read ahead in its entirety, regardless of the RPC size.

2.2. IOR

IOR (interleave or random) is an I/O benchmarking tool, commonly used in high performance computing (HPC). It provides special functionalities for HPC usage like Lustre specific `stripe_count` and `stripe_size` commands. For a better understanding of the benchmarks discussed in this thesis, a broad overview of the parameters is given (see Table 2.3).

Parameter	Meaning	Default
-a S (api)	API for I/O	POSIX, MPIIO, HDF5, HDFS, S3, S3_EMC, NCMPI
-b N (blockSize)	contiguous bytes to write per task	1048576
-F (filePerProc)	file-per-process	false
-i N	number of repetitions of test	1
-N N	number of tasks that should participate in the test	0 (all tasks)
-s N (segmentCount)	number of segments	1
-t N (transferSize)	size of transfer in bytes (e.g.: 8, 4k, 2m, 1g)	262144
-z	access is to random, not sequential, offsets within a file	false
-Z	changes task ordering to random ordering for readback	false
-C (reorderTasksConstant)	changes task ordering to N+1 ordering for readback	false
-e (fsync)	perform fsync upon POSIX write close	false
-O jsonOutput=out.json	write benchmark results to specific file	false

Table 2.3.: Important IOR Settings

In order to be able to post process the benchmark results a modified version of IOR was used [23]. When using the parameter *jsonOutput* together with an JSON output path, a file containing the benchmark results is created. In Listing 2.4 an example is shown. The

truncated `parameter` object contains information about the used benchmark options. The visualization shown in Section 4.2.2 is based on the Max, Min, Mean and StdDev variables shown in lines 7-10 and 15-18 respectively.

```
1  {
2     "test0": {
3       "parameter": {...},
4       "results": {
5         "write": {
6           "Operation": "write",
7           "Max": 108538239.11535607,
8           "Min": 96668213.190148085,
9           "Mean": 104545764.23861851,
10          "StdDev": 5570442.3230357831,
11          "MeanTime": 2.0119009017944336
12        },
13        "read": {
14          "Operation": "read",
15          "Max": 4592876324.8021049,
16          "Min": 4274658493.6838164,
17          "Mean": 4461210704.2883301,
18          "StdDev": 135585275.35404551,
19          "MeanTime": 0.047052780787150063
20        }
21      }
22    }
23  }
```

Listing 2.4: Truncated JSON output

In order to be able to benchmark a file system in a way that reflects the real world performance, one has to keep the effects of locking mechanisms and caching in mind.

Locking:
IOR differentiates between shared and file per process I/O. When benchmarking shared read/write performance each process handles blocks with the size defined by the `blockSize` parameter. The size of each operation on these blocks is defined by the `transferSize` parameter. Additionally the `segmentCount` indicates how often these blocks are handled. The total amount of data that is read and written during one iteration of a benchmark is therefore defined by `blockSize * segmentCount`. A visualization of this access pattern is shown in Figure 2.4. As several processes are operating on the same file, locking mechanisms are involved. This can be changed by adding the *filePerProc* parameter which eliminates the need for locking mechanisms as every process operates on its own file and therefore speeds up the I/O performance.

Caching:
When doing I/O benchmarks it is important to know when client side caching is involved. Whenever data is written to the file system a copy of that data stays in the DRAM. If the same Lustre client requests to read that data again, it is possible that the file system is not involved in the reading process, as the data can be retrieved by the memory. In this case a benchmark is not able to represent the performance of the file system, but the capabilities of the memory on the client. IOR implements several parameters to avoid these cases. By defining an offset with the `reorderTasksConstant` parameter one can make sure that nodes do not re-read data they have written in the previous write step. When activated, the task ordering is set to $N + 1$, meaning that node N writes data, while node N+1 reads the data.

Another caching mechanism is involved when a client writes data, as this process is successful as soon as the data has been written to the cache. However, this does not necessarily mean that the data already has been committed to the Lustre file system. IOR provides the *fsync* parameter to ensure, that the content of the cache is written to permanent storage.

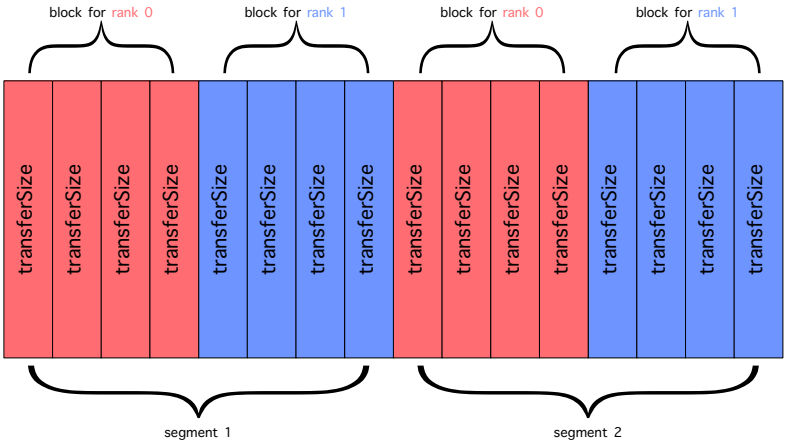

Figure 2.4.: IOR file access [inspired by 14]

3. Related Work

This chapter provides a broad overview over existing approaches to performance optimization in HPC.

3.1. Manual Benchmarking

The traditional way to benchmark a file system is to call the benchmark utility on its own. While this works for a small amount of benchmarks, the effort to gain insights and to actually improve the performance of the file system is significant. To provide some kind of automation a widely used approach is to write a bash file containing the path to the benchmark utility and some rules that set the benchmark options depending on the amount of nodes participating. An example is shown in the community repository for Lustre, where the *blocksize* is divided by the amount of participating nodes to keep the total size for the benchmark runs the same [see 3].

3.2. Intel Manager For Lustre

The goal of the Intel Manager For Lustre is to "provide a unified, consistent view of Lustre storage systems and simplify the installation, configuration, monitoring, and overall management of Lustre" [5]. Intel justifies their effort with the "grow in complexity" [8] of storage solutions required for today's applications. Like the tool developed in this thesis Intel combines the power of a CLI with the visual support only a Graphical User Interface (GUI) can provide. The focus relies more on the overall setup and monitoring of the Lustre components than on visualization of the similarities of Lustre configurations and their effect on the file system performance. The Intel Manager For Lustre also focuses on enforcement of HPC specific setups like high-availability (HA). Additionally, the GUI enables the administrator to specify some advanced *llite* settings like `max_cached_mb`, `max_read_ahead_mb` and `max_read_ahead_whole_mb`. Without further specifications the Lustre default values are used. As the Intel Manager For Lustre is released under the MIT License [6] efforts to integrate the provided web interface could be beneficial.

3.3. Automatic Configuration within Lustre

Lustre itself provides some kind of integrated mechanisms to tune some parameters on its own. For example, when the RPC size (`max_pages_per_rpc`) or the maximal amount of RPCs in flight (`max_rpcs_in_flight`) of an OSC is increased, the value for the maximal amount of dirty data that can be queued up on the client (`max_dirty_mb`) changes accordingly. This rule is implemented in [10], although it is obvious that the value for `max_dirty_mb` can only be increased, which might not be ideal in every use case. Implementing automatic tuning for every configuration option might not beneficial, as some options are depending on the running application. In general, a transparent and predictable behavior of the applied configurations is needed.

4. Design and Implementation

In this chapter, the tools developed in this thesis are shown in detail. The first section introduces the CLI and a JSON format to define Lustre configuration changes and IOR benchmarks. The second section gives an overview about the web interface that is used to visualize Lustre configurations and benchmark results. Additional implementation specific details are provided.

4.1. CLI

The CLI is the server-side tool developed during this thesis. Its purpose is to gather the Lustre configuration and to perform IOR benchmarks. The data collected during these actions is pushed to a master server. This tool was developed with two use cases in mind. First of all, it is possible to use the CLI on a single regular Lustre client. The functionality includes gathering the current Lustre configuration and the possibility to perform IOR benchmarks - either by passing an IOR command directly to the CLI or by providing a JSON file, which automatically applies Lustre settings and performs benchmarks. For most scenarios the second approach is recommended, as Lustre components are highly distributed. In order to be able to measure a whole parallel file system, the same configuration has to be applied to several components. For this use case a batch scheduling system, e.g. SLURM, is recommended. The CLI options are shown in Listing 4.1.

```
1 $./lustrehelper_run --help
2 --help this help message
3 --loadParams [=arg(=all)] load parameters (osc|llite)
4 --config arg (=../config.json) configuration file
5 --benchmark [=arg(=ior -w -r -o filename)] benchmark command
6 --automation [=arg(=../automation.json)] automatic
    ↪ benchmarking
```

Listing 4.1: CLI options

4.1.1. Automation JSON

To test different Lustre configurations and benchmarks in a single run a JSON file has been modeled. These automation files contain actions, which get configured independently. Each action contains an array of Lustre options and IOR benchmarks to run. This model provides a convenient way to configure complex Lustre configurations and benchmark setups. For example, a setup with three different Lustre configurations and two different IOR benchmarks would require three manual configuration changes within the Lustre clients and six manual runs of the IOR benchmark. The goal of these automation files is to simplify this time consuming process. In the configuration shown in Listing 4.2 one action is defined. It contains five sets of options which are evaluated by the benchmark specified in the array in lines 36-40. In line 42, the revert option ensures that the initial configuration is restored.

```
1  {
2    "actions": [
3      {
4        "options" : [
5          {
6            "osc": {
7              "max_rpcs_in_flight": 8,
8              "max_dirty_mb": 32
9            }
10         },
11         {
12           "osc": {
13             "max_rpcs_in_flight": 32,
14             "max_dirty_mb": 128
15           }
16         },
17         {
18           "osc": {
19             "max_rpcs_in_flight": 64,
20             "max_dirty_mb": 256
21           }
22         },
23         {
24           "osc": {
25             "max_rpcs_in_flight": 128,
26             "max_dirty_mb": 512
27           }
28         },
29         {
30           "osc": {
```

```
31              "max_rpcs_in_flight": 512,
32              "max_dirty_mb": 1024
33            }
34          }
35        ],
36        "benchmarks": [
37          {
38            "cmd": "ior -r -w -o /mnt/lustre/plehn/testfile
                ↪ -a POSIX -t 1m -b 1m -s 10000 -i 3 -F -C"
39          }
40        ],
41        "post": {
42          "revert": true
43        }
44      }
45    ]
46  }
```

Listing 4.2: Automation Configuration example

4.2. Web Interface

The web interface provides a convenient way to gather insights into the Lustre deployment. It is possible to compare the Lustre configurations with each other and to visualize the benchmarks. Furthermore, based on the benchmark results, recommendations for Lustre configurations are made.

4.2.1. Diff

Whenever the CLI is executed with the `--loadParams` or `--automation` option, every participating Lustre client sends their OSC and llite configuration to a central database. In order to be able to compare and evaluate Lustre client configurations and performance metrics, a diff algorithm is used. If the content of a line has changed it is highlighted in a light red/green color. The specific context of the change is highlighted with a darker color. For example in Figure 4.1 two different OSCs have been compared. The OSC identificators are highlighted with a dark color and therefore it is obvious that those OSCs belong to different Lustre clients. Performance metrics (like `MB_per_sec`) that differ from each other are highlighted as well.

In Figure 4.2 a Lustre client is shown at two different times. The only part that has changed is the value of `max_read_ahead_mb`.

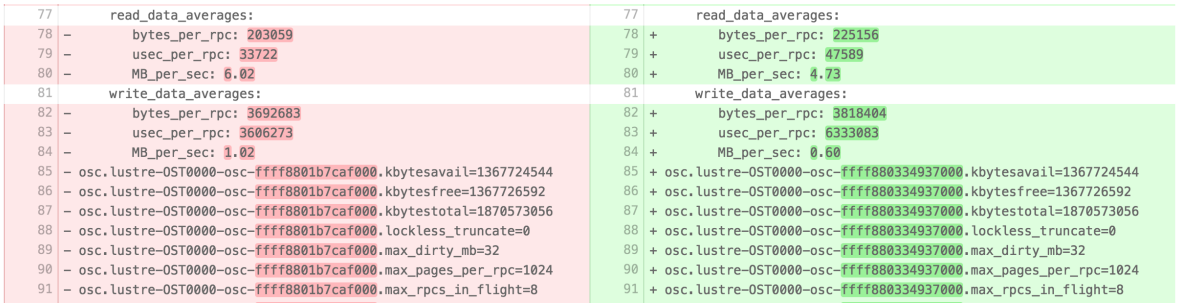

Figure 4.1.: Diff algorithm applied to two different OSCs

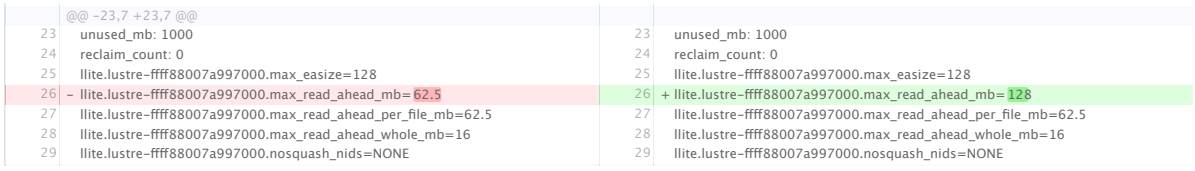

Figure 4.2.: Diff algorithm shows that the value for `max_read_ahead_mb` has been increased to 128 MB

4.2.2. Benchmark Visualization

The possibility to visualize IOR benchmarks is the main component of the web interface. After the specification of the IOR benchmark, the nodes that participated in the benchmark and an optional limitation on the time period, the visualization button is enabled. Additionally, a drop down menu is filled with Lustre configurations that have been tested with the criteria specified before. Upon visualization two charts are generated.

The purpose of the first one is to visualize the I/O performance and the standard deviation. The read and write values are mean values provided by the IOR JSON output shown in Listing 2.4. For easier comparison, the mean value of these read and write values is calculated. Therefore a single value can indicate whether the I/O performance increases or decreases. Additionally, each data point visualizes the current Lustre client configuration. Configuration changes that were made specific for this benchmark run are highlighted separately. An example is given in Figure 4.3 where several values for `max_read_ahead_per_file_mb` are benchmarked.

The purpose of the second chart is to compare the I/O performance of an individual Lustre configuration with a benchmark run performed on an unchanged configuration. This so called baseline is an indicator whether the tested configuration increases or decreases the file systems performance. An example is given in Figure 4.4 where compared to the current default configuration the performance decreases.

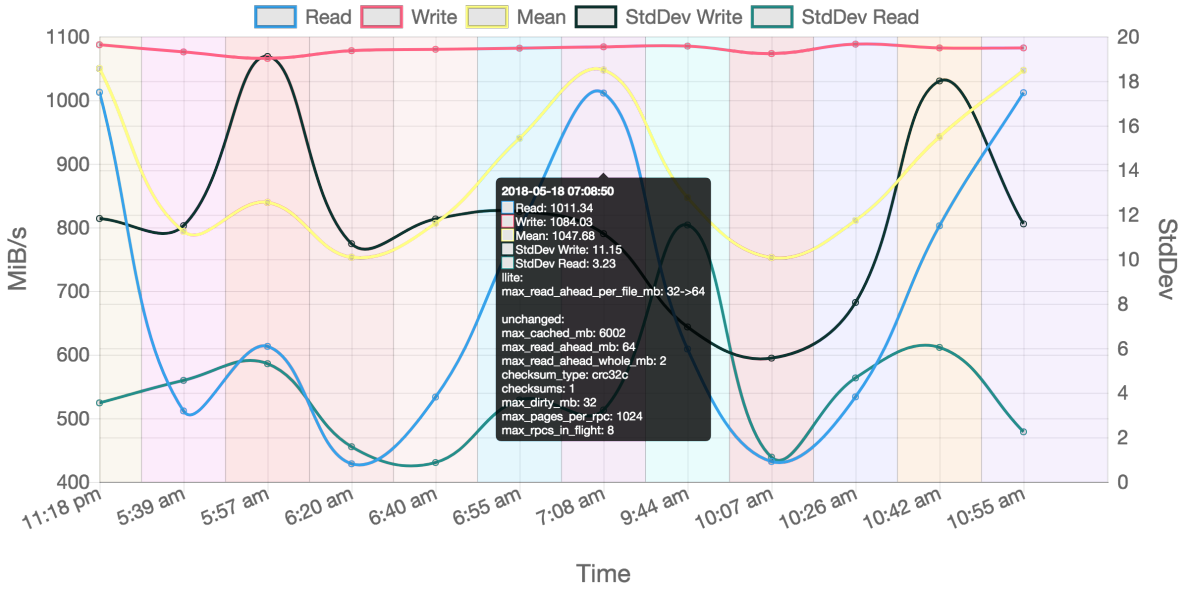

Figure 4.3.: Chart highlighting the effect of an increased value for `max_read_ahead_per_file_mb`

4.2.3. Recommendations

While it is relatively easy to spot client configurations that promise a performance improvement, an approach to automate this process has been made. The following steps are performed.

1. The mean value of all selected benchmark results is calculated
2. For every client configuration the biggest mean value is calculated
3. If the individual mean value is at least as big as the mean value of all benchmark results, a recommendation is made

In the end, a new automation JSON is generated, hopefully containing promising client configurations.

4.2.4. MongoDB

MongoDB is used as the database engine to store the Lustre configurations and the IOR benchmark results. MongoDB is a NoSQL database and the most used engine of its kind in 2017 [11]. The main difference between a NoSQL and a relative database engine like MySQL is the support for schema free data. MongoDB stores data in JSON-like documents and is therefore well suited for archiving and analyzing the automation JSON files shown in subsection 4.1.1 and the IOR results shown in Listing 2.4

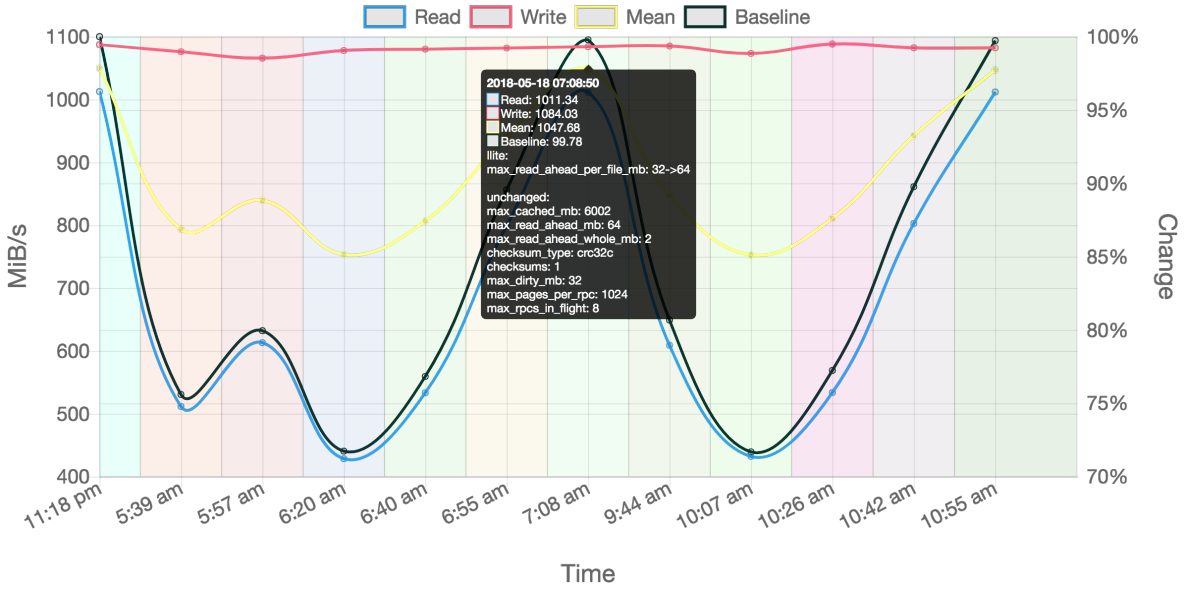

Figure 4.4.: Chart highlighting the effect of an increased value for `max_read_ahead_per_file_mb`

4.2.5. Docker

Docker is a tool to containerize applications. This allows those applications to run in a virtual environment with its own dependencies. It is recommended to use one Docker container for one purpose. Therefore a multi-container environment is provided. To simplify the start-up, networking and scaling of those independent containers, Docker Compose is used. A Docker Compose environment is specified in a YAML file. To install the web interface developed in this thesis a demo is given in Appendix B.

5. Evaluation

In this chapter, the possibilities to configure a Lustre client are shown. Furthermore, IOR is used to benchmark configuration changes and the results are put in perspective by visualizing them in the web interface.

5.1. Test setup

The following evaluation was performed on hardware provided by the workgroup of Scientific Computing at Universität Hamburg. Each server is equipped with 12 GB of memory, an Intel Xeon X5650 CPU and two 1 Gbit/s Ethernet interfaces. Ubuntu 16.04.4 LTS is used as the operating system and Lustre version 2.10.1 is installed. In the following tests 10 nodes are participating. As only one Ethernet interface per server is used for interconnection, the aggregated bandwidth is limited to $10 * 1$ Gbit/s resulting in an theoretical maximum bandwidth of 1280 MiB/s. When evaluating the bandwidth, one needs to consider the overhead introduced by other running services and protocol overheads. If not stated otherwise the stripe count is set to -1, which means that the file is striped across all available OSTs.

5.2. Automation configuration

In section 2.1.3, important Lustre client configurations were highlighted. In the following evaluation specific ranges of values are tested for each configuration option. Additionally, some configurations were tested at the same time as they influence each other. In Table 5.1 all tested client configurations are shown. The second column represents the current client configuration. Altogether nine different Lustre settings with 37 configuration changes in total are tested. Therefore to benchmark the effects of these changes in combination with just one IOR benchmark results in 37 independent runs.

As some configurations have direct influence on each other, groups have been tested. In Table 5.2 three groups are shown. This adds additional 16 benchmark runs for a single IOR configuration.

Configuration	Present value	Tested values
max_rpcs_in_flight (osc)	8	8, 32, 64, 128, 256
max_dirty_mb (osc)	32	32, 128, 256, 512, 1024
max_pages_per_rpc (osc)	1024	256, 1024
checksums (osc)	1	0, 1
checksum_type (osc)	crc32c	crc32, crc32c, adler
max_cached_mb (llite)	6002	3000, 6000, 9000
max_read_ahead_mb (llite)	64	0, 40, 64, 128, 512, 1024
max_read_ahead_whole_mb (llite)	2	2, 4, 8, 16, 32
max_read_ahead_per_file_mb (llite)	64	2, 4, 8, 16, 32, 64

Table 5.1.: Tested Lustre client configurations

Configurations	Tested values					
Group 1						
max_rpcs_in_flight (osc)	8	32	64	128	256	
max_dirty_mb (osc)	32	128	256	512	1024	
Group 2						
max_read_ahead_whole_mb (llite)	2	4	8	16	32	
max_read_ahead_per_file_mb (llite)	4	8	16	32	64	
max_read_ahead_mb (llite)	40	64	128	512	1024	
Group 3						
max_rpcs_in_flight (osc)	8	32	64	128	256	256
max_dirty_mb (osc)	32	128	256	512	1024	2047
max_pages_per_rpc (osc)	256	256	512	1024	1024	1024

Table 5.2.: Tested Lustre client configuration groups

5.3. Standard tests

The Lustre configurations shown in Table 5.1 and Table 5.2 have been benchmarked with the following three different IOR configurations. In all three cases the present, unchanged configuration was tested beforehand.

5.3.1. IOR benchmark with files-per-process and reordered read back configuration

```
IOR -r -w -o /mnt/lustre/testfile -a POSIX -e -t 1m -b 1m
   ↪ -s 14000 -i 3 -F -C
```

Listing 5.1: IOR benchmark with files-per-process and reordered read back configuration

This benchmark reads and writes 14000 times to a block with the size of 1 MiB. As the `transferSize` is set to 1 MiB the total number of read/write operations is 14000. One thing all benchmarks used have in common is that the file size per process is set to 14000 MiB. This exceeds the available memory on the clients with the intention of eliminating any unwanted cache effects. By using the POSIX interface in combination with the `-e` option fsync is triggered for every write close. Additionally the `-C` option is used to make sure that the processes do not read back their own files.

In Figure 5.1 this IOR configuration achieved a read and write performance of 1012 MiB and 1087 MiB respectively. Especially the low value of the standard deviation indicates a steady file system performance.

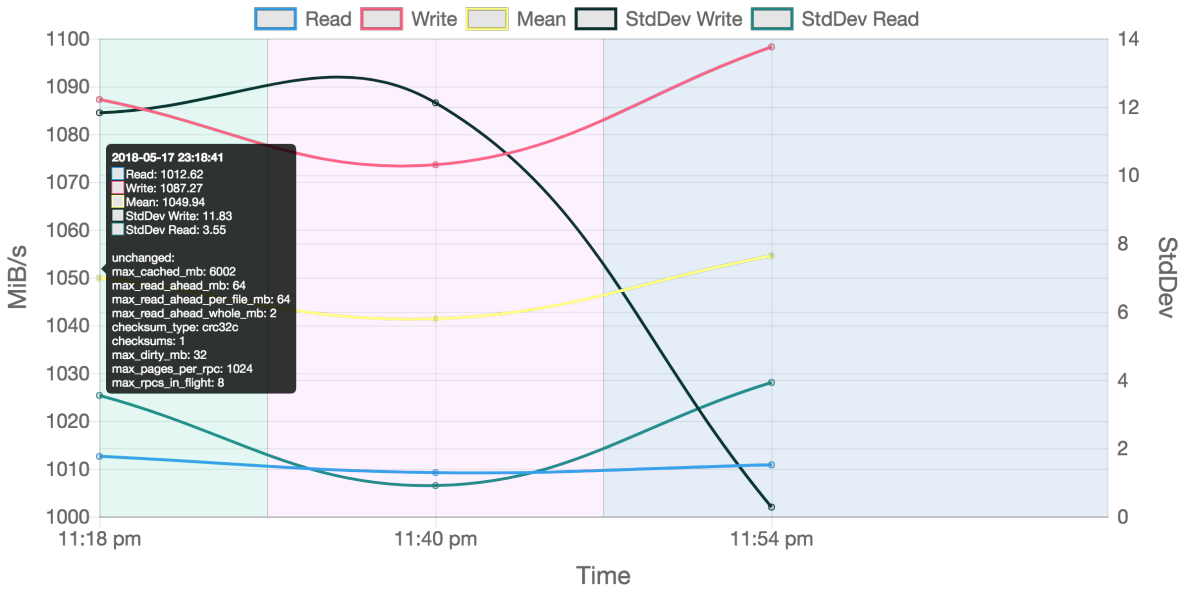

Figure 5.1.: IOR benchmark no. 1 without changes to the Lustre client

Configuration testing

RPCs

In Figure 5.2 three benchmarks are highlighted. The chart shown visualizes the performance difference between the current Lustre configuration and the tested configuration. The first highlighted data point is the only one that increases the performance slightly. However, as previously stated, fluctuation of 1-2% is expected in benchmarking. The second and third highlighted data points are of greater interest. Both data points share nearly identical performance metrics. In both cases the value for `max_dirty_mb` was enforced to 32 MiB. However, when tuning the values for `max_rpcs_in_flight` and

`max_pages_per_rpc` one has to keep in mind, that the RPCs are filled by the memory assigned to `max_dirty_mb`. In both cases these configurations are limited by the same configuration. Since Lustre 2.6 a mechanism is implemented that automatically increases the value for `max_dirty_mb` [22]. A potential issue could be that the current implementation never decreases the value for `max_dirty_mb` [see line 1293 21].

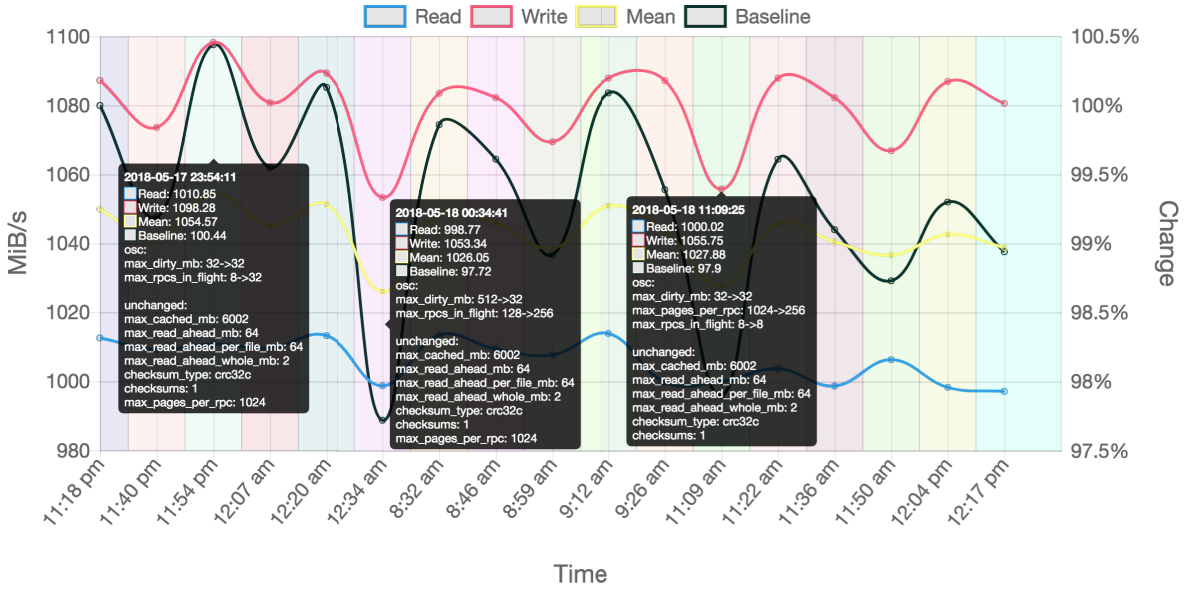

Figure 5.2.: Impact of `max_rpcs_in_flight`, `max_dirty_mb` and `max_pages_per_rpc` on each other

Checksums

In Figure 5.3 the impact of activated and deactivated checksums is shown. While the I/O performance basically stays the same, the value of the standard deviation changes. In theory, the reason for a doubled write standard deviation could be that data chunks are corrupted. However, it is more likely that the missing checksum header reduces the size of the RPCs requests. Therefore the RPCs are handled faster which could have an affect on the dirty pages.

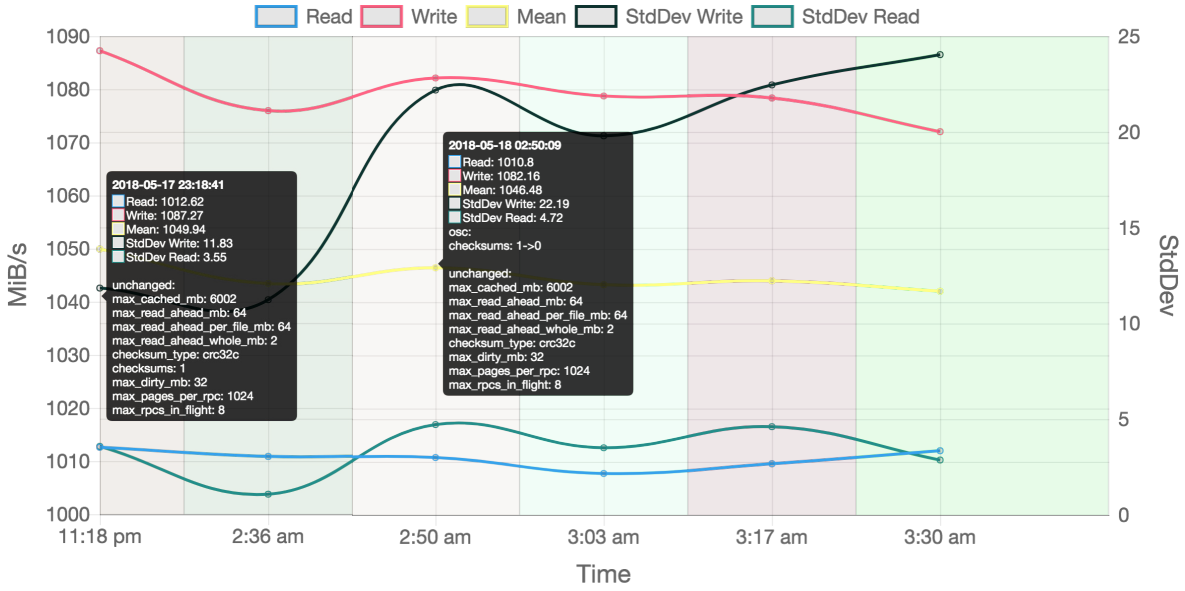

Figure 5.3.: Impact of activated and deactivated checksums

5.3.2. IOR benchmark with files-per-process, reordered read back configuration and continuous block size

```
IOR -r -w -o /mnt/lustre/testfile -a POSIX -e -t 1m -b
   ↪ 14000m -s 1 -i 3 -F -C
```

Listing 5.2: IOR benchmark with files-per-process, reordered read back configuration and continuous block size

This benchmark is similar to the previous one. However, this time the block size is set to 14000 MiB within a single segment.

As shown in Figure 5.4 this IOR benchmark achieved nearly identical I/O performance as the previous one. The main difference is, that the standard deviation for the read and write performance is very low in both cases. When comparing the further course of the line chart one assumption is, that the I/O performance is steadier than in the previous benchmark.

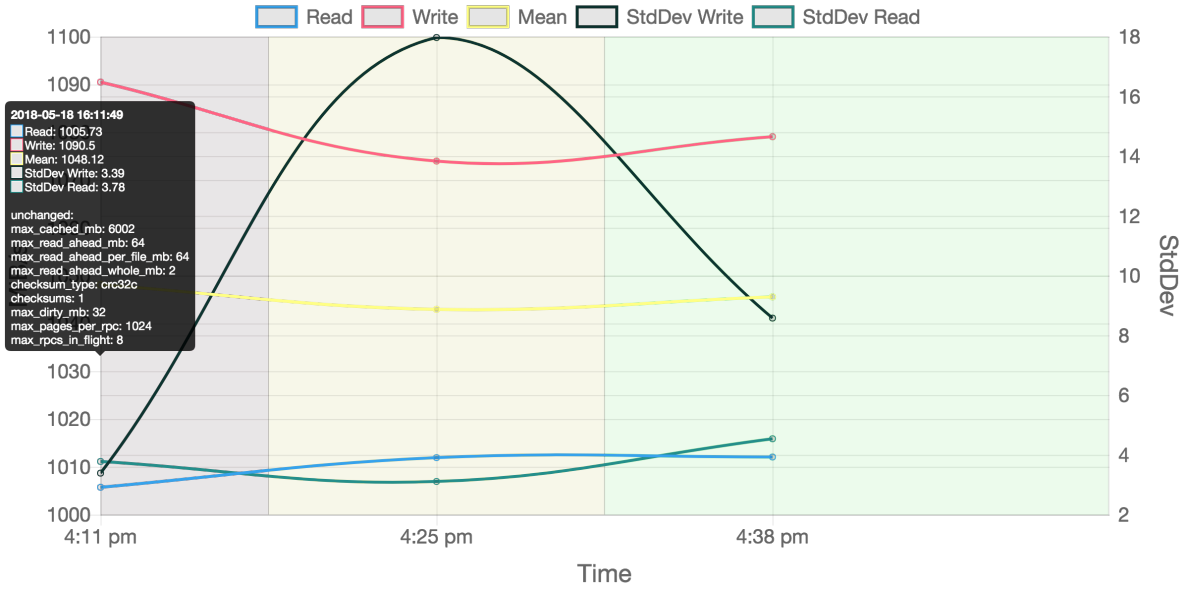

Figure 5.4.: IOR benchmark no. 2 without changes to the Lustre client

Configuration testing

Checksums

In Figure 5.5 the same effect is seen as in Figure 5.3. In this case, the impact of disabled checksums is even more obvious. The standard deviation rises from 22.10 to 33.78. As the same behavior is observable for a second time it is unlikely that this is a coincidence.

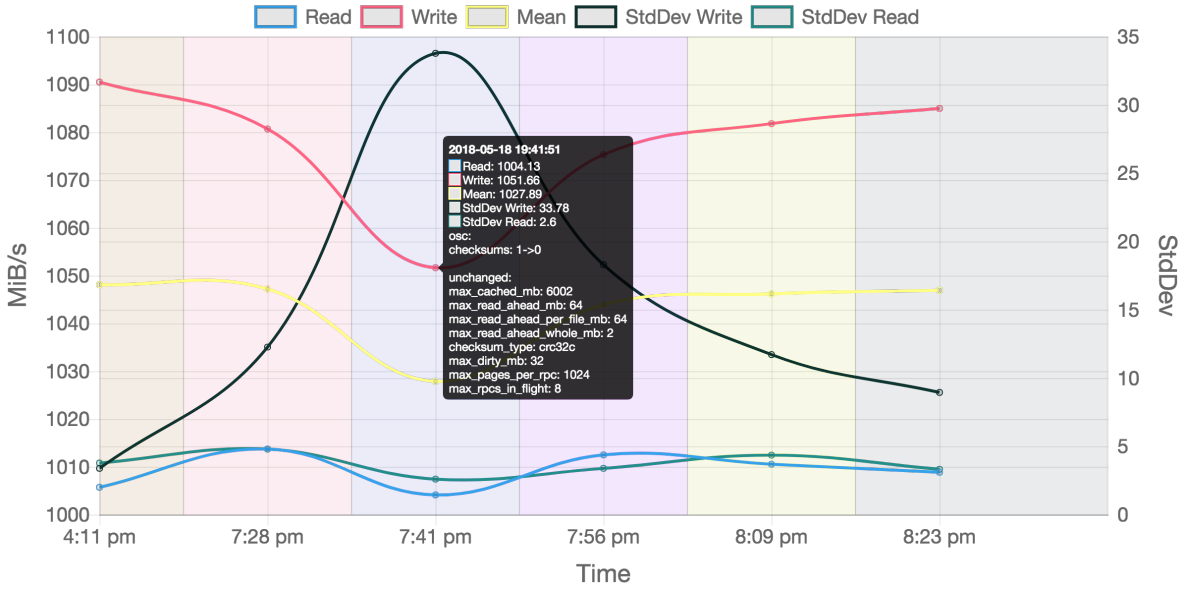

Figure 5.5.: Impact of activated and deactivated checksums

Read ahead

Figure 5.6 acts as a good example for poorly tuned parameters. It shows how essential the read ahead mechanism is for good file system performance. If the value for `max_read_ahead_per_file_mb` is cut in half, the performance drops by over 10%. On the other hand, it was not possible to increase the Lustre client performance. As shown in the second highlighted data point, increasing the value for `max_read_ahead_mb` to 1024 MiB is not helping either. One could assume that because every Lustre client is reading and writing 14000 MiB in a continuous block the increased read ahead cache should improve the performance.

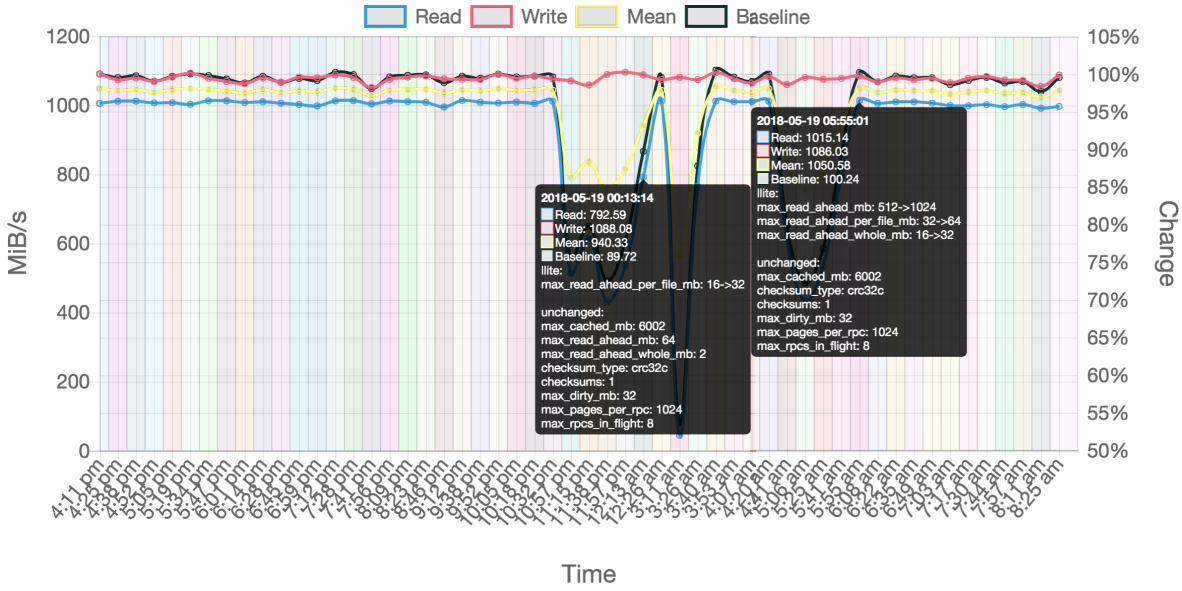

Figure 5.6.: Word case scenario for read ahead tuning

5.3.3. IOR benchmark with random access configuration

```
IOR -r -w -o /mnt/lustre/testfile -a POSIX -e -t 1m -b 1m
   ↪ -s 14000 -i 3 -z
```

Listing 5.3: IOR benchmark with random access configuration

This benchmark is using 14000 segments and 1 MiB for the the `transferSize` like the first IOR benchmark does. However, file access is set to random due to the random offset option (`-z`).

The benchmark results shown in Figure 5.7 are very different to the ones achieved before. Compared to the mean value of Figure 5.4 the I/O performance was reduced by nearly 60%. At the same time the read performance has overtaken the write performance.

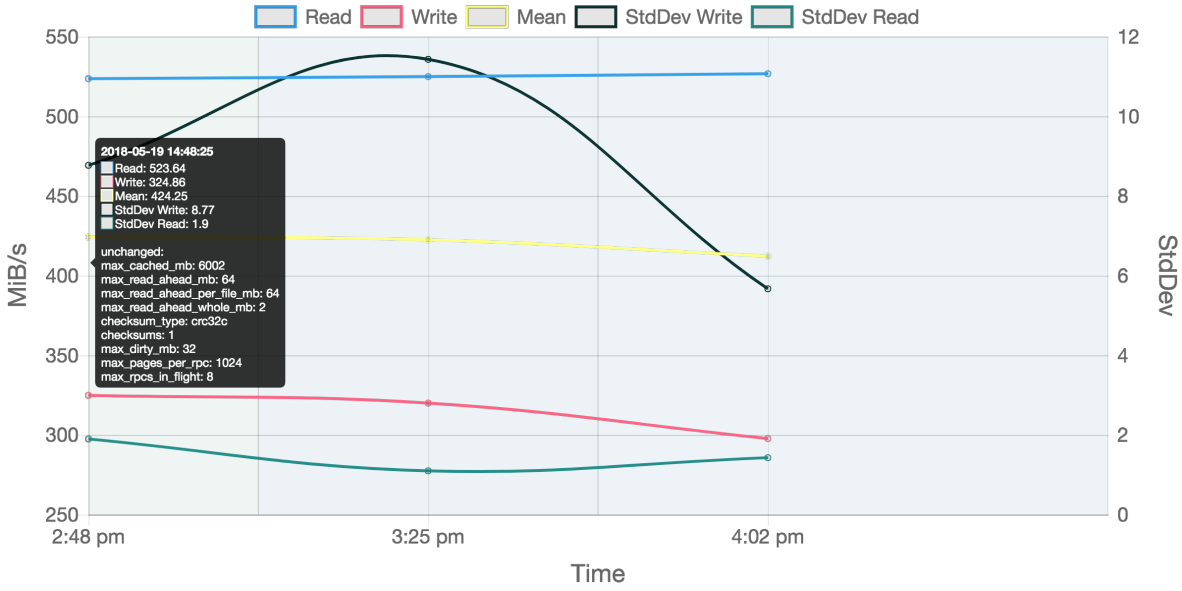

Figure 5.7.: IOR benchmark no. 3 without changes to the Lustre client

Configuration testing

Read ahead

As shown in Figure 5.8, when the read ahead cache is deactivated by setting `max_read_ahead_mb` to 0, the read performance drops to 44.89 MiB/s. This is interesting as the read access is set to random. The read ahead cache is only used if two sequential requests can not be fulfilled by the cache. One assumption is that even when reading random parts of a file, two sequential requests are made frequently. However, if this would be the case, the read performance should improve as the value for `max_read_ahead_mb` is increased. Thereby, a more realistic assumption is that Lustre fills the read ahead cache with data, even if no sequential requests have been made.

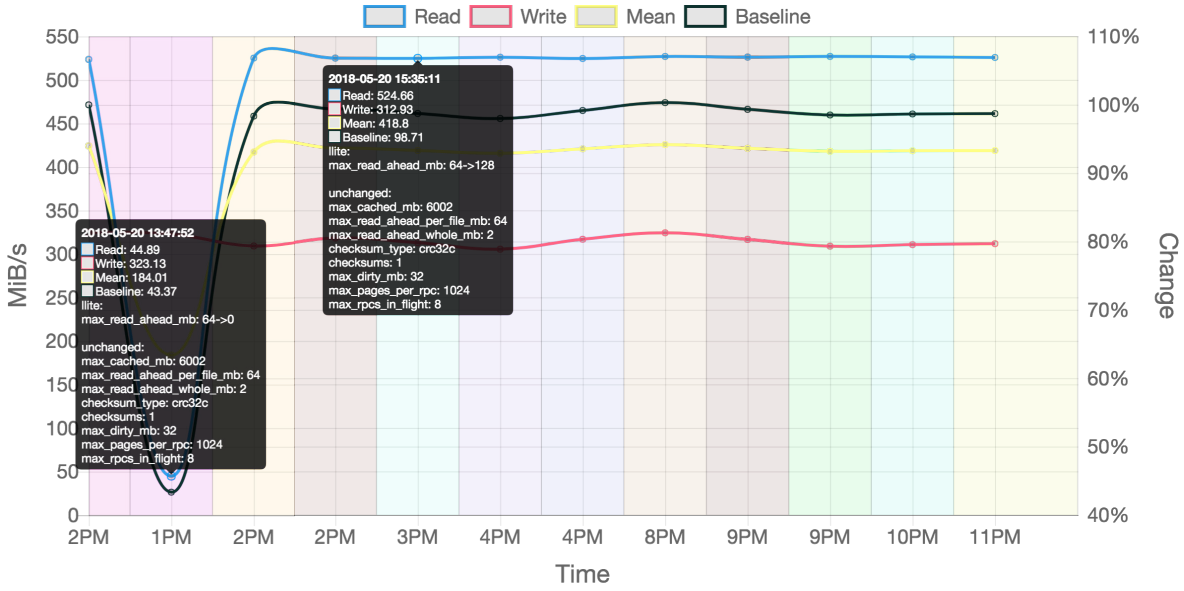

Figure 5.8.: Read ahead with random access

RPCs

In Figure 5.9 an interesting discovery has been made. Whenever `max_pages_per_rpc` is set to 256 the read performance overtakes the write performance. In those cases, the overall file system performance is improved by 30%. This behavior is repetitive and the performance improvements are constant.

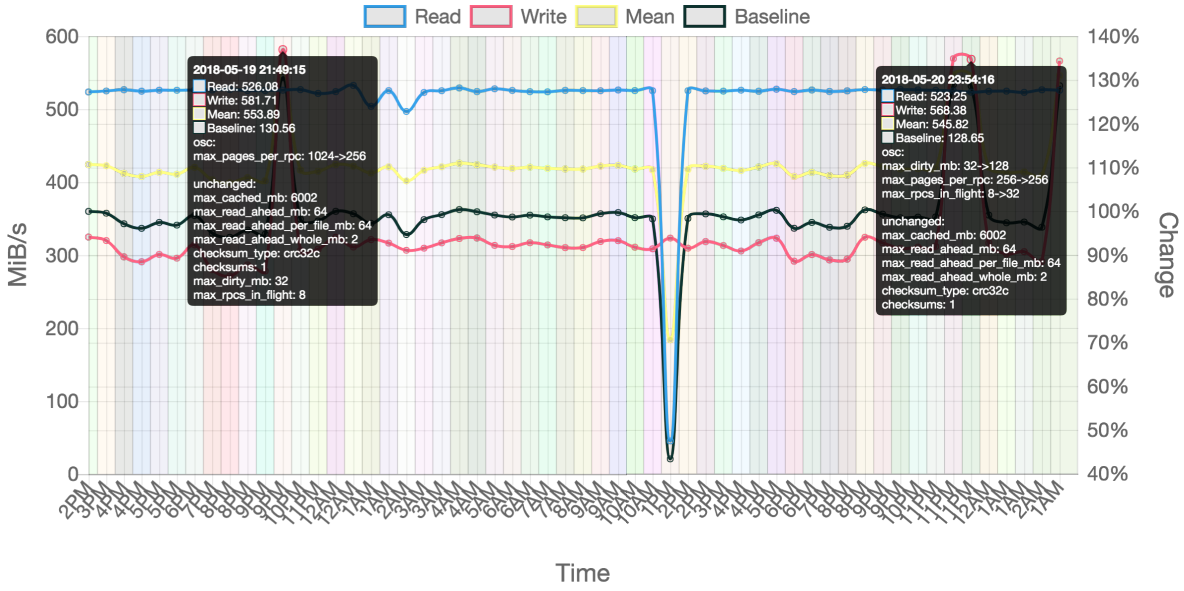

Figure 5.9.: Pages per RPC with random access

5.4. Stripe size tests

5.4.1. IOR benchmark with 4 MiB transfer size

```
IOR -r -w -o /mnt/lustre/testfile -a POSIX -e -t 4m -b 40m
   ↪ -s 350 -i 3 -F
```

Listing 5.4: IOR benchmark with 4 MiB transfer size and 40 MiB block size

This benchmark uses an increased transfer size of 4 MiB. The default Lustre stripe size is 1 MiB. Therefore a single chunk needs to get written to four OSTs. This adds a massive overhead as for every transfer four RPCs are required.

In Figure 5.10 the impact of this access pattern is shown. While the write performance is unchanged, the read performance drops to 244, 18 MiB/s. The second data point clarifies the importance of the RPC configuration. As soon as the value of `max_rpcs_in_flight` is set to at least 64 the read performance increases. In the third highlighted data point the I/O performance increases by more than 9%.

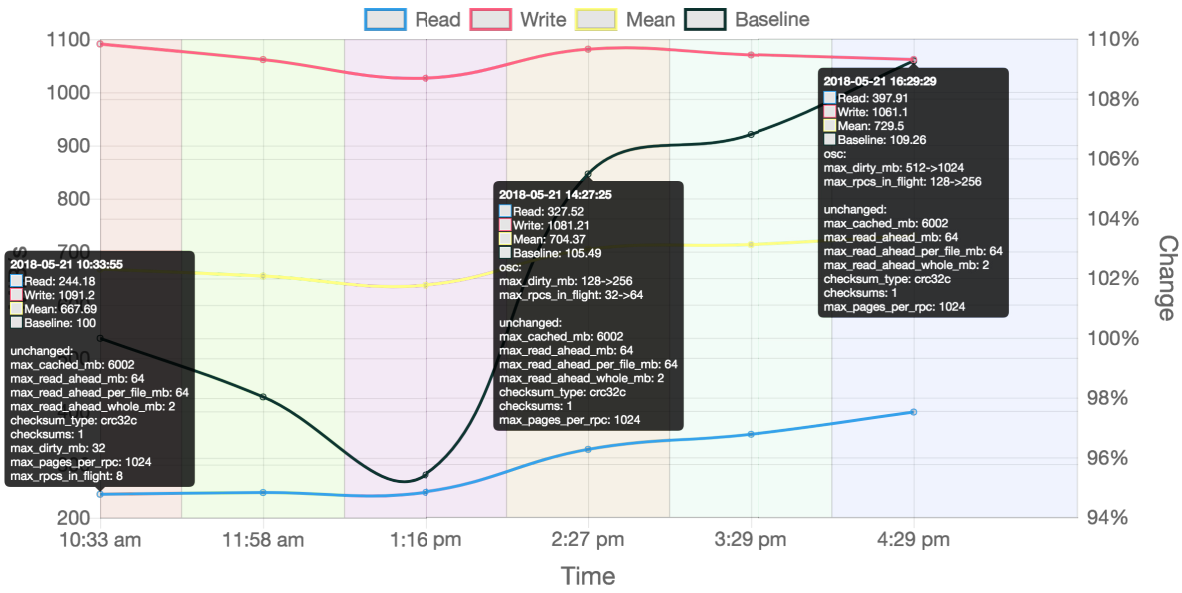

Figure 5.10.: Impact of additional RPCs

5.4.2. IOR benchmark with 4 MiB transfer size and 4 MiB stripe size

```
IOR -r -w -o /mnt/lustre/testfile -a POSIX -e -t 4m -b 40m
  ↪ -s 350 -i 3 -F -O lustreStripeSize=4m
```

Listing 5.5: IOR benchmark with 4 MiB transfer size, 40 MiB block size and 4 MiB stripe size

This benchmark is similar to the previous one, but this time the stripe size matches the transfer size. Thereby the client has to send ten RPCs to transmit a single continuous block.

The observed behavior in Figure 5.11 matches the previous one. As soon as `max_rpcs_in_flight` is set to 64 or higher, the read performance increases. This time, however, less RPCs are necessary to transmit a single block. The overall file system performance increases by 17.88%.

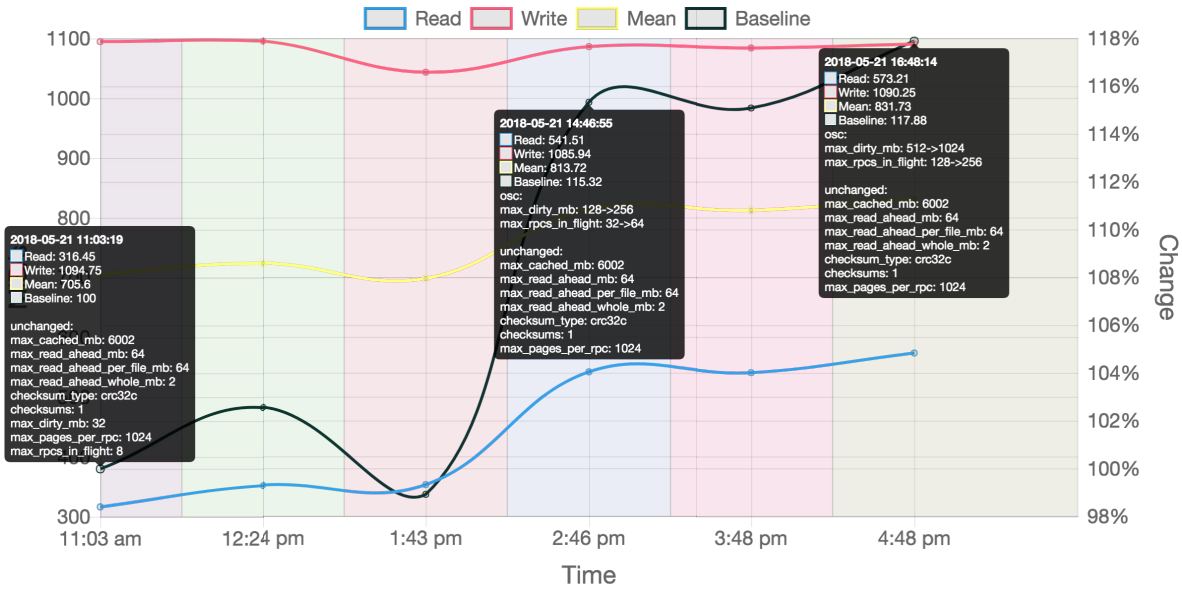

Figure 5.11.: Impact of additional RPCs

6. Conclusion

In this chapter, the overall experience using the decision support tool is summarized and evaluated.

The goal of this thesis was to develop a decision support system that helps the user to gain a better understanding of the behavior of the Lustre file system. This goal has been fulfilled, as the tools developed in this thesis provided several interesting insights. The automation files greatly improved the process of handling complex configuration changes. Once an automation file is prepared, the testing of different IOR benchmarks is a straight forward process. During evaluation 188 benchmarks have been run. Not every client configuration tested is meaningful in regards to performance optimization. However, this shows how convenient it is to be able to quickly experiment with configuration changes and benchmarks.

The approach shown in Section 4.2.3, where based on mean values another automation JSON is generated, has been quickly dismissed. In most cases the benchmark results were not meaningful enough to just use the mean value for a recommendation. However, different algorithms which pay attention to the standard deviation might have more success.

Altogether it was not possible to improve the I/O performance. That being said, many interesting trends were observed. Especially the usage for educational purposes seems promising.

7. Future Work

In this chapter an outlook for future work is given. The proposed features address ideas to configure and benchmark additional Lustre layers and components. Furthermore, integration into other HPC systems and existing interfaces is desired.

Integration of additional layers
Currently the llite and OSC layers are integrated into the decision support system. However, as shown in Figure 2.2, the Metadata Client (MDC) layer is missing in the current implementation. The MDC is responsible for the communication with the MDSs. It is the counterpart to the LOV layer as the MDC provides a single namespace for meta data like the LOV layer does for data. To benchmark the meta data performance, a benchmark like MDS-Survey [15] can be used and integrated into the decision support system.

Integration of additional Lustre components
The current implementation is only capable of configuring the Lustre clients. However, as this is a distributed system, many components depend on each other. For example, to fully benefit from an increased RPC size, the OSTs and OSCs should use the same configuration. Therefore, in the future, the decision support tool should be aware of all involved Lustre components.

Integration into existing management applications
As seen in Chapter 3, the Intel Manager For Lustre is able to manage production-grade Lustre installations. Due to an open source license, an integration in an already powerful management system is possible.

Data mining
The usage of the data gathered in the database is not limited to a simple recommendations mechanism as shown earlier. This data might also be suitable for data mining and further analyzes.

Bibliography

[1] Adler, Mark ; Gailly, Jean loup: ZLIB DATA COMPRESSION LIBRARY. (2017). – URL https://github.com/madler/zlib/blob/master/README#L87

[2] Chris Horn, Cray Inc.: LNet and LND Tuning Explained. (2015). – URL https://www.eofs.eu/_media/events/lad15/15_chris_horn_lad_2015_lnet.pdf

[3] Community, Lustre: Example: IOR Read/Write Test, Multiple Files per Process, Multiple Clients. (2018). – URL http://wiki.lustre.org/IOR#Example:_IOR_Read.2FWrite_Test.2C_Multiple_Files_per_Process.2C_Multiple_Clients

[4] Coweb, Malcolm: An introduction to Lustre architecture. (2015), 8. – URL https://nci.org.au/wp-content/uploads/2015/08/01-Introduction-to-Lustre-Architecture.pdf

[5] Intel: Intel Manager For Lustre. (2018). – URL https://github.com/intel-hpdd/intel-manager-for-lustre

[6] Intel: Intel Manager For Lustre License. (2018). – URL https://github.com/intel-hpdd/intel-manager-for-lustre/blob/master/LICENSE

[7] Intel: Intel® Solid-State Drive Data Center Family for PCIe. (2018). – URL https://www.intel.de/content/www/de/de/solid-state-drives/intel-ssd-dc-family-for-pcie.html

[8] Intel: Introducing Intel® Manager for Lustre* software. (2018). – URL https://intel-hpdd.github.io/Online-Help/docs/Introduction_1_0.html

[9] Intel, Oracle: Lustre* Software Release 2.x - Operations Manual. (2018). – URL http://doc.lustre.org/lustre_manual.pdf

[10] Intel, Oracle: OBD Code. (2018). – URL https://git.hpdd.intel.com/?p=fs/lustre-release.git;a=blob;f=lustre/include/obd.h;h=8098418bc237be48f83adf4560333db690f519b4;hb=HEAD#l1246

[11] IT gmbh solid: DB-Engines Ranking of Document Stores. (2018). – URL https://db-engines.com/en/ranking/document+store

[12] Kogge, Peter ; Bergman, Keren ; Borkar, Shekhar ; Campbell, Dan u. a.: ExaScale Computing Study: Technology Challenges in Achieving Exascale Systems. (2008), 9. – URL http://staff.kfupm.edu.sa/ics/ahkhan/Resources/Articles/ExaScale%20Computing/TR-2008-13.pdf

[13] Kuhn, Michael ; Kunkel, Julian ; Ludwig, Thomas: Data Compression for Climate Data. In: *Supercomputing Frontiers and Innovations* (2016), 06, S. 75–94. – URL http://superfri.org/superfri/article/view/101

[14] Lockwood, Glenn K.: Basics of I/O Benchmarking. (2016). – URL https://glennklockwood.blogspot.de/2016/07/basics-of-io-benchmarking.html

[15] Lustre: MDS-Survey Readme. . – URL https://git.hpdd.intel.com/?p=fs/lustre-release.git;a=blob;f=lustre-iokit/mds-survey/README.mds-survey;h=9f59ce25d53cc75085c7b10328378731650c5f42;hb=bdc5bb52c55470cf8020933f80e327c397810603

[16] Lustre: An introduction to Lustre architecture. (2017), 10. – URL http://wiki.lustre.org/images/6/64/LustreArchitecture-v4.pdf

[17] OpenSFS: Lustre® File System, Version 2.4 Released. (2013), 7. – URL http://opensfs.org/press-releases/lustre-file-system-version-2-4-released/

[18] Petersen, Torben K. ; Xyratex: Optimizing Performance of HPC Storage Systems. (2013). – URL https://chpcconf.co.za/files/2013/AFRICA/5%20Dec%202013/torben-OptimizingStorage.pptx

[19] Rutman, Nathan: Improvements in Lustre Data Integrity. (2012), 4. – URL http://cdn.opensfs.org/wp-content/uploads/2011/11/Improvements-in-Lustre-Data-Integrity.pdf

[20] Wang, Feiyi ; Oral, Sarp ; Shipman, Galen u. a.: Understanding Lustre Filesystem Internals. (2009), 4. – URL http://wiki.old.lustre.org/images/d/da/Understanding_Lustre_Filesystem_Internals.pdf

[21] Zhang, Hongchao ; Drokin, Oleg u. a.: LU-4933 changelog. (2014), 5. – URL https://review.whamcloud.com/#/c/10446/7/lustre/include/obd.h

[22] Zhang, Hongchao ; Xi, Li: Automatically tune the max_dirty_mb. (2014), 7. – URL https://jira.hpdd.intel.com/browse/LU-4933

[23] Zickler, Enno: *Option to output results in a json file.* 2018. – URL https://github.com/hpc/ior/pull/33

Appendices

A. Abbreviations

Abbreviations

HPC high-performance computing
MGT Management Target ... 8
MGS Management Server ... 8
MDS Metadata Server ... 8
MDT Metadata Target ... 9
MDC Metadata Client ... 41
OSS Object Storage Server ... 9
OST Object Storage Target ... 9
OSC Object Storage Client ... 10
VFS Virtual File System ... 10
RPC Remote Procedure Call ... 10
CLI Command Line Interface ... 2
GUI Graphical User Interface ... 20
HA high-availability ... 20
FLOPS Floating Point Operations Per Second ... 5
MPI Message Passing Interface ... 5
LNet Lustre Network ... 13
CRC cyclic redundancy check ... 13
ECC error-correcting code ... 13
DNE Distributed Namespace Environment ... 9
LOV logical object volume ... 11

B. Demo

Web interface demo

This section gives an introduction on how to install the web interface. By default this installation contains the Lustre client configurations and benchmark results that were used in the evaluation. More details, especially regarding the installation and usage of the CLI, can be found on this website: **https://git.wr.informatik.uni-hamburg.de/julius.plehn/LustreWeb**

```
git clone
   ↪ https://git.wr.informatik.uni-hamburg.de/julius.plehn/LustreWeb
cd LustreWeb && docker-compose up
```

Listing B.1: Demo installation

After the installation has been completed successfully, the web interface is available: **http://localhost:1026/**

List of Figures

1.1. Development of computational speed, storage capacity and storage speed [based on 13, p. 75] 6

2.1. Lustre file system overview [4, p. 2] 10
2.2. Lustre client layers [20, p.11] 12
2.3. Bulk Checksum Speeds, MB/s [19] 16
2.4. IOR file access [inspired by 14] 19

4.1. Diff algorithm applied to two different OSCs 25
4.2. Diff algorithm shows that the value for `max_read_ahead_mb` has been increased to 128 MB 25
4.3. Chart highlighting the effect of an increased value for `max_read_ahead_per_file_mb` 26
4.4. Chart highlighting the effect of an increased value for `max_read_ahead_per_file_mb` 27

5.1. IOR benchmark no. 1 without changes to the Lustre client 30
5.2. Impact of `max_rpcs_in_flight`, `max_dirty_mb` and `max_pages_per_rpc` on each other 31
5.3. Impact of activated and deactivated checksums 32
5.4. IOR benchmark no. 2 without changes to the Lustre client 33
5.5. Impact of activated and deactivated checksums 34
5.6. Word case scenario for read ahead tuning 35
5.7. IOR benchmark no. 3 without changes to the Lustre client 36
5.8. Read ahead with random access 37
5.9. Pages per RPC with random access 37
5.10. Impact of additional RPCs 38
5.11. Impact of additional RPCs 39

List of Listings

2.1. Lustre Mounting 9
2.2. lctl get_param and set_param 12
2.3. CRC example 15
2.4. Truncated JSON output 18

4.1. CLI options 22
4.2. Automation Configuration example 23

5.1. IOR benchmark with files-per-process and reordered read back configuration 29
5.2. IOR benchmark with files-per-process, reordered read back configuration and continuous block size 32
5.3. IOR benchmark with random access configuration 35
5.4. IOR benchmark with 4 MiB transfer size and 40 MiB block size 38
5.5. IOR benchmark with 4 MiB transfer size, 40 MiB block size and 4 MiB stripe size 39

B.1. Demo installation 46

List of Tables

2.1. LNet Credits [2, p. 14] 13
2.2. Adler-32 example 14
2.3. Important IOR Settings 17

5.1. Tested Lustre client configurations 29
5.2. Tested Lustre client configuration groups 29

Eidesstattliche Versicherung

Hiermit versichere ich an Eides statt, dass ich die vorliegende Arbeit im Studiengang Software-System-Entwicklung selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel – insbesondere keine im Quellenverzeichnis nicht benannten Internet-Quellen – benutzt habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen wurden, sind als solche kenntlich gemacht. Ich versichere weiterhin, dass ich die Arbeit vorher nicht in einem anderen Prüfungsverfahren eingereicht habe und die eingereichte schriftliche Fassung der auf dem elektronischen Speichermedium entspricht.

Ort, Datum	Unterschrift

Veröffentlichung

Ich bin damit einverstanden, dass meine Arbeit in den Bestand der Bibliothek des Fachbereichs Informatik eingestellt wird.

Ort, Datum	Unterschrift