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

In-situ Transformation for Input/Output Access Patterns by Applying Building Blocks of Optimization Schemas

Autor: Daniel Schmidtke
E-Mail Address: 0schmidt@informatik.uni-hamburg.de
MatrikelNr: 6250282
Studiengang: M.Sc. Informatik

Erstgutachter: Dr. Julian Kunkel
Zweitgutachter: Prof. Dr. Thomas Ludwig

Betreuer: Dr. Julian Kunkel

Hamburg, den April 19, 2017

Abstract

This thesis is about the finding of optimization strategies, that can be applied by in-situ transformation of input/output access patterns and the classification of these strategies. The found optimizations are then being implemented in SIOX and FUSE and evaluated with different benchmarks.
The optimization strategies found in this thesis are a demonstration of the possibilities that can be achieved using in-situ transformation.

Contents

1 Introduction

This section gives an introduction to the topic of the thesis. Section 1.1 is a motivation to the relevance of this topic. In the next section I discuss the goals of the thesis and the last section gives an overview of the structure of the thesis.

1.1 Motivation

In the field of high-performance computing (HPC) is a constant development, to reach even higher performance levels. In the beginning of the development, HPC had homogeneous systems, with a small number of components. As the development proceeded, the systems got more complex and with every leap new specialized parts were needed to push the performance.
Today there is a variety of special components for a HPC-system. In the Top500 list of the best Supercomputers in the world, all systems are build with a cluster architecture and not as one monolithic computer.
The HPC-system is usually made from two clusters, a compute cluster and a storage cluster.
The compute cluster contains compute nodes, which do the computation and communicate over a high bandwidth network. The compute nodes usually have no or a very small persistent memory. Because of that, the HPC-system has the storage cluster attached to it. The storage nodes are build to deliver a persistent storage for the Compute-Cluster.
Compute and storage clusters are connected over a separate high performance network to not interfere with the compute node to compute node communication while saving data to the disks.
Because of this complexity, a HPC-system has many components that can fail during a run of a program. This leads to the question how to still run a program on such a complex system.
In modern systems, the way to go is using check pointing to handle execution errors. This method has the advantage, that the program can be restarted from the last checkpoint in an event of a system failure and the user can inspect the results of the program using those checkpoints.
While checkpoints are dumping the system memory to storage in a sequential way, other write operations may use a random pattern. This kind of access is very slow on the usually utilized HDDs. An increase in computational time is the result of those two types of access to storage, when they are not optimized.
The development of persistent storage is a lot slower than development of computational power in form of CPUs. Therefore the checkpoint writing can lead to a very large increase in computation time while the computation nodes are idling and only writing

to storage.
To get a better understanding of how programs write to storage, there are a lot of tracing tools like Vampire, Darshan or SIOX. Those tools can inspect the run and give an insight look, but they can't modify and optimize. This needs to be done by the scientist, which is a challenging task and may take a long time for a small time reduction in the runtime.
The Scalable I/O For Extreme Performance Project (SIOX) was developed to generate traces and analyze program runs. Today SIOX is capable of optimize programs in situ. The project is a modular piece of software which can be equipped with plug-ins that can inspect and modify calls from the program on its way to the kernel. All calls are logged and with a replay plug-in, SIOX is capable of writing to and reading from disk without th actual program running, only using the trace files.
FUSE a file system, running in userspace. It has a kernel module, but the actual file system is running in the userspace. With fuse an in situ optimization can be implemented, for files written to the fuse file system.

1.2 Goals

Goal of this thesis is to research the feasibility of in situ optimization strategies of Input-Output Operations. This task is divided into:

1. Check FUSE and SIOX for their ability to manipulate operations in situ
2. Implement optimization strategies
3. Evaluating the basic performance of programs
4. Methodical evaluation of optimizations and comparing to the base performances

1.3 Structure of the Thesis

In the next chapter necessary background is introduced. Chapter 3 is about the design of algorithms used and methods applied to find optimizations. Chapter 4 elaborates interesting aspects of the implementation and describes techniques used in the plug-ins. The evaluation of the implemented plug-ins is described in chapter 5. Chapter 6 gives a conclusion and a prospect to future work.

2 Background and related work

This chapter is a description and a theoretical view of techniques used. Section 2.1 describes basic software used. Section 2.2 gives an introduction to the file formats and middle ware that is important to the evaluation in Section 5. The last Section 2.3 gives an overview for related work and state of the art.

2.1 Background

This section is an introduction to file systems in general and two systems that will be used in the course of this theses. Additional the I/O stack is explained, as it is essential for the understanding.

2.1.1 File-systems

File-systems are the essential part of a storage system, they manage where data is stored and how to retrieve it. There exists a variety of different file-systems, that all have specialized features. In Unix-Based systems there is EXT4 as to mention one local file-system and in the field of HPC Lustre, as further described in Section 2.1.2, is widely used as a parallel file system. Of course there are lots of others that are used more or less. These file systems run in the Kernel and are mountable only by privileged users. For non-privileged users there was the development of FUSE as described in Section 2.1.3

2.1.2 Lustre

Lustre is a parallel distributed file-system used in HPC. The development started as a research project in 1999 and is available and the GNU General Public License. The software is open-source, which leads to a wide usage as it is partially free of cost.
A storage Cluster with Lustre, can be seen in Figure 1. It is build from a small number of meta-data Servers (MDS) with their local storage system the meta-data targets (MDT) and a larger number of Object Storage Servers (OSS) with their local storage, object-storage targets (OST). For the cluster there has to be at least one management server (MGS).
When a client is accessing a Lustre file-system, as seen in Figure 2, it first accesses the MDS and receives the File IDs and OSTs (and OSS) on which the data resides. Then the Client only accesses the relevant OSSs to read or write the data. This behavior leads to the decisions in building the cluster. The MDS need high speed storage that is optimized for a access pattern of a vary large amount of very small files. The OSS on the other hand, contain the actual data and need to have a huge amount of local storage that needs to be optimized for a pattern of a small amount of large requests

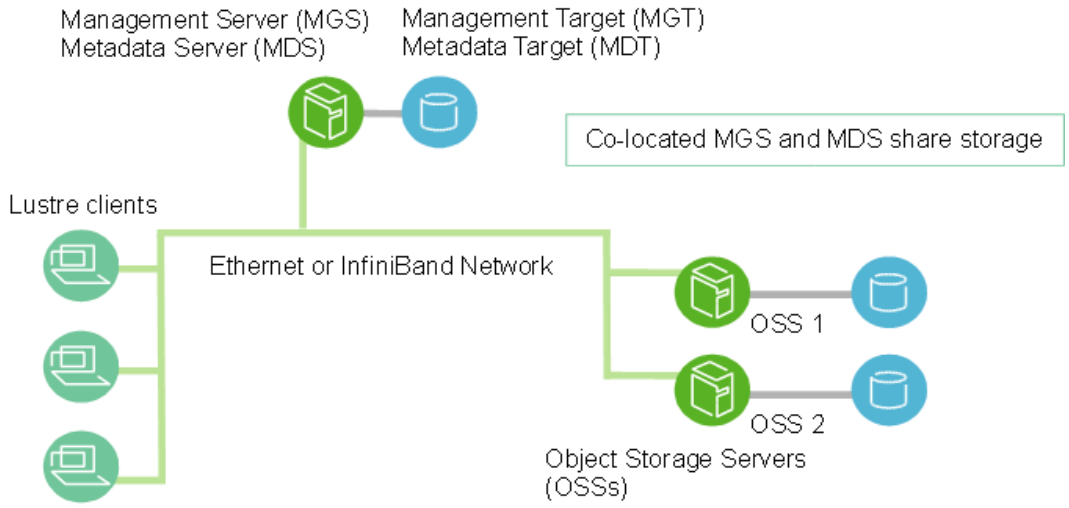


Figure 1: A basic Lustre Cluster [lus17]

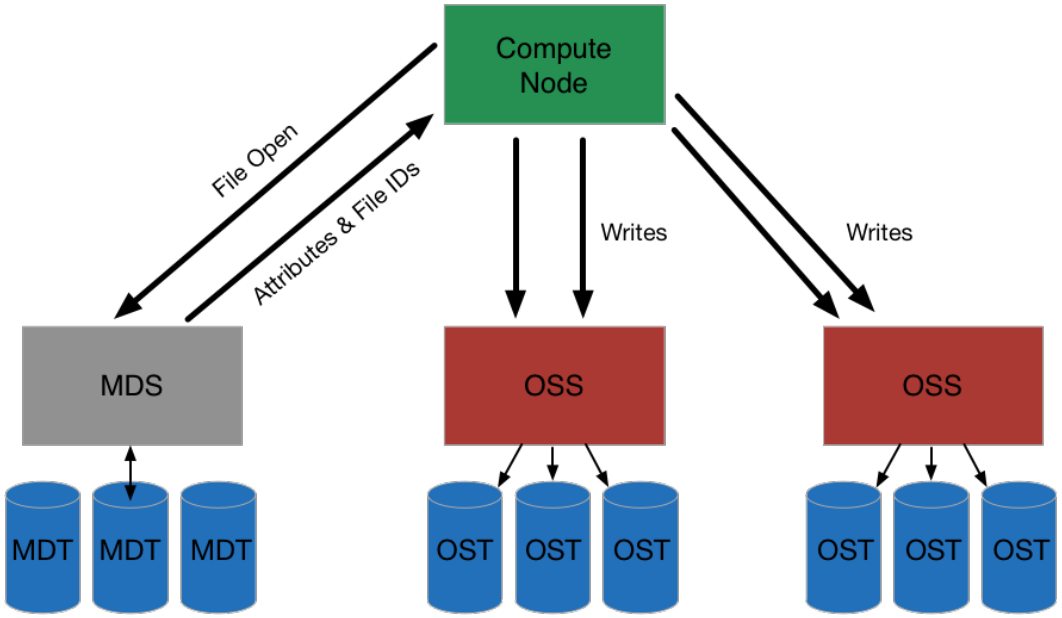


Figure 2: Access to a Lustre file-system [CRC17]

2.1.3 FUSE

FUSE is a file system, that was developed to give non-privileged users the right to create a file system. Regular file systems run in the Kernel, but FUSE only has a kernel module and runs in userspace. Because of the actual file-system being in userspace it is easy to develop a file system without the hassle of crashing the Kernel in case of an error.

FUSE allows users to analyze and optimize the access patterns of their programs. This can be easily done by using existing FUSE-File-systems and changing the code as needed.

While it is easy to write FUSE, it introduces a fair amount of overhead when switching from the Kernel to user-space and back for each I/O-Access. In the further develop-

ment of this thesis will be analyzed how well FUSE performs in comparison to SIOX, introduced in Section 2.3.1.

2.1.4 I/O-Stack

The I/O-Stack is made up of several Layers, as can be seen in Figure 3. The highest Layer is situated in the user space and contains the running application. This application is then sending I/O calls to the Virtual File System (VFS) which is running in kernel-space. In the VFS happens the determination of which actual file-system is running at the specified location, for example Lustre, NFS or FUSE. Even though the application can also use middleware for a higher level of abstraction when calling i/o functions.
The interaction of the application can therefore be interfered at various locations in the

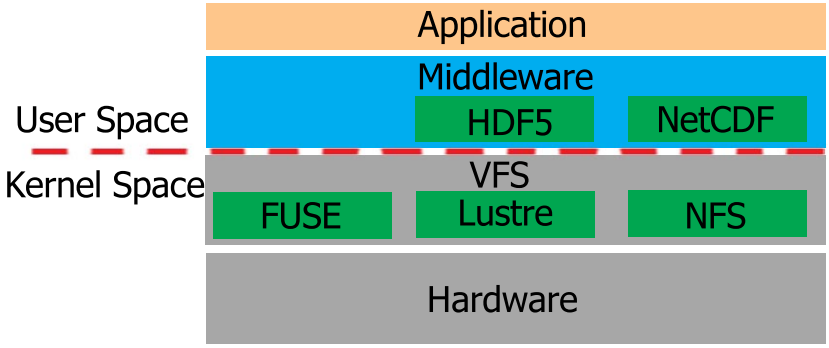


Figure 3: Basic I/O stack

I/O stack. As a short overview serves the following list, whose points will be explained in the following sections:

- The direct calls from the application to the VFS using POSIX interfered at the user space, Section 3.1.1 and at the kernel space, Section 3.1.2
- Interfering using the file system FUSE, Section 3.1.3
- Instrumenting the middleware, Section 3.1.4

2.2 File formats and middleware

In this section, the file format NetCDF and HDF5 are being introduced, as they will be used in the evaluation.

2.2.1 NetCDF

NetCDF, which stands for network Common Data Form, is a library for machine-independent data-formats, that are array-oriented and used in scientific applications for creation and sharing of the data. It is maintained at Unidata, which is a "community

of education and research institutions with the common goal of sharing geoscience data and the tools to access and visualize that data."[Uni17]
The main advantages of NetCDF are:

- It is a **self describing** file, which means, that meta information about the data contained in the file are also included
- The files are **portable**, meaning that it does not depend on the type of system the data was generated, it can be accessed by any other machine.
- Accessing the data is **scalable**, meaning accessing a smaller subset of the whole data is efficient

2.2.2 HDF5

HDF5 is a middleware, which consists of an api and data model to access data, which is saved in an own file format. The software and generated files are portable. With the meta data stored in the files, a understanding of the data, when it is stored in an archive and access at a later time, is possible. It also is used as input for analysis processes and needs to be accessed very efficiently, because these files usually have a high complexity and therefore a lot of data in them, which lead to a very large resulting file.
As stated in [KB15], the HDF Group, which is the maintainer of HDF5 and the predecessor, HDF4, was established in 1988. At first it was a part of the University of Illinois and later an independent non-profit company.
HDF5 files are accessible with an API, which is available for various programing languages, including C, FORTRAN, Python and others.The API consists of more than 300 functions, therefor there exists simple functions for access and more complex ones. The Parallel HDF5 library is build upon MPI and allows therefore a parallel file access, like MPI does. As seen in Figure 4 the HDF5 application is running on a number of compute nodes in the cluster and the underlaying HDF5 library is communicating using MPI. This leads to multiple interaction points for optimizations, like the HDF5 library on each node or the underlaying MPI communication. The interaction points are discussed in Section 3.1.

2.3 Related work and state of the art

While we are able to analyze programs, it still is a huge problem to do this in depth in the background at all times. A tool for a full time analyzes would be Darshan, described in Section 2.3.3 or SIOX, described in Section 2.3.1. While Darshan is widely used, it leaves us with no deeper insight only an overall overview. SIOX can give an insight view

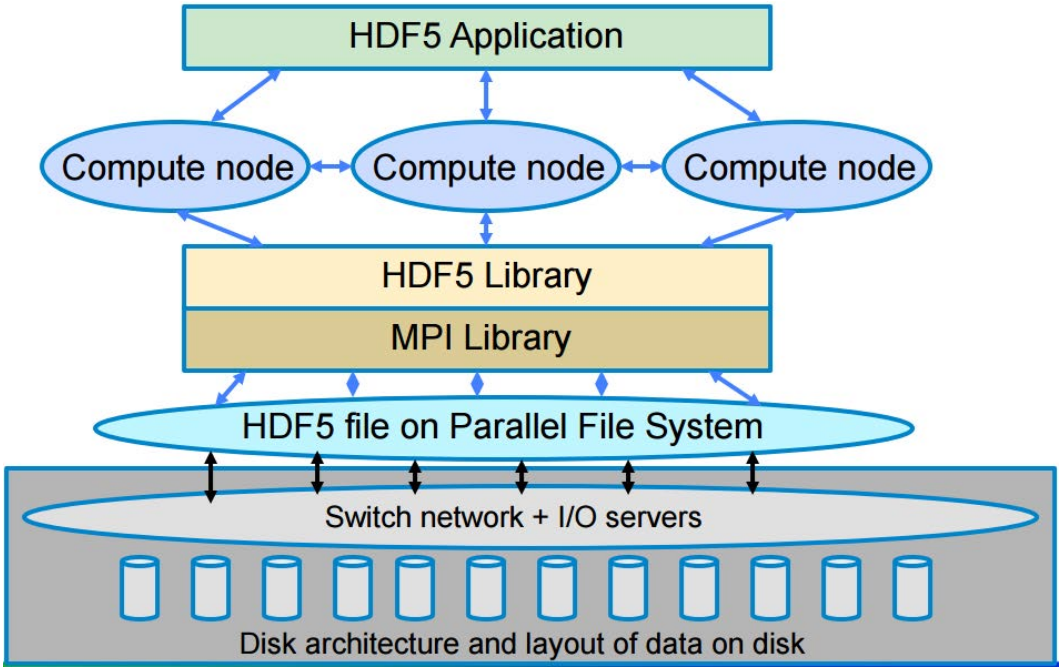


Figure 4: PHDF5 stack build on MPI [CRC17]

of the application, but is not used as widely as Darshan is. When trying to get a very deep insight of what applications are doing, the also widely used tool Valgrind can be used. The downside is, that it generates huge trace files and therefore the performance is dropping, as stated by [NS03]
The following is an introduction to state of the art tools and optimization strategies developed in related work.

2.3.1 SIOX

The Scalable I/O For Extreme Performance (SIOX) Project [KZH+14] is designed to measure and optimize Performance of programs in the field of high performance computing. SIOX was initially developed by the University of Hamburg, the ZIH at the TU Dresden and the HLRS in Stuttgart. It is a modular piece of software that, in its core, can process activities. An activity, in the case of SIOX, is an operation that a program calls to a programming layer.
In SIOX, all components are modular, including the integration for various APIs (layers). In SIOX, a layer describes the interface between the program and file system, that the user wants to inspect. The programmers included the most basic and commonly used layers as a baseline and for reference. Those are POSIX, MPI, HDF5 and NetCDF4. Whenever there is need for additional layers, one can write it.
SIOX uses a configuration file for starting the needed Modules. Each module has a basic structure with an loading function and initialization function. The initialization function also determines which calls to listen to and is then linking the corresponding functions to the linking. With this approach plug-ins in SIOX are able to determine on

their own what is interesting to them, or what they need to know of all the activities that go through the system. Another positive aspect of this approach is, that not all activities are slowed down by all plug-ins, but only by those, that are actually needing to know about an activity, which is a huge aspect in HPC, because we need a system that is not interrupting the system too much, otherwise the optimizations we make to optimize the performance would be outrun by the system used to make these optimizations.
A config file defines which modules will be loaded for the run. A number of various modules is shipped with SIOX, for example one to write trace files. Those traces can then be read by a replay engine like Feign, described in Section 2.3.2.
The easiest way to use SIOX is the integrated instrumentation program (siox-inst), which instruments the layer of interest on-the-fly and no change to the code has to be made.

2.3.2 Feign

When working with optimizations to an I/O workload, it is best to have a pattern, that does not change, so you can observe what effect the optimizations have. In that case, it is impractical to let the application, which possibly has random number generators or other influences in their code, run and test the optimizations. Therefore exists the Flexible Event imitation engine [Lü14]. This a tool that reads SIOX trace files and can replay them. We can inject optimization tools in the process between reading the trace and writing it again to the file-system, because SIOX is build modular.
Feign has been integrated into SIOX at a later state as the initial development as a prototype. This was possible, because, as stated in Section 2.3.1, even the layers are modular.

2.3.3 Darshan

High performance clusters are a complex collection of different computers. Therefore it is necessary to monitor the functionality of the cluster, possibly at all times. At the same time slowing down the cluster is not wanted. The need for a monitoring tool, that can run in background for every program, that is executed on the cluster, and is not interfering too much with the performance of the system, is urgent.
As introduced in [dar], Darshan is a tool, that was developed by the Argonne Leadership Computing Facility at the Argonne National Laboratory. It's lightweight design gives administrators a tool that can be used full time for analyzing I/O-Operations of all programs running in a cluster. When a program finishes its execution, Darshan generates a PDF-file with basic measurements graphically illustrated, that users can use to optimize their programs.
The lightweight of Darshan has some disadvantages too, when analyzing program runs

it can only be used as a first reference, because the results are very basic and do not reflect individual functions of a program. When diving in deeper into optimizing ones own programs, other trace-based like vampire or SIOX can be used, which give a more insight view of what happens in run.

2.3.4 Burst Buffers

When writing to the file-system, we need to wait for particularly the HDDs to start spinning and then rotating to the specified position. This introduces a lot of of overhead to the actual write. In addition the write is also slow, because the HDD has a vary limited write-speed compared to other technologies like SSD or memory. To overcome these problems, one proposal is the use of burst buffers [LCC+12]. They get integrated into the storage nodes and write the incoming data temporary to a local high performance storage module like SSDs or for even more performance NVRAM-based devices. With the approach of burst buffers, there is no need to change the applications. They still write to the I/O Nodes in the cluster, but those then decide whether to write to the buffer or directly. Therefore the only need, for the application, would be to give hints to the I/O Nodes what kind of write is coming next. In this way the node can decide for very small and very big writes what to do.

2.3.5 Optimizing NetCDF

The NetCDF library is not implemented for multi-threading and so the performance does not meed the maximum of the system. This will be tested and evaluated in Section 5.
Analysis for the read speeds of NetCDF4, as done in [JBLO16], show that the pattern and buffer size have significant impact on the speeds that can be achieved while reading from NetCDF4 files. Another factor is the programming language used, as they have tested with the C and Python libraries. The results were, that the C implementation outperformed Python and the read pattern has a significant impact on the performance.

2.3.6 Optimizing HDF5

Optimizing HDF5 for Lustre was analyzed in [HKK+10], they used different Benchmarking tools on a Cray HPC system for their analysis. The optimizations included modifications to the HDF5 and underlaying MPI-IO libraries. By adding API calls, that allow the user to give hints to the library, they introduced a way for optimizing performance by setting the alignment or optimizing the number of chunks, that will be created during the run. This lets the user get the best performance from the

underlying HPC system. In their research, they tested with different application benchmarks and on different HPC systems. The results from their test is, that the meta-data needs to be aligned and not fragmented, otherwise the amount of writes to the Lustre meta-data store decreases performance. With their modifications to the library they optimized this behavior, with an increase in overall performance by a factor of 1.4 to 33.

This chapter described the background and related work, that is relevant to this thesis. The next chapter is describing the design of optimizations to I/O operations.

3 Design

This chapter is going to describe the methods and algorithms used to determine optimization Plugins. Section 3.1 is giving an introduction to the methodology and the calls available to the different layers at which interference is possible. Section 3.2 is describing the possibility of FUSE and SIOX to implement an optimization library and how one could stack optimizations with those approaches. Section 3.3 introduces classifications for optimization strategies. Section 3.4 defines optimizations for demonstration purposes.

3.1 Methodology

There are numerous strategies to optimize applications for better usage of I/O and not all of them are equally helpful for all applications. We need a collection of plug-ins, that can be tested against different applications and then evaluated for their use cases. Usually we are trying to make optimizations stackable, so that for an application we are capable of using more than one strategy.
While researching optimization strategies, there are numerous approaches to interfere, when an application is calling I/O operations. For the interference, we are choosing a fitting approach. There is the choice between interfering in the user or the kernel space. The user space is where the application is running and the kernel space, where the actual calls are made. Writing programs for the user-space is usually less complicated, than writing them for the kernel.
The following sections are giving an insight understanding on the approaches of interfering with the application calls to I/O operations.

3.1.1 POSIX calls in user space

Applications have the choice of various libraries, when they are calling I/O functions. Therefore the instrumentation of POSIX in the user space requires a lot of functions to be caught and rewritten accordingly to their original implementation. This large number of functions makes it difficult to instrument all the functions, but having the information which function was called has a semantic meaning.
On the other hand implementing such a instrumentation in user space has the advantage of easy development and debugging of the instrumenting application. All errors in the code, only have the failure of the application as a result not the machine it is running on.
The instrumentation of the POSIX calls in user space are between the application and before the calls are switching to the kernel, as can be seen in Figure 5. The following is a list of POSIX calls, that have further relevance in this thesis and show the complexity

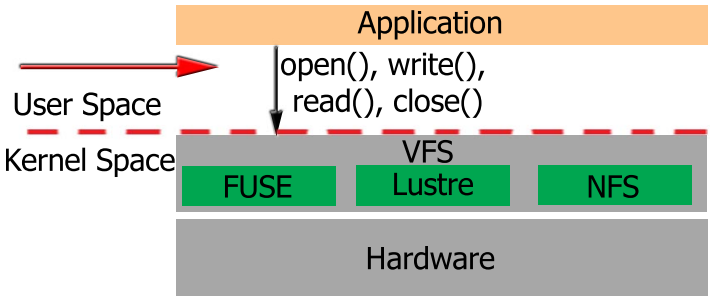


Figure 5: I/O stack, point of POSIX calls in user space

of instrumenting POSIX.

opening Opening a file is essential to I/O operations. Every function that is interacting with a file needs the file to be opened first. In user space there are two ways to open files. One is utilizes the sys_call directly and the other uses the stdio library. Each has a special purpose:

- **int open(const char* pathname, int flags)** is a system call, which receives the pathname at which the file is located and a flag matrix, which decides what mode the file is being opened. Those mode can be read/write only or both, create the file and others more
- **FILE* fopen(const char* path, const char* mode)** is a library call for streaming to a file. The parameters are equal to the sys_call, but the library is handling the interaction with the system calls.
- **FILE* fdopen(int fd, const char* mode)** is from the same library as fopen(). Instead of using the path, it uses an already opened file, that was opened using the sys_call.

closing When the interaction with a file is done, the file has to be closed again. Similar to the open, the close call has to be fitting. When using the stdio library, the close has to be accordingly. The following list gives an overview to the possible functions, but can not considered to be exhaustive:

- **int close(int fildes)** is the sys_call to close a file handle.
- **int fclose(FILE *fp)** is a library call to close a file. Upon close the user space buffer is flushed using fflush(), before actually closing the file. Even though the flush is called, the buffer is not necessarily saved to storage if the kernel is still buffering it.

- **int fcloseall(void)** is a function, that closes all open streams from the calling process. Even the standard streams, stdin, stdout, stderr. Before closing all output is flushed using fflush and all buffered inpt is discarded

writing When writing to a file, a variety of functions is usable. Those are system calls or from the stdio library. In the following an excerpt of functions:

- **ssize_t write(int fildes, const void *buf, size_t nbyte)** is a system call, that is writing a number of bytes (nbyte paramter) from the buffer (buf parameter) to the file handle (fildes paramter) and returns the actual number of bytes written or -1 to indicate an error.
- **ssize_t pwrite64(int fildes, const void *buf, size_t nbyte, off64_t offset)** is the system call to write to a file, with the addition, that the file pointer is not changed and the position, at wich the write is taking place is the offset, given as parameter. The offset is a 64bit value to enable the interaction with large files.
- **ssize_t pwrite(int fildes, const void *buf, size_t nbyte, off_t offset)** was the system call prior to kernel 2.6 after that, it was renamed to the new function pwrite64().
- **size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream)** is one of the write functions from the stdio library. it writes the number of elements (nmemb parameter) with a length of the size parameter in bytes, from the pointer (ptr parameter) to the file stream. The return value is the number of items written.
- **int fputc(int c, FILE *stream)** this is one of multiple put functions, that are writing a character or string (fputs()) to a stream without the terminating null character.
- **int printf(const char *format, ...)** is a family of functions, that are getting a format and are writing it to different locations. The location is determined by the name of the function, this is stdout for printf, a stream for fprintf, a string for sprintf and snprintf which is also writing to a file but with a maximum length parameter.

reading Reading from a file is similar to writing. The number of functions is comparable to the write functions. The following list is the mapping from write to read functions:

- write() -> read()
- pwrite() -> pread()

- pwrite64() -> pread64()
- fwrite() -> fread()
- fputc() -> fgetc()
- printf() -> scanf()

seeking When accessing files, many functions use the current position in the file to read or write data. This file position can be manipulated without accessing the data. As with the other functions exists multiple methods in doing so. The following list is giving an overview of functions:

- **off_t lseek(int fildes, off_t offset, int whence)** is the system call, that interacts with the file offset. The offset parameter can have different meanings depending on the whence parameter.
- **int fseek(FILE *stream, long offset, int whence)** is a function from the stdio library that is equal to the lseek() for streams.
- **int fsetpos(FILE *stream, fpos_t *pos)** is equal to fseek() with the whence parameter pre set to SEEK_SET, with the result of the offset parameter being the new file position
- **long ftell(FILE *stream)** is a function to receive the current offset in the stream.

3.1.2 POSIX calls in kernel space

The kernel space only has a very limited number of calls that need to be instrumented, those are described in the previous section. This advantage is annihilated by the disadvantage of the complexity, that is introduced by implementing a kernel module. In kernel modules every error leads to a crash of the machine, or at least to serious consequences in the stability.
Another problem is, that with a decrease in number of functions, the semantical informations, that come with calling a certain function, gets lost.

3.1.3 FUSE file system

Using FUSE to instrument an application has the advantage of having a limited number of functions, that can be called, because the kernel is calling FUSE and this reduces this number. In addition the complexity of implementing a FUSE file system is limited and therefore good.
Like the previous approaches, FUSE has disadvantages, this is the lack of semantic

information in the reduced number of functions and the unknown performance impact, which will be analyzed in Section 5.5.1.
When writing a FUSE file system, the point, where the interaction with the calls from the application is happening can be seen in Figure 6. There the calls went from the user space to the kernel space and there to the VFS, which decided, that the file system of the path is a FUSE file system. Then the call is approaching the FUSE kernel module and from there to the user space and to the actual FUSE implementation.

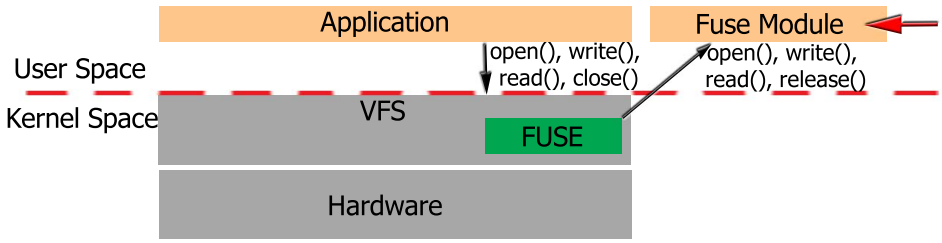


Figure 6: I/O stack, point of FUSE in user space

A number of important calls is described in the following list. The functions are used in the high level API of fuse, where the kernel module is calling the functions with a path parameter instead of a request and an inode. A FUSE function name is composed of the name of the file system followed by the call. For this list, the file system is called fuse:

- **static int fuse_open(const char *path, struct fuse_file_info *fi)** is the open call, which takes the path and a file_info struct and returns 0 on successful opening or -1 on an error. The struct can be used to save additional information, like the file handle.

- **static int fuse_release(const char *path, struct fuse_file_info *fi)** is the close call. It uses the saved file handle from the fuse_file_info struct and is calling the close function on it.

- **static int fuse_write(const char *path, const char *buf, size_t size, off_t offset, struct fuse_file_info *fi)** is the call to write to a file. The call not dealing with a file pointer, but instead is taking the offset, at which position the write should be executed. It is similar to the *pwrite()* of POSIX. The path parameter can be used to open the file if no file handle exists. The return value is the bytes written.

- **static int fuse_read(const char *path, char *buf, size_t size, off_t offset, struct fuse_file_info *fi)** reading from a file works in the same way writing does. The buffer is taken to save the read information. The return value is the bytes read.

3.1.4 Middleware instrumentation

Using a middleware as instrumentation layer, is an implementation in user space, which could reduces the complexity. But this reduction is annihilated, by the functions that can get called and are usually very complex in a middleware. In addition, there are a lot of different middlewares on the market. In this thesis I am using HDF5 and NetCDF. The performance reduction while running an instrumented middleware is not very high. When instrumenting a middleware, like HDF5, the point of instrumentation is between the application and the middleware, as can be seen in Figure 7. The application is calling middleware function, which then get instrumented.

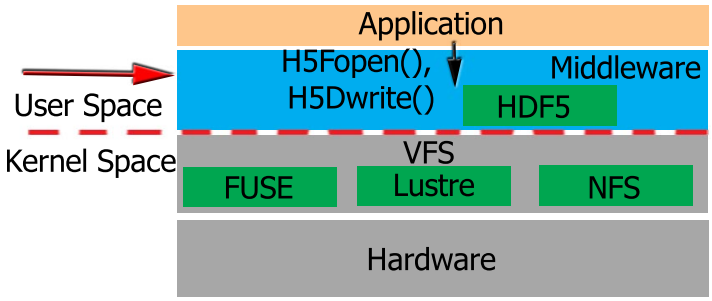


Figure 7: I/O stack, point of middleware in user space

The following list of functions is an excerpt of needed functions, to be instrumented, on the example of HDF5, for a possibility to manipulate I/O calls:

- **create a file**, to create a HDF5 file the following three functions have to be called:
  1. H5Fcreate(), creates a new HDF5 file
  2. H5Screate_simple(), creates a simple data space
  3. H5Dcreate2(), creates a data set
- **opening an existing file**, when data is already in an existing HDF5 file, this file has to be opened with the following functions:
  1. H5Fopen(), open an existing HDF5 file
  2. H5Dopen2(), to open the existing dataset in a file
- **accessing data**, after opening or crating the dataset, it can be written and read from it with the following functions:
  - H5Dwrite(), to write new data into a dataset
  - H5Dread(), to read existing data from a dataset
- **closing a file**, after the access to a HDF5 file, the different opened instances have to be closed again using the following functions:

1. H5Dclose(), to close the dataset
2. H5Sclose(), to close the data space
3. H5Fclose(), to close the HDF5 file

The calls in this list are generating a multidimensional memory space in a file space. The calls of the middleware are more complex than FUSE or POSIX.

3.1.5 Decision of layer

All the discussed layers in this chapter have their advantages and disadvantages. In Table 1 is a comparison between the different layers as a summary. The characterization of each column is in the following list:

- **Universality** is a measure of how well a certain implementation can be used to solve other problems, or how well it can be adapted.
- **Complexity** describes the difficulty of an implementation of a certain layer. This includes the number of functions and their own complexity.
- **Performance** is a measure of the expected performance gain, when implementing on the layer. And the loss of performance, that comes with the choice.
- **Semantic information** while a small number of functions can reduce complexity, it clearly reduces the amount of information, that is given by a programmer in using it. For example writing to the stdout and using *printf()*.

As can be seen in Table 1, the complexity of both POSIX at the user space and FUSE is good, the other two layers have a higher complexity, therefore and for the performance gain is not bad for those layers, I decide to use them.

	Universality	Complexity	Performance	Semantic information
Middleware	- -	- -	+	++
POSIX user space	+	+	+	+
POSIX kernel	++	- -	++	-
FUSE	++	++	O	-

Table 1: Comparison between the different instrumentation layers

3.2 Optimization library

Implementing the Optimizations for a certain system is not the only important task. We also need to make the optimizations available in a library. When applying optimizations to a problem, one might not be enough to get the best performance. Therefore it is necessary to have the possibility to stack the optimizations. In the following I will introduce this task for FUSE and SIOX.

3.2.1 FUSE

In FUSE, the trivial method of stacking optimizations, as shown in Figure 8a, is to use multiple instances of FUSE and having an input path and output path. With this method the FUSE instances are stacking by having multiple folders, for each optimization one. I will outline the steps needed to write to the destination folder:

- The application is writing to its output path, which is a FUSE file system
- A switch to the kernel space is required and the FUSE kernel module is calling the first FUSE file system. There the switch to user space takes place and the optimizations from the first FUSE instance are made. This instance is writing to an output path, which is again in a FUSE file system.
- The call returns to the kernel and because the output path was a FUSE file system, the FUSE kernel module is redirecting the call to the next instance.
- This switching is repeated for every optimization in the stack.
- Last the call gets returned to the application and has a return value, which the application can use.

Because the instances of optimizations have to switch between kernel and user space a lot, this type of setup is slow and inefficient.
The second approach, as shown in Figure 8b, is to use modules. Which is a non documented feature, but it is used in the libraries of the FUSE source code. By writing a basic file system in FUSE, which is providing a path to which an application can write and an output path, which FUSE is then writing to. The optimizations are then written in modules. Every call to one of the implemented file system functions need to call the function with the preword "fuse_fs_" and then the name of the currently implemented function, after the optimization. This call is then calling the next module in an optimized way. This is needed in every function. After the last module is called, the main file system is called with all optimizations, that got stacked. This method is more optimized than the previous one.

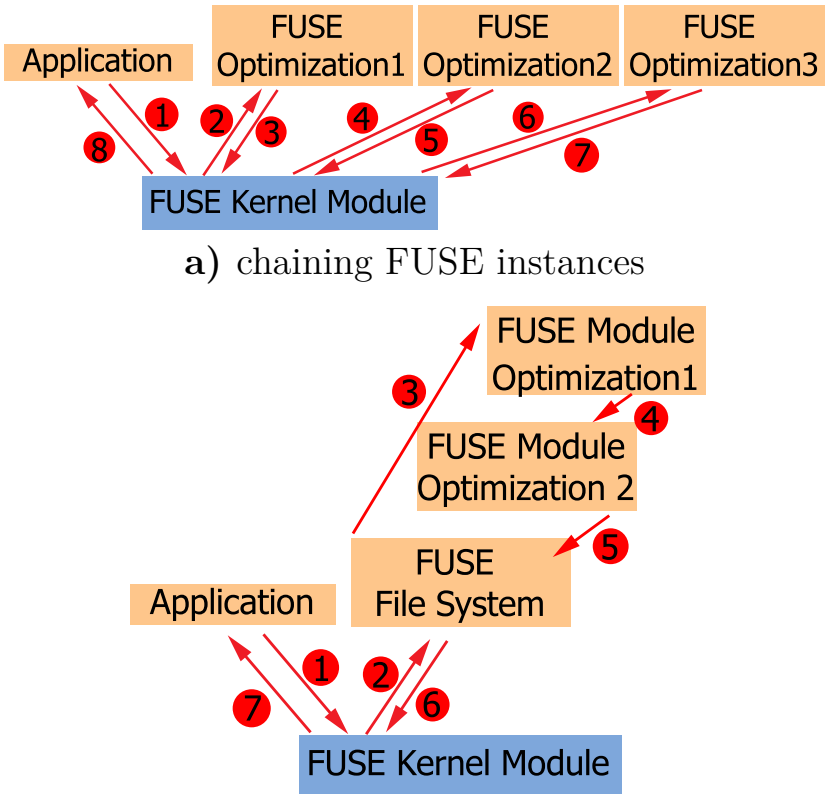


Figure 8: FUSE stacking optimization methods

3.2.2 SIOX

SIOX is a system that is able to stack optimizations, this comes from the way it is build, as a modular system. The config file is loading the plug-ins and other system relevant modules at startup.
A multiplexer is a module in SIOX that is receiving activities and then redirecting them to the configured plug-ins. The system is not limited to only one multiplexer, but can have multiple as appropriate. Some plug-ins, that are changing or removing activities, can do this only, when they send them to a new multiplexer. Otherwise activities can not be removed and they would be redirected to the next plug-in.
To optimize I/O operations, there exists an online and an offline path. The online path is using a special dlsym, that is instrumenting the I/O calls and generating activities from them. The way of those activities through SIOX is displayed in Figure 9. The last plug-in that is required for the online path to function, is the ReplayOnline plug-in.
The following steps are taken on the way through SIOX, when a call is instrumented:

1. The posix-deedless-dlsym is catching the I/O request by instrumentation and generating an activity with the known values, according to the arguments of the function, that was called. Timers in SIOX are started, to measure the time a call is taking including passing through SIOX and calling the actual function to make

the I/O call to the file system. Lastly the dlsym is passing the activity to the global multiplexer.

2. The global activity multiplexer is receiving the activity and passing it to all registered plug-ins, which is done during initialization, and user controlled with a configuration file. The multiplexer can either be synchronous or asynchronous for a faster handling of activities.

3. The plug-ins are receiving the activity and handling it according to their configuration. For monitoring plug-ins the activity gets processed and the path ends. For modifying plug-ins like the OptimizeBlocksize plug-in the activity gets processed and if necessary, gets handed to the next multiplexer instance.

4. The second multiplexer is handling the activity the same way the first one, it sends it to the next plug-ins, according to the configuration. The points two and three can be chained as needed, if the need for more than one modifying plug-in is needed. TraceWriter plug-ins can be added to the multiplexer as needed, to generate a trace file after each manipulation of activities.

5. The last plug-in in the chain is the ReplayOnline plug-in, which is executing the incoming activities accordingly, to their specific configuration with the values saved in them. This plug-in is also filling in the return value for the activity. The activity is returned to the POSIX-deedless wrapper, which extracts to output value and returns it to the function.

The offline path through SIOX is using the siox-trace-reader, which is reading a trace file, generated by the TraceWriter plug-in, and is generating activities that are going the same way through SIOX as in the online path, with the exception, that the Replay plug-in has to be loaded, for the activities to being actually written to the file system.

3.3 Classification of Optimization strategies

When classifying optimizations, there are various approaches. I will be using a number of the approaches introduced in the following:

- **Scope**: The scope is showing which type of data is being modified. This can be either meta data or data.
- **Interference with I/O operation**: Optimizations can interfere with a single I/O operation or with multiple, while multiple can be divided into actively modifying multiple accesses and passive because of the modification of the open call.

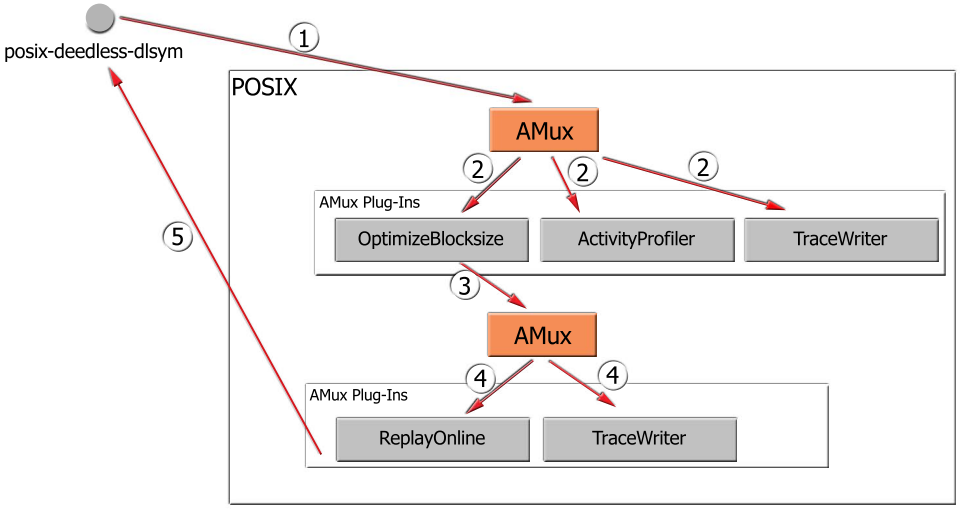


Figure 9: SIOX Example configuration for online modifications

- **State**: While modifications to I/O calls are made, optimizations sometimes need to know about previous I/O calls or keep a state of the file.
- **Semantics**: When changes to the semantics of a file are made the application might not be able to read anymore.

3.4 Optimization Strategies for demonstration purposes

One of the goal of this thesis is the abstract cover and implementation of optimization strategies. To have a variety and cover the classifications from Section 3.3. I decided to use three different optimization strategies as a demonstration.
Optimizing file access patterns require insight knowledge about the underlaying system. Therefore it is essential, before optimizing ones application, to know what kind of optimization, with which parameter is going to have the best performance. By testing those strategies first, the performance gain can be evaluated before the implementation.
The first strategy will remove seeking and is described in Section 3.4.1. The second one will rewrite the path and is described in Section 3.4.2. Lastly I am describing a write back optimization in Section 3.4.3.In Table 2 is the association of those optimization strategies to the classification from Section 3.3

strategy	Scope	Interference with I/O operations	State	Semantics
Seek free	data	yes	stateless	yes
Rewriting path	meta data	indirectly following	stateless	no
Write back	data	no	stateful	no

Table 2: Optimization startegies classification matrix

This section is giving an insight about the different strategies to optimize I/O patterns.

3.4.1 Seek free

Writing data while execution of an application is sometimes random or processes are writing their result to a file, which leads to an access pattern with small writes and seek to get to the next position to write. Those kinds of pattern are not optimized. The goal for an optimization can be to get a sequential access pattern from an incomming random pattern.
When analyzing an access pattern as seen in Figure 10, there each process has a recurring part in the file. When analyzing only process 1, the access pattern is a write, then a seek of two times the size of the write and then a write again. Seeking is essential for such a pattern, but if not optimized, the seeks can reduce write performance significantly.

Figure 10: Access pattern with multiple processes

A basic optimization strategy would therefore be, to eliminate any seeking from the access pattern. This will lead to a change in semantics and the file will not be readable anymore.
As a first approach, to get a baseline of performance, this optimization should get the theoretical maximum performance from the system, as a sequential access would.
In the following is an analysis of which kind of calls this optimization has to handle, to achieve the required demands.

lseek Seeking in a file leads to moving the file-pointer to the position of the offset. Therefore every call to lseek has to be eliminated, to gain an access pattern that is sequential.

pwrite While the write function is writing at the file-pointer, and moving the pointer according to the bytes written. The pwrite function is not using the file-pointer as a reference, but is using an offset, so it can write at any position in the file, without interfering other i/o-operations to the same file. Therefore, to eliminate seeks, it is necessary, to also interferer with the pwrite, with the goal of manipulating the offset, to have it be at the last position, the file was accessed. The interference could also be switching the function from pwrite to a regular write and remove the offset.

pread The pread function is similar, to the pwrite function. It also has to be interfered with, with the purpose of removing any seeking that might happen by using the offset.

The optimizations done to remove any seeking is to modify the stream of i/o functions. regular read and writes get passed through as they are. Every incoming seek operation gets dismissed and pwrite and pread functions get modified, in a way that the offset is at the current file-pointer and that the file-pointer is moved accordingly, as if the pwrite and preads would be a regular write and read.
This leads to a sequential access-pattern which is the best for rotating disks, as they are used in most file-systems

3.4.2 Rewriting path

When a Checkpoint or any other File is written to disk, it can be better to first write it to a local file-system and in the background, when the program is not waiting for the end of the I/O Operation, combine the local files in a shared file-system
In a first step as the rewriting file-path optimization is, the file-path is rewritten to a different location. This could ether be from a slow NFS-share to a Lustre or when the local disk has enough speed, the file-path could be temporarily moved to this local storage. In most applications the local storage has possibly not enough space and the destination storage could be a network caching node.
In check-pointing the basic rewrite of the path is not a problem. But the optimization is turned on for the whole execution of the application and usually not only writes occur, but also reads. Therefore we always need a method to exclude certain input path from the rewriting.

open The open function is the one, which is generating the file-handle from a path. Therefore interfering with the open call is enough to rewrite the path of the file. All other calls are implicitly changed, because they use the file-handle.

close While the open call can move the destination of the file, the file hast to be moved to the final destination, after it is closed. Therefore it is only logical to move the file after the close call to the destination, that is desired.

3.4.3 Write back

When writing to a file-system like Lustre, it is best to know the stripe-size with which Lustre is working, so that one process can access exactly one OSS. For every folder this can be configured differently, but there exists a default configuration, which leads to possibilities for scientists and programmers, but also for a lot more work done by hand, that is not directly connected to the application that they are writing. Most of them will not make use of this overhead of work to figure out, what might perform best and therefore will stick to the default configuration, which might lead to longer execution times.
The Optimization can only apply to user generated files and not to the synchronization and initialization files of, for example MPI. This files need to be written and read by different processes, which could lead to malfunctioning of the initialization process. Another factor is that synchronization calls are not going to be very large and not to many and therefore are not going to impact the i/o times this much.
Optimizing the block-size is a process, which needs to analyze all i/o functions. It then checks whether it should interfere with the current call and last the actual optimization is taking place. This optimization has to handle each function independently, respecting their use case and the output of each function.

write The write call is a function which gets the file-handle, a buffer and a size and then writes the data from the buffer with the size of count to the file-handle at the current file pointer position and changes the position of the file pointer according to the size written.
Therefore the optimization can check the size and compare it to the maximal cache size for the decision if the write is going to be cached. When this is the case it also needs to check at which position the last write was and if the file pointer is also at this point. The difference might come from functions, which are not moving the file pointer according to the last write like described in Paragraph pwrite. The write also has to check how much the buffer is already filled and then flush it if necessary.

pwrite The thread safe pwrite call has a similar functionality as the write with the exception that it has an additional input parameter, the offset, which is the position in the file to which the write is being performed and it is not changing the current file pointer.
When calling the pwrite function, to optimization is going to check whether the last access position is equal to the offset and, if this is the case, it can write into the buffer the same way the regular write is performing. In the case, that the last access is not equal to the offset, the current cache has to be flushed and the the regular write would be called, with the exception, that the file pointer cannot be modified.

read Reading from a file is a function, which can be optimized by returning the current cache if the read is in the cache, or by calling the read to the file system and returning the result to the caller. When calling the read function the file pointer from the current file-handle is moved according to the byte to be read by the function. This has to be taken into account with the caching of the writes, because the next write might not be at the same position the last ended and the cache has to be flushed.

pread This function also performs a read from the file-system, but like the pwrite it has an additional parameter for the offset to read at and is thread safe. It is also not moving the file-pointer When reading using pread, the cache might be read and can therefore be returned to the caller or the read has to be performed on the file-system and the result has to be returned.

lseek Seeking is the performance killer when it comes to I/O operations, because rotating disks have to move to the correct part of the file which costs time and therefore waiting cycles for the CPU. When an lseek occurs the optimization has to check whether it is actually moving to a different position in the file or if the lseek can be removed. This might be the case if an application is writing to disk using write and is then checking if the data got to the disk correctly using read. It hast to seek to the beginning and read the file. Those accesses can utilize the optimization perfectly, by caching the write and if a seek in combination with a read occurs, the cached write is returned.

This chapter is about the design of optimization strategies and the decision of the layer, on which the optimization is working. The next chapter is about the implementation of the described optimization strategies.

4 Implementation

This chapter is going to describe the most important concepts of the implementation in a paradigmatic way. Section 4.1 is describing the Architecture of SIOX. Section 4.2 is giving an overview of the implementation of the seek free optimization. Section 4.3 is about the implemntation of the write back optimization. Needed modifications to SIOX are described in Section 4.4.

4.1 Architecture of SIOX

For SIOX being an existing piece of software, it is crucial to implement code at the right place. SIOX uses plug-ins for interacting with Activities. Therefore, every Optimization that will be implemented is going to be located in a single Plug-in. This way it is later easier to evaluate the performance gains of single or multiple Plug-ins. In the stack of SIOX, the plug-ins developed in this theses will be located at the monitoring multiplexer.

4.2 Seek free

The following subsections describe the implementation of the optimization strategy to remove seeking from I/O operations. as described in Section 3.4.1.

4.2.1 FUSE file system

For our test, we needed to implement the removal of seeks in FUSE. This would give the best performance when writing randomly to disk. Therefore I introduced a variable in FUSE, that saves the current position in the opened File and when a write occurs the offset, which the write function gets, gets overwritten by the last offset, that we accessed. The FUSE-Function for write can be seen in Listing 1. FUSE has no function for the seek command, because this is not handled by the high level API of FUSE. It only has the write call, which has a parameter called offset, as can be seen in line 3. By having an offset handled by the user file system instead of the automatically handled offset, it is possible to increment the offset by the amount, the write is having in its size parameter, the sequential write is achieved. No other functions have to be modified, because the goal is to have a sequential write.

```
1  static off_t c_write_offset = 0;
2  static int cache_write(const char *path, const char *buf, size_t size,
3              off_t offset, struct fuse_file_info *fi)
4  {
5      int res;
6          c_write++;
7
8      (void) path;
9      res = pwrite(fi->fh, buf, size, fi->c_write_offset);
10     fi->c_write_offset += size;
11     if (res == -1)
12         res = -errno;
13
14     return res;
15 }
```

Listing 1: Ignoring seeking in writes using FUSE

4.2.2 SIOX plug-in

In SIOX instead of modifying the write operation and removing the offset, there is an Activity, that seeks through the file. By ignoring this kind of activity and only listening to write we accomplish to ignore seeks. The Plug-in has two activity-Multiplexer, one that inputs all Activities to the Plug-in and one that gets all important activities and gives them to following plug-ins.
This at least seems to be as easy as it sounds for SIOX, but while for the observing part of a benchmark we can just redirect reads. A benchmark is also initializing a lot of values and therefore needs to read and write to other files in different locations. Therefore we need to make sure those values can be correctly read and written. When using MPI for example, even if it is just a included library, with the approach of SIOX, when instrumenting our benchmark, this also instruments all libraries leading to messing with initialization phases of them.
The following list is an explanation of Listing 2, which includes all interference necessary to remove seeking:

- When an *open()* call is made, lines 2-14 are being executed. In line 3 the open is redirected to the next multiplexer and executed. Lines 4-7 are retrieving the filename and file handle from the activity. Line 8 is checking whether the filename is included in the intervention path. Lines 9 and 10 are initializing the global variables, that are needed later.

- Lines 18-22 are checking if the activity is an lseek and if the path is supposed to be seek free. Then the activity is removed and not send to the next multiplexer.

- Lines 24-27 are called if the activity is closing the file handle. The global variables are cleared then.
- If an *pwrite64()* is called, lines 28-39 are being executed. It is being checked is the file is supposed to be interfered, line 31, and then the global variable *filePosition* is used to modify the *pwrite64()* to have a offset equal to the file pointer. The *filePosition* is set accordingly to the number of bytes to write.
- Line 40 is sending all activities that are not removed before to the next multiplexer instance.

```
1  int ucaid = activity->ucaid();
2  if (ucaid == posix_open) {
3    out->Log( activity );
4    int fnAttID = facade->lookup_attribute_by_name(interface,"descriptor/filename").aID;
5    const VariableDatatype * attFileName = activity->findAttribute( fnAttID );
6    int fhAttID = facade->lookup_attribute_by_name(interface,"descriptor/filehandle").aID;
7    const VariableDatatype * attFileHandle = activity->findAttribute( fhAttID );
8    if (prefix(intervention_path.c_str(), attFileName->toStr().c_str())) {
9      sequalizableFiles[attFileHandle->uint32()] = 1;
10     filePosition[attFileHandle->uint32()] = 0;
11   }
12   else { sequalizableFiles[attFileHandle->uint32()] = 0; }
13   return;
14 }
15
16 int fhAttID = facade->lookup_attribute_by_name(interface,"descriptor/filehandle").aID;
17 const VariableDatatype * attFileHandle = activity->findAttribute( fhAttID );
18 if (ucaid == posix_lseek) {
19   if (sequalizableFiles[attFileHandle->uint32()]) {
20     return;
21   }
22 }
23
24 else if (ucaid == posix_close) {
25   sequalizableFiles.erase(attFileHandle->uint32());
26   filePosition.erase(attFileHandle->uint32());
27 }
28 else if (ucaid == posix_pwrite64) {
29   int fhAttID = facade->lookup_attribute_by_name(interface,"descriptor/filehandle").aID;
30   const VariableDatatype * attFileHandle = activity->findAttribute( fhAttID );
31   if (sequalizableFiles[attFileHandle->uint32()] == 1) {
32     int qbAttID = facade->lookup_attribute_by_name(interface,"quantity/BytesToWrite").aID;
33     const VariableDatatype * attBytes = activity->findAttribute( qbAttID );
34     int fpAttID = facade->lookup_attribute_by_name(interface,"file/position").aID;
35     VariableDatatype * newFilePosition = new VariableDatatype(filePosition[attFileHandle->uint32()]);
36     activity->replaceAttribute(Attribute(fpAttID, *newFilePosition));
37     filePosition[attFileHandle->uint32()] += attBytes->toUint64();
38   }
39 }
40 out->Log( activity );
```

Listing 2: SIOX handling an activity, when ignoring seeking

4.3 Write behind

The development of an optimization, that is caching the write requests, consists of multiple steps. At first the Optimization has to decide, whether the writes belong to a path that needs optimization, because MPI initialization phases should not be optimized. This decision is already made when opening the file, as can be seen in Listing 3. There the first action is to send the not modified activity to the next instance where the actual open can be performed, this leads to a better system performance for files

that will not be cached and is necessary for the optimization, because we need the file-handle for further processing. In the next step the filename is retrieved from the activity, with the purpose of deciding whether the file is configured to be optimized. After this decision follows a sequence of initial commands that allow to decide later on, what the plug-in needs to do with incoming activities. Those are done in the following order:

1. Generate a struct for saving the needed attributes
2. Allocate memory for the buffer
3. Set all byte in the buffer to zero, so no remaining data is in the buffer
4. Retrieve the file-handle from the activity, that was written there by the actual write in background.
5. Save the generated struct in an object, with the file-handle as key, for a fast access

After the initialization, the plug-in returns, because no further computation is needed for any open calls. This initialization introduces the possibility to check every new activity, whether it needs optimization, by just looking in the *openFiles* object and check if the file handle was inserted.

```
1  out->Log( activity );
2  OntologyAttributeID fnAttID = facade->lookup_attribute_by_name(interface,"descriptor/filename").aID;
3  string filename(activity->findAttribute( fnAttID )->str());
4  if ( inputpath != "" && filename.find( inputpath ) == 0 ) {
5    buff_struct buff_tmp;
6    buff_tmp.buffer = (char*) malloc(config_buffer_size);
7    memset(buff_tmp.buffer, 0, config_buffer_size);
8    OntologyAttributeID fhOAttID =  facade->lookup_attribute_by_name(interface,"descriptor/filehandle").aID;
9    uint32_t filehandleOpen = activity->findAttribute( fhOAttID )->uint32();
10   openFiles.insert(
11      std::pair<uint32_t,
12      buff_struct>(filehandleOpen, buff_tmp) );
13 }
14 return;
```

Listing 3: Opening a file in the write-behind plug-in in SIOX

When a write activity enters the system, a short preprocessing has to be done, but the very first check is, if the write needs to processed. This is done by checking in the *openFiles* object if the file handle is in it. Then what type of write the activity is dealing with, this leads from the differences in the calls. The main types are *write(), pwrite(), pwrite64()*. For the write, because it changes the file pointer we need to adjust the struct in the *openFiles* object. This is done as seen in Listing 4, by retrieving the file-pointer and amount of bytes to write and then just adding the amount of bytes to write to the file pointer But first there is the exception check if the last offset is not equal to the file pointer in which case we sync the data with the file system. This exception can occur, when *pwrite() and write()* calls come in mixed.

```
1  OntologyAttributeID fhAttID = facade->lookup_attribute_by_name(interface,"descriptor/filehandle").aID;
2  uint32_t filehandle = activity->findAttribute( fhAttID )->uint32();
3  OntologyAttributeID btwAttID = facade->lookup_attribute_by_name(interface,"quantity/BytesToWrite").aID;
4  uint64_t bytesToWrite = activity->findAttribute( btwAttID )->uint64();
5  if ( openFiles[filehandle].filePointer !=
6       openFiles[filehandle].last_offset ) {
7    ourSync(filehandle);
8  }
9  openFiles[filehandle].filePointer += bytesToWrite;
10 return optimizeWrite(activity);
11 }
```

Listing 4: Handling an incoming write in the write-behind plug-in in SIOX

A *pwrite()* call does not modify the file-pointer therefore there are different exceptions to handle than in the write, as can be seen in Listing 5. Here we need the file-position to which the write will start writing. This position, then gets written to the struct as the last offset. This offset gets added with the bytes to write in the *optimizeWrite* function. In the case where the last offset is not equal to the file position we need to sync the data, because the data is not written sequential.

```
1  OntologyAttributeID fhAttID = facade->lookup_attribute_by_name(interface,"descriptor/filehandle").aID;
2  uint32_t filehandle = activity->findAttribute( fhAttID )->uint32();
3  int fpAttID = facade->lookup_attribute_by_name(interface,"file/position").aID;
4  uint64_t filePosition = activity->findAttribute( fpAttID )->uint64();
5  if ( filePosition != openFiles[filehandle].last_offset ) {
6    ourSync(filehandle);
7  }
8  openFiles[filehandle].last_offset = filePosition;
9  return optimizeWrite(activity);
10 }
```

Listing 5: Handling an incoming pwrite in the write-behind plug-in in SIOX

While differences in the functions get handled separately, the actual writing can be handled in a function that gets invoked by all the write function. This shared function gets discussed in the following. When writing to the file system, the optimization has three different cases of what to do with the incoming buffer:

1. The incoming buffer has a size larger then the cache, that is reserved.
2. The incoming buffer and the already cached size are combined larger than the cache, that is reserved
3. The incoming buffer fits into the rest of the cache

When the first case occurs, as can be seen in Listing 8 the optimization is going to sync the current cache to the file-system by calling *ourSync* with the file-handle of the opened file and is then redirecting the incoming buffer to the replay engine of SIOX. This is necessary, because chopping the large buffer in smaller ones would not result in a better performance, than just writing it out a as is.

```
if (bytesToWrite >= config_buffer_size) {
  ourSync(filehandle);
  out->Log( activity );
  return;
}
```

Listing 6: Handling an incoming write, that is larger than the cache in the write-behind plug-in in SIOX

The second case occurs when the data in the cache combined with the incoming data exceeds the size of the buffer. Then the copy of the new data and flushing of the old one is done in a two step process. In the first step, the following actions have to be done:

1. Calculate how much space is left in the buffer
2. Calculate the position inside the buffer where new data can be written
3. Copy the new data into the buffer until the buffer is full
4. Set the new fill size of the buffer to maximum and call the sync function

In the second step, the rest of the data has to be copied as follows:

1. Calculate the address inside the new data, that is the beginning of the not already copied data.
2. Copy the rest of the data into the buffer
3. Calculate and save the new fill position of the buffer
4. Save the amount of data written to the attribute, so that a calling program has access to the return value of the write call.

```
if ( bytesToWrite + openFiles[filehandle].buffer_fill_position > config_buffer_size ) {
  size_t written_to_buffer = config_buffer_size - openFiles[filehandle].buffer_fill_position;
  char* currBuffPosition = openFiles[filehandle].buffer + openFiles[filehandle].buffer_fill_position;
  memcpy(currBuffPosition, (void*)dataMemoryAddress, written_to_buffer);
  openFiles[filehandle].buffer_fill_position = config_buffer_size;
  ourSync(filehandle);
  uint64_t restAddress = dataMemoryAddress + written_to_buffer;
  memcpy(openFiles[filehandle].buffer, (void*)restAddress, bytesToWrite - written_to_buffer);
  openFiles[filehandle].buffer_fill_position = bytesToWrite - written_to_buffer;
  Attribute attr(oa_bytesWritten.aID, bytesToWrite);
  activity->attributeArray_.push_back(attr);
  return;
}
```

Listing 7: Handling an incoming write, that is smaler or equal to the cache, but exceeds it becuase of the already filled data in the write-behind plug-in in SIOX

The last case is important in all other scenarios, namely when the current buffer filling combined with the new incoming data will not exceed the maximal size of the buffer or just reaches the maximum. In this case, the current position in the buffer will be calculated, to which the new data can be written and then the data will be copied to this position. Next the new fill position is calculated and tested if it has the maximal size. If this is the case, the cache will be written, by calling the *ourSync* function. Last the return value is written to the activity, so the calling function has the appropriate return value.

```
1  else {
2    char* currBuffPosition = openFiles[filehandle].buffer + openFiles[filehandle].buffer_fill_position;
3    memcpy(currBuffPosition, (void*)dataMemoryAddress, bytesToWrite);
4    openFiles[filehandle].buffer_fill_position += bytesToWrite;
5    if (openFiles[filehandle].buffer_fill_position == config_buffer_size) {
6      ourSync(filehandle);
7    }
8    Attribute attr(oa_bytesWritten.aID, bytesToWrite);
9    activity->attributeArray_.push_back(attr);
10   return;
11 }
```

Listing 8: Handling an incoming write, that will just fill the buffer or stay below the maximal size of the buffer in the write-behind plug-in in SIOX

With this code, the plug in is able to cache the write call, but in Linux, there are a lot more calls that can be invoked to make I/O operations. Like the *pwrite()* or *pwrite64()* call. In difference to the *write()* call, the file pointer is not being moved when *pwrite()* calls occur. Therefore, there are a lot more incoming activities that need to be handled by the plug-in, than just the write call or even all the calls that can be optimized.
In the following, I give a more insight into calls that will not be optimized, but need further handling. Those are the reading calls, in particular the *read()* and *pread()*. They need to access the data, that might be still in the buffer. For distributed files to be accessed correctly, we have a build in functionality, that checks whether the path is for user data, which is a configuration made by the user, or in any other file.
When a read occurs, as can be seen in Listing 9, the data that is cached is synchronized and the file positions, that we maintain are set accordingly. A *read()* access moves the file pointer by the bytes read and for our cache we also need to set the last access position. A *pread()* is working similar, but is not moving the file-pointer therefore, the last two lines from Listing 9 are not invoked in a *pread()* and just the *ourSync* is called.

```
1 OntologyAttributeID fhAttID = facade->lookup_attribute_by_name(interface,"descriptor/filehandle").aID;
2 uint32_t filehandle = activity->findAttribute( fhAttID )->uint32();
3 OntologyAttributeID btwAttID = facade->lookup_attribute_by_name(interface,"quantity/BytesToRead").aID;
4 uint64_t bytesToRead = activity->findAttribute( btwAttID )->uint64();
5 ourSync(filehandle);
6 openFiles[filehandle].last_offset += bytesToRead;
7 openFiles[filehandle].filePointer += bytesToRead;
```

Listing 9: Handling an incoming read in the write-behind plug-in in SIOX

Our plugin also needs to respect the calls of *fsync() and lseek()*. An *fsync()* is calling the sync function the same way the *pread()* is doing and the *lseek()* is similar to the read, because it also moves the file pointer to a new position. The *lseek()* has in addition to the handling like described in Listing 9, an exception handling, when the new position is the same as the position of the last access.
The last access that gets handles by the optimization plug-in is the close in which the cache is written and the struct for the current file handle is removed from the *openFiles* object.

4.4 Modifying SIOX

In this section, the changes, that had to be made to SIOX, for optimization plug-ins to be able to work are explained. In Section 4.4.1 a new function was added to the activity class, to change an existing attribute, without having to implement the traversal of the attribute vector. Section 4.4.2 describes the wrapper, together with Section 4.4.3, which describes the online replay plug-in, which are needed to modify incoming I/O operations.

4.4.1 Mutability of activity attributes

While for trace writing and speed measurement it is not necessary to mutate the activities, when trying to optimize there is the need to be able to change attributes in an activity. This change to the SIOX-Core makes the life of plug-in developers easier, since the activity will be capable of replacing attributes and not anymore the plug-in
For a plug-in in SIOX to be able to optimize a program it needs to be able to mutate attributes in activities. Therefore there was the need for a function in the activity class, that is able to mutate an attribute by its name without the need of plug-ins knowing how the attributes are stored in the activity.

In SIOX attributes are stored in a vector in the activity, therefore we need to iterate over the vector to find the current value of the attribute. Because Attributes are not mutable we need to remove the attribute from the vector and then insert the new attribute, as seen in Listing 10.

```
1  inline void replaceAttribute(const Attribute & replace){
2      for(auto itr=attributeArray_.begin();
3          itr != attributeArray_.end();
4          itr++)
5      {
6          if( itr->id == replace.id ){
7              *itr = replace;
8              return;
9          }
10     }
11     attributeArray_.push_back(replace);
12 }
```

Listing 10: Function to modify an attribute in an activity in SIOX

4.4.2 POSIX-deedless wrapper

The wrapper is handling incoming calls to POSIX functions and is generating an activity from them. In Listing 11 is the instrumentation to the call to the *write()* function. The following list is describing the function calls in Listing 11:

- Lines 3-6 are initializing the times, that are being used to measure the time needed to execute a function call to the file system and measure the overhead introduced by SIOX
- Line 7 is checking if SIOX is activated and initialized. Otherwise the *write()* call is executed in line 46-49
- A write call always has a parent, this parent can belong to the network component or the global component. Global parents are being generated by *open()* calls. Network parents are being generated by *socket()* calls. The determination of the type is done in lines 8-31 with locking were needed. Additionally a new activity is generated with the appropriate component (global or network).
- Lines 33-36 are needed to add the parameters of the *write()* call to the activity.
- Line 41 is sending the activity to the multiplexer and is keeping the activity in memory to get the return value in line 44, when it is available.
- Line 45 is cleaning up the activity and removing it from memory.
- Lines 50-52 are adjusting the global timers for the profiler.
- Line 53 is returning the result of actual *write()* call to the user function.

```
1  ssize_t write(int  fd , const void *  buf , size_t  count ) {
2    ssize_t ret = 0;
3    siox_timestamp t_tmp = 0;
4    siox_timestamp t_fkt_start = 0;
5    siox_timestamp t_start = siox_gettime();
6    siox_timestamp t_overhead = 0;
7    if( siox_monitoring_namespace_deactivated() && global_layer_initialized && siox_is_monitoring_enabled() ){
8      g_rw_lock_reader_lock(& lock_activityHashTable_int);
9      siox_activity_ID * parent = (siox_activity_ID*) g_hash_table_lookup(
10       activityHashTable_int, GINT_TO_POINTER(fd) );
11     g_rw_lock_reader_unlock(& lock_activityHashTable_int);
12     // now decide to which component the activity parent belongs to
13     // we expect it is likely to belong to the first component
14     siox_activity * sioxActivity = NULL;
15
16     if ( parent == NULL ){
17     // check if it belongs to the other component
18     g_rw_lock_reader_lock(& lock_activityHashTable_network_int);
19     parent = (siox_activity_ID*) g_hash_table_lookup(
20       activityHashTable_network_int, GINT_TO_POINTER(fd) );
21     g_rw_lock_reader_unlock(& lock_activityHashTable_network_int);
22
23     if( parent != NULL ){
24       sioxActivity = siox_activity_begin( network_component, write_network );
25     }else{
26       // unknown so we keep the first component
27       sioxActivity = siox_activity_begin( global_component, write_global );
28     }
29   }else{
30     sioxActivity = siox_activity_begin( global_component, write_global );
31   }
32
33   siox_activity_link_to_parent( sioxActivity, parent );
34   siox_activity_set_attribute( sioxActivity, fileHandle, &fd );
35   siox_activity_set_attribute( sioxActivity, memoryAddress, &buf );
36   siox_activity_set_attribute( sioxActivity, bytesToWrite, &count );
37   t_tmp = siox_activity_start(sioxActivity);
38   t_overhead = t_tmp - t_start;
39   t_fkt_start = t_tmp;
40   t_tmp = siox_activity_stop( sioxActivity );
41   siox_activity_end_keep( sioxActivity );
42   t_start = t_tmp;
43   t_fkt_write += t_tmp - t_fkt_start;
44   siox_activity_get_attribute( sioxActivity, bytesWritten, &ret );
45   siox_activity_kept_delete( sioxActivity );
46   }else{
47     if(initialized_dlsym == 0) sioxSymbolInit();
48     ret = (__real_write)(fd, buf, count);
49   }
50   calls_write++;
51   t_overhead += siox_gettime() - t_start;
52   t_overhead_write += t_overhead;
53   return (int)ret;
54 }
```

Listing 11: SIOX POSIX-deedless wrapper, function to instrument a write call

4.4.3 OnlineReplay plug-in

The *OnlineReplay* plug-in is used to execute the incoming activities and save the return values in the activities. The usage of the online path in SIOX, requires the use of the POSIX-deedless wrapper from Section 4.4.2.
In the following list is the description of the execution of a write, as seen in Listing 12, when an activity is received by the *OnlineReplay* plug-in:

- In line 1 is the determination, if the activity is a write call.
- Lines 2-5 initialize the needed variables to call the *write()* function.

- Lines 6-8 retrieve the saved parameters from the activity and save them to the appropriate variables
- The call of the *write()* function is in line 10.
- Error handling is done in lines 11-14 where SIOX is printing a message to console with the name of the plug-in and the message of the error.
- Lines 15 and 16 are saving the return value from the *write()* call to the activity.

This chapter gave an overview of the implemented functions. The next chapter is evaluating the implemented optimization strategies with different benchmarks.

```
1  if (activity->ucaid() == posix_write) {
2    int ret;
3    int fd, errnum;
4    void *buf;
5    size_t count;
6    count = getActivityAttributeValueByName(activity, SUB_bytesToWrite).SUB_CAST_bytesToWrite();
7    fd = getActivityAttributeValueByName(activity, SUB_fileHandle).SUB_CAST_fileHandle();
8    buf = (void*) getActivityAttributeValueByName(activity, SUB_memoryAddress).SUB_CAST_memoryAddress();
9
10   ret = write(fd, buf, count);
11   errnum = errno;
12   if (ret < 0) {
13     printf("ReplayOnline(%d): write failed: %d, message: %s\n", __LINE__, errnum, strerror( errnum ));
14   }
15   Attribute attr(oa_bytesWritten.aID, convert_attribute(oa_bytesWritten, &ret));
16   activity->attributeArray_.push_back(attr);
17 }
```

Listing 12: SIOX OnlineReplay plug-in, function to execute a write activity

5 Evaluation

Section 5.1 is an introduction to the evaluation. Section 5.2 is describing the cluster used to evaluate the optimization strategies. Section 5.3 is introducing the benchmarks used in the evaluation. Section 5.4 gives an overview of the different configurations used with the different benchmarks. Section 5.5 is evaluating the results.

5.1 Method

For measuring the performance gain, by using the implemented optimization strategies, it is necessary to measure the performance of the system and the overhead, by using the systems.
When using FUSE as the file system, the performance gets lost with the switching between user and kernel space. Using SIOX the performance is lost by the modules, that are interacting with the activites, which go through the system. Therefore, it is necessary to measure the Overhead on a memory file system, because there the file system is not introducing new overhead.
When evaluating optimizations, the performance loss by the systems has to be taken into account. The performance without the system and the performance after the optimization has to be compared.

5.2 Test System

This section is describing the used cluster and is analyzing the maximum theoretical performance.

5.2.1 System description

The test-system used, is a cluster seen in Figure 11, that is composed of 21 nodes, from which we will be using the 10 Westmere nodes. Each node has a network connection of two time 1GBit-Ethernet. Each node also has one internal hard drive with a speed of roughly 100 MiByte/s. The network-speed has a maximum throughput of 127MiByte/s per interface.
In addition to the local drive, each node is connected to the Lustre file-system, offered by the Sandy nodes, and has a local memory backed file-system mounted at /dev/shm. The Westmere nodes have two Intel Xeon X5650 CPU with 6 cores each and HT enabled. They have a speed of 2.67GHz.

The Cluster has a lustre file system available which is connected with a GiB/s Ethernet connection and consists of 10 OSSs with one OST each and one MDS for the system. The clients utilize a write-behind cache and therefore cache a certain amount of write data before actually writing the data to disk. This leads to a better performance in writing, but can impact read speeds.

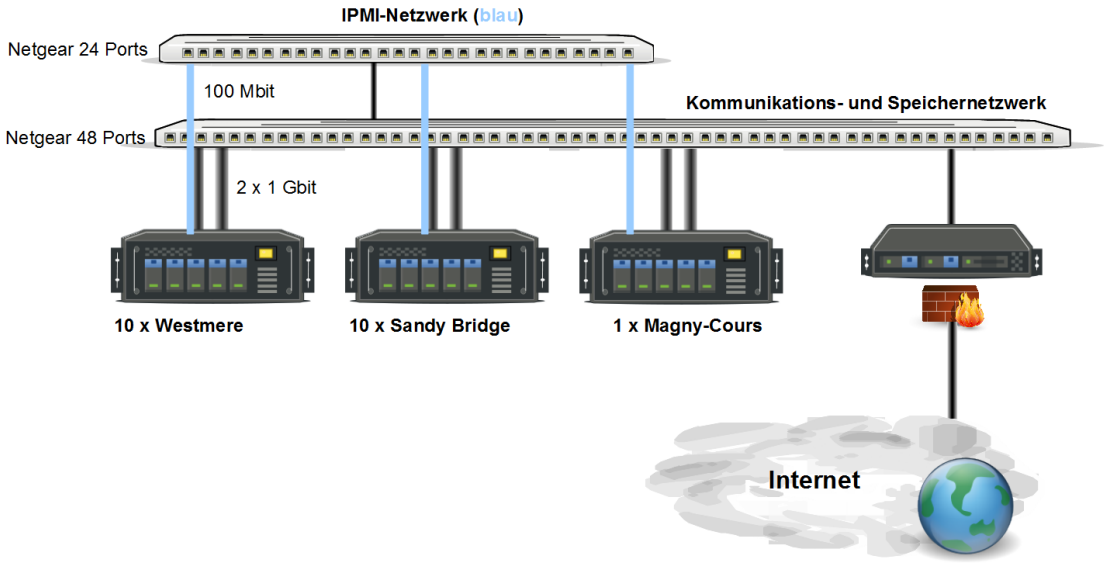


Figure 11: Cluster at the group Scientific Computing from the Department Informatics at UHH [Ham17]

5.2.2 Theoretical System Analysis

When using a system for testing, it is important to know what performance is achievable by the system on a theoretical basis. In the following, I am analyzing the performance for the access to the Lustre file-system.

Network When accessing Lustre all call have to go through the network connection. Therefore it is crucial to have network, that is capable of speeds above the speed of the hard drive, so that the bottleneck is not at the network layer.
The important measures for the network are the round-trip-time, calculated from this measure the messages per second, and throughput. The messages per second can be calculated by Equation (1). The round-trip-time was measured, using ping as root with the Adaptive Ping option.

$$messages\ per\ second = \frac{1}{round - trip - time} = \frac{1}{0.1ms} = 10msg/ms = 10000msg/s \tag{1}$$

The throughput is a fixed value, that can be taken from the interconnect used in the cluster. In the cluster that will be used in the testing for this thesis there is a gigabit Ethernet interconnect used. This leads to a maximum data rate of 1GiB/s which is equal to about 117MiB/s.

Storage Device The storage devices in the OSSs from the used Lustre are HDDs which have a typical IOPS (input output operations per second) value of about 150, which correlates with the average latency of 7ms, and a data rate of about 120MiB/s. For a rotating disk to respond to a request the disk has to first rotate to the required position and then read or write the data from or to the disk. The time for this action is called the access time and has a typical value, for rotating disks with a speed of 10k rotations per minute, of about 3.5ms.

Calculating the maximum data rate for Lustre When analyzing the data rate, there is a huge impact from the access sizes of the requests to the file system. When accessing the file-system with a large request, at best the stripe size of Lustre, the limiting factor of the access speed is the network's maximum throughput, as can be seen in Figure 12. When accessing the file system with small block sizes, the limiting factor is the latency of the network. This behavior can be read from the Figure. The maximum throughput is small for small block sizes and is growing rapidly with an increase in block size, until it slowly approaches the maximum throughput of the network.

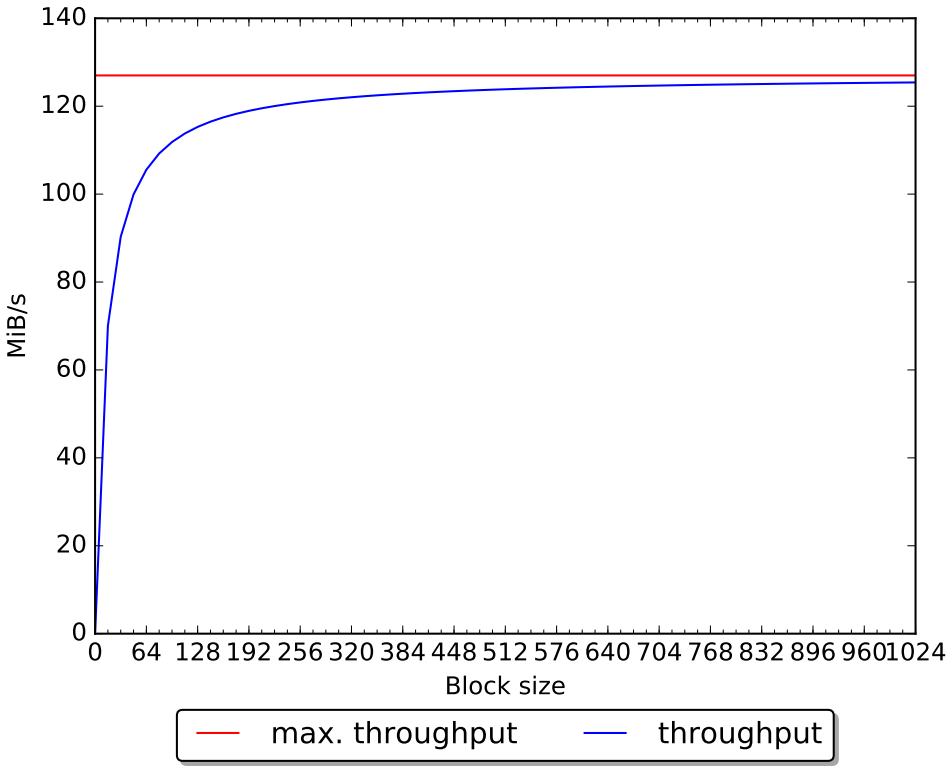


Figure 12: Theoretical bandwidth with varying access sizes

5.3 Benchmarks

For testing what performance a HPC-System can do, there exists a variety of benchmarks. In the terms of I/O-Benchmarks there will be an introduction of various Benchmarks we used to evaluate the performance gains the optimization strategies from Section 3 provide us with.

5.3.1 IOR

IOR is a benchmark for analyzing the performance of parallel file-systems using the commonly used interfaces POSIX, MPIIO and HDF5. It offers a variety of options for changing the access pattern to once needs. It generates a synthetic access pattern and measures the performance while it is running by timing the execution and from the input parameters, knowing the aggregate file-size, calculating the speeds that could be achieved. At the end of each run, the read and write speeds and respectively the times and aggregate file-size, are being output to console.

5.3.2 MACSio

MACSio is a performance I/O benchmark [mac], its acronym represents Multi-purpose, Application-Centric, Scalable I/O Proxy Application. It was developed at the Lawrence Livermore National Laboratory with the need of a benchmark that operates at a high abstraction layer, as can be seen in Figure 13 and has a new degree of freedom for the users.

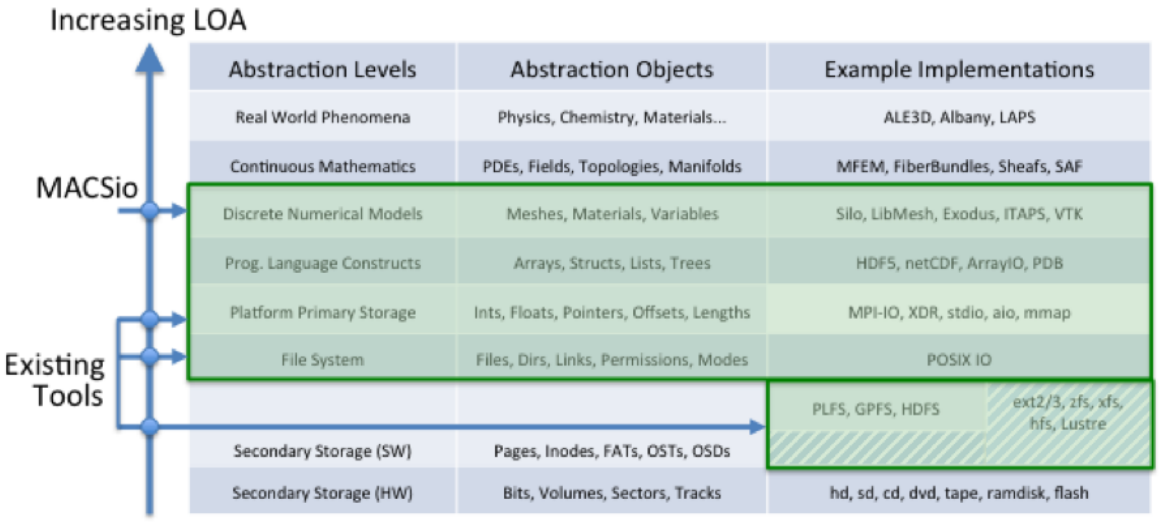


Figure 13: Level of abstraction of MACSio [Lab17]

5.3.3 NetCDF-bench

NetCDF-Bench is a tool for measuring the read and write speeds for the access of NetCDF files. It was developed by the DKRZ for the need to optimize file access to NetCDF-Files and to test those optimizations.
In this thesis I will also be using this tool to measure Access speeds to NetCDF-data, because this is one of the major type of Files used in Climate-Science.

5.4 Configurations

All measurements are run on the cluster with an exclusive use of the nodes. The file system is a shared resource and it can therefore not be enforced, that only the tests are running. On startup of the tests, a script made sure, that no other node of the cluster was assigned to a different job. Therefore, the Lustre file system was most likely been used only by the benchmarks.
The Tests were started with different parameters and configurations of nodes. The following sections are describing which configurations were used for the different benchmarks in general. The specific configuration for the tests are explained in the beginning of the evaluation for each test scenario.

5.4.1 IOR

The configuration parameters of IOR determine the block size and aggregate file size of the run. To have a stable result, the test should be running for a couple of minutes. The block sizes used were 16KByte, 256KByte or 1MByte for all tests. The number of nodes was varied between one and 10 nodes, which is the maximum number in the cluster.
IOR supports multiple APIs. For the tests were MPIIO and HDF5 used.
The tests were performed on a single-shared file. Accessing the file was with random offset, sequential and sequential with caching disabled.

5.4.2 MACSio

The MACSio benchmark is writing double values to a file. The amount of values written in each write can be determined by a parameter. The goal is to have a block size of 16KByte and 1MByte for each access. The parameter is therefore set accordingly. The number of nodes is varied from one to 10 nodes (1, 3, 5, 8, 10) and on each node the number of processes is varied from one to 12.

5.4.3 NetCDF-bench

NetCDF is using multi dimensional arrays to access its data. For this test the arrays are chosen in a way, that block sizes of 16KByte and 1MByte are accessed.
The two measures of the access are the block geometry and the data geometry both have four variables. Those are the time and x, y and z axis. The block geometry defines the block size. For the test with a block size of 16KByte, it is chosen to be one time step and a three dimensional array with the values 16:32:4 for x:y:z components. The access of 1MByte has one time step and 32:256:16 for x:y:z components.
With an increase in processes, the data geometry is being increases. The total amount of data accessed is increased, but the access size is kept constant. Every process is supposed to access 100 time steps in total and multiple times the array of x, y and z components.
The access type is independent I/O and the target of access is the Lustre file system.

5.5 Evaluation

Using the configurations described in Section 5.4, the different runs are being evaluated in the following sections. The first evaluation is about the overhead introduced, by using FUSE or SIOX as a system to implement the optimization strategies.

5.5.1 Overhead

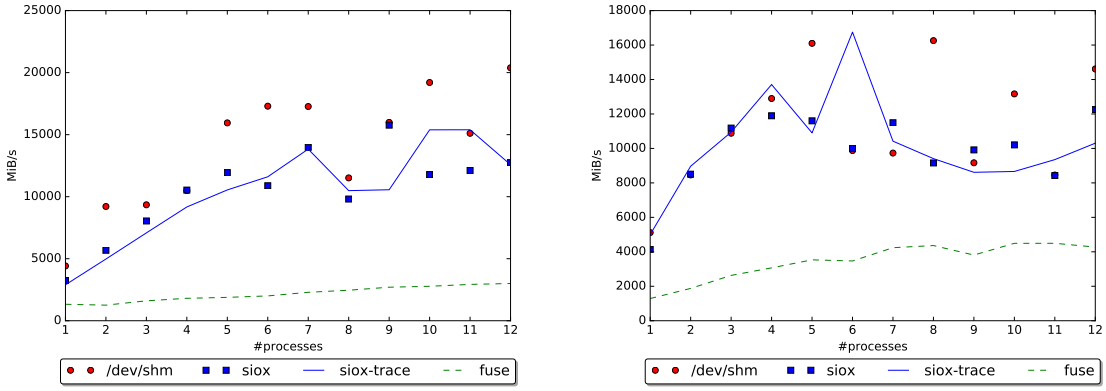
These first runs will be performed on /dev/shm which is a memory backed file-system and therefore eliminates the wait-cycles for any physical drive. The following charts show the read and write-speeds on /dev/shm with different block-sizes and with different configurations of SIOX and FUSE. For comparison the speeds of accessing /dev/shm without either of those systems is in the figures as well. The benchmark used in this test is IOR, one node of the cluster is utilized and the memory backed file-system on /dev/shm is the target of I/O operations. The block-size differs between the two tests, the first one has access blocks of 16KByte and the second one 1MByte. The aggregate file-size of all accesses is 2.38GiB, which leads to the number of accesses per process, which can be calculated by 2.38GiB/1MByte.
To keep the aggregate file-size constant the number of segments differ according to the number of processes utilized.

Observations:

- While reading or writing with a small block-size, FUSE is by magnitudes slower than SIOX or base access. (Figure 14a and Figure 15a)
- Accessing /dev/shm with a large block size of 1M shows a better performance in all three systems. The most benefit has FUSE, this is coming from less switching

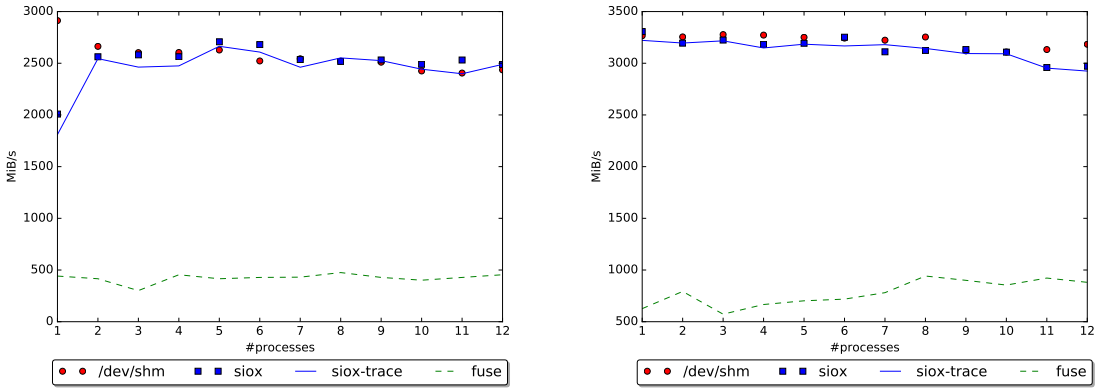
between Kernel and User-space, which is taking some time and is limited by the performance of the hardware. (Figure 14b and Figure 15b)

- When comparing SIOX with and without trace writing, the performance is varying in the margin of error, they can be assumed as having the same performance.
- There are quite some fluctuations in speeds, but at this high level of performance every little access to the memory by the system itself can influence the performance.



a) block size of 16KByte **b)** block size of 1MByte

Figure 14: read-speeds on /dev/shm



a) block size of 16KByte **b)** block size of 1MByte

Figure 15: write-speeds on /dev/shm

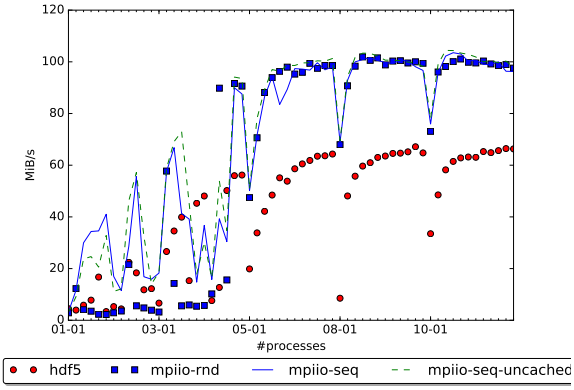
In the next test, the overhead of SIOX for different APIs is being evaluated. The goal is to have a understanding, in which measure SIOX influences the execution.
The test is going to be executed on the NFS share of the cluster and different APIs are used. SIOX is mostly used as a tracing software and therefore tracing will be enabled and written in the same file system the test data is written. In this way, the usual

performance loss with SIOX can be evaluated.
The access was handled by HDF5 and MPIIO, while in MPIIO multiple access patterns were generated, namely a random access, a sequential access and for the read tests a sequential access without caching. IOR was run on one, three, five and ten nodes and with one to twelve processes per node. The aggregate file-size was kept constant for every test, so that with an increase in processes the number of segments is being reduced. The two test scenarios differ by the block and transfer-size. The first scenario has a block and transfer size of 16kByte, the second one has a block size of 256kByte and a transfer-size of 1MByte.
Observations for the read access as seen in Figure 16:

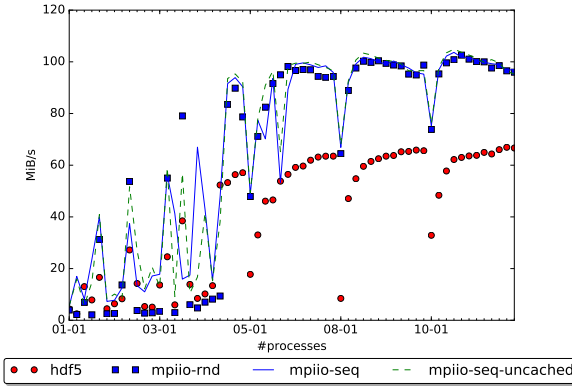
- For MPIIO and a sequential pattern, a small block-size has an advantage over a larger one.
- The random access with MPIIO is faster with a larger block-size for a small number of processes and nodes, but with an increase in the number of nodes, a smaller block-size leads to a better performance.
- HDF5 gains speed with an increase in processes. for only one node the larger block-size is faster but starting with more nodes, the smaller block-size increases performance.
- An increase in processes per node, leads to a better performance in speed neglect able the block-size.
- The performance for MPIIO starting from five nodes onward and the small block-size is being capped by the performance of the NFS-share.
- Even though SIOX is writing it's traces to the same file-system the test-data is read from, the performance is not notable lower than without SIOX.

Observations for the write access as seen in Figure 17:

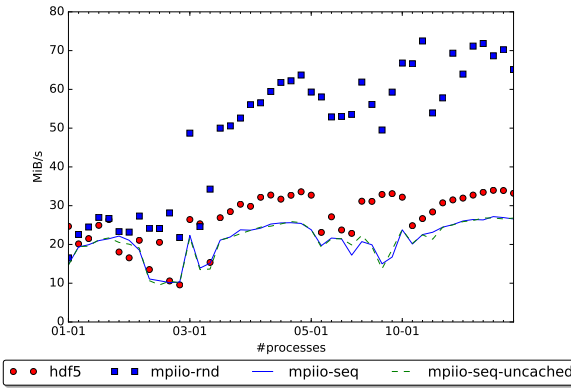
- The larger block-size has a better performance for all numbers of nodes and processes per node.
- Sequential writes have the best performance in all tests and are capped by the performance of the file-system while using 10 nodes.
- Random writes with a small block-size and MPIIO, while using only one or three nodes, is by magnitude slower than every other test. With a larger block-size the performance is better and closer to the sequential writes.
- Using HDF5 while writing is only a little slower than MPIIO and is again capped by the file-system when using a large block-size and 10 nodes.



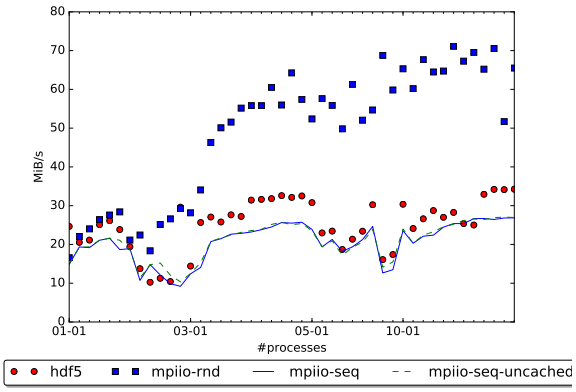
a) Block size of 16KByte without SIOX enabled



b) Block size of 16KByte with SIOX enabled



c) Block size of 256KByte without SIOX enabled



d) Block size of 256KByte with SIOX enabled

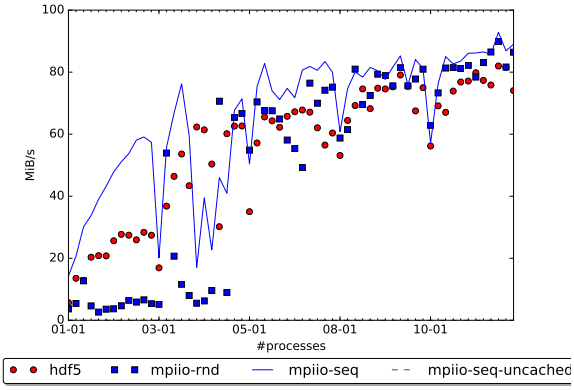
Figure 16: Read speeds on NFS

- SIOX is again not lowering te performance of the system even though it is writing traces to the same file-system.

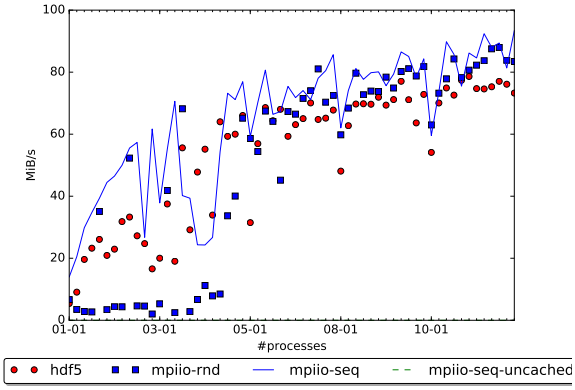
5.5.2 Seek free

In an other test suite we tested the influence of optimizations to a stream of data. Our goal was to measure which amount of overhead SIOX or FUSE introduce, when we adding plug-ins. The first optimization was to remove any seeking overhead, that would be added while writing randomly to disk. Therefore our plug-in analyzes the data stream and removes any seeking. This optimization can not be applied in a real world scenario, because any data would be destroyed when written to the wrong places in a file.

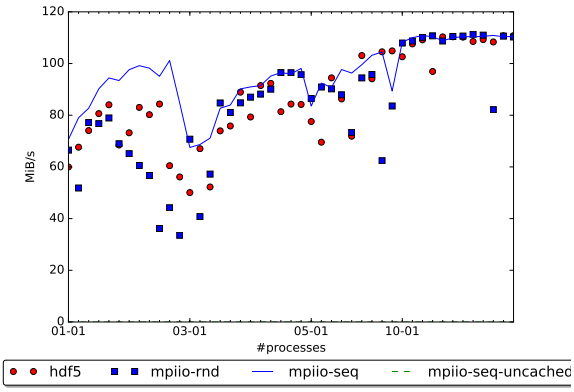
The testing was performed as write tests only. All tests are run on the Lustre file system. The target file was a single-shared file and the benchmark is writing with a random access to the file only. Block sizes used in the two scenarios are 16 kByte and



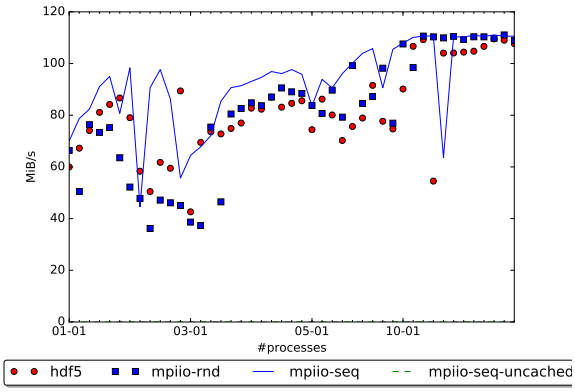
a) Block size of 16KByte without SIOX enabled



b) Block size of 16KByte with SIOX enabled



c) Block size of 256KByte without SIOX enabled



d) Block size of 256KByte with SIOX enabled

Figure 17: Write speeds on NFS

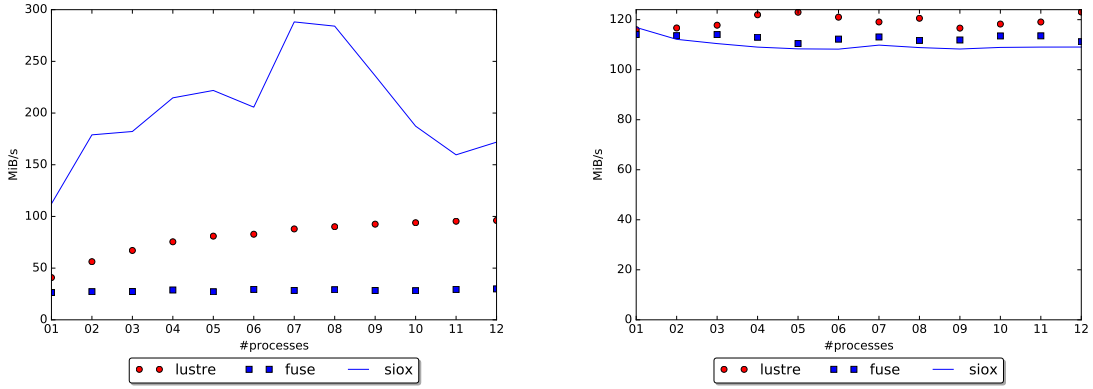
1 MByte.

Observations from Figure 18:

- Using a small block size results in less throughput.
- Writing to Lustre without optimization has a higher throughput, than optimized writes using fuse.
- Using SIOX to optimize the writes, the throughput using one process is close to the theoretical performance maximum of large block sizes, as seen in Figure 12
- When using SIOX the throughput of multiple processes is higher, than the network throughput. This results from caching mechanisms, because all processes write to the start of the file.
- Block size of 1MByte is at the maximum throughput, when writing to Lustre without optimization.

- The large block size is not suffering from the additional processing done by SIOX or FUSE.

What we learn from such a plug-in is, what can be achieved in an optimal case.



a) block size of 16KByte **b)** block size of 1MByte

Figure 18: write speeds on Lustre without seeking

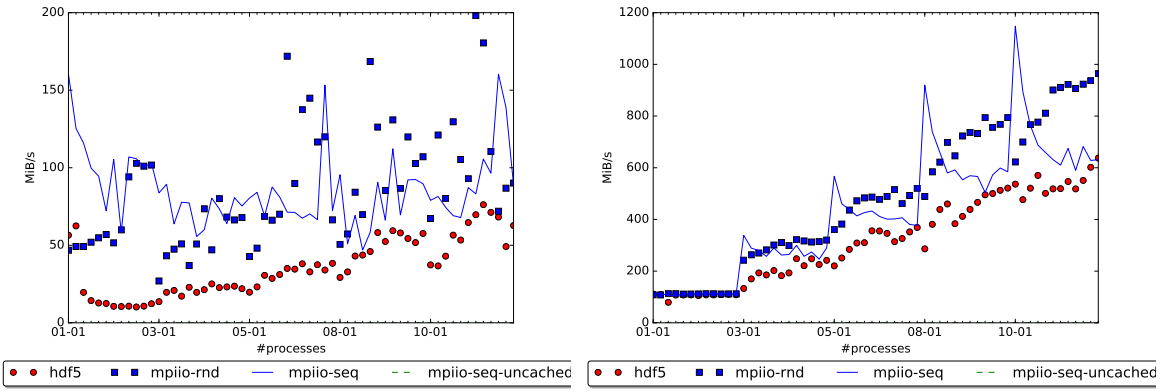
5.5.3 Write-back with IOR

The test from Figure 19 shows what impact the block-size has on writing speeds. The parameters were the cluster with lustre as described in Section 5.2.1. While initializing the test a folder, which was striped over a specific amount of OSSs, was created, so that each node, that was involved in the current run would write to a dedicated OSS, to maximize performance.
The system was writing 6GBiByte for the large and 15 GiByte for the small block-size Observations:

- When using a small block-size, the system has a steady write performance, which is not expected, as there is an increase in nodes and therefore network performance. This leads to the conclusion, that the amount of network packets that is needed for the small writes, are slowing down the overall performance of the system.

- Looking at the large blocks of 1 MByte, the write performance is increasing with the number of nodes, that gets added to the system. While one process per node gets the best performance, the decrease is reasonable for a small number of nodes. When the node count increases, the performance drops by nearly 50% comparing one Process per node and multiple processes per node, that are writing to the lustre file-system

- Comparing sequential and random writes with mpiio, the random writes are slightly faster when using multiple processes per node, this is due to lustre utilizing the write-back-cache and therefore optimizing the writes.



a) Block size of 16KByte **b)** Block size of 1MByte

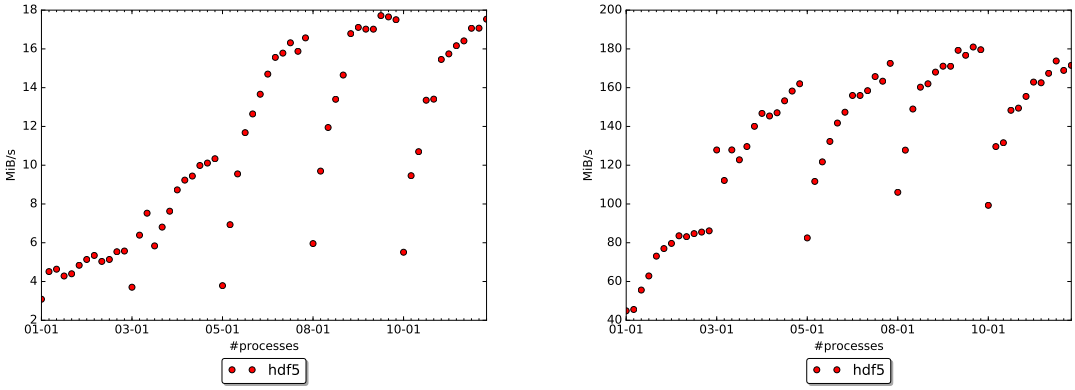
Figure 19: Write speeds on LUSTRE with different block sizes

5.5.4 Write-back with MACSio

The tests were run with one file per process and constant number of files, with the goal of keeping the overall written size of the runs constant. Therefore with an increase in processes used, the number files that ware written per process decreased. This means, that for the small accesses a total of 1.04GiByte was written to file and for the larger accesses, a total of 23GiByte was written to the file system. As an interface, HDF5 was chosen. The results are visualized in Figure 20 Observations:

- The performance of small writes is very low, this is due to the way MACSio is writing, as will be described below.
- Larger writes increase the performance by a factor of 10, but cannot get close to the system performance
- With an increase in processes per node, the performance increases almost linear.
- An increase in nodes does not lead to a better performance, except when increasing from one to three nodes.

MACSio is writing every call of write in a separate file, therefore a large access size is decreasing the overhead, which is introduced by opening and closing files this can be seen in Listing 13 line 4 and 10. The optimization that is trying to increase the access size, can not get a better performance. It just introduces new overhead by interfering



a) Block size of 16KByte **b)** Block size of 1MByte

Figure 20: Write speeds on LUSTRE using MACSio with different block sizes

with the open and creating an object in which the buffer and other needed variables have to be initialized. When the write gets written to the buffer, a close will flush this buffer immediately and remove the object, which in turn is again introducing overhead to the system.

For this kind of access patter a different type of optimization would be needed, which could buffer the write calls beyond the close and combine them to a single file.

```
1  ================================================================
2  SIOX Reports
3  ================================================================
4  POSIX/calls/close = 30007
5  POSIX/calls/fileno = 30012
6  POSIX/calls/flock = 30000
7  POSIX/calls/ftruncate = 2
8  POSIX/calls/lseek = 420000
9  POSIX/calls/open = 60006
10 POSIX/calls/pwrite = 30003
11 POSIX/calls/read = 10
12 POSIX/calls/setbuf = 1
13 POSIX/calls/unlink = 1
14 POSIX/calls/write = 11
```

Listing 13: SIOX Report for the MACSio run with one process on one node, only POSIX, calls no times

5.5.5 Write-back with NetCDF

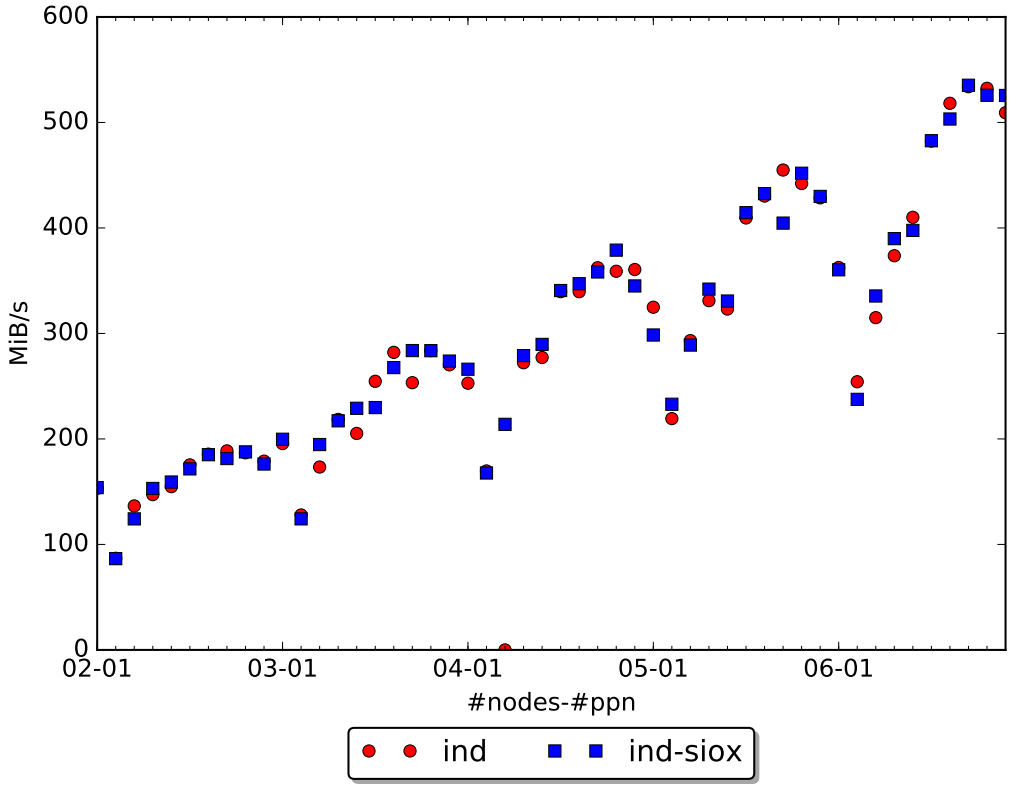
One major type of Files used in Climate science is NetCDF data. Therefore analyzing and optimizing its access speed is being discussed.

The first test is the comparison between the speed of the base access to a NetCDF file and the speed when the base SIOX system is logging the trace files. This already

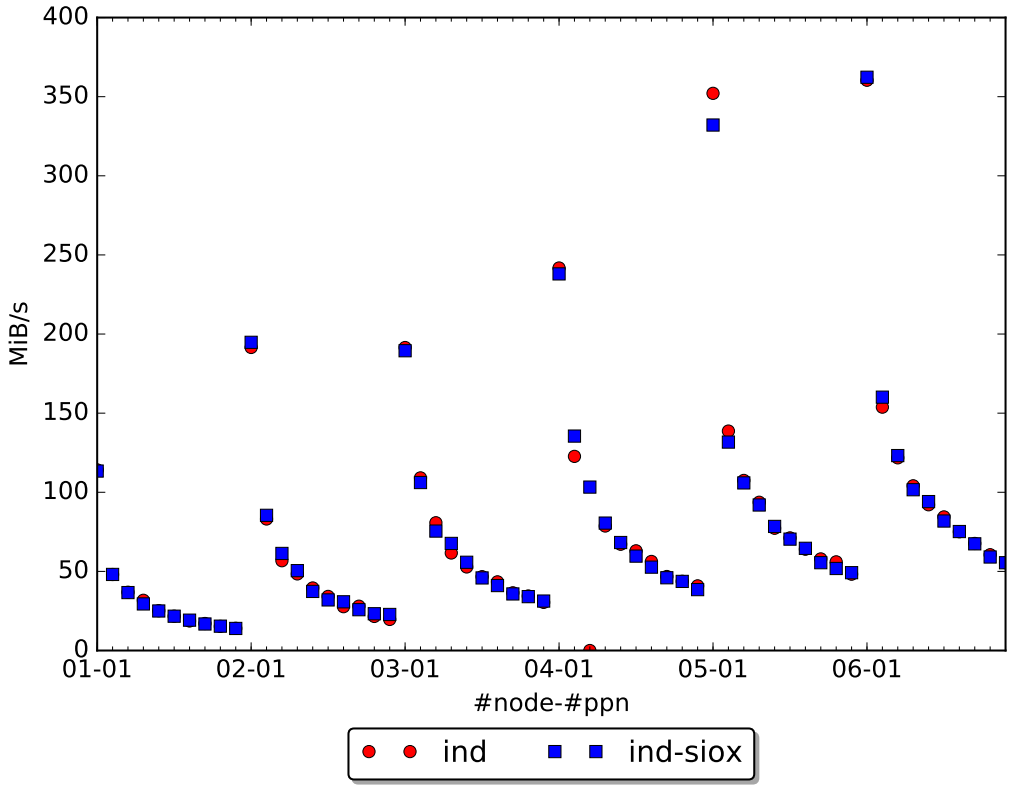
shows, that SIOX is not introducing overhead to the measured speeds. All values are within the same range.
Observations from Figure 21

- Read-speeds are increasing with the number of nodes and number of processes per node. The total number of processes have a higher impact on the total read-speed, than the number of nodes.
- Read-speeds are not hitting the theoretical maximum, that was calculated in Section 5.2.2
- Write speeds with one process per node have the highest througput, even though it is not at the theoretical maximum.
- Write speeds are decreasing with the number of processes.

The observations from the tests lead to the conclusion, that the optimization strategies can not improve the throughput as much as a strategy, that would combine the accesses to one node, could. When working with NetCDF files, using one Process per Machine to access the file and later distribute it locally with OpenMPI-Threads or MPI, leads to a better performance, than letting each process read its own file.



a) Read test



b) Write test

Figure 21: Accessing NetCDF files with a block size of 1MByte

6 Summary and Conclusion

This chapter gives a summary over the achievements and results that could be made over the course of this thesis. And will also introduce future work that is still to be done.

This thesis analyzed the performance of a cluster in a practical way using synthetic benchmarks, primary IOR but also MACSio and netcdf-bench. With the results gathered from runs with different parameters of the tools and using SIOX to analyze the runs.
Using SIOX to optimize runs of the benchmarks had a greater performance gain, than using FUSE. This is due to the overhead introduced with FUSE, which could not be removed with optimizations.

6.1 Future Work

While I implemented a basic optimization strategy and improved SIOX ability to interfere with the stream of I/O operations, a lot more work has to be done to improve SIOX, to instrument all major I/O calls and process them correctly.
A list of work, that could be added to extend the library of optimizations, that was demonstrated in this thesis:

- Extend the list of optimizations by finding new appropriate one for different problems.
- Classify those optimizations.
- Implementation of the POSIX-deedless wrapper has to be finished, to interfere with more POSIX calls and the *OnlineReplay* plug-in has to be modified accordingly.

The optimizations from this thesis were only a demonstration, to have a variety in terms of classifications. The implementation of more optimization strategies, which can then be added to the library could be done. This can help improve programs and save a lot of work, by testing the optimizations first and then implementing the best directly to the programs of scientist.

References

[CRC17] https://www.rc.colorado.edu/sites/default/files/images/figure2.png

[dar] *Welcome to the Darshan project.* http://www.mcs.anl.gov/research/projects/darshan/2009/07/31/welcome-to-the-darshan-project/

[Ham17] Hamburg, Scientific C. o.: *Physische Sicht.* https://wr.informatik.uni-hamburg.de/_detail/teaching/ressourcen/physische-sicht.png. Version: 04 2017

[HKK+10] Howison, Mark ; Koziol, Quincey ; Knaak, David ; Mainzer, John ; Shalf, John: Tuning hdf5 for lustre file systems. In: *In Workshop on Interfaces and Abstractions for Scientific Data Storage*, 2010

[JBLO16] Jones, Matthew ; Blower, Jon ; Lawrence, Bryan ; Osprey, Annette: Investigating Read Performance of Python and NetCDF When Using HPC Parallel Filesystems. In: *International Conference on High Performance Computing* Springer, 2016, S. 153–168

[KB15] Koziol, Quincey ; Breitenfeld, Scot: *A Brief Introduction to Parallel HDF5.* 2015

[KZH+14] Kunkel, Julian ; Zimmer, Michaela ; Hübbe, Nathanael ; Aguilera, Alvaro ; Mickler, Holger ; Wang, Xuan ; Chut, Andrij ; Bönisch, Thomas ; Lüttgau, Jakob ; Michel, Roman ; Weging, Johann: The SIOX Architecture - Coupling Automatic Monitoring and Optimization of Parallel I/O. In: Kunkel, Julian (Hrsg.) ; Ludwig, Thomas (Hrsg.) ; Meuer, Hans (Hrsg.) ; ISC events (Veranst.): *Supercomputing* ISC events, Springer International Publishing, 2014 (Supercomputing). – ISBN 978–3–319–07517–4, S. 245–260

[Lü14] Lüttgau, Jakob: *Flexible Event Imitation Engine for Parallel Workloads.* Online https://wr.informatik.uni-hamburg.de/_media/research:theses:jakob_l__ttgau_flexible_event_imitation_engine_for_parallel_workloads.pdf, 03 2014

[Lab17] Laboratory, Lawrence Livermore N.: *Level of Abstarction of MACSio.* https://codesign.llnl.gov/images/macsio_loa.png. Version: 02 2017

[LCC+12] Liu, Ning ; Cope, Jason ; Carns, Philip H. ; Carothers, Christopher D. ; Ross, Robert B. ; Grider, Gary ; Crume, Adam ; Maltzahn, Carlos: On the role of burst buffers in leadership-class storage systems. In: *MSST*, 2012

[lus17] *Lustre file system components in a basic cluster.* http://doc.lustre.org/figures/Basic_Cluster.png. Version: 01 2017

[mac] *MACSio: A Multi-purpose, Application-Centric, Scalable I/O proxy application.* https://codesign.llnl.gov/macsio.php

[NS03] Nethercote, Nicholas ; Seward, Julian: Valgrind. In: *Electronic Notes in Theoretical Computer Science* 89 (2003), Nr. 2, 44 - 66. http://dx.doi.org/http://dx.doi.org/10.1016/S1571-0661(04)81042-9. – DOI http://dx.doi.org/10.1016/S1571–0661(04)81042–9. – ISSN 1571–0661

[Uni17] Unidata: *What is Unidata.* https://www.unidata.ucar.edu/about/tour/index.html. Version: 04 2017

List of Figures

List of Tables

List of Listings

Eidesstattliche Erklärung

Eidesstattliche Erklärung zur Master-Arbeit

Ich versichere, die von mir vorgelegte Arbeit selbstständig verfasst zu haben. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder nicht veröffentlichten Arbeiten anderer entnommen sind, habe ich als entnommen kenntlich gemacht. Sämtliche Quellen und Hilfsmittel, die ich für die Arbeit benutzt habe, sind angegeben. Die Arbeit hat mit gleichem Inhalt bzw. in wesentlichen Teilen noch keiner anderen Prüfungsbehörde vorgelegen.

$Unterschrift$: $Ort, Datum$: