Analyzing File Create Performance in IBM Spectrum Scale

Marc-André Vef

March 18, 2016

Johannes Gutenberg University Mainz

Faculty for Physics, Mathematics and Computer Science
Efficient Computing and Storage Group

Master's Thesis

Analyzing File Create Performance in IBM Spectrum Scale

Marc-André Vef

1. Reviewer	Prof. Dr.-Ing. André Brinkmann Efficient Computing and Storage Johannes Gutenberg University Mainz
2. Reviewer	Prof. Dr. Miguel Andrade Computational Biology and Data Mining Johannes Gutenberg University Mainz
Supervisors	Dr. Vasily Tarasov and Dr. Dean Hildebrand

March 18, 2016

Marc-André Vef
Analyzing File Create Performance in IBM Spectrum Scale
Master’s Thesis, March 18, 2016
Reviewers: Prof. Dr.-Ing. André Brinkmann and Prof. Dr. Miguel Andrade
Supervisors: Dr. Vasily Tarasov and Dr. Dean Hildebrand

Johannes Gutenberg University Mainz

Efficient Computing and Storage Group
Faculty for Physics, Mathematics and Computer Science
Anselm Franz von Bentzel-Weg 12
55128 Mainz, Germany

Abstract

It is known that metadata operations, such as file creation, do not scale well in modern parallel file systems with an increasing number of nodes in *High Performance Computing* (HPC) clusters. Specifically, many scientific applications require fast concurrent file creation in a single directory, which will become more important in the future when used with larger environments. The aim of this study is to investigate file create performance in IBM's *Spectrum Scale* parallel file system, formerly known as *General Parallel File System* (GPFS), in order to understand its complex file create process and the corresponding bottlenecks. We investigate the process by modifying the popular *mdtest* metadata benchmark to reveal the per-process create performance at any time during concurrent file creation and by investigating Spectrum Scale's source code and trace records. This allowed us to understand the impact of various internal mechanisms and to divide the file create process into logical tasks. The generated latency distributions showed the percentage of time consumed by each task in respect to the total time. Overall, we are able to locate those tasks that consume the most time during concurrent file creation on our test cluster. However, with different hardware available, such as a faster network, a task's contribution to the total time may differ. Therefore, file creation should be further investigated for different hardware with the tools provided by this study which will be continuously developed in the future.

German abstract

Es ist bekannt, dass Metadaten-Operationen wie z.B. Dateierstellungen, nicht mit der Anzahl der Knoten in Hochleistungsrechnern skalieren. Viele wissenschaftliche Anwendungen benötigen jedoch schnelle parallele Dateierstellungen in einem einzigen Verzeichnis – ein Fakt, der in den kommenden Jahren, hinsichtlich der steigenden Knotenanzahl in Großrechnern, immer mehr an Bedeutung erlangen wird. In dieser Arbeit wird die Erstellung einer Datei in *Spectrum Scale*, einem parallelen Dateisystem von IBM, das ehemals unter dem Namen *General Parallel File System* (GPFS) bekannt war, untersucht, um den komplexen Ablauf und die Skalierungsprobleme nachvollziehen zu können. Zunächst einmal wurde die bekannte Metadaten-Benchmark-Anwendung *mdtest* modifiziert, um einen Einblick in das Verhalten jedes Prozesses zu jedem beliebigen Zeitpunkt während einer parallelen Dateierstellung zu erhalten. Ferner wurde der Quellcode von Spectrum Scale und die dazugehörigen Trace-Einträge analysiert. Diese wurden dazu verwendet die Auswirkungen verschiedener interner Mechanismen zu untersuchen und den Dateierstellungsablauf in mehrere Abschnitte zu unterteilen. Für jeden Abschnitt wurden Latenzverteilungen erstellt, die darstellen welcher Abschnitt die meiste Zeit benötigt. Insgesamt sind wir dazu in der Lage, jene Abschnitte aufzuzeigen, die im parallelen Dateierstellungsprozess in unserem Test-Hochleistungsrechner für den Hauptzeitverbrauch verantwortlich sind. Allerdings können diese Abschnitte abhängig von der Hardware, z.B. bei einem schnelleren Netzwerk, in unterschiedlichen Zeiten differieren. Der Dateierstellungsablauf sollte somit mit den vorgestellten Werkzeugen, die in Zukunft weiterentwickelt werden, für abweichende Hardware wiederholt werden.

Acknowledgments

First of all, I would like to thank my advisor Professor Dr.-Ing. André Brinkmann for the exciting topic of this thesis, his support, the great discussions and for his valuable input throughout the thesis that steered the project into the right direction, while allowing me the freedom of pursing many new leads that arose during my research. I want to thank him for his help, not only associated with this work, and especially for introducing me to the interesting field of storage and systems that ultimately led to the decision of facing the challenges of this thesis.

I would also like to acknowledge Professor Dr. Miguel Andrade as the second reader of this thesis and I am thankful for his valuable comments.

I wish to express my most sincere gratitude and appreciation to my supervisor, Dr. Vasily Tarasov with IBM, for not only mentoring me through the internship at IBM but also for continuing to guide me with his knowledge to the end of this thesis. I am grateful for the great discussions in the weekly meetings and his invaluable feedback in the project as well as during the writing of the thesis.

Furthermore, I thank Dr. Dean Hildebrand with IBM for suggesting and discussing directions of the project and especially for agreeing to review the thesis as well. I am gratefully indebted for his very valuable comments. I would also like thank the IBM Almaden Research Center for the possibility of a challenging and fun summer internship making this project possible. Moreover, I want to acknowledge Dr. Frank Schmuck with IBM for taking the time and answering many questions regarding my research.

I want to thank Thomas Kemmer for the great discussions, invaluable constructive criticism, and repeatedly proofreading the thesis. I am also gratefully indebted to Sarah Kupfer for her support and proofreading the thesis. Furthermore, I want to acknowledge Jürgen Kaiser for the great office atmosphere, good smelling tea and his help, as well as Markus Tacke for patiently answering my questions related to hardware. I also thank Jennifer Leclaire and Federico Padua for their support.

Finally, I express my profound gratitude to my family and girlfriend for providing me with unfailing support throughout the years of my studies at the university.

Contents

Abbreviations

CPU	Central Processing Unit
DLM	Distributed Lock Manager
EXT4	Fourth Extended File System
FGDL	Fine-Grained Directory Locking
GPFS	General Parallel File System
HDD	Hard Disk Drive
HPC	High Performance Computing
I/O	Input/Output
LUN	Logical Unit Number
MMU	Memory Management Unit
mtime	Last modified time
MPI	Message Passing Interface
NSD	Network-Shared Disk
NTFS	New Technology File System
NVRAM	Non-Volatile Random-Access Memory
POSIX	Portable Operating System Interface
RAM	Random-Access Memory
SAN	Storage Area Network
SAS	Serial attached SCSI
SATA	Serial AT Attachment
SCSI	Small Computer System Interface
SNC	Shared Nothing Cluster
SSD	Solid State Drive
VFS	Virtual File System

Introduction

1

In *High Performance Computing* (HPC) many scientific applications create a large number of files simultaneously in a single directory. One example from the structural bioinformatics field is the process of writing the current state of an entire simulated system to disk, such as the 64 million atom coordinates of a HIV capsid [1, 2]. This process is commonly known as *checkpointing* and finds frequent usage in HPC clusters where (long-running) applications protect themselves from component failure by saving their progress to persistent storage [3]. With growing HPC clusters, this use case will be even more important in the coming exascale era as component failure becomes the rule rather than the exception [4, 5, 6]. Nowadays, checkpointing already represents the most important I/O workload in supercomputers [7].

Metadata operations, such as file creation, are generally problematic if the file system must handle many of them concurrently. As a result, modern parallel file systems distribute metadata across multiple *metadata servers* to achieve better performance. However, directories with a large number of files still share a single metadata server, which has to perform many metadata operations during concurrent file creation [3]. It has been shown that file creates do not scale well with an increasing number of compute nodes [8, 9, 10]. Therefore, file creates account for a non-negligible part of an application's runtime [8]. One possible contributor to slow metadata performance is the *Portable Operating Systems Interface* (POSIX) [11] standard which introduces a set of guidelines for operating systems to ease cross-platform development of systems and applications. For example, it requires created files to be immediately accessible to all nodes in the cluster, implying a large amount of overhead that may heavily degrade the file create performance. Consequently, a number of parallel file systems do not implement particular guidelines to improve various aspects of the file system [12]. However, because many applications depend on a POSIX compliant file system, relaxing the standard is often not desirable.

One such POSIX compliant parallel file system is *IBM Spectrum Scale*, formerly known as *General Parallel File System* (GPFS), which is widely used in HPC and data-intensive applications and is also available for Linux [13]. Spectrum Scale utilizes distributed locking to allow concurrent shared disk access while still maintaining consistency. Furthermore, it reaches scalability by distributing data and metadata across all disks [13]. However, for more than two decades parallel file systems have focused on providing scalable performance for large files instead of large directories [7]. Despite optimizations, the scalability of concurrent file creation with an increasing number of nodes still has to be improved [8].

Since file creation is a complex process with a large portion of network communication and I/O operations it is hard to understand where potential bottlenecks are located. Furthermore, a file create does not only consist of the creation itself: To maintain consistency the parallel file system has to verify whether a file with the same name does not already exist in the directory beforehand, involving an additional lookup operation. In this thesis, we aim on building an accurate understanding of the file create process in Spectrum Scale. To achieve this goal, we divide the file create process into logical tasks to generate a time distribution for each task. In this way, we can reveal potential bottlenecks that degrade file create performance.

As a first step, we gain a general understanding of Spectrum Scale's file create behavior which can be achieved with the well-known *mdtest* metadata benchmark tool [14]. However, in its current state it only provides the creates per second at the end of each benchmark run. This is unsuitable for such an investigation since it may hide potential differences in create performance during the experiment. Therefore, we modify the mdtest benchmark, providing us with the fine-grained per-process performance at any user specified interval. This approach allowed us to show the impact of internal mechanisms on the file create performance in Spectrum Scale, such as directory block splits and inode pre-allocation. Moreover, we investigated the influence of different storage hardware.

In the second step, we analyze the source code and trace records of Spectrum Scale to generate a call graph of its file create process. This enables us to split it into logical tasks whereas a corresponding *latency distribution* can be generated for each task via trace analysis. With this fine-grained investigation, we were able to locate those tasks that are responsible for the scaling problem in concurrent file creation. Additionally, we identified a particular Spectrum Scale configuration setting that was causing some files to consume significantly more time than others. Disabling this configuration increased the file create performance by the factor of two on our test system.

The subsequent chapters of the thesis are structured as follows: Chapter 2 introduces the theoretical background of our work. Specifically, we give detailed information about POSIX, metadata, and Spectrum Scale, including its internal mechanisms that might have an impact on the file create process. Next, we give an overview of the related work and other parallel file systems, exemplified by Lustre and CephFS. In Chapter 3 we describe the used methodology and present the modifications of the mdtest benchmark and how the trace records were analyzed. Chapter 4 discusses the results, starting with a black box investigation of mdtest's output. This includes the evaluation of client scalability as well as the impact of internal mechanism and different storage on the file create process. Additionally, we present Spectrum Scale's file create protocol and the evaluation of the trace analysis which shows the time distribution of all tasks. Based on these results, we discuss possible improvements. In Chapter 5 we conclude this thesis with an outlook on the future work.

2 Theoretical background

This chapter provides background information that is required to understand parallel file systems and the issues arising in the case of parallel file creation. First, we provide basic knowledge about POSIX, file systems, and metadata, followed by an overview and deeper elaboration on specific components of IBM Spectrum Scale. Finally, we discuss the related work on the topic and describe the architecture of other parallel file systems.

2.1 POSIX

The *Portable Operating System Interface* (POSIX) is a set of standards which is developed by the *Institute of Electrical and Electronics Engineers* (IEEE) [15] and *The Open Group* [16]. It was originally released in 1988 by the IEEE. Its intent is to introduce guidelines for operating systems to ease cross-platform development of systems and applications. This is achieved by defining a common operating system interface and environment, including a command interpreter (shell) [11]. In the most recent release *POSIX.1-2008*, POSIX is divided into four major components, each described in a separate volume (c.f. [11] for details):

1. *The Base Definitions volume* describes general terms, concepts and interfaces.
2. *The System Interfaces volume* gives definitions for system service functions, portability, and error handling for the C programming language.
3. *The Shell and Utilities volume* explains the standard source code-level interface to command interpretation services (shell).
4. *The Rational (Informative) volume* contains information about features that were discarded or included during iterations of the POSIX standard by its developers.

Next, we discuss POSIX I/O, included in The System Interfaces volume, that defines the I/O interface and is especially interesting in the case of distributed storage systems.

2.1.1 POSIX I/O

POSIX I/O describes the set of standards that define the *Input/Output* (I/O) interface for POSIX compliant applications, such as `read()` and `write()` functions, including their error messages. When POSIX was defined in 1988, a single computer, incapable of any concurrency, always owned its file system. Consequently, POSIX I/O was created with local file systems in mind, requiring write operations to be done sequentially consistent. The latter requires all instructions to be executed in order and that every write operations becomes immediately visible throughout the system. In addition, all write operations must appear atomic to any reader and from any location [11]. This guarantees consistent results from the I/O system in case multiple processes write to overlapping data regions.

On a single system, the disadvantages of these semantics are not apparent since all files are accessed through the same machine. Consequently, a locking mechanism that only allows one process to write to the same data region can be efficiently used to coordinate atomic access. On the other hand, maintaining consistency in any distributed environment where multiple computers may access the same resource concurrently is a communication-intensive and difficult process. Thus, parallel file systems commonly maintain consistency with one of the following locking subsystem approaches[1] [17]:

Centralized Management
All client requests go to a single node (broker server) that serializes access to the same data region and performs isolation if necessary.

Distributed Locking
A dedicated *Distributed Lock Manager* (DLM) hands out locks to clients that request exclusive access to a particular data region. Therefore, only one client can access a memory address at the same time.

2.2 File system

On the basic input/output (I/O) system level, disk partitions use equally sized, numbered sectors[2] to refer to the bytes of the physical storage. Each sector is a group of consecutive bytes and represents the minimum addressable unit of the storage. In this kind of representation each partition is one big dataset without any definition of files or a directory structure. The *file system* forms sectors into files and can be associated with a name while also providing a directory structure

[1]Section 2.5.4 describes Spectrum Scale's locking subsystem.
[2]Today, the sector size for most used block devices is usually 512 bytes, while it is 4096 bytes for modern devices.

and methods to organize the file system. Commonly, file systems use an additional abstraction, chaining sectors together to form a *file system block*. Advantages of blocks are, for example, faster disk reads due to sequential reads, resulting in less disk seeks, or less metadata (see Section 2.3) which also requires disk space.

However, file systems never provide physical memory addresses to a user application. The obvious reason is security, because processes would not be restricted to an exclusive address space. Another reason is that applications would need to manage memory by themselves, i.e., moving data from disk to memory, that would change their physical address, for instance. Instead, applications use *virtual addresses*, which are mapped to their physical counterparts by the *memory management unit* (MMU), an independent part of the CPU. This saves applications from managing a shared address space while also preventing applications from accessing the virtual address space of other running processes. The virtual address space appears continuous to the process, while, in reality, the data can be scattered among physical addresses on disk or memory.

In practice, there exists a wide variety of file systems for different operating systems and use cases. For the user, they differ in the maximum usable length of the file name, the maximum file size, supported folder encryption etc. However, internally, they vary greatly in data structures or features, such as extents [18]. Common consumer local file systems for Windows and Linux are the *New Technology File System* (NTFS) and the *Fourth Extended File System* (EXT4), respectively[3].

2.3 Metadata and inodes

In Unix or Linux file systems, information about a file, e.g., its owner and permissions, size, or last modification time (mtime) are referred to as metadata and are stored in a so-called *inode*. In addition, it stores the block addresses to the data of the file. Commonly, the default block and inode sizes are 4 KiB, as it is in EXT4, for example. In EXT4 and other Linux file systems, the inode can store up to 12 direct data block pointers that can store a total of 48 KiB and 3 indirect pointers that link to so-called *indirect blocks*. They may have the same size as regular blocks and contain pointers to the actual data blocks but can also refer to another indirect block (double indirect block) or to an additional indirect block (triple indirect block), which contains pointers to more indirect blocks before pointing to the actual data block. Each of the mentioned 3 pointers link to a single, double, and triple indirect block, respectively. With this structure, the file size can grow up to 16 TiB for a 4

[3]In this thesis, we will only talk about Linux file systems, since we use IBM Spectrum Scale with Linux.

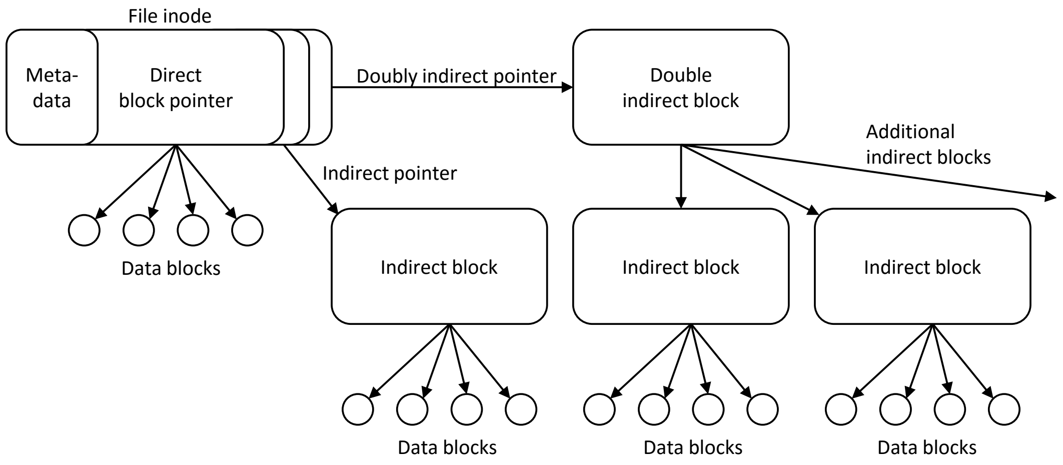

Fig. 2.1. – An example file inode structure where a double indirect block is in use.

KiB block file system. Figure 2.1 visualizes an inode in an example with a single and double indirect block structure.

Since data is stored in blocks, many processes are able to efficiently write to different data regions of a file in parallel as long as two processes do not write to the same block. If multiple processes had write access to the same block and they updated the block without coordination in their cache and *flush* (write) it back to the disk, the result would be undefined. This is due to the fact that only one process succeeds while the changes of other processes are discarded without their knowledge. This is known as *false sharing* and has to be avoided in order to maintain file system consistency. Furthermore, every change to the data of a file modifies its metadata as well. Attributes like the mtime or file size need to be updated on every write to the same file. Since this information is only stored in a single block, the inode, only one process is allowed to access this block at the same time. To serialize write access locks can be efficiently used in local environments. On the other hand, using locks in a similar fashion in parallel environments can significantly decrease performance due to lock conflicts if multiple clients and their processes access one inode concurrently. This is especially true when many files are created in a single directory at the same time, because the inode of the created file has to be inserted into the directory's data blocks as a new *dentry*, short for directory entry. The dentry essentially associates the inode of the file represented with its filename. *Directory blocks* are similar to indirect blocks except that they contain directory entries instead of pointers to data blocks. Therefore, concurrent creation of files in the same directory is not possible and they are essentially serialized as a result of the locking mechanism. Section 2.7 describes how parallel file systems, specifically Spectrum Scale (see Section 2.5.4), try to solve this problem.

2.4 File creation

When a file needs to be created the file system has to ensure that a file with the same name does not already exist within the directory with a lookup operation. Hence, the file create process can be logically split into two steps: The lookup and the creation of the file. Both steps are briefly explained in the following paragraphs.

In Linux the lookup operation searches for a file with a given name (in its path) in a directory and will return the corresponding dentry with the associated inode of the file if it exists. If possible, this process is performed in the *dcache* where looked up dentries are cached for quick access. The first step is to traverse the namespace either from the root directory (for absolute paths) or from the current working directory (for relative paths). The first component of the path is looked up in the starting dentry which yields to the next dentry in the path [19]. This process, called *path walking*, is continued until the end of the path is reached. However, not all path components might be present in the dcache, requiring the file system to do a more expensive lookup operation via the parent's inode. If the lookup succeeds, the resulting dentry with its associated inode pointer will be put into the dcache. If the dentry cannot be found, the file system will also add the dentry to the dcache with a `NULL` inode pointer – the so-called *negative dentry*. Negative dentries can improve performance as repeated lookup operations do not need to invoke the underlying file system [20].

When the lookup operation returns a negative dentry the file can be created. Then, the file system allocates a new inode for the new file and initializes its metadata. Next, the `NULL` inode pointer in the dentry is replaced by a pointer to the new inode. Finally, the dentry is added to the parent directory and the file is created.

2.5 IBM Spectrum Scale

In this section, we give an introduction into clustered environments and IBM's Spectrum Scale and describe general components as well as pointing out which are important for file creation.

2.5.1 Overview

Clustered environments consist of a number of nodes which are connected in a network. Every node can run an application on its own or in conjunction with multiple nodes. Further, each node is connected to a shared storage system that stores all data and metadata and is generally accessible from any cluster node[4]. Notably, a hardware device in a shared storage system can refer to any kind of storage technology, such as *solid state drives* (SSD), *hard disk drives* (HDD), or *non-volatile random-access memory* (NVRAM) [21]. Henceforward, we will use the term *disk* if the storage type is unimportant in the given context. However, many disks on their own, or as a combined array, cannot be used effectively without an appropriate file system which exploits and combines the I/O bandwidth of all disks equally. After all, I/O speeds are still an issue in parallel computing as I/O-bound applications are easily limited by the speed of the underlying storage hardware, regardless of the CPU performance of combined nodes. To solve this issue, parallel/distributed file systems were created to utilize the I/O speed of all available disks in a cluster as best as possible.

Spectrum Scale, formerly know as *General Parallel File System* (GPFS), is a closed source parallel, shared-disk file system developed by IBM and is used in High Performance Computing (HPC) and data-intensive applications in clustered environments. The parallel file system started as a research project in the early 1990s, called *Tiger-Shark* on IBM's proprietary operating system *AIX*. In 2002, GPFS was first introduced for Linux by Schmuck et al. [13]. Since then, advanced features, such as GPFS Native RAID, information lifecycle management, or wide-area caching and replication are continuously introduced. Nowadays, it also supports heterogeneous clusters, including AIX, Windows, and Linux. In order to satisfy today's requirements of demanding scientific and commercial applications, Spectrum Scale utilizes parallelism and efficient data sharing between nodes in clustered environments, while being completely POSIX [11] compliant with minimal overhead [21].

Spectrum Scale is based on the shared storage model – a framework for describing storage architectures, defined by the Storage Networking Industry Association (SNIA) [22] in 2003, that captures the functional layers and properties of a storage system, regardless of the underlying design, product, or installation. The shared storage model itself does not define a specific structure but rather describes architectures which also make them comparable with each other [23]. The three typical shared storage models are the *Storage Area Network* (SAN), the *Network Shared Disk* (NSD), and the *Shared-Nothing Cluster* (SNC) models (see Figure 2.2). Today, the NSD model is the most common because it avoids the cost and administrative overhead of maintaining two separate networks [21]. In this model, a node is called a *NSD server* when it provides disk access to other nodes in the cluster, while *NSD clients* access data through an NSD server. Hence, the NSD servers accept requests from clients to read and write data from and to disk, respectively, and reply to them with the desired data. However, in the case of multiple NSD servers, it would not be a good idea if each server could only access a subset of the total number of disks

[4]Some cluster hierarchies only allow specific nodes to access the file system to handle the I/O requests.

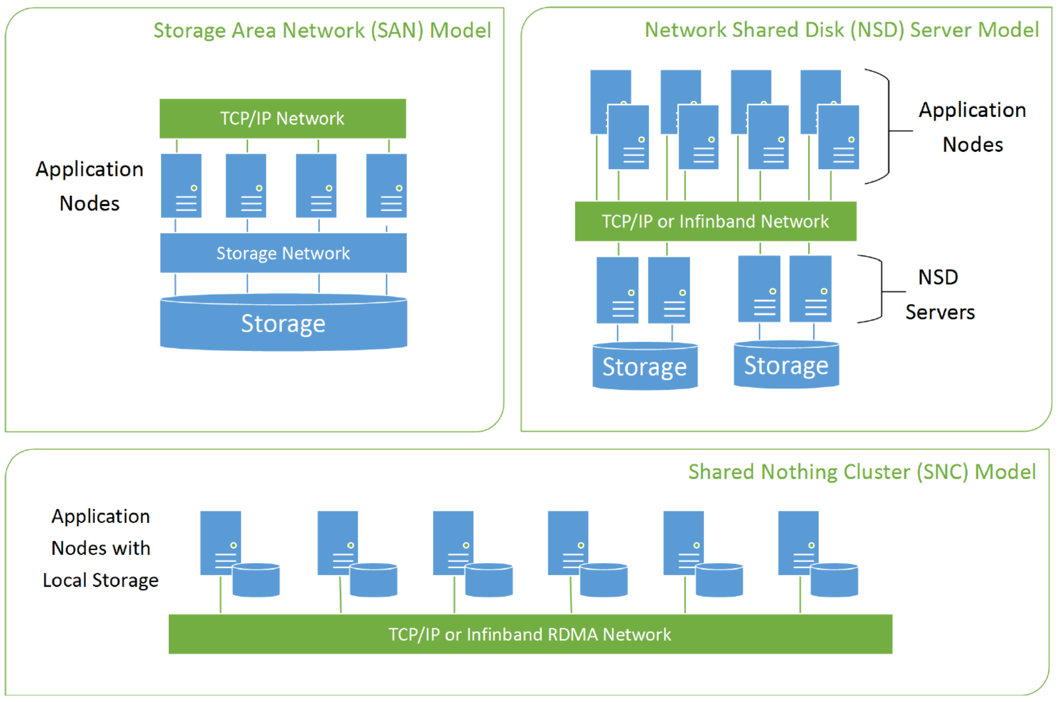
Storage Area Network (SAN) Model
TCP/IP Network
Application Nodes
Storage Network
Storage
Network Shared Disk (NSD) Server Model
Application Nodes
TCP/IP or Infinband Network
NSD Servers
Storage
Storage
Shared Nothing Cluster (SNC) Model
Application Nodes with Local Storage
TCP/IP or Infinband RDMA Network

Fig. 2.2. – Typical shared storage models [21]. *[High Performance Parallel I/O, D. Hildebrand and F. Schmuck, 2014, p. 33–43]*

available as some data cannot be accessed in the case of a NSD server node failure. Thus, a common scheme is to use redundant connections between the NSD servers and the disks, known as *multipathing*, which allows the data to be accessed through different NSD servers. This has two advantages: Node failures do not introduce data unavailability and load balancing between the NSD servers can be efficiently used to avoid network congestion at one of the servers. As we will see in Section 3.1, we use the NSD server model in our experiments.

Spectrum Scale provides concurrent shared disk access to a single global namespace to all nodes of a cluster. The file system achieves scalability by distributing data and metadata evenly across all available disks to maximize I/O bandwidth. This process is referred to as *wide striping*. Specifically, large files are divided into equally sized blocks and are distributed across all disks in a round-robin fashion. In addition, distributed locking (see Section 2.5.4) guarantees the consistency of the file system without the need of a centralized server, and maintaining cache coherency while allowing local caching of data and metadata. Data loss is prevented with traditional hardware RAID solutions or *GPFS Native RAID* [24], while recovery logs are implemented for metadata consistency and heartbeats check for node failures [21]. Finally, Spectrum Scale uses inodes and indirect blocks to store a file's metadata and data block addresses [13] similar to other file systems, such as EXT4.

2.5.2 Basic Spectrum Scale structure

Each node of the Spectrum Scale cluster needs to have the Spectrum Scale software installed. It consists of three basic components: administration commands, a kernel extension, and a multithreaded daemon. To enable the daemon to communicate with the Linux kernel, Spectrum Scale uses a loadable open source kernel module, the *portability layer*, which can be build for a wide variety of Linux kernel versions and configurations to ensure Spectrum Scale's portability [24] with most Linux distributions. We briefly describe the three components in the following sections.

Administration commands

Administrative commands are executed by any node in the cluster that is part of the file system. They control configurations and the Spectrum Scale's operation by providing commands to add or remove nodes from the file system, for example. In general, any command can be executed on any node, as the user specifies which nodes a command should affect. Spectrum Scale ensures that it is propagated appropriately across the cluster.

Kernel extension

Linux provides a *virtual file system* (VFS) layer that is used to register any Linux file system. This modular structure allows any application running outside the kernel, i.e., in user space, to use the same system calls, regardless of the underlying file system. Internally, the kernel maps a system call to the file system's kernel extension. Spectrum Scale also implements this standard mechanism and, thus, appears to all applications just as another file system. Spectrum Scale either satisfies a system call by the resources available in the kernel extension or sends a request to the daemon to complete the request [24]. Moreover, the VFS provides facilities, such as the earlier mentioned dcache (see Section 2.4), that can be used by file systems.

Daemon

The multithreaded Spectrum Scale daemon runs on every node and is responsible for all I/O operations and buffer management, such as read-ahead for sequential reads and asynchronous writes. All I/O operations have to follow the rules set by Spectrum Scale's locking management (see Section 2.5.4), toward ensuring file system consistency. In addition, the daemon sends requests to other nodes (such as manager nodes) and propagates configuration changes by communicating with other daemons in the cluster.

2.5.3 Directory blocks and directory entries

Similar to other file systems, Spectrum Scale uses directory entries which are stored in directory blocks to store the association between a directory and its files. The parallel file system uses extensible hashing [25] to organize directory entries within a directory that define the connection to the inode of the corresponding file. As soon as a directory block is full Spectrum Scale splits a block in two whereas each block holds half of the entries of the initial block. In Section 2.5.4, we will discuss the difficulties that arise with directory splitting and locking. To determine the corresponding directory block (bucket) of a file, Spectrum Scale uses extendible hashing [26] on the filename. Extendible hashing is dynamic as it allows insertions and bucket splitting as well as deletions and bucket combination without resulting in poor performance after many of these operations [27]. Each split of the same directory block increases the hash tree level. Current versions of Spectrum Scale allow a hash tree level of 20 with a maximum of 2^{20} directory blocks [28].

2.5.4 Distributed locking and metadata management

With respect to maintaining POSIX semantics and file system consistency, Spectrum Scale is based on distributed locking to efficiently synchronize access to data and metadata. Therefore, a lock is required for every file system operation that updates data or metadata in order to preserve read and write atomicity, as defined by POSIX [11]. Distributed locking is achieved by using a centralized server, referred to as the *token manager*, that hands out *tokens* to local lock managers that run on every file system node. Tokens are provided for directories, files, or other file system objects. When a node holds a token for a resource, no additional network communication is required as long as no other node wants to access the same object. However, tokens are not only used for maintaining POSIX semantics and consistency, but also for introducing a cache coherency mechanism. For instance, a node can request an exclusive write token, which only one node is allowed to hold at a time, and update the locked object in its cache. Other nodes that need to modify an already locked object have to wait until the lock holder is finished and releases its token. Furthermore, exclusive write tokens can also be downgraded to read-only tokens. This type of token can be held by multiple nodes simultaneously and allows them to cache the read-only object until one node requires exclusive access.

Nonetheless, in the common scenario where many nodes need to modify the same file, exclusive write locks per file would result in a lock conflict on every write operation. Spectrum Scale solves this issue by utilizing a *shared write lock* to allow multiple writers access to one file simultaneously. The following example describes how multiple nodes are writing to non-conflicting regions of a file: When writing to a file, the first node initially receives a token for the complete file. As soon as another node also requires access to some region of the same file, the first node is contacted by the token manager to give up the exclusive write lock for the conflicting region. If the first node does not require access to the conflicting region, it releases parts of its token. Both nodes now hold non-conflicting tokens for different regions of the file and are allowed to write to the same file concurrently, while maintaining consistency and POSIX semantics. Locking a specific region of a file is known as *byte-range locking*[5] and enables Spectrum Scale to perform simultaneous writes to the same object, maximizing the I/O bandwidth. A byte-range token allows the holder to lock one or multiple consecutive blocks. One important constraint is that a byte-range token cannot be smaller than one block and it is always rounded to block boundaries. This is because every node allocates disk space independently when creating a new file [13]. In addition, byte-range tokens ensure that only one data block is allocated by a single node.

When the data of a file is updated new data needs to be written to the inode as well, changing metadata attributes, such as file size, mtime, and indirect blocks, which hold pointers to the data blocks of the file. Consequently, byte-range locks cannot be efficiently used for metadata and indirect blocks as every write would result in a lock-conflict, essentially serializing writing to a file. In other words, a different solution is needed to handle metadata updates. Similar to other parallel file systems, Spectrum Scale uses a so-called *metanode* to solve this issue by collecting updates to the inode from other nodes. However, defining a single metanode for the whole file system could easily introduce a bottleneck for the file system while being under heavy load. Instead, Spectrum Scale specifies one metanode per file, while any client that accesses this file can act as its metanode. In the example of two nodes writing concurrently to the same file, the first accessing node will become the metanode. The second node will send all updates concerning the file's metadata to the metanode while independently writing to the data region for which it holds a byte-range token. Note that the metanode for a file can change in case it does not require access to the file anymore. This allows great scaling, regardless of metadata activity throughout the file system.

On the other hand, directory write sharing is much more common in the scenario of parallel file creation in a single directory [21]. Additionally, creating a file in a directory requires much finer grained locking than writing to an existing file, since every create inserts only a new single entry into a directory block. As mentioned

[5] To achieve almost near raw disk performance, distributed locking can also be bypassed completely if applications support their own locking mechanism [21].

earlier, the smallest unit of a byte-range lock is one block, while a single directory block can contain thousands of entries. Hence, locking the whole block by one node will introduce many token conflicts if other nodes need to insert entries into the same directory block. This issue gets worse with bigger block sizes[6]. To allow more fine grained locking, Spectrum Scale uses a technique called *fine-grained directory locking* (FGDL) [29]. FGDL allows nodes to obtain a token only for a specific filename. In other words, if a node needs to insert an entry into the directory block, it is able to lock the filename hash in the directory block separately instead of locking the whole block. This guarantees that two nodes cannot create a file with the same name while still allowing other nodes to obtain a token for a different filename in the same block. This is done by using a byte-range offset that is computed from the filename hash. However, as mentioned before, it is not possible to write parts of a block to disk. Instead, the metanode of the directory collects all updates to the directory block and writes the changes to disk on behalf of the actual creating nodes. Note that the creating nodes are still allowed to write the created inode of the new file to disk, as it does not influence the directory's metadata. This method allows the metanode to cache directory blocks and reserve space in a directory block for pending creates. Non-metanodes are able to cache entries covered by FGDL tokens and to insert entries locally before sending them to the metanode [29]. However, despite the advantages of using FGDL, it is incompatible with byte-range locking. Thus, a directory can either be in byte-range locking or in FGDL mode. Switching between these modes can be expensive because it involves invalidating all associated tokens, possibly introducing heavy token traffic across the network. Another scenario, which involves invalidating the FGDL tokens of a directory block, is when a directory split occurs in the case of a full directory block. Since only the metanode is allowed to write directory blocks to disk, it is convenient that it is also responsible for splitting a full directory block whereas each block will contain half of the entries of the pre-splitted directory block. During the split, exdentible hashing determines which files are put into the appropriate block. Additionally, in the case of concurrent file creation, directory blocks tend to get full at the same time which can result in heavy token traffic at the time of directory splitting due to invalidations. This may lead to metanode contention if a large number of blocks needs to be split simultaneously.

2.5.5 Logging

Comparable to other journaled file systems, Spectrum Scale uses a log to provide atomic updates to the metadata on disk.[7]. Usually, operations such as creating or removing a file require a series of separate write operations. If a system is interrupted

[6]By default, Spectrum Scale uses a block size of 256 KiB.
[7]Each node manages its own journal.

during these operations due to a power outage or a system crash, for example, it may be left in an intermediate, invalid state which is much more time consuming to restore. Instead, journaling file systems utilize a write-ahead log, recording all changes that are to be written at a later point in time. In the case of a crash the file system can recover easily by reading the log and replaying the journaled operations until the system is back in a consistent state. Depending on the implementation, a file system may log data as well as metadata. Spectrum Scale, on the other hand, only records metadata updates since they affect the file system's consistency, while user data are not logged [13].

2.5.6 Tracing

In general, tracing allows users to retrieve event logging information about an application's execution. Traces, in contrast to event logging, are mostly read and analyzed by developers because they contain intricate information about file system internals that only developers can interpret. Therefore, traces are often fairly technical and low-level. Trace messages are defined in the source code and can be enabled at runtime or compile time. However, enabling tracing usually slows down the application severely, mainly owing to a large amount of trace messages.

The Spectrum Scale source code contains over 20,000 trace points which are separated into *subsystems* and can be enabled or disabled by the root user per node while the file system is running. Traces of such a set can contain all messages of the NSD server or the metanode, for example. Yet, the number of trace messages of a subsystem may still be too large. For this reason and because different traces of a subsystem may be more important than others, the user is able to limit the number of messages by providing a level of details in form of an integer, whereas 1 enables the most important traces within a subsystem. A bigger integer provides additional messages, while still including the traces from all higher levels of detail. Trace analysis does not only help for debugging purposes but also helps studying bottlenecks of an application and is the most useful tool in understanding various components of Spectrum Scale and specifically, in this thesis, the process of file creation.

2.6 Related work

For over two decades, research on file systems was focusing on delivering scalable performance for large files rather than for directories with a large number of files [7]. Although most directories contain less than 8,000 files [30], the majority of files are located in big directories [29, 31] involved in a number of use cases for data-intensive applications [7].

One example is checkpointing in clustered environments [3, 10, 8, 9, 29] where the application's state is written into files in a single directory per-process. This use case will become even more important in the coming exascale era, with applications that run on up to a billion CPU cores [3]. However, scaling metadata performance still poses a challenge in local and parallel file systems. Literature has shown that metadata operations, including single directory file creation, do not scale well with an increasing number of nodes [8, 10, 32] on parallel file systems. Alam et al. [8] describe this effect as the metadata wall, spoiling the file system's performance potential.

As a result, software solutions were developed focusing on metadata scalability and have shown performance improvements for metadata operations [7, 33, 34]. However, these solutions are not available in architectures of current parallel file systems, such as Spectrum Scale [13] or Lustre [35]. Alternatively, Frings et al. [9] built an additional software layer between an application and the underlying parallel file system to avoid contention on metadata servers during file creation without decreasing the I/O bandwidth. In this thesis, our goal is to provide information about Spectrum Scale's file create behavior to help the developer team to optimize those mechanisms which consume the most time in the process. In the end, applications should be able to rely on the underlying file system that supports fast concurrent file creation without the need of additional software.

2.7 Related parallel file systems

This section presents a brief overview of two other parallel file systems, Ceph [12] and Lustre [35], which also support the distribution of metadata in single large directories.

2.7.1 Lustre

Lustre is an open-souce[8], POSIX-compliant parallel file system, initially designed at Carnegie Mellon University, PA, USA, in 1999 and is based on Linux. It is currently mainly developed by *Intel's High Performance Data Division* (Intel HPDD) [36, 35] whereas many other companies, such as Fujitsu [37] or NASA [38], also contribute to Lustre's codebase [39]. Its development is primarily sponsored by the *Open Scalable File Systems Inc.* (OpenSFS) [40] and the *European Open File System* (EOFS) [41].

Lustre is based on separating metadata operations from actual data I/O operations. As visualized in Figure 2.3, the main components in Lustre's architecture are: the

[8]Lustre is licensed under the GPL 2.0 license.

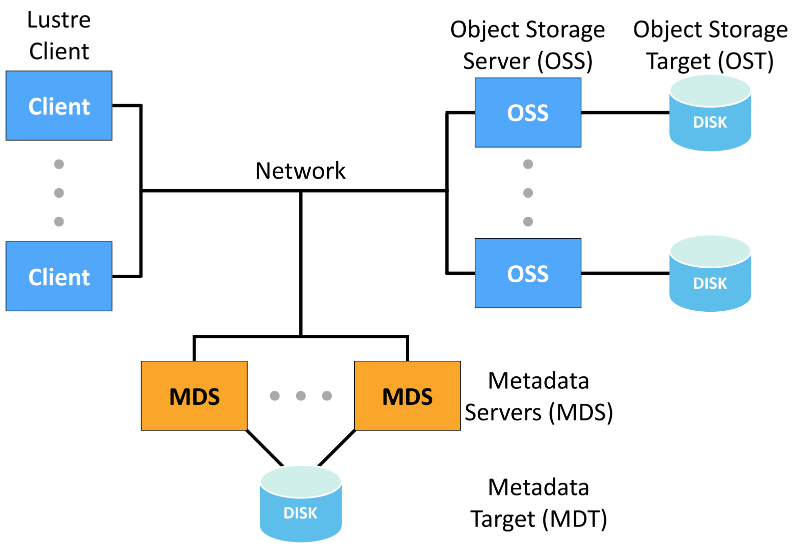

Fig. 2.3. – Sample architecture of the Lustre file system

client, the *metadata servers* (MDS)[9], and the *object storage servers* (OSS). The clients run applications, while the MDSs are responsible for managing all the metadata operations of the entire file system. The OSSs are handling the I/O data transfer. The metadata for all clients is stored on the *metadata targets* (MDTs) which act as the storage component of the MDSs. Similarly, the *object storage targets* (OSTs) store all data which is segmented logically to different storage devices, i.e., striped over all disks [42].

To achieve parallelism, each MDS and OSS is running an instance of the *Lustre Distributed Lock Manager* (LDLM) which utilize so-called *intents* for the better locking decisions [43]. For example, if a client wants to read an existing file in the file system, it contacts the MDS with a read intent. The MDS checks the users permissions and sends back a list of OSTs where its stripes are located. For a read operation no locking is required on the MDS, since all content is provided by the OSTs. Then, the client enqueues a lock request with a read intent to the corresponding OSTs. The client now holds a lock for each stripe on each OST, while the LDLM on the OSSs are managing the locks. When the client is finished, it issues the OSSs to release the locks [43]. For a file create, the client sends an *open and create* intent to the MDS with the corresponding path and name of the file. The MDS locks the parent directory and creates the file on the clients behalf. Eventually, the MDS responds to the client that the file was created, including a lock on the file so that it can be opened by the client afterwards [43]. This is dissimilar to Spectrum Scale's file create behavior in which Spectrum Scale clients only forward directory block updates to the metanode while they still create the file themselves.

[9]Multiple metadata servers are only possible in Lustre 2.4 and beyond for better metadata scaling.

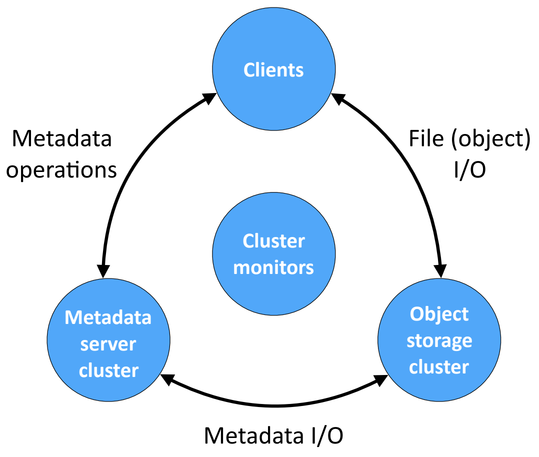

Fig. 2.4. – Conceptual architecture of CephFS

2.7.2 Ceph

CephFS is an open-source[10], nearly POSIX-compliant [44] parallel file system. It began as an PhD research project by Sage Weil at the University of California, Santa Cruz, CA, USA, in 2010, who is now working for *Red Hat* [45] as the chief architect of the Ceph project. CephFS is one component of the Ceph ecosystem which is build on top of the *Reliable, Autonomic Distributed Object Store* (RADOS), responsible for all the data in a Ceph cluster [46].

Figure 2.4 shows the conceptual architecture of CephFS, divided into four components: The *clients*, the *metadata servers* (MDSs; handles caching and synchronization of distributed metadata), the *object storage cluster* (OSDs; responsible for storing data and metadata), and the *cluster monitors* (implement monitoring functionality) [47]. From the client's point of view, metadata operations, such as *open* and *rename*, are performed by the MDSs while OSDs provide direct file I/O (*reads and writes*) to the underlying storage without additional MDS communication [12]. In contrast to other file systems, CephFS does not rely on mappings between the metadata and the blocks on a disk to a given file (allocation lists). Instead, the file system depends on a pseudo-random mapping algorithm called *Controlled Replication Under Scalable Hashing* (CRUSH) [48] to assign objects to storage devices, simplifying the design of the system and reducing the metadata workload [12, 47].

Contrary to Spectrum Scale and Lustre, CephFS does not utilize a distributed lock manager and rather issues *RADOS locks* which are enforced by the OSDs. The read (shared) and write (exclusive) locks are implemented as object attributes, behaving like any other object update for lock acquisition and release. For consistency and safety reasons, the locks are replicated across all OSDs [49]. This is in contrast to Spectrum Scale which uses distributed locking to synchronize access to data and

[10]CephFS is licensed under the LGPL 2.1 license.

metadata on a shared disk [21]. For concurrent file creation in a single directory, Ceph hashes the shared parent directory and relaxes the directory's mtime coherence to scatter the workload over all MDS nodes [12]. However, CephFS is not yet recommended to be used for storing important data as it is lacking robust disaster recovery tools, which are under development [50].

Methodology 3

This chapter provides information about the methodology to analyze file creation in Spectrum Scale. First, we illustrate our experimental setup followed by a discussion about both the data acquisition and processing, including a description of the used tools.

3.1 Experimental setup and file system configuration

Running metadata benchmarks in a productive environment is usually not feasible and difficult to control as they should not interfere with the system that may even need to be restarted during the benchmark. Instead, all experiments were run on a small test cluster which consists of six compute nodes. All of them have the Spectrum Scale software installed in version 4.1.1.0. Two of the nodes (Dell R710) are directly connected to the storage and act as Spectrum Scale NSD servers, named *argon* and *kalium*. The Dell R710 machines are equipped with two quadcore E5620 CPUs @ 2.4 GHz and 24 GB of memory. A Dell MD3200 array is used for storage with two controllers whereas each RAID controller provides 2 GB of cache for a total of 4 GB. They are used in a dual controller configuration which is mirrored with the other controller's cache for high availability. Furthermore, the controllers are protected with a battery assisted persistent cache backup to non-volatile media in the case of a power failure [51]. The MD3200 uses three MD1200 expansion shelves with a total of 48 2 TB 7,200 RPM SAS connected hard drives (HDDs). They are formatted into four dynamic disk pools of 11 disks and 4 hot spares. For improved rebuild performance, a 8+2 RAID6 setup distributes its stripes over all 11 disks of each pool. In addition, the disk pools are split into four partitions, each associated with a logical unit number (LUN) which is then used in Spectrum Scale. All HDDs are accessible through both NSD servers redundantly since the MD3200 is shared. Alternatively, each NSD server can utilize a single solid state drive (SSD; Intel SSD 520 Series 240 GB), connected via SATA, or a RAMDisk of 8 GB. A RAMDisk describes a portion of the system memory that is treated as if it were a disk drive. However, the Dell MD3200 is not controlling the SSDs and RAMDisks. Thus, they are only accessible by the corresponding NSD server.

The remaining four nodes act as Spectrum Scale clients, named *boron*, *chlorine*, *lithium*, and *neon*. Each of them is equipped with an eight core E3-1230 @ 3.3 GHz

CPU and 16 GB of memory. All machines, i.e., clients and NSD servers, are connected through a 10 Gbit ethernet via a SuperMicro SSE-X24S 24 port 10 Gbit switch.

In all our experiments, we use similar Spectrum Scale file system configurations with 44 HDDs, unless indicated otherwise. Table 3.1 lists those configurations that might impact metadata performance. Moreover, three of the above presented storage subsystems hold data and metadata alike[1].

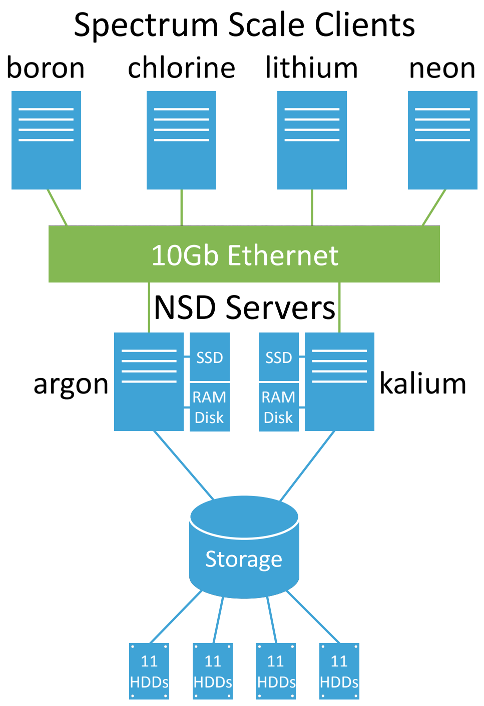

Fig. 3.1. – Simplified test cluster representation without facilities, such as the controllers.

Configuration	Value
Block size	256 KiB
Inode size	4 KiB
Log size	16 MiB
Inode limit	2,200,000
Pre-allocated inodes	2,200,000
Metadata replicas	1
Max. metadata replicas	3
Exact mtime	yes

Tab. 3.1. – The used Spectrum Scale file system configuration parameters.

3.2 Data acquisition

The process of data acquisition consists of two steps: Firstly, the file create process is benchmarked in a black box approach to investigate the overall performance which is visible to the user. The second step requires a deeper analysis of Spectrum Scale's internal behavior with the help of the source code and the trace records. These two steps are discussed in the following sections.

[1]It is possible to assign disks for only keeping storage metadata or data.

3.2.1 Mdtest metadata benchmark

We use the metadata benchmark *mdtest*, developed to evaluate file create performance. In the following, we discuss its functionality, usage, issues, and modifications.

Overview

Mdtest is a popular micro benchmarking tool, developed by the Lawrence Livermore National Laboratory [14] and used by many large HPC sites [32]. The mdtest in version 1.9.3, henceforth called original mdtest, was used in this work and was released in December 2013. It measures metadata performance of the operations *create*, *stat*, *read*, and *remove* on files and directories, called *items*. The *Message Passing Interface* (MPI) coordinates the operations on multiple compute nodes and to consolidate the results, which are given in operations per second. Mdtest measures each operation sequentially and calculates its results afterwards. For that, the benchmark divides the number of items (workload) by the elapsed time for each operation at the end of an experiment, that is, one execution of the program. Each operation is embedded in two *MPI_Barriers*. This ensures that faster processes do not continue until all processes have finished their workloads. The duration of an operation is measured between the two barriers to calculate the operations per second for the workload of all processes. In addition, mdtest is able to run multiple consecutive experiments and compute the mean and standard deviations over all executions. Figure 3.2 shows a sample excerpt of the output, whereas 32 total tasks, distributed on four nodes, were utilized to create 4,096 files over 20 iterations. Each row and column in the result table represent an operation and its operations per second, respectively. For files, the user can also define the size of the created files. Nonetheless, metadata performance for file creation can be best evaluated when zero-byte files are created. Writing additional data would introduce a delay between two created files, relaxing the workload on the metanode. Therefore, we exclusively create zero-byte files in all benchmark runs.

Naturally, there are other metadata benchmarks available, e.g., Filebench [52] or Bonnie++ [53]. Besides the metadata evaluation, those benchmark suites also offer additional functionality. We decided to use mdtest due to its focus on metadata and easy-to-modify source code as its whole functionality is consolidated in a single C file.

```
mdtest-1.9.3 was launched with 32 total task(s) on 4 node(s)
Path: /gpfs/createtest
FS: 14.2 TiB   Used FS: 0.0%   Inodes: 14.2 Mi   Used Inodes: 0.0%

32 tasks, 4096 files

SUMMARY: (of 20 iterations)
Operation                  Max          Min          Mean        Std Dev
---------                  ---          ---          ----        -------
File creation :        398.583      177.422       305.453         93.658
File stat     :   12688234.257   259260.080   4178164.927    5207511.648
File read     :    4039470.770   408169.855   2396671.658    1232374.203
File removal  :        392.336      156.385       291.746         98.000
```

Fig. 3.2. – Sample excerpt of mdtest's output. 32 tasks, distributed over 4 nodes, a were used to create 4096 files. Each row in the result table represents the operation and its operations per second.

Usage

Mdtest is used on the command line while MPI defines the number of processes that are started on each assigned node. The benchmark tool itself utilizes a variety of arguments that can be combined almost arbitrarily. The following configurations are mandatory for executing mdtest: Before each experiment, the program creates a so-called directory tree (user defined by its depth and branches), which is always generated in case the user needs to create files or directories in multiple independent directories. The number of files or directories to be created is specified per node. In addition, the user has to provide the working directory on which the benchmark will take place, for instance, a parallel file system's mount point.

Issues

Unfortunately, mdtest suffers from a set of issues that are not immediately apparent to the user. The most problematic ones include non-functional arguments and flawed results. The intransparency of false results can potentially compromise research results. Moreover, the project experiences a sparse release cycle with gaps of up to 20 month between releases (e.g., from version 1.9.3 to 1.9.4). The most recent version of the benchmark was released in July 2015 to support the restful S3 interface[2]. However, no changes were made to solve the discussed issues.

[2]The mdtest project was continued on Github [54].

Modifications and scripts

A single data point per run and operation of the original mdtest output is challenging since the operations per second might differ on a per-process basis during the benchmark. Therefore, we modified the benchmark to provide us with the creates per second at any point in time, called *periodic output*. The periodic output is a user defined interval and can be set to a number of files or seconds. Each point in the interval contains the per-process information about its number, the node's identity, the elapsed time in seconds, and the number of created files. Furthermore, it is important that the data collection does not induce any overhead, for example, due to disk I/O which would disrupt the benchmark and falsifies the results. Therefore, all data points are collected in memory and written to disk after the total workload is finished. Henceforward, the term *modified mdtest* refers to the version of mdtest that was modified with the abovementioned functionality. The modified mdtest will be further maintained on *GitHub*[3] [55] while the periodic output will be extended to all other operations in the future.

The periodic output of the modified mdtest allows the fine-grained investigation of the create performance of Spectrum Scale whose findings will be discussed in Section 4.1. In addition, this information is not only useful for visualizing the per-process performance, but more importantly, for pointing out time frames that require a deeper analysis of the corresponding trace records. Without the modified mdtest, these time frames would remain unnoticed. Lastly, a *Python* [56] wrapper script automates the execution of multiple benchmark runs with varying clients and numbers of processes. They introduce functionality, for example, enabling trace collection, restarting Spectrum Scale, or remounting the file system after each benchmark while providing a user friendly command line interface (CLI).

3.2.2 Tracing and source code

Trace records give insight into the internal mechanisms of Spectrum Scale, e.g., its file creation protocol, which the modified mdtest is unable to provide. However, due to the large number of available trace points in the parallel file system, it is difficult to filter those records that are the most relevant in the create process. In addition, traces are ineligible to give reasons for specific design decisions or Spectrum Scale's implementation, that can only be answered with its source code. With the source code available, we are able to investigate every step of the file create process by enabling the corresponding submodules with the least required level of details of the relevant trace records. Collecting the correct traces in the right context is crucial for

[3]`https://github.com/marcvef/mdtest`

understanding which part in the file create process is consuming most of the time and for analyzing potential bottlenecks.

3.3 Data processing

The following sections present the processing of the acquired data, which includes the periodic output of the modified mdtest and trace records as well as its usage.

3.3.1 Plotting

The previously mentioned wrapper script of the modified mdtest (see Section 3.2.1) does not only provide an improved user interface and automation but also contains additional functionality for post-processing the acquired data. The periodic output files of a modified mdtest run, executed through the wrapper, are automatically processed and generate four different types of graphs with the *Gnuplot* [57] application. They show the combined and per-process performance in regard to the accumulated creates per second and the current creates per second between the set interval for all data points. Moreover, the script generates comparable graphs of all its executed benchmark runs with reference to node and process scalability. These features allow the user to get a general idea of how a file system performs in the scenario of concurrent file creation.

3.3.2 Trace analysis

The trace analysis consists of two parts: Firstly, with the help of the source code the file create protocol is investigated to retrieve the relevant trace records with their corresponding subsystems and their level of details. Secondly, the achieved knowledge about the trace points is used to generate latency distributions of defined parts of the create process. The two steps are discussed in the following two sections.

File create protocol and call graphs

Based on the source code and the traces, we manually built call graphs of the file create process in Spectrum Scale. The call graphs do not only contain information about the arguments and mechanisms of the called functions but also about the particular location of the trace records in the file create process. Next, we divided the call graph into several tasks while each step accounts for a particular logically connected segment. However, even with the large number of trace points in the file system, not every function contains trace points in its beginning and end which would ease the determination of the elapsed time. Thus, an additional requirement for the segment is that it gives us the ability to compute its time contribution to the total file create process, resulting in the merging of some tasks.

Latency distribution

As outlined in Section 1, the goal of this thesis is to understand in which parts of Spectrum Scale the time is spent during parallel file creation. Because mdtest only gives an overview of the create performance, we utilize the previously achieved trace points to measure each task's time consumption. However, one cannot make a general statement about the time consumption of each segment of the file create process. In other words, some files might be created faster while others require significantly more time due to token traffic or metanode communication.

In order to achieve this goal, we generated *latency distributions*[4] from all tasks of the create process. In such a distribution the elapsed time of a function is measured in multiple cases (files in our case) and scattered on the corresponding bins which represent the latency of the function. We defined 50 bins that range non-linearly from 1 microsecond to 500 milliseconds. As seen in Section 4.4, this range captures all file creates in a sufficient granularity while still being able to visualize the distribution in a graph. Furthermore, it is important to understand that latency distributions do not allow conclusions of the parallel behavior of an application as each file is measured individually. With this definition, we can easily observe if and how many files take longer for each segment of the file create process. Moreover, if a time consuming segment is found, the methodology can be reused for a deeper investigation. In this case, the acquired call graphs prove useful as they contain further information about a suspicious segment as well as its trace points.

Generating the latency distributions of all create tasks requires trace analyses. More specifically, the trace records need to be filtered for the determined trace points. This is done by a set of interactive Python [56] scripts, which initially parse given trace files for a number of chosen trace points. Trace files contain the trace records of all processes of a single node with timestamps relative to the time the trace collection was enabled. Therefore, the scripts filter all traces per process and cache them for further processing. Next, the user can choose a particular task that consists of a beginning and an end trace point. The scripts search for matching pairs through all filtered traces while computing their latency. Finally, the latency distributions will be created for a user chosen time frame for which the created graphs from wrapper

[4]Latency distributions are typically represented as a histogram.

scripts of the modified mdtest are helpful. In addition, the latency distributions are automatically plotted by *Matplotlib* [58].

Results and discussion 4

In this chapter we take a closer look at the file create process. More specifically, we analyze and discuss the results obtained using the tools presented in Chapter 3. First, we evaluate the output of the modified mdtest and interpret the created graphs. Next, we show general aspects of tracing Spectrum Scale in addition to interesting traces that were collected during the file create experiments. Then, we present the complete file create protocol, i.e., the logical steps which are required to create a file in the successful case, including the initial lookup. Finally, we discuss the latency distribution for the logical steps and some important internal functions called in the file create process.

4.1 Mdtest metadata benchmark

In Section 3.2.1, we described the modifications of mdtest which allow the analysis of fine-grained per-process performance at any specific time frame during an experiment. As a result, we discard mdtest's original output while only using the periodic output which the modified mdtest version provides. As a post-processing step, the periodic output, which describes the create performance at any point during the experiment, is used to generate multiple graphs. The modified mdtest will not only help with a preliminary investigation into file creates but is also a useful tool for the subsequent trace analysis (see Section 4.2). In this section, we will present the modified mdtest evaluation and discuss the corresponding graphs in detail. We will explain how different storage hardware influences the create performance and explore the impact of byte-range locking mode and FGDL. In addition, we will investigate internal mechanisms, influencing file create performance, e.g., directory block splits, inode pre-allocation, or metanode activity.

4.1.1 Workload and test sets

Before starting the mdtest evaluations it is important to define an appropriate workload, i.e., the total number of files that are to be created in a single benchmark run in a single directory. At the time of writing this thesis the Spectrum Scale developer team investigated and optimized the burst create performance. However, since redundant work is not desirable and burst performance cannot be maintained for an arbitrary workload, we focused on the sustained case which, among others, involves disk I/O and potentially more network communication. Both are, in distributed environments, the most expensive operations and can easily introduce a bottleneck for an application. We found that the creation of two million zero-byte files is sufficient for investigating the sustained performance on our test cluster. This workload requires more than 10 minutes to execute while the system is measurably doing I/O and constantly sending messages through the network. However, as further elaborated in Section 4.2.1, we will use a smaller workload later as two million files introduce problems during trace collection.

In a single run mdtest creates a number of zero-byte files a single directory, defined by a given workload, while measuring the current creates per second for a defined interval of files or seconds. When the benchmark is finished it removes all files and their corresponding parent test directory it generated earlier. A *test set* defines a single execution of the script which involves a number of mdtest runs that differ only in the number of clients as well as the number of processes used per client. In all those variations, the problem size remains fixed and is distributed onto all participating processes of all clients. This allows to evaluate Spectrum Scale's ability to exploit parallelism whenever additional nodes are used for the same workload. In HPC this methodology is also called *strong scaling* and it defines how the solution time varies with the number of processors for a fixed total problem size. We deliberately decided against *weak scaling*, i.e., keeping the number of files fixed per process as it would change the total workload in every benchmark execution, making the results incomparable for investigating parallelism.

At the start of every mdtest run, it is also important that the parallel file system remains in a similar state in each benchmark execution within a test set. Therefore, Spectrum Scale is restarted prior to each run. However, a scenario whereby an application is only executed after a file system restart is far from reality. In practice, many applications are accessing the file system simultaneously and rely on its availability. Furthermore, caching mechanisms, which increase the overall performance of the file system, could not be effectively used. In order to simulate a scenario in which applications are executed on a running system while also using a warm cache, the same mdtest run is executed prior to the actual measured mdtest execution. Henceforth, to avoid ambiguous terms we call the former the *warm up* and the latter the *experiment*. The pseudo code in Figure 4.1 shows a sample execution of a single test set, which executes a total of 12 experiments with a varying number of clients and processes. Note that the numbers of clients, processes, and files are multiples of two in order to evenly distribute the total number of files onto the participating processes.

We also evaluated the possibility of dropping various caches in addition to the Spectrum Scale restart. Generally, file systems utilize a so-called *page cache*, which is

```
clients := [1, 2, 4];
processes := [1, 8, 16, 32];
workload := 2097152;
for c in clients do:
  for p in processes do:
    unmount_filesystem();
    restart_spectrum_scale();
    mount_filesystem();
    execute_mdtest(c, p, workload); //warm up
    execute_mdtest(c, p, workload); //experiment
  od
od
```

Fig. 4.1. – The above pseudo code shows the workflow of an executed test set. Spectrum Scale is restarted prior to the warm up, whereas the measured experiment follows after the warm up.

stored in memory and used, for example, to cache dirty pages, i.e., modified pages that need to be flushed to the disk. However, file systems can also exist without a cache, though their performance might decrease. Linux also allows file systems to implement their own cache if the operating system's default file system cache is not feasible. Spectrum Scale is one example for implementing its own page cache. The so-called *page pool* is similar to the page cache but, among others, allows Spectrum Scale to have control over its size, contrary to the Linux page cache. Moreover, the page pool is part of the shared space for the kernel and user space. For entering a similar file system state before starting an experiment we also considered clearing Spectrum Scale's page pool. Yet, to the best of our knowledge, the parallel file system does not provide a mechanism for dropping the cache other than restarting the daemon. There is also the possibility to clear the Linux caches which include the page or dentry cache[1]. However, invalidating caches results in undefined behavior and may not even impact Spectrum Scale directly but applications that are used in conjunction with parallel file system. In this case, possible performance degradation may not be introduced by Spectrum Scale but rather by the programs which rely on cached information. In the end, dropping all caches is a scenario inapplicable in practice. Hence, we decided not to explicitly drop caches.

4.1.2 Benchmark strategy

With the workload described above, we ran dozens of test sets on our test cluster (defined in Section 3.1) with a varying numbers of clients and processes per client for different file system configurations and storage hardware. Each test set ran a total of 12 experiments where the workload was distributed on one, two, and four

[1]Clearing the Linux caches can be done with the command "`sync && echo 3 > /proc/sys/vm/drop_caches`".

clients, each with 1, 8, 16, and 32 processes per client. For every experiment, we decided to use an interval of five seconds for the periodic output of the modified mdtest. In other words, every data point contains information about the creates per second for the last five seconds per process (running performance) as well as creates per second since the start of the experiment (accumulated performance). In contrast to the total runtime, it is not meaningful to provide a standard deviation for each data point of an experiment. The create performance that is shown at those points is generally differing heavily in each experiment, depending on current and previous token traffic or metanode activity. For this reason, we do not present the standard deviation for the data points of the periodic output of a single experiment. However, it is noteworthy that different executions of the same experiment express very similar create performance with a standard deviation of ~2% for 10 experiment iterations with 4 participating clients (see Section 4.1.10). Nonetheless, we do not focus on the total runtime but on understanding the create behavior at any point during the experiment, which will implicitly explain fluctuations in the total runtime.

4.1.3 The warm up

As mentioned earlier, the warm up is used to provide a more realistic scenario for the experiments. We also compared the create performance of the warm ups with their consecutive experiments briefly. Commonly, the experiment is slightly faster than the warm up by up to 5% runtime. However, in some cases we could observe an unusually increased performance by up to 20% as compared to its usual runtime that also made them faster than the following experiment. This can be explained by undefined behavior due to the dropping of caches during a Spectrum Scale restart because subsequent experiments do not express a similar behavior and show a great consistency in their runtime. Nonetheless, we did not further analyze the above described performance of the warm up since it almost never is an issue in practice. In addition, understanding the influence of multiple caches requires a more complex investigation that was not within the scope of this study.

4.1.4 Client and process scalability

To investigate the scalability of file creates for a different number of clients, we first compared the accumulated performance for 1, 2, and 4 clients with 16 processes per client in each experiment, visualized in Figure 4.2. 16 processes should provide optimal parallelism for 16 cores with enabled hyper-threading per client. The workload amounts to 2,097,152 files that are created in each experiment on 44 HDDs. Every line shows the combined performance of all processes in a single experiment. The x-axis represents the time elapsed since the start of the experiment. The y-axis shows the accumulated average of creates per second since the beginning

# of clients	# of processes in total	creates/second	runtime in seconds
1	16	~4693	446
2	32	~3144	667
4	64	~2657	789

Tab. 4.1. – Original mdtest output: Average creates per second and runtime for three experiments with equal workload but different numbers of used clients and 16 processes per client.

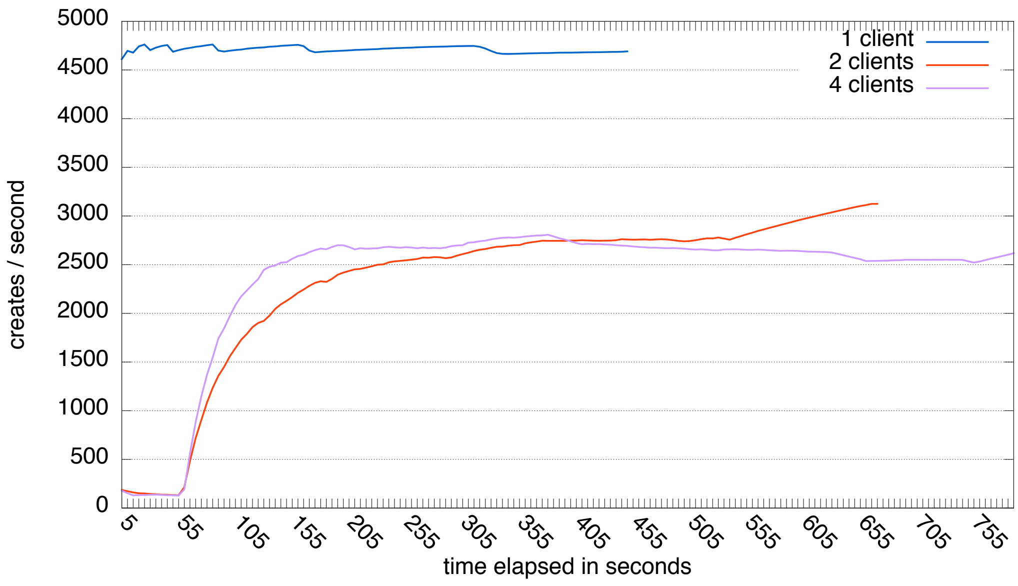

Fig. 4.2. – Investigation of client scalability: Comparison of the average accumulated creates per second in 3 experiments for 1, 2, and 4 participating clients with 16 processes per client with a workload of 2 million files.

of the experiment. Evidently, there seems to be a drastic difference in performance and runtime between one and multiple clients. A single client is able to create the given workload between a third to half of the time faster than multiple clients. In addition, four clients require significantly more time than two clients, even though four clients seem to achieve a higher create performance during the middle of the experiment. The graph indicates that multiple clients start off remarkably slowly for the first minute of the experiment in which only approximately 8,000 files are created. Therefore, ~0.4% of the workload accounts for ~9% and ~7.6% of the total runtime for two and four clients, respectively. In contrary, a single client does not show a similar behavior in the beginning with mostly stable performance throughout the experiment. Notably, the original mdtest can only provide one value for each experiment, that is, the respective last data point (c.f. Table 4.1), leaving the user without knowledge of the mid-experiment performance. In the following sections, we will investigate every aspect of the observed behavior to understand the cause of this performance while also suggesting possible solutions.

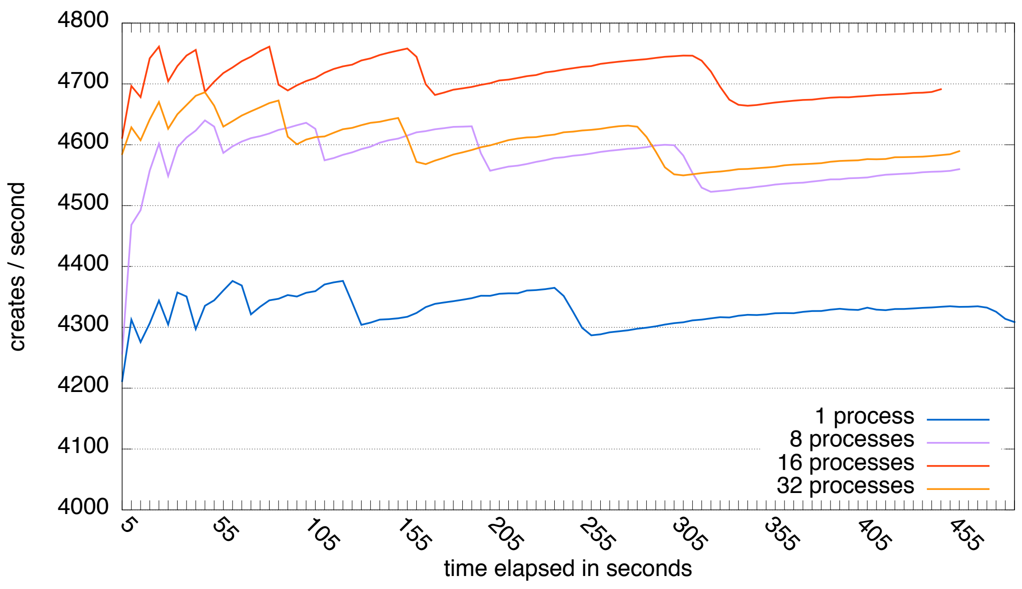

Fig. 4.3. – Investigation of process scalability: Comparison of the average accumulated creates per second in 4 experiments for 1 participating client with 1, 8, 16, and 32 processes per experiment with a workload of 2 million files.

Figures 4.3 and 4.4 demonstrate the process scalability examplarily for one and two clients, whereby each figure compares the average accumulated create performance for 1, 8, 16, and 32 processes per client to investigate the above assumption that 16 processes per client are optimal. Similar to Figure 4.2, the x-axis and y-axis show the elapsed time and the creates per second, respectively. Both figures show that 1 process per client is not able to provide the same create performance as 8, 16, and 32 processes per client which exhibit a similar create behavior consistently in all our executed test sets. However, we can observe a slight performance advantage (∼2%) in favor of 16 processes per client against 8 and 32 processes. This supports the assumption that the number of used processes per compute node should be equal to the number of available cores to maximize parallelism. Due to this observation, we will only present results of experiments in which 16 processes per client are used. In the case of one process per client we assume that the slightly worse performance is caused by missing parallelism that Spectrum Scale will exploit if multiple clients are utilized in an application.

4.1.5 Running per-process performance

Investigating combined-process accumulated create performance is only a preliminary attempt to explain the previously shown scaling problem with multiple clients. Clearly, it does not allow us to understand smaller differences in creates per second at a specific time frame, nor can it provide information about the individual per-process behavior. In Figure 4.5 we present the data gathered by the modified mdtest. In contrast to the previous figures in which multiple experiments were compared, it

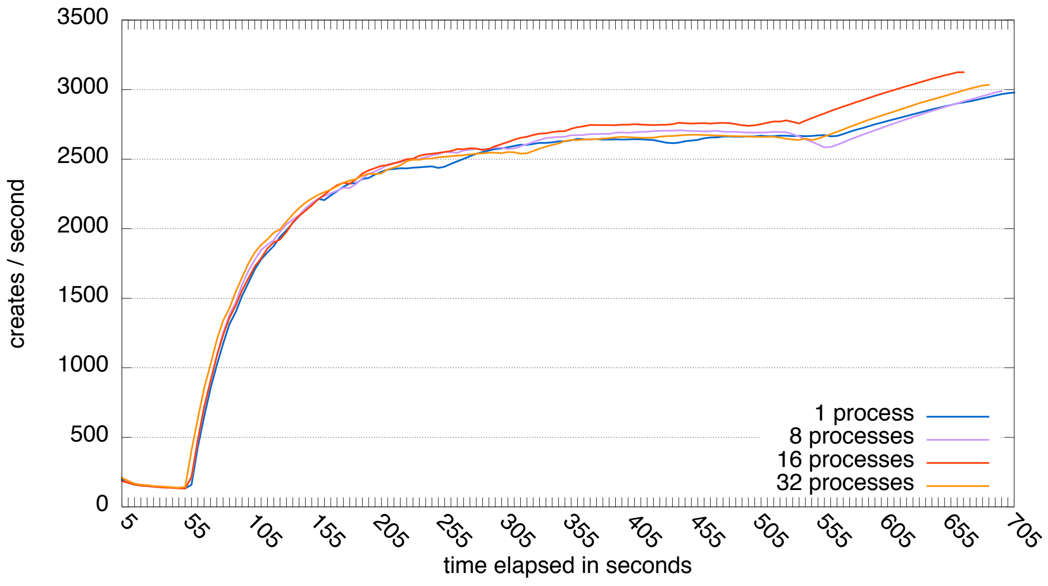

Fig. 4.4. – Investigation of process scalability: Comparison of the average accumulated creates per second in 4 experiments for 2 participating clients with 1, 8, 16, and 32 processes per experiment with a workload of 2 million files.

displays the per-process performance for a single experiment whereas a total of 2 million files are created by 4 clients, each of them utilizing 16 processes. The y-axis represents the creates per second for each process. The x-axis shows the elapsed time since the beginning of the experiment and, in addition, the number of created files by all processes at a specific point in time, shown in parentheses. Each data point of any process contains information about the measured creates per second for the last five seconds. For simple distinguishability, each process per line of the same color specifies the affiliation to the same client. At first glance, we observe that the create performance of one client is occasionally significantly higher than of the other clients. Consequently, a faster client finishes its workload sooner since every process is assigned a fixed number of files to create. In other words, client *lithium*, shown in blue, created all its files in roughly 385 seconds and then idled until the other three clients finished their workloads, essentially wasting parallelism.

In Section 2.5.4 we described that Spectrum Scale designates one cluster node as the metanode for each file or directory. In particular, the metanode role is assigned based on the first-touch policy, i.e., the first node who accesses the file will act as its metanode, defined by a specific (metanode) token. Other clients that access a file or directory with an active metanode initially request a metanode token from the token manager, receiving the active metanode's identity instead for future requests. When the metanode is inactive for 30 seconds a new metanode is selected from the accessing clients. Therefore, all metanode processes can locally read and update the directory's metadata, including directory blocks. On the other hand, non-metanodes have to send requests over the network for the same tasks. This advantage of the

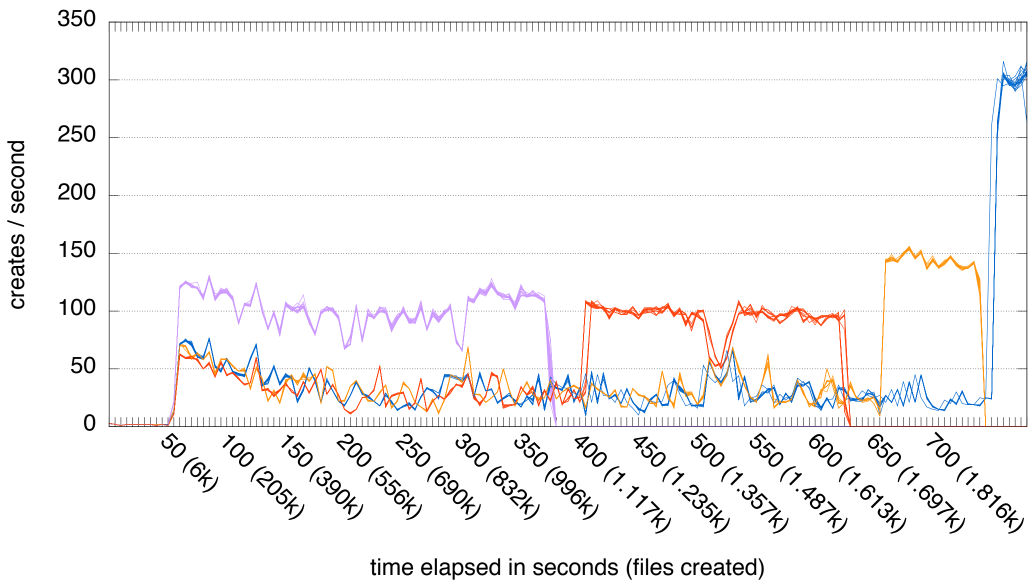

Fig. 4.5. – Investigation of the running per-process create performance in a single experiment with 4 clients and 16 processes per clients where 2 million files are created in total. Each line represents a single process while its color visualizes its client affinity: Lithium (purple), neon (red), chlorine (orange), and boron (blue). Every data point shows the create performance per-process for the last five seconds.

metanode suggests that exactly one client should outperform all of the others at any time and, in fact, this can be seen in Figure 4.5.

Second 385 shows the first acting metanode (lithium) finishing its workload. After a grace period of 30 seconds, a new metanode is selected out of the remaining clients, called a *metanode transition*. While the metanode's create performance does not seem to influence the other clients, it conversely improves the fewer clients that are actively creating files. This is especially visible in the last seconds of the experiment where a single client is left to finish its workload. The above assumption can be confirmed by investigating the traces. In Section 4.4, we will further investigate metanode performance with trace analysis and point out what is causing the above described behavior.

What is left to explain is the poor performance of all clients during the first 60 seconds of the experiment in which only approximately 8,000 files are created (refer to Figure 4.5). We can explain this behavior when we remember that a directory can remain in either of the two locking modes: Byte-range locking and FGDL, described earlier in Section 2.5.4. By default, the former is used for any file whereas directories are able to transition into FGDL at a later time. FGDL is a recent feature, specifically designed to improve file create performance. Considering that the directory starts off in byte-range locking mode, there has to be a mechanism which also triggers the transition into FGDL. Because mdtest's results cannot provide an answer for this issue, deeper investigation is needed to understand this mechanism which we will further discuss in Section 4.2.2. However, we forestall that traces can indeed confirm that the directory remains in byte-range locking mode for the first ∼8,000 files being created. A directory block can contain thousands of entries, each of them representing a file in the directory. The byte-range locking mode forces every process to request an exclusive write token to modify the directory block in which an entry will be inserted, essentially serializing the access to the directory block. During that time tokens will be granted to a client but invalidated almost immediately when other clients also require access to the same directory block. Therefore, leading to countless token conflicts (known as *token thrashing*) and many token requests, slowing down overall performance significantly, as can be seen in the first minute. This also explains why experiments with a single client are not expressing a similar issue since an exclusive write token has to be granted only once.

4.1.6 Logging

In the previous section, we have examined the overall create performance in a four client experiment and interpreted the behavior solely with modified mdtest's results which we could also confirm with traces. However, we did not discuss the cause of small performance fluctuations that can be observed at any data point for every client (see Figure 4.5). Apparently, every process experiences a shift of up to ∼30 creates per second in both directions in consecutive data points. One possible reason for this behavior could be the flushing of dirty inodes and directory blocks from memory to disk, disrupting file creation every few seconds. This process can be invoked by three conditions: Firstly, a node has to flush dirty data when it runs out of memory. However, since inodes and directory blocks are relatively small in size (in our case with 4 KiB and 256 KiB, respectively) and each client has 15 GiB of memory available, it is unlikely that it runs out of memory with a workload of 2 million files. The second mechanism is a periodic background synchronization feature which is triggered for metadata every 30 seconds to ensure consistency of the file system. Nonetheless, the fluctuations can be observed much more frequently. Finally, the last mechanism, the so-called *log wrap*, triggers the flushing process as soon as the journal is full.

When log records are written they are initially stored into a log buffer in the memory of a client. Its size is dependent on the configured block size of the file system. When the log buffer is full it is flushed to an on-disk circular log whereas its size is configurable per file system[2]. The data on the circular log can only be discarded if its contents are already consistently stored on disk. As long as the on-disk log and buffer are full no file is allowed to be created. When this occurs, log-wrap is triggered. In the current state of Spectrum Scale, a single process is assigned to flush the log

[2]Our file systems use a log size of 16 MiB.

by requesting the daemon process to send the I/O request to the NSD server node. During the flushing of log records the daemon blocks further execution until the I/O is complete to preserve atomicity. Unfortunately, mdtest's results in conjunction with different file system configurations do not allow a complete explanation without a deeper investigation into trace information which we will discuss in Section 4.4.5.

4.1.7 Directory block splits

The earlier presented Figure 4.5 shows not only constant fluctuations in creates per second but also occasional and sudden performance drops. Interestingly enough, we observed these dips (c.f. to Figure 4.5 at seconds 210, 300, and 525) in every experiment at a similar number of created files. This behavior can best be seen in experiments with a single active client since it does not express token or metanode communication as visualized in Figure 4.6. The given figure presents the abovementioned problem and shows an experiment with 44 HDDs whereas 1 client with 16 processes creates 2 million files. The x- and y-axis representation is equal to Figure 4.5. Earlier in Section 2.5.4 we elaborated on the complexities that arise during a directory block split. In summary, a split causes Spectrum Scale to invalidate all FGDL tokens corresponding to the affected directory block from all non-metanodes as the metanode requires exclusive access to it. Any client trying to create a file with a filename that hashes to this directory block has to wait until the metanode completes the directory split.

Due to the randomness of hashing it is reasonable to assume that all directory blocks fill up equally, blocking all clients to create files at roughly the same time due to directory block splits. Moreover, each event is splitting twice as many blocks than previously, prolonging the required time to split all blocks. This can be easily observed in Figure 4.6 as every dip introduces a greater drop in performance than the prior one. Supposing the above also implies that all directory blocks can contain twice as many entries and files than after the preceding split because the number of blocks is doubled with each split. This would allow us to predict future directory block splits at a particular number of files. However, according to Figure 4.6 this is clearly not the case. As it turns out, the original implementation of mdtest generates filenames of increasing length depending on the numbering of the file. The benchmark labels the filename with the process number and the proceeding file number. Thus, the longer the experiment runs the longer the filenames will be, resulting in larger directory entries. Consequently, we suspected that more entries can be placed into a directory block in the beginning of the experiment (rather than the end) due to the increasing filename length.

In order to confirm the above stated assumption, we adjusted mdtest for generating filenames of equal length (30 characters) throughout the experiment (see Figure 4.7).

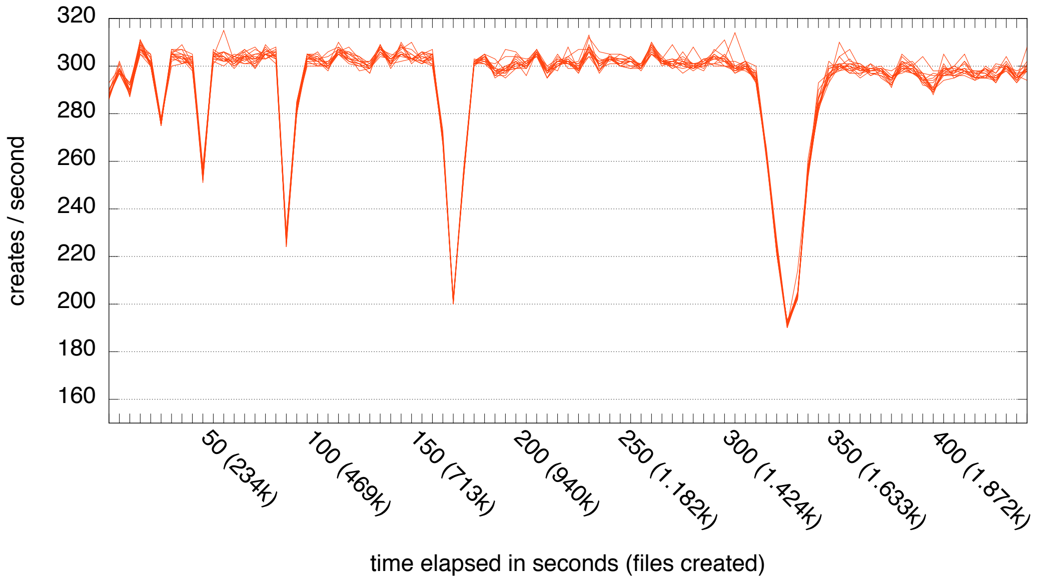

Fig. 4.6. – Investigation of the running per-process create performance in a single experiment with 1 client and 16 processes where 2 million files are created in total. Every data point shows the create performance per-process for the last five seconds. In the beginning of the experiment files with a shorter filename length are created compared to the end of the experiment.

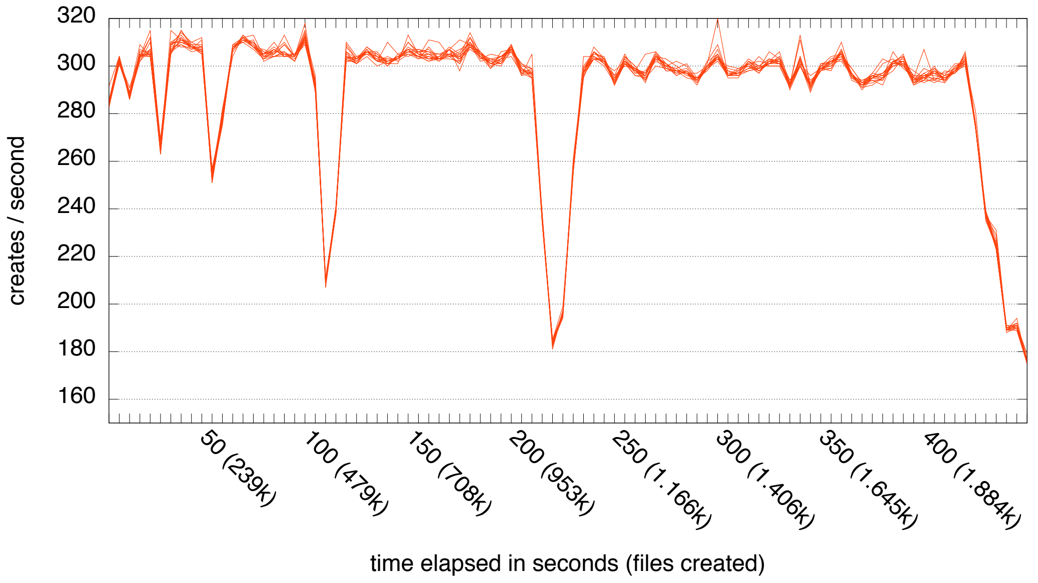

Fig. 4.7. – Investigation of the running per-process create performance in a single experiment with 1 client and 16 processes where 2 million files are created in total. Every data point shows the create performance per-process for the last five seconds. Files with an equal filename length are created throughout the experiment.

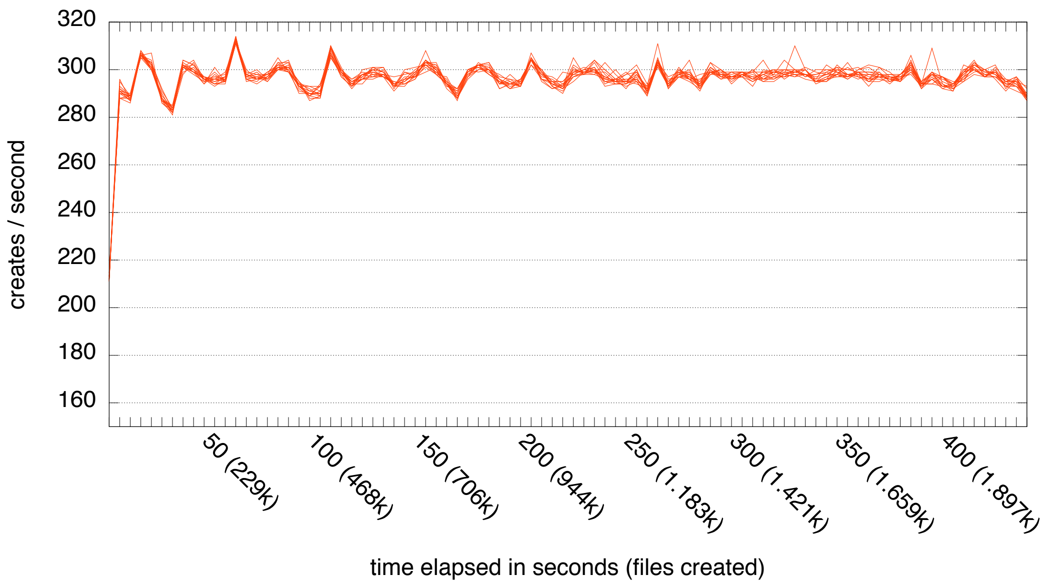

Fig. 4.8. – Investigation of the running per-process create performance in a single experiment with 1 client and 16 processes where 2 million files are created in total with pre-allocated directory blocks. Every data point shows the create performance per-process for the last five seconds. Files with an equal filename length are created throughout the experiment.

Spectrum Scale's own `tsdbfs` tool revealed that the file system allocates multiples of 16 bytes for the filename (with a minimum of 32 bytes). Over the course of the original mdtest experiment, filenames consumed 32 to 64 bytes of space. With the modified mdtest, all filenames consumed a constant amount of 64 bytes. While obviously reducing the number of files per directory block, the latter experiment also sped up the initial byte-range locking mode duration by 30 seconds. Considering that the first split now happens with approximately 4,000 files created[3], we concluded that the splitting event at second 220 generates 512 directory blocks in total that contain up to almost two million files. Towards the end of the experiment, we can observe another directory split containing the remaining number of files that did not occur in earlier experiments with the same workload. Due to the additional block split the runtime is also reduced by 1.7% in respect of the experiment shown in Figure 4.6. Finally, traces can confirm this behavior.

Assuming an average per-thread performance of 305 creates per second and neglecting the need of directory splitting, we could see an improvement of up to 4% in total runtime when compared with the experiment shown in Figure 4.6[4]. Unfortunately, Spectrum Scale provides no functionality, such as pre-allocating a number of directory blocks in advance. Furthermore, Spectrum Scale collapses and

[3]These numbers only apply for file systems with a block size of 256 KiB and with a filename length that requires 64 bytes in a directory entry.

[4]We did not calculate the theoretical runtime in the scenario of multiple clients due to unforeseeable behavior of non-metanodes and token traffic.

merges splitted directories as soon as enough files are removed from the directory by default, which we disabled with a particular configuration. Thus, with further mdtest modifications, we could evaluate experiments whereby all required directory blocks were already allocated in the beginning of the experiment (see Figure 4.8). Despite using pre-splitted blocks, we are not able to observe a noteworthy performance improvement. In fact, the overall creates per second for each process is slightly lower than before. This was especially apparent in multiple client experiments whereas create performance steadily decreased over time for the metanode. Considering that enabling the mentioned mechanism is not the default, we did not deeper investigate this behavior.

Furthermore, we evaluated the total number of files that a directory can contain with a filename length of 64 bytes. In concept, assuming a maximum of 2^{20} directory blocks (see Section 2.5.3) and a file system block size of 256 KiB, a directory can contain up to 2^{32} files with a filename length of 64 bytes. However, since this number exceeds the 32 bit offset to the `readdir()` function [28], the real limit decreases to 2^{31} files.

4.1.8 Inode pre-allocation

As described in Section 4.3.2, the first step in creating a file is to allocate the corresponding inode. When creating a Spectrum Scale file system, the user is allowed to set the maximum number of available inodes as well as the number of pre-allocated inodes. The former configuration prevents applications from running out of inodes which would prohibit the creation of files and directories regardless of available disk space[5]. If the number of pre-allocated inodes is not given by the user, Spectrum Scale will set it accordingly, depending on the maximum number of inodes. When no more pre-allocated inodes are available the *allocation manager* will allocate a batch of inodes automatically, called *inode expansion*. This method avoids lock conflicts on the inode allocation map which would occur if each process tried to allocate its inode individually. Logically the inode allocation map is a bit map in a sparse file whereas its logical size is dependent on the maximum number of inodes. An inode remains in one of four states, represented as bit pairs: *In-use*, *not in-use*, *being-created*, and *being-deleted*. The last two states are transitional because of situations that make atomic allocation and deletion ineligible.

When creating a file Spectrum Scale has to ensure that two nodes are not using the same inode. The file system solves this issue by locking the inode allocation map. However, locking the whole map would results in a lock conflict on every access. For this reason, it is by default divided into 32 separately lockable segments. The

[5]The maximum number of inodes can also be updated after the creation of the file system.

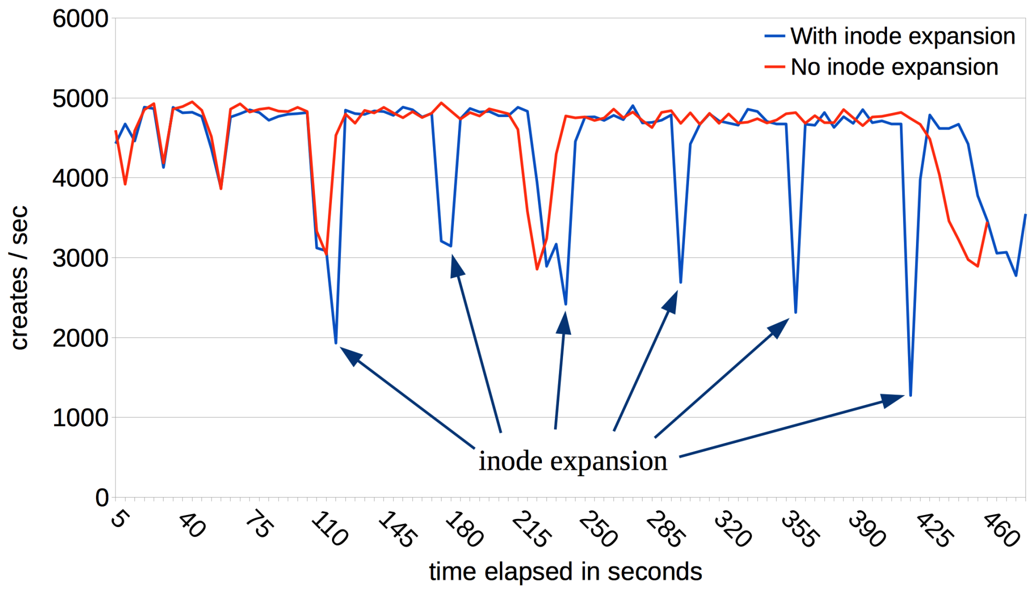

Fig. 4.9. – Investigation of the running performance of two experiments, each creating 2 million files in total with a single client and 16 processes. Every data point shows the create performance of all processes for the last five seconds. The experiment (shown in red) had enough inodes pre-allocated while the client in the other experiment (shown in blue) had to expand the inode allocation map several times (pointed out with arrows).

number of segments is dependent on the expected number of used nodes in the file system which can be set by the user when creating the file system. In order to further reduce locking conflicts on the segments, clients try to lock the same segment as long as free inodes are available in them. Therefore, the allocation manager node has to lock the whole map in order to expand all segments equally. As a result, all clients have to wait until the process is finished, blocking operations that require access to the inode allocation map.

Toward analyzing the impact of expanding the inode allocation map, we executed two types of experiments: Firstly, with enough pre-allocated inodes available and, secondly, with the reverse case where only 500,000 inodes are allocated and the inode allocation map need to be expanded during the file create process. The Figure 4.9 exemplarily compares two experiments in which a single client was used for creating two million files. For clarity, all processes are merged together and the corresponding inode expansion events are pointed out. In the cases of one and two participating clients inode expansion accounts for ~4% and ~1.2% of performance degradation, respectively. Note that inode expansion and directory block splits can overlap (see seconds 120 and 230 in Figure 4.9). In the case of four clients we found that inode expansion does not measurably impact the total runtime in our experiments.

4.1.9 CPU utilization

Although the file creation task has not been found to be particularly CPU demanding, extensive monitoring shows a different CPU utilization based on the node role: In a 4 client experiment with 16 processes per client, non-metanodes and NSD servers almost never exceeded 5% and 2% of total CPU usage, respectively. The metanode consistently showed a total CPU utilization of ~20%[6]. This increase of CPU usage can be explained by the requests that the metanode has to answer.

We evaluated the impact of the CPU speed by decreasing the maximum clock frequency for all abovementioned node roles. However, this did not measurably influence the runtime of our experiments. As a result, we did not further investigate CPU utilization. Nonetheless, the CPU can play a bigger role in environments where hundreds of nodes are participating in the file create process in which the metanode has to respond to many more nodes than in our test cluster, potentially becoming overwhelmed in the process.

4.1.10 Performance of varying storage

Before starting with the trace analysis, this section investigates the impact of different storage subsystems on the overall performance of creating two million files in a single directory. Each workload was run 10 times repeatedly for each storage subsystem with 1, 2, and 4 participating clients. Five different storage systems were used: 44, 22, and 11 HDDs, as well as 2 SSDs, and 2 RAMDisks. As described in Section 3.1, the HDDs are enclosed in a storage array, protected by a shared non-volatile cache, and redundantly accessible through both NSD servers. However, the SSDs and RAMDisks are not part of this architecture as one SSD or RAMDisk is directly connected to each NSD server. Therefore, each SSD and RAMDisk is only accessible through their controlling machine. Note that each storage subsystem requires a separate file system.

Figure 4.10 presents the create performance of all above experiments. The x-axis shows the five storage subsystems whereas the y-axis represents their creates per second. Each data point represents the average creates per second over 10 iterations, including the corresponding standard deviation.

All storage subsystems show consistently that four clients require considerably more time to finish their workload as compared to others. Furthermore, all storage variations express similar performances for the number of used clients, suggesting that the creates per second are not limited by the I/O latency in our test cluster

[6]One and two client experiments expressed a similar CPU utilization.

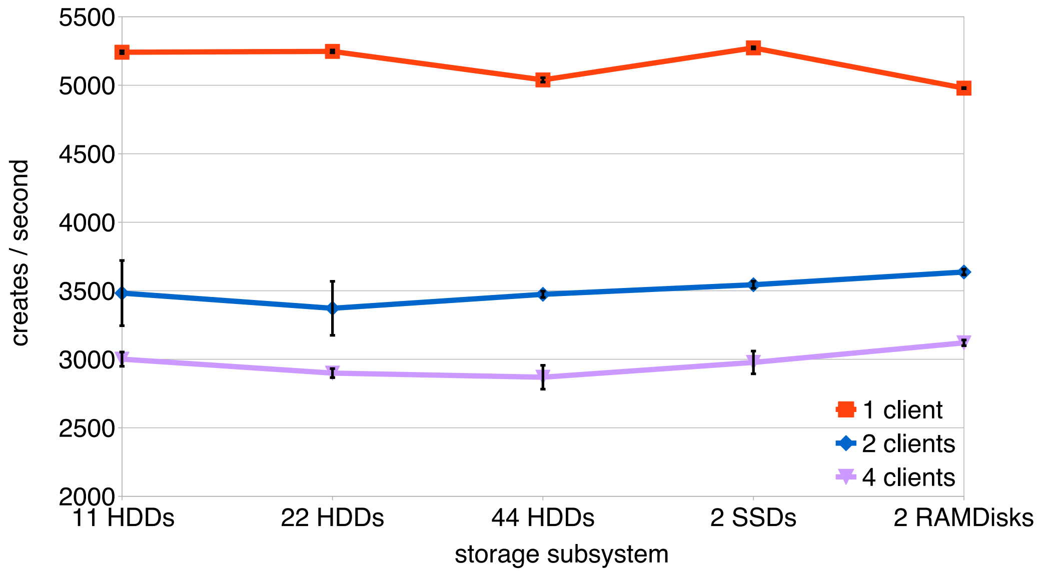

Fig. 4.10. – Spectrum Scale's file create performance for creating 2 million files on different storage subsystems with a varying number of participating clients and 16 processes used per client. Each data point shows the average creates per second over 10 experiments.

setup. However, it is important to note that all HDD configurations represent the I/O performance of the non-volatile controller cache rather than the hard disk's. This is because data written to the cache is already considered persistent, although it may not have been stored on the hard disks. In this case the cache size is sufficient to buffer the written data, explaining the similar performance compared to SSDs and RAMDisks.

4.2 Trace analysis

In the previous section we have seen how the modified mdtest can be used to make certain assumptions about the metanode's behavior. However, mdtest is not only a tool for a black box create performance evaluation, but is also useful for trace analysis. For instance, the modified mdtest helps us to understand at what time during an experiment a particular event happened which we can then easily analyze with the generated traces. As described in detail in Section 3.2.1, the wrapper scripts of the modified benchmark enables tracing automatically during the create process which allows to align the timestamps of the modified mdtest with those of the traces. In the following sections, we will discuss challenges that arise when working with traces and investigate when Spectrum Scale is switching from byte-range locking to FGDL mode.

4.2.1 Performance degradation

Spectrum Scale provides two tracing modes: blocking and overwrite. The former is used by default and collects traces in a configurable buffer of up to 16 MiB. Flushing the buffer to the disk causes all applications to be temporarily blocked. This implies that increasing the output (e.g., by enabling a large number of trace points) negatively affects the file system's performance. Consequently, the blocking mode is only applicable if either the performance is of no interest or if the storage system is fast enough to prevent application blocking. The overwrite tracing mode does not issue any I/O to the underlying storage and instead utilizes a ring buffer in memory, beneficial for preserving the file system's performance. When the ring buffer is filled it continues writing from the beginning of the buffer, overwriting previous collected trace records. Because a 16 MiB buffer (by default) is saturated almost instantly it can be set to a higher size to guarantee that no information is lost. When tracing is stopped all traces are flushed to the disk in a compressed state on a per-client basis.

Unfortunately, two million created files are already generating too many traces even with a single subsystem active, despite an overwrite buffer of 8 GiB, which already accounts for half of each client's memory. This led to the decision to reduce the workload to half of a million files, which is still an adequate number to provide a stable latency distribution over multiple executions (refer to Section 4.4). However, reducing the workload only solves a part of the problem since the trace size is not the only concern. File create performance can already degrade if only two subsystems with a high level of details are enabled. For instance, an experimented created 2 million files with 4 clients and all traces enabled in overwrite mode with a level of detail of 8 and required over 26 hours to complete with an average of 21.8 creates per second. This corresponds to a slow-down of a factor of 120 in comparison to an experiment without tracing enabled, taking only $\sim$13 minutes to complete. Although this is an extreme example, it shows how expensive tracing can be even without the additional I/O caused by the blocking mode.

Nonetheless, in most use cases the user is not interested in a large number of trace records which are provided by a single subsystem. Usually, a whole subsystem is activated just for the sake of a single trace message, leaving much overhead due to the remaining, undesired messages. However, Spectrum Scale does not yet provide a mechanism to specifically turn on a particular trace. Such an option could not only benefit analyses of specific bottlenecks but might also introduce the possibility of enabling tracing in productive environments, as the performance degradation of a single trace message is negligibly small.

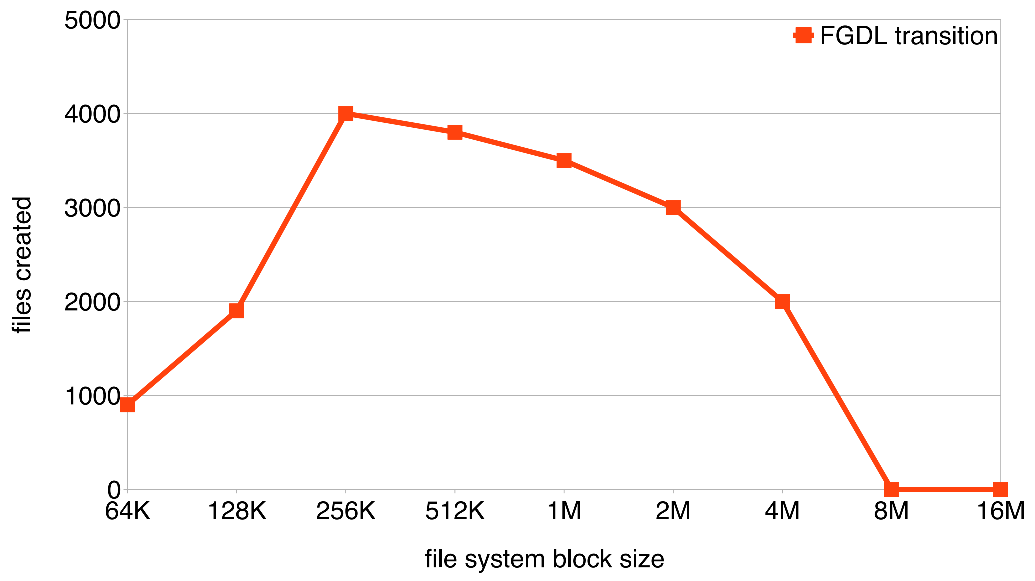

Fig. 4.11. – Number of files that are required to transition into FGDL for all available block sizes in Spectrum Scale.

4.2.2 FGDL and file system block size

In Section 4.1.7 we showed that directory splits are responsible for sudden performance drops although they do not account for much of the overall runtime. We observed that the performance increase of FGDL aligns roughly to the first directory split, concluding that FGDL is enabled as soon as two directory blocks are in use. Thus, different block sizes should have an influence on the time when FGDL is activated because smaller blocks need to be split sooner. For file systems with a block size of 256 KiB, the modified mdtest shows that FGDL is activated after approximately 4,000 files are created with a filename of 64 bytes (confirmed by traces). Therefore, file systems with a block size of 128 KiB should transition into FGDL after 2,000 files are created whereas more files need to be created for larger block size file systems, prolonging the byte-range locking phase. Figure 4.11 presents different Spectrum Scale file systems with all its supported block sizes and points out how many files are required to trigger the transition into FGDL. As expected, directories in file systems with a block size smaller than 256 KiB transition into FGDL with less files. However, this is apparently not the case for larger block sizes. In addition, 8 MiB and 16 MiB block size file systems seem to start in FGDL mode from the beginning of the experiment. This behavior can be explained considering that Spectrum Scale is able to allocate smaller fragments of a block size for files. The size of one fragment is 1/32 of the actual block. The main benefit of using fragments instead of whole blocks is to save disk space. Without fragments large block size file systems would waste a lot of space if the contents of a file only accounts for a few bytes of disk space.

The source code and traces revealed that a directory block usually corresponds to the block size with an upper limit of 256 KiB. We found that Spectrum Scale enables FGDL for a directory as soon as either the last fragment of a block is allocated (applies to block sizes of 64, 128, and 256 KiB) or a fragment of a block is allocated that would increase the size of the block to the largest allowed directory block size. The latter applies for file systems with larger block sizes of 256 KiB as less fragments are required to reach the maximum directory block size. For example, with a block size of 2 MiB the third fragment corresponds to 192 KiB, allowing up to 3,000 files to be created. The fourth fragment already allocates the complete directory block with 256 KiB, enabling FGDL with 3,000 files in the directory (see Figure 4.11). Therefore, FGDL is enabled before the first directory split, which is only triggered when the directory block is completely filled with entries. For file systems with a block size larger than 4 MiB, the first fragment accounts for at least 256 KiB, which is the reason FGDL is immediately in-use with the first fragment.

From Spectrum Scale version 4.2 onwards the parallel file system introduced a new configuration setting, called *AvoidDirFragments*. It disables the allocation of fragments in the case of directory blocks only. Therefore, the first directory block is always completely allocated when a directory is created, resulting in the immediate use of FGDL for the first created file in the directory, regardless of the block size. However, our test cluster uses Spectrum Scale version 4.1.1 in which this configuration is not yet available. Nonetheless, we were able to confirm the benefit of *AvoidDirFragments* on an IBM test cluster with an installed Spectrum Scale developer version.

4.3 File create protocol

The creation process, initialized by an `open()` (with an `O_CREAT` flag) or `creat()` (deprecated) system call, involves three consecutive VFS function calls:

Lookup – `gpfs_i_lookup()`

The function is called when the file system needs to look up an inode in a parent directory. The path and filename to look up are given in an otherwise empty dentry object. The function is called for every directory in the path while their dentry objects will be cached for future lookup calls. The lookup returns a positive error code and the found inode in the given dentry if the lookup succeeds. Otherwise, a negative error code (`ENO_EXIST`) is returned with a `NULL` inode in the given dentry object – the so-called *negative dentry*.

Create – `gpfs_i_create()`

The function is called when the file system needs to create a new inode in a parent directory. The given dentry is a negative dentry. The function will return the dentry with a newly created inode if successful.

Open – `gpfs_f_open()`
The function is called when the file system needs to open an inode in a parent directory. The function creates a new file object and links it to the corresponding inode object. The existing inode and empty file objects are given as arguments. The function will return the newly initialized file object if successful.

In the following sections, we will present the logical tasks for the creation of a file in the successful case, that is, the lookup returns a negative dentry and the file can be created. As we will see in Section 4.4, `gpfs_f_open()` hardly contributes to the total time required to create the file. As a result, we did not further investigate the performance of `gpfs_f_open()` and its protocol as part of this study. The protocols for `gpfs_i_lookup()` and `gpfs_i_create()` were achieved by creating the corresponding call graphs with the help of traces and manually investigating the source code. Each logical task represents a particular abstracted segment of the create process and contains a number of subsequent functions calls.

4.3.1 Lookup

The lookup is executed in the kernel-space, except where indicated otherwise. The following enumeration presents the logical tasks of `gpfs_i_lookup()`.

1. Fast lookup
Firstly, the client tries to quickly look up a file in a directory. It is an optimistic, fast version that only targets a single directory block. However, there are a number of conditions that have to be met beforehand. Among others, those conditions require that the directory inode and directory block are already cached and valid. For file creation this is generally not the case. If at least one condition is not met, the fast lookup is skipped.

2. Lock the directory file
The client obtains a read-only lock on the directory inode in which a file is looked up to access the permission data on the directory and ensure that other nodes cannot delete the directory while the client has a lock on it.

3. Verify permissions
The client verifies that the process can be granted access to look up the file in the directory.

4. Acquire FGDL token and directory entries

The client hashes the filename and determines (with extentible hashing) in which directory block it should reside. Next, the client sends a message to its daemon to acquire an FGDL token for the corresponding directory entry of the filename. Because the client might create a file later, implying the insertion of a new entry into the directory block, it always requests an exclusive-write token (although it would only require a read-only token for looking up the file). After obtaining the token, the daemon will request the directory entries of the whole block from the metanode and cache them. If the client is the metanode, it will have local access to all up-to-date directory blocks.

5. Lock the directory block

The client obtains a read-only lock on the directory block that it retrieved in the fourth task. If the FGDL token is invalidated between the two tasks, it will repeat the prior task. In addition, the file system will drop access to any previously locked block.

6. Search for the filename in the directory block

Before the client searches for the filename, it verifies that the correct block is locked multiple times and checks for directory corruption, for example. Next, the client looks for an entry that matches the searched filename in the looked directory with the help of a hash table. The hash table avoids the need to iterate over all entries in the block. If the filename is not found, the function will return `E_NOENT`.

7. Release locks

Finally, the client releases all locks that it has obtained earlier. Note that it does not release the obtained tokens.

4.3.2 Create

The `gpfs_i_create()` function is run in the kernel-space while, in the beginning, sending a message to the Spectrum Scale daemon. The whole create process is then executed in the Spectrum Scale daemon (user-space) as it is allowed to do I/O. Thus, the following logical tasks are an excerpt of the daemon's functionality to create a file.

1. Allocate inode

The client obtains a token for a particular segment of the inode allocation map and locks it. If there is no free allocated inode available, it will first try to target other segments which may be locked by other clients. If there are still free allocated inodes available, it will choose a free one randomly and set it to the transitional state *being-created*. If approximately 80% of the allocated inodes are in use, the inode allocation map expansion is triggered. In addition, parts of the log buffer are reserved and the corresponding records are written.

2. Lock file and directory file

The client obtains a read-only lock on the directory inode, similar to the second task in the lookup process. Furthermore, it requests a token for the allocated inode and locks it.

3. Get the inode of the directory

The client validates the cached inode of the directory. If it is not valid and the current node serves as the metanode, it will be read from disk and revalidated. Non-metanodes request the metanode to send them the up-to-date inode.

4. Acquire FGDL token and directory entries

If the client has lost the FGDL token between the lookup and the create function, it will request the FGDL token again and the metanode to send the entries of the corresponding directory block, similar to the fourth task of `gpfs_i_lookup()`.

5. Search for the filename in the directory block

The client acquires a read-only lock on the directory block and searches for the filename, similar to the sixth task of `gpfs_i_lookup()`.

6. Various verifications

The client verifies various properties: For example, it checks that the inode of the new file is valid and not in use, it verifies that the filename does not exist in the parent directory again, and that the process is allowed to create files in the parent directory.

7. Create metadata of the new file

The client initializes the metadata in preparation for a new file that is being created. Inode header fields will be initialized and metadata attributes set, such as *creation date and time*.

8. Reserve space in the directory block (metanode only)

The metanode of a directory reserves space for a new entry in the corresponding directory block. If the directory is in FGDL mode and the directory block cannot contain additional entries, it will split the block. Splitting is done by copying the directory block and removing those entries which do not belong in either block, being determined by extendible hashing. After the split is finished, the metanode flushes its buffer, writing any required log records to the disk. Until the split is finished no entries can be added to the affected blocks.

9. Ask metanode to reserve space in the directory block (non-metanode only)

Non-metanodes will request the metanode to reserve space for a new entry in the corresponding directory block on their behalf.

10. Finish file create and finalize log

The client finishes the initialization of the cached inode and marks it as dirty. In addition, parts of the log buffer are reserved and records are written for the directory block changes and the inode updates.

11. Insert entry into the directory block

The client updates the directory block with the new entry for which it has reserved space earlier and marks the directory and inode update as complete. Log records describe the directory block change. Note that the non-metanodes do not explicitly send a message to the metanode that describe the directory block insertion within the create process. Instead, when the flushing process is triggered on non-metanodes by one of the three mechanisms discussed in Section 4.1.6, the writing of the corresponding directory blocks will be redirected to the responsible metanode that collects all directory updates.

12. Release locks

The client indicates that the update to the cached inode and directory block was successful and that the corresponding log records have been written. In addition, the inode state is changed from *being-created* to *in-use*. Finally, the client releases the log reservation and all locks. Then, it sends a message back to the kernel process that the file was successfully created, initializing the corresponding virtual inode.

4.4 Latency distribution

In the previous sections, we analyzed the overall create performance in different cases and presented reasons that explain Spectrum Scale's behavior at any time during an experiment. In addition, we looked at potential performance degrading mechanisms, such as directory block splits. Finally, we introduced the internal file create protocol of Spectrum Scale whose logical tasks are needed in the following sections to understand the time consumption in each step by presenting their corresponding latency distribution. It is important to understand that a latency distribution cannot provide data about parallelism as each task will be treated individually. Thus, we are able to understand which tasks are the most time expensive ones, allowing us to point out bottlenecks that might be worth optimizing. In the following sections, we discuss the latency distributions for the most important logical tasks of the lookup and the create processes. In addition, we investigate logging and I/O performances on different storage hardware. Except otherwise indicated, all experiments create 500,000 files on 44 HDDs. We will show a varying number of clients, whereas each client always runs 16 processes.

4.4.1 Constraints

Generating latency distributions is subject to unavoidable constraints because they can only be created by analyzing and processing trace records. For example, enabling traces on a Spectrum Scale cluster is an asynchronous process since every client writes its own trace file, resulting in slightly different starting times of up to a few seconds.

Other difficulties arise from potential performance degradations caused by tracing in general. As discussed previously in Section 4.2.1, collecting many traces can have a disastrous effect on the creates per second in an experiment and may introduce storage issues when using overwrite mode due to the danger of overwriting traces in the buffer. In consequence, we used a workload of 500,000 files and carefully evaluated the appropriate trace subsystems and levels of detail for investigating each logical task to preserve create performance as best as possible. However, this led to the inconvenience of executing multiple experiments for creating the latency distribution of consecutive or connected logical tasks. For example, two experiments with different trace submodules enabled had to be run for evaluating the tasks of the lookup function. Nonetheless, we treat the impact of this inconvenience as negligible. In all our experiments, Spectrum Scale provides a consistent performance, producing almost identical latency distributions for repeated benchmark executions, including smaller workloads of 100,000 files.

Moreover, a latency distribution can only be provided for a single experiment due to preserving comparability and correctness. Consequently, we will only exemplarily show a single experiment per investigation, i.e., experiments with a specific trace subset enabled, in the following sections. They represent one latency distribution out of the dozen that we have generated in our analysis while the differences in time consumption of each logical task is minor and generally consistent.

4.4.2 VFS layer

Initially, we traced the virtual file system layer of Spectrum Scale which represents the entry point of Linux VFS to Spectrum Scale VFS for achieving a first overview of the latency distribution for each file create. As described in detail in Section 4.3, the create process involves the three consecutive VFS function calls `lookup()`, `create()`, and `open()` that correspond to the Spectrum Scale VFS func-

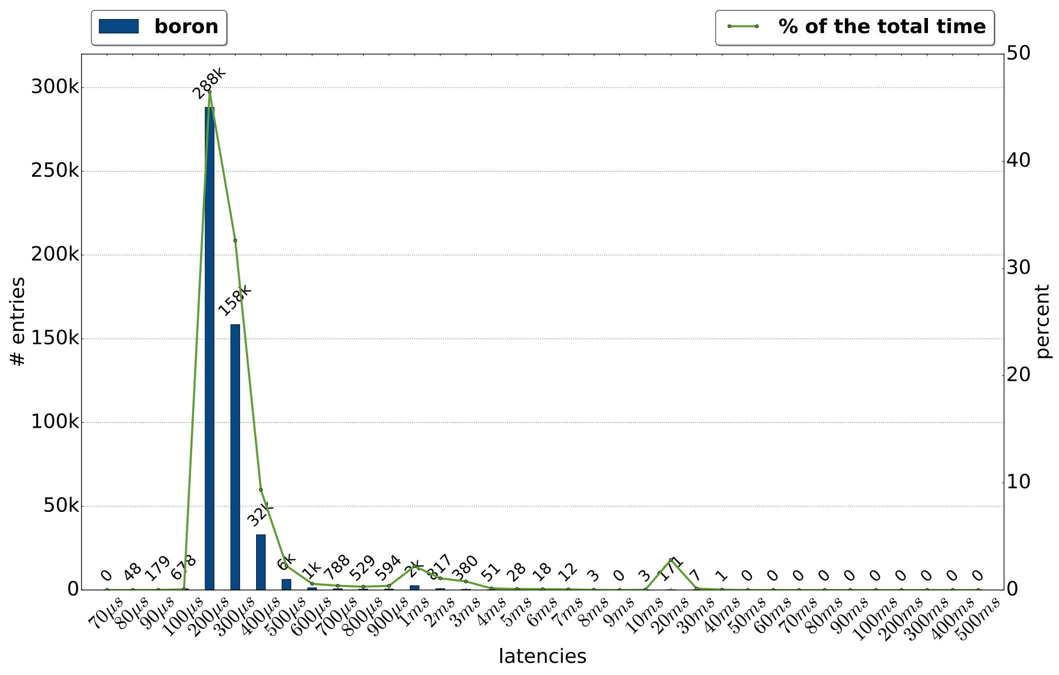

Fig. 4.12. – Latency distribution of a single client creating 500,000 files. Each entry in a bucket corresponds to the time a file consumes to be created, i.e., from `gpfs_i_lookup()` until `gpfs_f_open()`. The total number of entries of each bucket is placed above each bar.

tions `gpfs_i_lookup()`, `gpfs_i_create()`, and `gpfs_f_open()`. In the following paragraphs, we will present the results for a complete and successful file create process with one, two, and four clients, involving all mentioned functions.

Figure 4.12 visualizes the distribution of one client for the create process, i.e., the time required from the call of `gpfs_i_lookup()` until the completion of `gpfs_f_open()`. The x-axis lists an array of bins, representing the latency (or time) that was measured for the tracked event. The scale starts at 70 μs and increases non-linearly to 500 ms[7]. With this scale we are able to distribute all latencies appropriately while each entry is matched to its bin, determined by rounding its value. The number of entries per bin are represented as bars whereas they are aligned to the primary, left y-axis. The green line represents each bin's contribution to the total time, aligned to the secondary, right y-axis. Furthermore, we exclude the traces of the beginning of an experiment in all following latency distributions because of token trashing. Token trashing is caused by byte-range lock conflicts that does not show the sustained case on which we are focusing on. Moreover, IBM seems to be aware of this problem as they already introduced functionality in order to use FGDL from the very beginning (see Section 4.2.2). Hence, the latency distributions are generated for the duration of an experiment in which FGDL is activated, resulting

[7] The scale does not show the bins from 1 to 60 μs as they do not contain any entries.

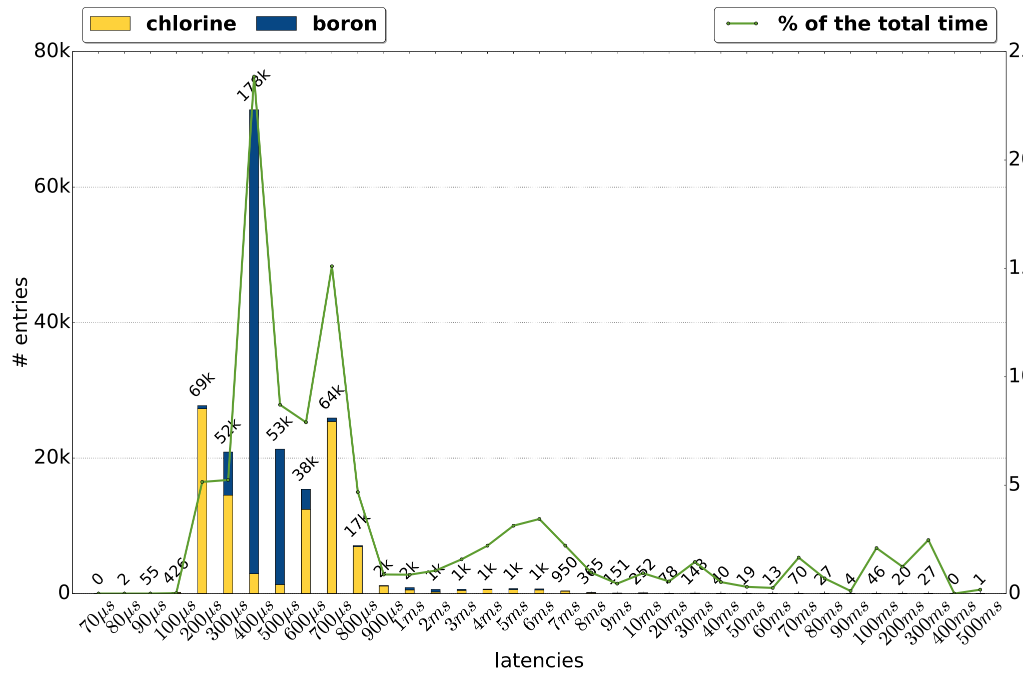

Fig. 4.13. – Latency distribution of two clients creating 500,000 files. Each entry in a bucket corresponds to the time a file consumes to be created, i.e., from `gpfs_i_lookup()` until `gpfs_f_open()`. Stacked bars are used to show the most represented client in each bin. The total number of entries of each bucket is placed above each stacked bar.

in slightly less than 500,000 entries. The information for this interval is provided by the modified mdtest.

According to Figure 4.12, most files in the single client experiment require between 200 μs and 400 μs, which all together contribute to almost 90% of the total time. Also, 171 files took 20 ms, accounting for already 3% of the time. Next, we compare the distribution of one client with Figure 4.13 in which two clients were used for the same workload. In this context, there are two effects to mention: Firstly, both clients show different create performances as they are not equally represented in each bin. This fact is also observable from the modified mdtest's results because one client is acting as a metanode first, achieving higher creates per second. In this particular experiment, the *boron* client was selected to be the first metanode, populating the bins 300 μs to 500 μs, while the *chlorine* client during that time achieved only a performance of 600 μs to 800 μs per file. However, chlorine was also faster than boron at some point in the experiment, visible by the entries in the bins 200 μs and 300 μs. These entries correspond to the time frame when boron was already finished with its workload and chlorine could finish its files without any other client interfering in the process. In Section 4.1.5, we presented a similar case with four clients whereas the last remaining client showed single client create performance towards the end of the experiment.

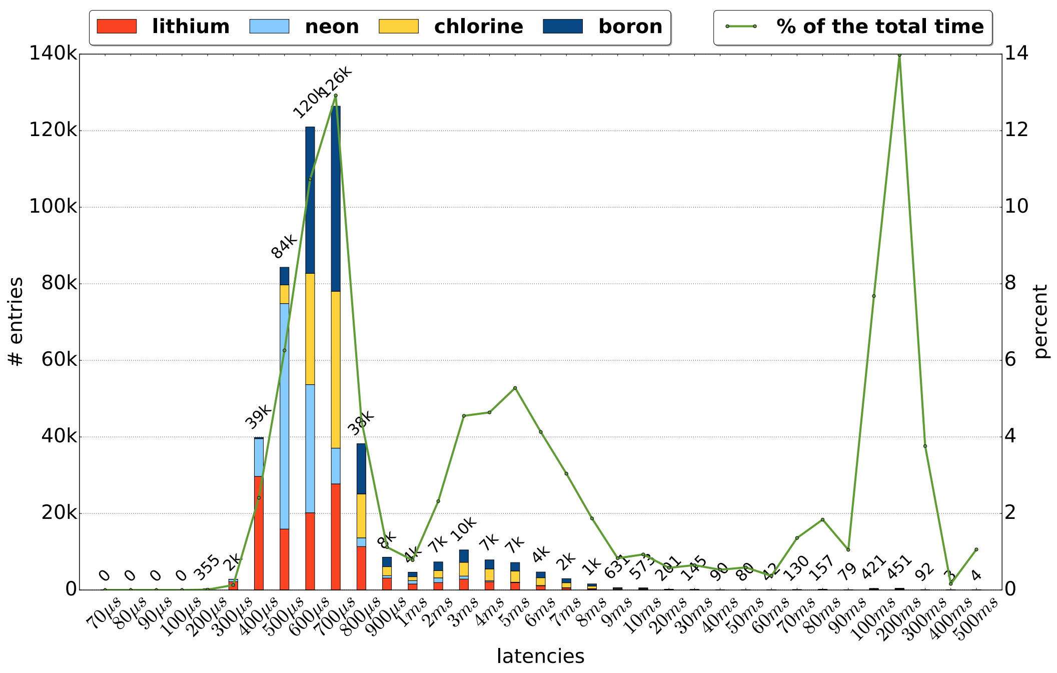

Fig. 4.14. – Latency distribution of four clients creating 500,000 files. Each entry in a bucket corresponds to the time a file consumes to be created, i.e., from `gpfs_i_lookup()` until `gpfs_f_open()`. Stacked bars are used to show the most represented client in each bin. The total number of entries of each bucket is placed above each stacked bar.

The second effect corresponds to all bins from 1 ms to 500 ms, showing that only ~13,000 files (~2,6% of the workload) consume almost 30% of the total time. Observing the latency distribution for four clients, as evident from Figure 4.14, shows that this effect is getting worse with additional clients participating in the experiment. We can observe ~50,000 files (10% of the workload), henceforth called *outliers*, scattered onto the bins from 1 ms to 500 ms, contributing to ~62% of the total time. This behavior also explains why two clients are showing a better performance than four clients (refer to Section 4.1.4). However, it is important to note that these outliers clearly represent two groups: The first group (bins 70 ms to 500 ms) accounts for ~31% of the time with only 1,337 files (~0.3% of the workload). The second group (bins 1 ms to 60 ms) contributes also to ~31% of the total time but with ~48,500 files (~10% of the workload). All clients are represented equally in the first group while the entries in the second group contain mostly non-metanodes. In addition, the number of entries in the second group seems to correlate with the amount of time a client spent as a non-metanode during the experiment. In summary, both groups show a large relevance to the total runtime in an experiment and will be further investigated in the following sections to understand their cause. Furthermore, we will focus on experiments with four participating clients since they show the outliers most clearly.

VFS functions **44 HDDs used**	**1 client (in s)**	**2 clients (in s)**	**4 clients (in s)**
1. `gpfs_i_lookup()`	14.6% (∼19)	55.7% (∼164)	75.2% (∼507)
2. `gpfs_i_create()`	61.3% (∼82)	34.8% (∼102)	20.8% (∼140)
3. `gpfs_f_open()`	24.0% (∼32)	9.5% (∼28)	3.9% (∼26)

Tab. 4.2. – Time distribution for the sequentially called Spectrum Scale VFS functions `gpfs_i_lookup()`, `gpfs_i_create()`, and `gpfs_f_open()`. Different experiments are shown per column while their rows display the total time each individual function call consumed.

In the first step, we measured the three VFS functions separately to understand if only one particular task is responsible for these outliers. Thus, we created the latency distribution for experiments with different numbers of clients for each of the three functions, which revealed that only `gpfs_i_lookup()` was causing them. For the `gpfs_i_create()` function, ∼3200 entries were placed into the same bins like those of the outliers. However, they did only account for ∼2.8% of the total time. The `gpfs_f_open()` function did not show any outliers. The latency distributions of the three Spectrum Scale VFS functions in a four client experiment are shown in the appendix (refer to Figures A.1, A.2, and A.3).

According to the time distribution Table 4.2[8] (based on their corresponding latency distributions), the more clients are used in an experiment the higher is the percentage of the total time that is consumed only by the `gpfs_i_lookup()` function. In absolute numbers, we can observe that the `gpfs_f_open()` function requires the same amount of time, regardless of the number of clients used in the experiment, suggesting that it only operates locally. The amount of time that is required by the `gpfs_i_create()` function increases more linearly with the number of participating clients compared to the exponential time increase for the `gpfs_i_lookup()` function. Therefore, we will firstly investigate the `gpfs_i_lookup()` function's behavior to clarify what is causing the abovementioned two groups of outliers in the following sections. Later, we analyze the `gpfs_i_create()` function to understand which logical tasks consume most of the total time while focusing on the tasks that do not scale. Because of the increasing insignificance of the `gpfs_f_open()` function in accordance with a higher number of clients (see Table 4.2), we did not further investigate its logical tasks as part of this thesis.

4.4.3 Lookup

As shown in Section 4.3.1, we divide the lookup function into seven subsequent logical tasks. These were analyzed for experiments with 1, 2, and 4 clients, each

[8]Similar time distributions for 2 SSDs and 2 RAMDisks of Spectrum Scale's VFS functions are listed in the Appendix (see Tables A.1 and A.2).

`gpfs_i_lookup()` tasks 44 HDDs used	1 client (in s)	2 clients (in s)	4 clients (in s)
1. Fast lookup	57.9% (∼8)	2.5% (∼4)	0.6% (∼3)
2. Lock the directory file	5.4% (∼1)	0.5% (∼1)	0.2% (∼1)
3. Verify permissions	0.6% (<1)	0.1% (<1)	0% (<1)
4. Acquire FGDL token and directory entries	2.8% (<1)	91.6% (∼148)	97.5% (∼435)
5. Lock the directory block	12.6% (∼2)	3.1% (∼5)	0.9% (∼4)
6. Search for the filename in the directory block	13.8% (∼2)	1.5% (∼2)	0.5% (∼2)
7. Release locks	8.0% (∼1)	0.8% (∼1)	0.3% (∼1)

Tab. 4.3. – Time distribution for the sequentially called `gpfs_i_lookup()` functions. Different experiments are shown per column while their rows display the percentage of the total time each individual function call consumed, including their absolute numbers in seconds.

with a total workload of 500,000 files. Table 4.3 presents the time distribution for all tasks of the three experiments[9]. In experiments with 2 or 4 active clients, task 4 accounts for almost the total lookup time with 97.5% and increases exponentially with the number of clients. Both groups of the previously mentioned outliers are caused by this task. Toward understanding the origin of this behavior, we used a conjunction of traces and the file system's source code to determine the task's exact functionality.

Task four can be split into two consecutive operations: The acquisition of an FGDL token from the manager node and the entries of their directory block from the metanode. In the case of a client being the metanode of the directory, it only has to request a FGDL token from the token manager to lock the corresponding range afterwards. Generally, the client requests an exclusive token from the token manager although it only requires a read-only token to look up the file. This is because Spectrum Scale has, at this point, no knowledge about the client's intentions after the lookup. For creating a file later, the client would already possess a token strong enough to update the directory entry. Otherwise, it would need to request an exclusive write token later, implying additional network traffic. However, Spectrum Scale is currently not using a mechanism allowing to make a more suitable token choice under certain circumstances, as granting an exclusive write token might induce an overhead if the client only looks up a file.

[9]Similar time distributions for 2 SSDs and 2 RAMDisks of `gpfs_i_lookup()` are listed in the Appendix (see Tables A.3 and A.4).

As soon as the metanode is in possession of the token, it can look for the filename since it owns all up-to-date directory blocks[10]. Nevertheless, non-metanodes, in addition to acquiring a FGDL token, need to request the metanode to send the entries of the directory block in which the token was acquired. Those are necessary for searching the entry of the corresponding filename in the directory block. In general, the token and metanode requests are answered immediately by the token manager and metanode, respectively, while some of them take longer and can be categorized as the before mentioned outliers. In addition, we could observe that the metanode's respond times to client requests often differ, resulting in an up to 50% slower create performance for some non-metanodes compared to others. Finally, various latency distributions can confirm that delayed token responses are causing the first group of outliers, while the second group accounts for slow responses of metanode requests. This explains the earlier observation that all clients are represented equally in group one whereas mostly non-metanodes can be found in the second group.

The delayed token responses of the first group of outliers are a result of token invalidations in which the token manager requests a particular client to revoke a set of his tokens. For example, the token manager will request clients to invalidate specific FGDL tokens if a directory block is split or if the metanode changes, potentially also influencing lookup operations. We could count almost 2,000 individual token revoke requests throughout an experiment. The second group of outliers is indirectly caused by token invalidations as well. This has two reasons: Firstly, the metanode has to execute a particular function (`dgetentries()`) that scans the entries of a directory block. This can take up to five milliseconds and is frequently called in an experiment. Secondly, all clients whose tokens got invalidated will have to reverify all held FGDL tokens. The verification of a single token usually consumes less than one microsecond. However, since the function is called for batches of FGDL tokens, the required time can increase to up to five milliseconds. In a sample experiment, we could observe ~310,000 batch executions with approximately 46,000,000 token verifications in total.

On the other hand, the presumed block splitting is not responsible for most of the time consuming token invalidations, as discussed in Section 4.4.4. Instead, they are caused by a particular configuration: (`fgdlleavethreshold`), set to `1,000` by default, defining that Spectrum Scale transitions out of FGDL mode into byte-range locking mode every 1,000 file creates. This transition is triggered frequently throughout an experiment. The purpose of this configuration is that Spectrum Scale can detect when the number of participating clients during file creation changes to a single client [28]. In this case byte-range locking mode is more effective as processes

[10]Spectrum Scale tries to hold the directory blocks in memory, avoiding disk I/O as good as possible to fetch directory entries. However, as seen from Section 4.4.5 many directory reads are issued.

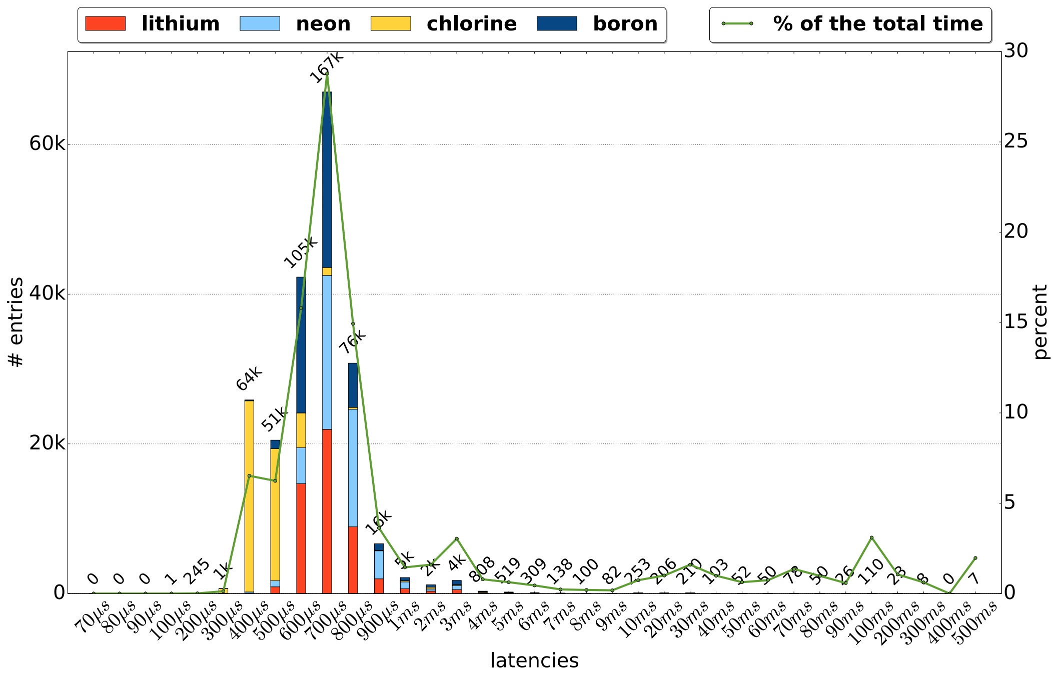

Fig. 4.15. – Latency distribution of four clients creating 500,000 files with `fgdlleavethreshold` disabled. Each entry in a bucket corresponds to the time a file consumes to be created, i.e., from `gpfs_i_lookup()` until `gpfs_f_open()`. Stacked bars are used to show the most represented client in each bin. The total number of entries of each bucket is placed above each stacked bar.

do not have to request FGDL tokens since the client already holds a byte-range lock on the whole directory.

Setting `fgdlleavethreshold` to zero disables this mechanism, resulting in a speed up of a factor of two in our experiments with four participating clients. Because two clients do not show many outliers and the configuration seems to only affect them, the performance improvement is limited to the factor of 1.05. Also, with the disabled configuration, four client experiments finish their workload slightly faster compared to single client experiments. This was not the case in earlier experiments as a four client experiment took significantly longer than one and two client experiments (see Section 4.1.4). Moreover, almost all outliers disappear from both groups as seen in the latency distribution, measured from `gpfs_i_lookup()` until `gpfs_f_open()`, for four clients, in Figure 4.15. Finally, entries of the bins from 1 ms to 500 ms now account for ∼20% of the total time with ∼3% of the workload. Although these entries still correspond to task four of the lookup process, we did not further analyze their cause as part of this study. However, we suspect that the remaining outliers are caused by token invalidations due to directory splits and metanode transitions. In addition, we could observe that setting `fgdlleavethreshold` to zero reduces the minor performance fluctuations during the experiment that were shown earlier in Section 4.1.5, resulting in a more stable performance per process, especially for non-metanodes (see Figure A.4 in the Appendix). In summary, with `fgdlleavethreshold` disabled ∼97% of the workload is bound by the network latency, introduced by token and metanode communication, as the token manager and the metanode generally answer requests without further delay.

From Spectrum Scale version 4.2 onward `fgdlleavethreshold` uses an advanced algorithm that should greatly reduce the disruptive token revokes. The new version no longer relies on the number of created files since its frequency increases with better file create performance. Instead, it uses a time-based approach where the directory transitions out of FGDL every 10 seconds by default [28].

4.4.4 Create

As shown in Section 4.3.2, we divide the create function into 12 subsequent logical tasks. Their analyses involved experiments with 1, 2, and 4 clients participating in creating 500,000 files. Table 4.4 presents the time distribution for all tasks of the three experiments[11]. Apparently, two tasks (four and nine) are consuming an increasing amount of time the more clients are participating in an experiment. All other tasks require a similar amount of time. Similar to task four of the `gpfs_i_lookup()` function, task four of the `gpfs_i_create()` function will search for the filename in the directory block to verify that a file with the same name is not yet created. This is necessary due to the separation of lookup and create in the Linux VFS which Spectrum Scale has to implement. Consequently, there is the possibility of losing the token between the two VFS functions and, furthermore, allowing other clients to create the same file during that timespan. This is generally not an issue, visible by the low amount of total time required by task four compared to the same task in the `gpfs_i_lookup()` function. However, token invalidations can still occur due to directory block splits and metanode transitions that cause the loss of a token between the two VFS functions.

The second mentioned ninth task is also noticeable as it consumes more time with an increasing number of clients. As shown in Section 4.3.2, task nine is executed by non-metanodes only, requesting the metanode of a directory to reserve space for a new entry. Naturally, single client experiments do not show any time consumption in this step since only one client (its own metanode) is working for the whole experiment. In multiple client experiments, the time needed for a request is introduced by the network latency as well as the time that the metanode requires to complete the request, which can be delayed due to directory splits or metanode transitions. Although some requests (1%) take the metanode longer than 30 μs to answer, they only account for less than 10% of the total time of the `gpfs_i_create()` function,

[11]Similar time distributions for 2 SSDs and 2 RAMDisks of `gpfs_i_create()` are listed in the Appendix (see Tables A.5 and A.6).

`gpfs_i_create()` tasks 44 HDDs used	1 client (in s)	2 clients (in s)	4 clients (in s)
1. Allocate inode	14.3% (∼9)	9.2% (∼9)	6.6% (∼9)
2. Lock file and directory file	9.4% (∼6)	4.8% (∼4)	2.4% (∼3)
3. Get the inode of the directory	24.2% (∼16)	15.7% (∼15)	12.4% (∼16)
4. Acquire FGDL token and directory entries	7.3% (∼5)	19.8% (∼18)	17.3% (∼23)
5. Search for the filename in the directory block	4.0% (∼3)	2.7% (∼3)	1.8% (∼2)
6. Various verifications	4.9% (∼3)	3.4% (∼3)	2.5% (∼3)
7. Create metadata of the new file	6.4% (∼4)	4.4% (∼4)	3.2% (∼4)
8. Reserve space in the directory block (metanode only)	10.8% (∼7)	6.2% (∼6)	4.6% (∼6)
9. Ask metanode to reserve space in the directory block (non-metanode only)	0% (0)	20.0% (∼19)	38.2% (∼50)
10. Finish file create and finalize log	5.3% (∼3)	3.6% (∼3)	2.8% (∼4)
11. Insert entry into the directory block	8.8% (∼6)	6.9% (∼6)	5.7% (∼8)
12. Release locks	4.6% (∼3)	3.2% (∼3)	2.4% (∼3)

Tab. 4.4. – Time distribution for the sequentially called `gpfs_i_create()` functions. Different experiments are shown per column while their rows display the percentage of the total time each individual function call consumed, including their absolute numbers in seconds.

while ∼99% of the requests are answered in 7 μs to 30 μs by the metanode. However, the cause of the increased time consumption with using more clients is not simply explained by delayed responses but by the amount of time non-metanodes are active during an experiment. For instance, in two client experiments task nine is only called as long as two clients are creating files while the first client finishes its workload sooner. In four client experiments, on the other hand, non-metanodes work for ∼80% of the duration of the benchmark, increasing the time spent in this particular task. Therefore, the required time in task nine correlates with the duration of actively working non-metanodes and will likely not increase further with an higher number of clients than available in our experimental setup.

4.4.5 I/O and Logging

In the previous sections, we have shown that the file create performance in our test cluster is limited by the network. Moreover, we presented the overall file create performance for a variety of storage subsystems with the consensus that different storage does not impact the create performance (see Section 4.1.10). In contrast to the modified mdtest results, traces are able to visualize the size of data that is written to the shared storage, including its latency. In Spectrum Scale, all read and write operations to the storage subsystem go through the same traceable function which is called by the daemon of each client. Thus, the latency of each trace record includes the time that the NSD server requires to read or write the data in addition to the round-trip network time from the client to the NSD server. Furthermore, the trace record contains information about the amount of bytes that are being read or written.

The read and write operations are separated into four operations. The following enumeration includes the average number of requests and average amount of data being read and written for each operation in a workload of 500,000 files with four participating clients in multiple experiments:

4 KiB write
This operation represents the writing of the inode of a newly created file, issued by any creating client. During the experiments 500,000 requests were measured with a total of 1,953 MiB being written in average.

256 KiB write
This operation represents the writing of a directory block, issued by the metanode only. During the experiments ∼30,000 requests were measured with a total of ∼8,000 MiB being written in average.

Various writes

This operation represents the writings of various sizes, including log records, issued by any creating client. During the experiments ~4,100 requests were measured with a total of ~160 MiB being written in average.

256 KiB read

This operation represents the reading of an directory block, issued by the metanode only. During the experiments ~3,600 to ~6,500 requests were measured with a total of ~1,500 MiB being read in average.

Avg lat/op in μs	4 KiB write	256 KiB write	Various writes	256 KiB read
44 HDDs	~861	~1774	~991	~1162
2 SSDs	~430	~1236	~536	~1284
2 RAMDisks	~344	~766	~395	~601

Tab. 4.5. – Average latency per operation in microseconds with different storage subsystems for an experiment in which four clients create 500,000 files concurrently.

Table 4.5 shows the average latency per operation in microseconds for an experiment in which 500,000 files were created by four clients, based on their latency distributions. The table does not include the results for 11 and 22 HDDs as their average latency is similar to 44 HDDs due to the non-volatile controller cache (see Section 4.1.10). The table visualizes the impact of faster storage subsystems in the scenario of parallel file creation for all operations. For instance, RAMDisks show at least a 50% decrease in latency compared to HDDs in all cases. However, we did not further investigate the I/O behavior, for example, the cause of the diverse number of directory block reads, as part of this thesis and did not show an impact on the experiments runtime.

4.5 Possible improvements

This chapter has shown the time distribution for every logical task and investigated particular mechanisms further, such as directory block splits. Based on our observations, we present two ideas to improve file create performance. However, we will not further elaborate on optimizing mechanisms that affect the enabled FGDL duration, such as *AvoidDirFragments* (see Section 4.2.2) and *fgdlleavethreshold* (see Section 4.4.3). In conclusion, FGDL ought to be enabled with the first created file in the directory and not be disrupted during the parallel create process.

4.5.1 Multiple metanodes per directory

Section 4.1.5 revealed that metanodes achieve at least a twice as good create performance compared to non-metanodes. This effect leads to an imbalance in which metanodes finish their workloads earlier, essentially removing one client at a time from the concurrent create process. But this performance difference is not the only concern since the required transition time of 30 seconds to a different metanode wastes performance as well. Applications could use this knowledge to ensure a dynamic workload distribution on the participating clients at runtime to keep the same metanode active. However, a solution on the application level is not desirable as legacy applications will likely not be modified.

Another solution is that Spectrum Scale provides an equal performance for all clients which create files in a single directory by using multiple metanodes per directory. Each metanode controls a number of directory blocks and still collects the directory entry updates from other nodes that need to insert a new entry into the metanode's controlled directory block. Due to extendible hashing, which distributes the files equally to all directory blocks, this solution would lead to a similar performance and participation for all clients for the whole create duration. Note that the number of directory blocks increases with the amount of files in a directory, allowing more metanodes to be used with a bigger workload. Moreover, with a pre-allocation of directory blocks, as discussed in Section 4.1.7, the scalability could further improve for smaller workloads where the number of nodes is higher than the number of directory blocks. This method may pay off in exascale environments in which a single metanode per directory may be overloaded due to a large number of client requests. However, implementing this optimization may require a large implementation overhead because metanodes have to be determined per directory block as opposed to per directory. Since metanodes are not only used in the create process, changing them would also require modifications to other parts of the file system.

In addition, the described methodology can be further improved in the future, exploiting that metanodes create files faster than non-metanodes (see Section 4.1.5). Specifically, the metanodes should prefer adding new entries into their controlled directory blocks. This would allow all clients to achieve the create performance of single client experiments because all metanodes would generally insert entries into their locally accessible directory blocks while minimizing metanode and token traffic over the network. On the other hand, this conflicts with the current implementation of extendible hashing which would need to be replaced in the process.

4.5.2 File create intentions

Another potential performance improvement is a mechanism that considers the client's intention of creating a new file or opening an existing one. As discussed in Section 4.4.3, the lookup of a file is done twice during the create process due to the definition in Linux VFS. Because Spectrum Scale has to ensure that the file does not already exist in the directory, it repeats the lookup in `gpfs_i_create()` although it was already done in `gpfs_i_lookup()`. This step is necessary since all locks are released in the end of `gpfs_i_lookup()` and the corresponding tokens might have gotten reassigned between the steps. Hence, if Spectrum Scale knows the client's intentions, it can skip `gpfs_i_lookup()` in the case of a file creation, avoiding the overhead of two consecutive and similar lookup operations while additionally reducing token traffic.

5 Conclusion and future work

The performance of concurrent file creation in a single directory will become more important in the future. With the number of nodes in HPC clusters increasing the *mean time between failures* (MTBF) of cluster components, e.g., HDDs, will decrease to the order of minutes [4, 5]. This makes applications, such as checkpointing, even more relevant. Therefore, metadata performance of modern parallel file systems has to improve to handle such workloads, which is currently limited by metadata scalability issues [8]. Although other file systems try to improve metadata performance by relaxing POSIX semantics [12], it is not feasible for applications depending on these guidelines. Therefore, a solution to improve file create performance by not violating POSIX compliance of a file system is desirable. Nonetheless, this is a challenging task due to the complexity of the process, involving a large portion of network communication and I/O operations, in which many aspects are not fully understood.

In this thesis, we have investigated the file create process of IBM's Spectrum Scale parallel file system with its source code, the well-known mdtest metadata benchmark, and trace records. Our goal has been to provide a detailed insight into the process and to achieve an understanding about Spectrum Scale's behavior during concurrent file creation in a single directory. This has enabled us to pinpoint bottlenecks that are worth optimizing, providing support to the Spectrum Scale developer team to improve the file system.

To achieve this goal, we initially have modified the mdtest benchmark to receive fine-grained per-process information about the creates per second at any point in time during an experiment for a user-specific time interval. In addition, we have developed tools to automatically generate graphs of the modified mdtest's output. The modified mdtest results have verified the abovementioned scalability issues measured by previous work and even showed that the runtime increases with an increasing number of clients for the same workload. But more importantly, the results have revealed various aspects of the file create performance and their impact, such as directory block splits and inode pre-allocation, accounting for an performance increase of up to $\sim$4% in both cases if they do not occur. Moreover, we have shown that metanodes finish their workload faster due to their better performance compared to non-metanodes, resulting in a loss of participating nodes during an experiment and in expensive metanode transitions. Finally, we have observed the impact of different storage hardware on file creation with the conclusion that neither SSDs nor RAMDisks affect the runtime of the ran experiments noticeably compared to the HDDs in our test setup.

In the second step, we have discussed the implications of trace collection and have presented the file create protocol of Spectrum Scale. With latency distributions for each logical task of the file create process, we have identified the `gpfs_i_lookup()` function as the main time contributor whose absolute time increases exponentially with the number of clients participating in the process. Further investigations have revealed that ~98% of the time in `gpfs_i_lookup()` is spent for token and metanode communication in a four client experiment which is expected to increase with a larger number of participating nodes. We have identified the configuration setting `fgdlleavethreshold` as the cause, issuing frequent disruptive token revokes and resulting in file creates that consume significantly more time than usual. Disabling this setting has decreased the runtime by 50%, outperforming experiments with a single client participating. Moreover, we have presented the average latency for various read and write operations, showing that the RAMDisk's latency is outperforming HDD's at least by a factor of two. We have concluded that file creation in Spectrum Scale is limited by the 10 Gbit network in our test setup. More specifically, latency distributions of the `gpfs_i_lookup()` function and the `gpfs_i_create()` function (see Figures A.1 and A.2 in the Appendix) have indicated that runtime is driven by the accumulated network latencies of token and metanode requests. Overall, most tasks in the Spectrum Scale's file create process seem to scale well for up to the available four nodes in our test cluster.

However, due to the complexity of the file create process we could not follow every lead that has arisen during our research. For example, we did not further investigate the I/O behavior since it did not show any impact on the experiment's runtime. Nonetheless, faster storage subsystems may play a more important role in environments with a lower network latency, that should be investigated by future work. We also recommend to repeat the trace analysis for newer Spectrum Scale versions (that is, newer than 4.1.1.0) as important mechanisms that influence Spectrum Scale's file create performance greatly (e.g., `fgdlleavethreshold`) have been updated [28]. As the development of the presented tools will be continued[1], future work should further explore the scalability for every logical task of the file create process in a cluster with a larger number of nodes. Those results may prove valuable for enhancing metadata performance in the future with respect to the upcoming exascale era.

[1]The sources of the modified mdtest and the trace analyzer are available at `https://github.com/marcvef/mdtest` and `https://github.com/marcvef/tracealyzer`

Appendix A

The appendix contains additional latency distributions of Spectrum Scale's VFS functions `gpfs_i_lookup()`, `gpfs_i_create()`, and `gpfs_f_open()` as well as their time distributions with 2 SSDs and 2 RAMDisks, similar to those presented in Section 4.4. Furthermore, the per-process performance of a four client experiment is shown with `fgdlleavethreshold` disabled. The below Table summarizes the Appendix' content.

Figures:

Tables:

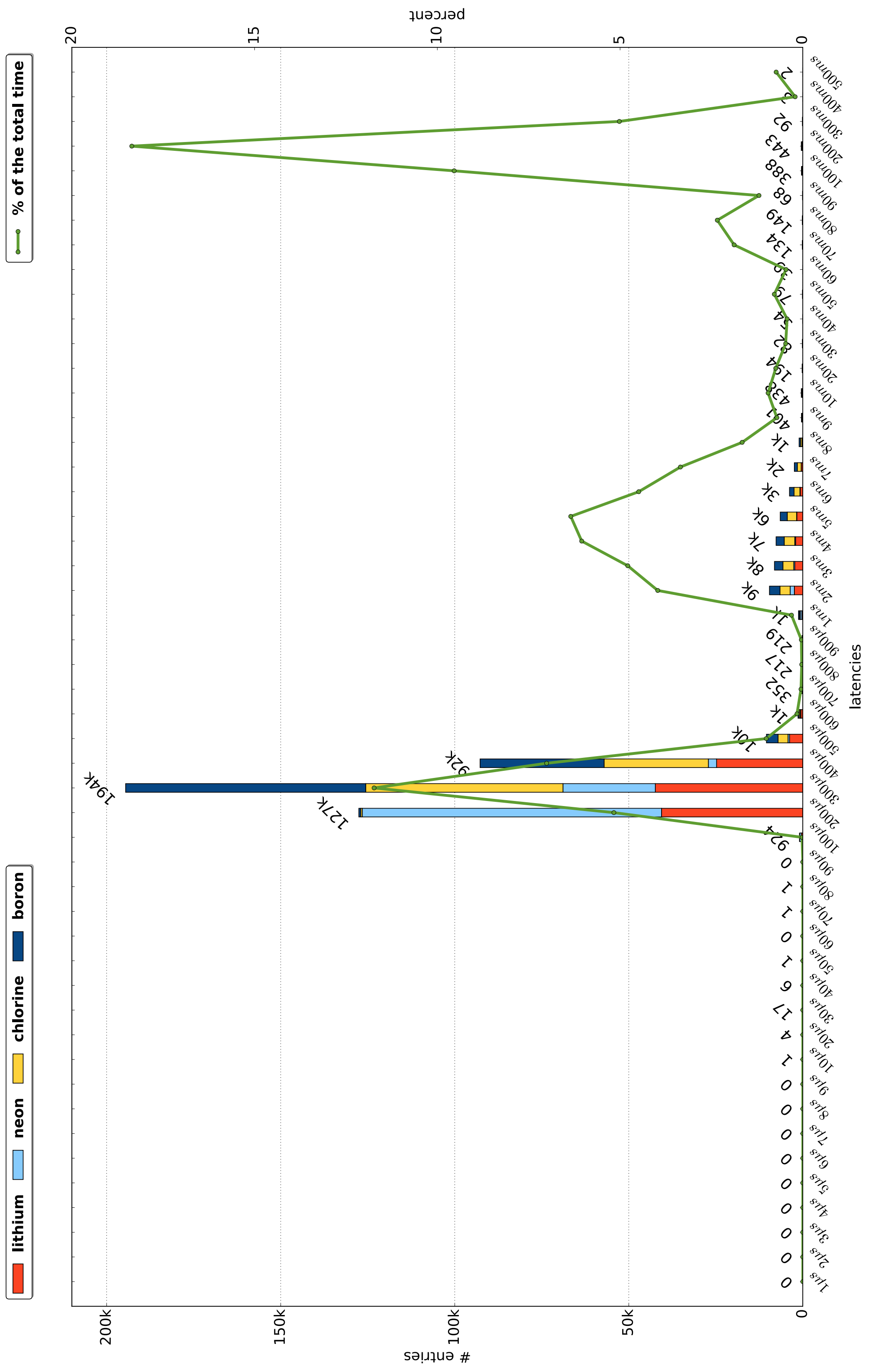

Fig. A.1. – Latency distribution of `gpfs_i_lookup()` for four clients creating 500,000 files. Each entry in a bucket corresponds to the time the `gpfs_i_lookup()` function consumes when looking up a file. Stacked bars are used to show the most represented client in each bin. The total number of entries of each bucket is placed above each stacked bar.

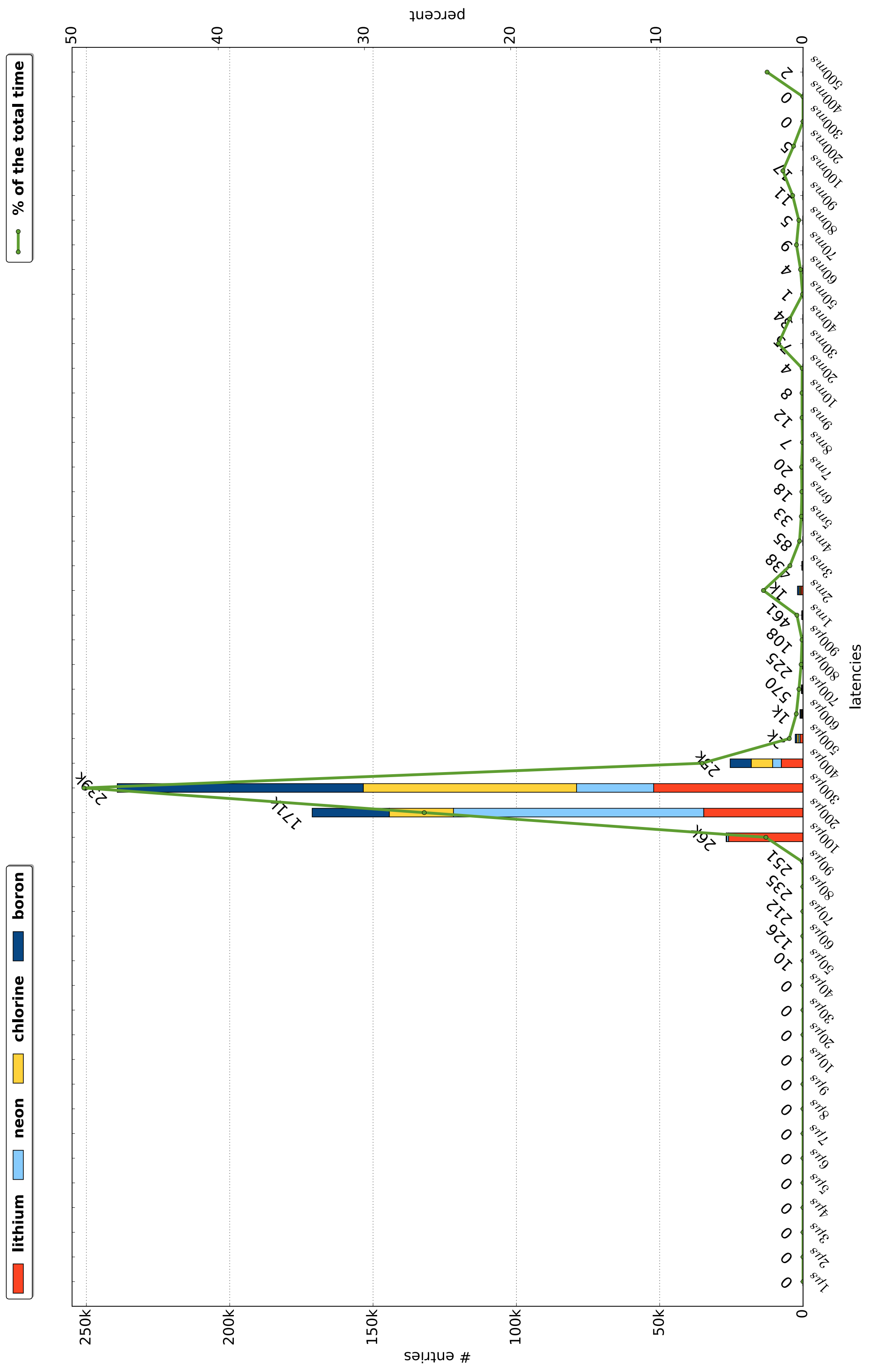

Fig. A.2. – Latency distribution of of `gpfs_i_create()` for four clients creating 500,000 files. Each entry in a bucket corresponds to the time the `gpfs_i_create()` function consumes when creating a file. Stacked bars are used to show the most represented client in each bin. The total number of entries of each bucket is placed above each stacked bar.

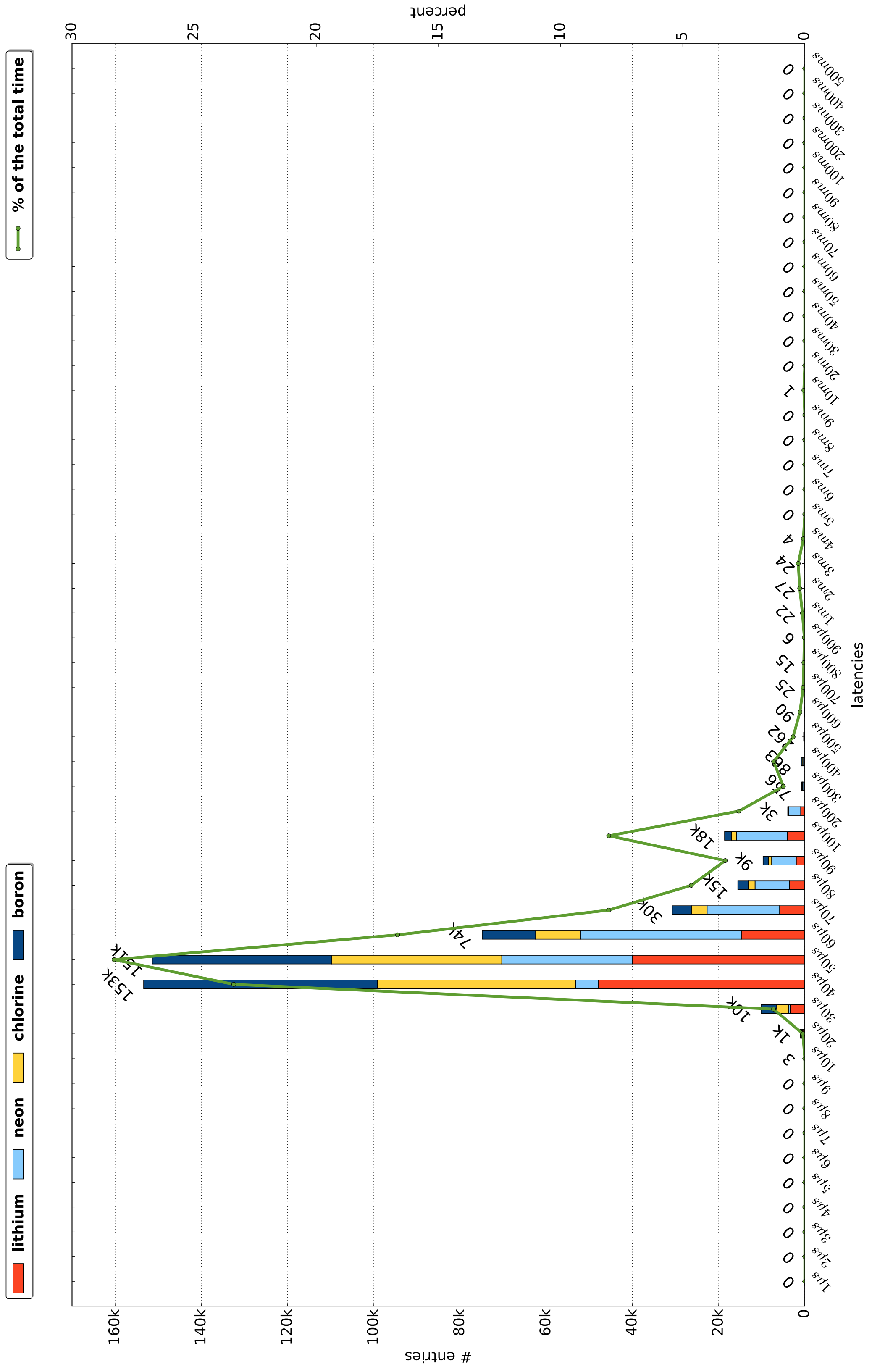

Fig. A.3. – Latency distribution of of `gpfs_f_open()` for four clients creating 500,000 files. Each entry in a bucket corresponds to the time the `gpfs_f_open()` function consumes when opening a file. Stacked bars are used to show the most represented client in each bin. The total number of entries of each bucket is placed above each stacked bar.

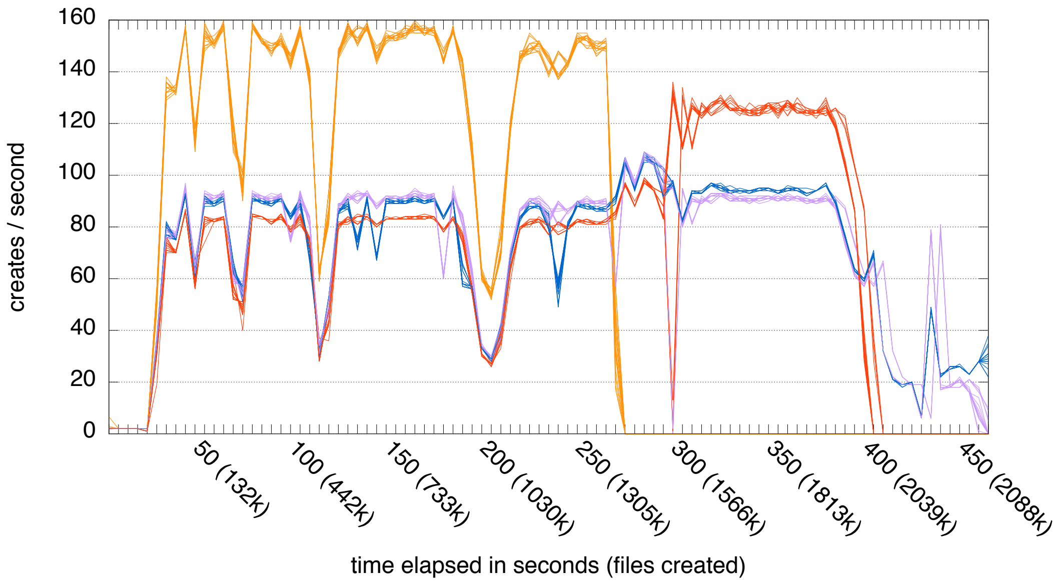

Fig. A.4. – Investigation of the running per-process create performance with `fgdlleavethreshold` disabled in a single experiment with 4 clients and 16 processes per clients where 2 million files are created in total. Each line represents a single process while its color visualizes its client affinity. Every data point shows the create performance per-process for the last five seconds.

VFS functions **2 SSDs used**	**1 client (in s)**	**2 clients (in s)**	**4 clients (in s)**
1. `gpfs_i_lookup()`	13.5% (∼19)	75.5% (∼224)	73.0% (∼509)
2. `gpfs_i_create()`	57.2% (∼78)	36.0% (∼107)	22.3% (∼155)
3. `gpfs_f_open()`	23.6% (∼32)	9.7% (∼28)	4.14% (∼29)

Tab. A.1. – Time distribution for the Spectrum Scale VFS functions `gpfs_i_lookup()`, `gpfs_i_create()`, and `gpfs_f_open()` with two used SSDs. Different experiments are shown per column while their rows display the total time each individual function call consumed.

VFS functions **2 RAMDisks used**	**1 client (in s)**	**2 clients (in s)**	**4 clients (in s)**
1. `gpfs_i_lookup()`	14.8% (∼19)	51.0% (∼133)	64.4% (∼316)
2. `gpfs_i_create()`	58.1% (∼74)	37.3% (∼97)	28.9% (∼142)
3. `gpfs_f_open()`	24.7% (∼31)	10.5% (∼27)	5.5% (∼27)

Tab. A.2. – Time distribution for the sequentially called Spectrum Scale VFS functions `gpfs_i_lookup()`, `gpfs_i_create()`, and `gpfs_f_open()` with two used RAMDisks. Different experiments are shown per column while their rows display the total time each individual function call consumed.

`gpfs_i_lookup()` tasks 2 SSDs used	1 client (in s)	2 clients (in s)	4 clients (in s)
1. Fast lookup	53.9% (∼8)	1.9% (∼4)	0.6% (∼3)
2. Lock the directory file	5.4% (<1)	0.4% (∼1)	0.2% (∼1)
3. Verify permissions	0.6% (<1)	0% (<1)	0% (<1)
4. Acquire FGDL token and directory entries	3.1% (<1)	86.2% (∼185)	95.2% (∼458)
5. Lock the directory block	13.8% (∼2)	9.9% (∼21)	3.18% (∼15)
6. Search for the filename in the directory block	15.4% (∼2)	1.1% (∼2)	0.5% (∼3)
7. Release locks	7.6% (∼1)	0.6% (∼1)	0.3% (∼1)

Tab. A.3. – Time distribution for the sequentially called `gpfs_i_lookup()` functions with two used SSDs. Different experiments are shown per column while their rows display the percentage of the total time each individual function call consumed, including their absolute numbers in seconds.

`gpfs_i_lookup()` tasks 2 RAMDisks used	1 client (in s)	2 clients (in s)	4 clients (in s)
1. Fast lookup	56.5% (∼8)	2.8% (∼4)	0.8% (∼2)
2. Lock the directory file	5.2% (<1)	0.6% (∼1)	0.3% (∼1)
3. Verify permissions	0.6% (<1)	0% (<1)	0% (<1)
4. Acquire FGDL token and directory entries	2.9% (<1)	92.4% (∼118)	97.2% (∼324)
5. Lock the directory block	12.9% (∼2)	1.6% (∼2)	0.7% (∼1)
6. Search for the filename in the directory block	14.7% (∼2)	1.67% (∼2)	0.7% (∼3)
7. Release locks	7.3% (∼1)	0.9% (∼1)	0.4% (∼1)

Tab. A.4. – Time distribution for the sequentially called `gpfs_i_lookup()` functions with two used RAMDisks. Different experiments are shown per column while their rows display the percentage of the total time each individual function call consumed, including their absolute numbers in seconds.

`gpfs_i_create()` tasks 2 SSDs used	1 client (in s)	2 clients (in s)	4 clients (in s)
1. Allocate inode	15.0% (∼10)	9.6% (∼9)	6.3% (∼9)
2. Lock file and directory file	5.6% (∼4)	3.8% (∼4)	2.0% (∼3)
3. Get the inode of the directory	24.2% (∼16)	17.0% (∼17)	12.1% (∼18)
4. Acquire FGDL token and directory entries	7.6% (∼5)	19.7% (∼19)	25.4% (∼36)
5. Search for the filename in the directory block	4.2% (∼3)	2.7% (∼3)	1.8% (∼3)
6. Various verifications	5.2% (∼3)	3.6% (∼3)	2.5% (∼4)
7. Create metadata of the new file	6.7% (∼4)	4.6% (∼4)	3.2% (∼4)
8. Reserve space in the directory block (metanode only)	11.7% (∼8)	6.7% (∼6)	3.7% (∼5)
9. Ask metanode to reserve space in the directory block (non-metanode only)	0% (0)	18.4% (∼18)	32.8% (∼48)
10. Finish file create and finalize log	5.5% (∼3)	3.7% (∼3)	2.6% (∼4)
11. Insert entry into the directory block	9.3% (∼6)	6.9% (∼6)	7.7% (∼8)
12. Release locks	4.9% (∼3)	3.3% (∼3)	3.3% (∼3)

Tab. A.5. – Time distribution for the sequentially called `gpfs_i_create()` functions with two used SSDs. Different experiments are shown per column while their rows display the percentage of the total time each individual function call consumed, including their absolute numbers in seconds.

`gpfs_i_create()` tasks 2 RAMDisks used	1 client (in s)	2 clients (in s)	4 clients (in s)
1. Allocate inode	15.7% (∼9)	9.7% (∼9)	7.0% (∼9)
2. Lock file and directory file	6.1% (∼4)	2.9% (∼3)	2.8% (∼3)
3. Get the inode of the directory	24.3% (∼15)	18.1% (∼16)	13.0% (∼16)
4. Acquire FGDL token and directory entries	7.6% (∼5)	12.8% (∼11)	9.5% (∼12)
5. Search for the filename in the directory block	4.2% (∼3)	2.8% (∼2)	2.0% (∼2)
6. Various verifications	5.2% (∼3)	3.7% (∼3)	2.7% (∼3)
7. Create metadata of the new file	6.7% (∼4)	4.7% (∼4)	3.5% (∼4)
8. Reserve space in the directory block (metanode only)	10.5% (∼6)	5.8% (∼5)	3.7% (∼4)
9. Ask metanode to reserve space in the directory block (non-metanode only)	0% (0)	24.8% (∼21)	43.9% (∼54)
10. Finish file create and finalize log	5.5% (∼3)	3.8% (∼3)	3.0% (∼4)
11. Insert entry into the directory block	9.4% (∼6)	7.6% (∼7)	6.4% (∼8)
12. Release locks	4.8% (∼3)	3.3% (∼3)	2.5% (∼3)

Tab. A.6. – Time distribution for the sequentially called `gpfs_i_create()` functions with two used RAMDisks. Different experiments are shown per column while their rows display the percentage of the total time each individual function call consumed, including their absolute numbers in seconds.

Bibliography

[1] Diana Yates. *Wit, grit and a supercomputer yield chemical structure of HIV capsid*. May 29, 2013. URL: `https://news.illinois.edu/blog/view/6367/204804` (visited on Mar. 1, 2016) (cit. on p. 1).

[2] Glenn Lockwood. *DNA sequencing: Not quite HPC yet*. Mar. 3, 2015. URL: `http://www.nextplatform.com/2015/03/03/dna-sequencing-not-quite-hpc-yet/` (visited on Mar. 1, 2016) (cit. on p. 1).

[3] John Bent, Garth Gibson, Gary Grider, et al. „PLFS: a checkpoint filesystem for parallel applications“. In: *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. ACM. 2009, p. 21 (cit. on pp. 1, 17).

[4] Jack Dongarra et al. „The international exascale software project roadmap“. In: *International Journal of High Performance Computing Applications* (2011), p. 1094342010391989 (cit. on pp. 1, 67).

[5] Peter Kogge, Keren Bergman, Shekhar Borkar, et al. „Exascale computing study: Technology challenges in achieving exascale systems“. In: (2008) (cit. on pp. 1, 67).

[6] James N Glosli, David F Richards, KJ Caspersen, et al. „Extending stability beyond CPU millennium: a micron-scale atomistic simulation of Kelvin-Helmholtz instability“. In: *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*. ACM. 2007, p. 58 (cit. on p. 1).

[7] Swapnil Patil and Garth A Gibson. „Scale and Concurrency of GIGA+: File System Directories with Millions of Files.“ In: *FAST*. Vol. 11. 2011, pp. 13–13 (cit. on pp. 1, 2, 16, 17).

[8] Sadaf R Alam, Hussein N El-Harake, Kristopher Howard, Neil Stringfellow, and Fabio Verzelloni. „Parallel I/O and the metadata wall“. In: *Proceedings of the sixth workshop on Parallel Data Storage*. ACM. 2011, pp. 13–18 (cit. on pp. 1, 2, 17, 67).

[9] Wolfgang Frings, Felix Wolf, and Ventsislav Petkov. „Scalable massively parallel I/O to task-local files“. In: *High Performance Computing Networking, Storage and Analysis, Proceedings of the Conference on*. IEEE. 2009, pp. 1–11 (cit. on pp. 1, 17).

[10] K Fitzgerald, M Gary, RM Hedges, and DM Stearman. *Comparison of Leading Parallel NAS File Systems on Commodity Hardware*. Tech. rep. Lawrence Livermore National Laboratory (LLNL), Livermore, CA, 2010 (cit. on pp. 1, 17).

[11] The IEEE and The Open Group. *POSIX*. 2001. URL: `http://pubs.opengroup.org/onlinepubs/9699919799/` (visited on Feb. 1, 2016) (cit. on pp. 1, 5, 6, 10, 13).

[12] Sage A Weil, Scott A Brandt, Ethan L Miller, Darrell DE Long, and Carlos Maltzahn. „Ceph: A scalable, high-performance distributed file system“. In: *Proceedings of the 7th symposium on Operating systems design and implementation*. USENIX Association. 2006, pp. 307–320 (cit. on pp. 1, 17, 19, 20, 67).

[13] Frank B Schmuck and Roger L Haskin. „GPFS: A Shared-Disk File System for Large Computing Clusters.“ In: *FAST*. Vol. 2. 2002, pp. 231–244 (cit. on pp. 1, 10, 11, 14, 16, 17).

[14] Brett Kettering, Ruth Klundt Alfred Torrez, and William Loewe. *mdtest HPC benchmark*. Dec. 23, 2013. URL: `http://sourceforge.net/projects/mdtest/` (visited on Jan. 30, 2016) (cit. on pp. 2, 23).

[15] IEEE. *IEEE*. 1963. URL: `https://www.ieee.org/` (visited on Feb. 14, 2016) (cit. on p. 5).

[16] The Open Group. *The Open Group*. 1995. URL: `http://www.opengroup.org/` (visited on Feb. 14, 2016) (cit. on p. 5).

[17] Samuel Lang. *Parallel File Systems*. Guest Lecture. Argonne National Laboratory, Sept. 20, 2010 (cit. on p. 6).

[18] Hai Pomeranz. *Understanding EXT4: Extent Trees*. Mar. 28, 2011. URL: `https://digital-forensics.sans.org/blog/2011/03/28/digital-forensics-understanding-ext4-part-3-extent-trees` (visited on Mar. 6, 2016) (cit. on p. 7).

[19] Jonathan Corbet. *Dcache scalability and RCU-walk*. Dec. 14, 2010. URL: `https://lwn.net/Articles/419811/` (visited on Mar. 6, 2016) (cit. on p. 9).

[20] Paul E. McKenney, Dipankar Sarma, and Maneesh Soni. *Scaling dcache with RCU*. Jan. 1, 2004. URL: `http://www.linuxjournal.com/article/7124` (visited on Mar. 6, 2016) (cit. on p. 9).

[21] Dean Hildebrand and Frank Schmuck. „GPFS“. In: *High Performance Parallel I/O*. Chapman & Hall/CRC, 2014, pp. 33–43 (cit. on pp. 10, 11, 14, 20).

[22] Wayne Rickard et al. „Shared Storage Model—A framework for describing storage architectures“. In: *SNIA Technical Council, snia. org* 41 (2003) (cit. on p. 10).

[23] Storage Networking Industry Association. *The SNIA Shared Storage Model*. 2016. URL: `http://www.snia.org` (visited on Feb. 5, 2016) (cit. on p. 10).

[24] IBM. *IBM Knowledge Center*. 2016. URL: `https://www-01.ibm.com/support/knowledgecenter/?lang=en` (visited on Feb. 6, 2016) (cit. on pp. 11, 12).

[25] Frank B Schmuck, James Christopher Wyllie, and Thomas E Engelsiepen. *Parallel file system and method with extensible hashing*. US Patent 5,893,086. 1999 (cit. on p. 13).

[26] Ronald Fagin, Jurg Nievergelt, Nicholas Pippenger, and H Raymond Strong. „Extendible hashing—a fast access method for dynamic files“. In: *ACM Transactions on Database Systems (TODS)* 4.3 (1979), pp. 315–344 (cit. on p. 13).

[27] Lucia Moura. *Extendible Hashing I*. 2002. URL: `https://www.site.uottawa.ca/~lucia/courses/2131-02/lect18.pdf` (visited on Mar. 1, 2016) (cit. on p. 13).

[28] Frank Schmuck. „Private communication“. Mar. 2, 2016 (cit. on pp. 13, 41, 58, 60, 68).

[29] Ted Anderson. „Fine-grained Directory Locking (FGDL)". Mar. 21, 2014 (cit. on pp. 15–17).

[30] Shobhit Dayal. „Characterizing HEC storage systems at rest". In: *Parallel Data Lab, Carnegie Mellon University, Pittsburgh, PA, USA* (2008) (cit. on p. 16).

[31] Ric Wheeler. *One Billion Files: Scalability Limits in Linux File Systems*. URL: `http://events.linuxfoundation.org/slides/2010/linuxcon2010_wheeler.pdf` (visited on Feb. 28, 2016) (cit. on p. 16).

[32] Shuichi Ihara. *Lustre Metadata Fundamental Benchmark and Performance*. DataDirect Networks Japan Inc. Sept. 22, 2014. URL: `https://eofs.gsi.de/fileadmin/lad2014/slides/03_Shuichi_Ihara_Lustre_Metadata_LAD14.pdf` (visited on Feb. 28, 2016) (cit. on pp. 17, 23).

[33] Nawab Ali, Ananth Devulapalli, Dennis Dalessandro, Pete Wyckoff, and P Sadayappan. „Revisiting the metadata architecture of parallel file systems". In: *Petascale Data Storage Workshop, 2008. PDSW'08. 3rd*. IEEE. 2008, pp. 1–9 (cit. on p. 17).

[34] Jing Xing, Jin Xiong, Ninghui Sun, and Jie Ma. „Adaptive and scalable metadata management to support a trillion files". In: *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. ACM. 2009, p. 26 (cit. on p. 17).

[35] Various. *Lustre*. 2016. URL: `http://lustre.org/` (visited on Feb. 28, 2016) (cit. on p. 17).

[36] Intel. *Intel HPDD*. 2016. URL: `https://wiki.hpdd.intel.com/display/PUB/HPDD+Wiki+Front+Page` (visited on Feb. 28, 2016) (cit. on p. 17).

[37] Fujitsu. *Fujitsu*. 2016. URL: `http://www.fujitsu.com` (visited on Feb. 28, 2016) (cit. on p. 17).

[38] National Aeronautics and Space Administration. *National Aeronautics and Space Administration*. 2016. URL: `http://www.nasa.gov` (visited on Feb. 28, 2016) (cit. on p. 17).

[39] Roland Laifer. *The parallel file system Lustre*. Karlsruher Institut für Technologie. Apr. 16, 2014. URL: `https://www.scc.kit.edu/scc/docs/Lustre/scc_storage_workshop_20140416.pdf` (visited on Feb. 29, 2016) (cit. on p. 17).

[40] Open Scalable File System Inc. *Open Scalable File System*. 2016. URL: `http://opensfs.org/` (visited on Feb. 28, 2016) (cit. on p. 17).

[41] European Open File System. *European Open File System*. 2016. URL: `http://www.eofs.org/` (visited on Feb. 28, 2016) (cit. on p. 17).

[42] Torben Kling Petersen. *Inside the Lustre File System*. Seagate. 2014. URL: `http://www.seagate.com/files/www-content/solutions-content/cloud-systems-and-solutions/high-performance-computing/_shared/docs/clusterstor-inside-the-lustre-file-system-ti.pdf` (visited on Feb. 28, 2016) (cit. on p. 18).

[43] Feiyi Wang, Sarp Oral, Galen Shipman, et al. *Understanding Lustre Filesystem Internals*. URL: `http://info.ornl.gov/sites/publications/Files/Pub15218.pdf` (visited on Feb. 28, 2016) (cit. on p. 18).

[44] Red Hat Inc. *Differences from POSIX*. 2016. URL: `http://docs.ceph.com/docs/master/cephfs/posix/` (visited on Feb. 29, 2016) (cit. on p. 19).

[45] Red Hat Inc. *Red Hat Ceph Storage*. 2016. URL: http://www.redhat.com/en/technologies/storage/ceph (visited on Feb. 29, 2016) (cit. on p. 19).

[46] Red Hat Inc. *Development Guide – Architecture*. 2016. URL: http://docs.ceph.com/docs/master/dev/#architecture (visited on Feb. 29, 2016) (cit. on p. 19).

[47] IBM. *Ceph: A Linux petabyte-scale distributed file system*. June 4, 2010. URL: http://www.ibm.com/developerworks/library/l-ceph/ (visited on Feb. 29, 2016) (cit. on p. 19).

[48] Sage A Weil, Scott A Brandt, Ethan L Miller, and Carlos Maltzahn. „CRUSH: Controlled, scalable, decentralized placement of replicated data“. In: *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*. ACM. 2006, p. 122 (cit. on p. 19).

[49] Sage Weil. „Ceph: Reliable, Scalable, and High-Performance Distributed Storage“. PhD Dissertation. University of California, Santa Cruz, CA (cit. on p. 19).

[50] Red Hat Inc. *Ceph Filesystem*. 2016. URL: http://docs.ceph.com/docs/master/cephfs/ (visited on Feb. 29, 2016) (cit. on p. 20).

[51] Dell. *Technical Guide Book for PowerVault MD3200MD3220*. 2010. URL: http://www.dell.com/downloads/global/products/pvaul/en/powervault-md3200-md3220-technical-guidebook-en.pdf (visited on Feb. 26, 2016) (cit. on p. 21).

[52] Vasily Tarasov, Jim Mauro, and Spencer Shepler. *Filebench*. Feb. 19, 2015. URL: http://filebench.sourceforge.net/ (visited on Feb. 24, 2016) (cit. on p. 23).

[53] Russel Coker. *Bonnie++*. Apr. 16, 2013. URL: http://bonnie.sourceforge.net/ (visited on Feb. 24, 2016) (cit. on p. 23).

[54] Unknown. *s3mdtest*. June 8, 2015. URL: https://github.com/MDTEST-LANL/mdtest (visited on Jan. 31, 2016) (cit. on p. 24).

[55] Github Inc. *Github*. 2016. URL: https://github.com/ (visited on Feb. 22, 2016) (cit. on p. 25).

[56] Python Software Foundation. *Python*. Dec. 5, 2015. URL: https://www.python.org/ (visited on Feb. 30, 2016) (cit. on pp. 25, 27).

[57] *Gnuplot*. Dec. 24, 2015. URL: https://www.gnuplot.info/ (visited on Feb. 30, 2016) (cit. on p. 26).

[58] John Hunter, Darren Dale, Eric Firing, and Michael Droettboom. *Matplotlib*. Feb. 8, 2016. URL: http://matplotlib.org/ (visited on Feb. 25, 2016) (cit. on p. 28).

List of Figures

List of Tables

Colophon

This thesis was typeset with $\LaTeX\,2_\varepsilon$. It uses the *Clean Thesis* style developed by Ricardo Langner with slight modifications by Thomas Kemmer and Marc-André Vef. The design of the *Clean Thesis* style is inspired by user guide documents from Apple Inc.

Download the *Clean Thesis* style at `http://cleanthesis.der-ric.de/`.

Declaration

I hereby declare that I have written the present thesis independently and without use of other than the indicated means. I also declare that to the best of my knowledge all passages taken from published and unpublished sources have been referenced. The paper has not been submitted for evaluation to any other examining authority nor has it been published in any form whatsoever.

Mainz, Germany, March 18, 2016

Marc-André Vef

Erklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Die aus anderen Quellen oder indirekt übernommenen Daten und Konzepte sind unter Angabe der Quelle gekennzeichnet. Die Arbeit wurde bisher weder im In- noch Ausland in gleicher oder ähnlicher Form in einem Verfahren zur Erlangung eines akademischen Grades vorgelegt.

Mainz, Deutschland, 18. März 2016

Marc-André Vef