Berufsakademie Sachsen
Staatliche Studienakademie Leipzig

ZFS unter Linux - Evaluierung und Optimierung

Bachelorthesis

zur Erlangung der staatlichen Abschlussbezeichnung eines

Bachelor of Science

in der Studienrichtung Informatik

Eingereicht von: Markus Then
Schwartzestr. 27
04229 Leipzig
cs13-2
Matrikelnr.: 5000352

Erstgutachter: Herr Dr. Helmut Hayd
Max-Planck-Institut für
Kognitions- und Neurowissenschaften
Stephanstraße 1a
04103 Leipzig

Zweitgutachter: Herr Prof. Dr. Ingolf Brunner

Leipzig, 17. Juni 2016

Inhaltsverzeichnis

Abkürzungsverzeichnis

ARC	Adjustable Replacement Cache
BSD	Berkeley Software Distribution
Btrfs	B-tree File System
CDDL	Common Development and Distribution License
COW	Copy-On-Write
CPU	Central Processing Unit
CSV	Comma-Separated Values
DDT	Dedup Table
ECC	Error-Correcting Code
L2ARC	Level 2 Adjustable Replacement Cache
LVM	Logical Volume Manager
NFS	Network File System
MRT	Magnetresonanztomographie
NTFS	New Technology File System
OS	Operating System
PPA	Personal Package Archive
RAID	Redundant Array of Independent Disks
RAM	Random-Access Memory
ReFS	Resilient File System
SATA	Serial Advanced Technology Attachment
SDS	Software-defined storage
SHA	Secure Hash Algorithm
SLOG	Separate Intent Log
SSD	Solid-State-Drive
TXG	Transaction Group
USV	Unterbrechungsfreie Stromversorgung
vdev	Virtual Device
ZLE	Zero Length Encoding
ZIL	ZFS Intent Log
ZoL	ZFS on Linux

Abbildungsverzeichnis

Listingverzeichnis

Tabellenverzeichnis

1 Einleitung

In der heutigen Zeit werden immer größere, leistungsfähigere und gut skalierbare Speicherlösungen benötigt. Es fallen immer mehr Daten in immer kürzeren Zeiträumen an, die sicher gespeichert werden müssen. Dabei ist es nicht mehr ausreichend und angemessen diesen Anforderungen durch immer bessere und kostenintensivere Hardware gerecht zu werden. Aus diesem Grund werden Softwarelösungen benötigt, die eine effiziente Nutzung und Verwaltung von sehr großen Speichermengen ermöglichen. Die Verlagerung der Funktionen des Speichermanagements von der Hardware in die Software wird als Software-defined storage[5] (SDS) bezeichnet.

Mit den sogenannten „Next Generation File Systems“ wurden deshalb neue Dateisysteme entwickelt, deren Funktionsumfang den herkömmlicher Dateisysteme weit überschreitet. Sie vereinen meist mehrere Komponenten und können auf diese Weise optimale Geschwindigkeiten erreichen. Dabei werden Funktionen angeboten, die bei der Kombination unabhängig voneinander entwickelter Soft- und Hardwarekomponenten nicht möglich wären. So werden z. B. häufig die Aufgaben des RAID-Controllers übernommen und eng mit dem eigentlichen Dateisystem verknüpft, um zusätzliche Sicherheitsmechanismen realisieren zu können. Beispiele für diese Generation von Dateisystemen sind beispielsweise Btrfs[6], ZFS[7] und ReFS[8].

In den nachfolgenden Kapiteln wird ZFS als vielversprechendster Vertreter dieser neuen Dateisysteme behandelt. Es wurde aufgrund der weiten Verbreitung, des fortgeschrittenen Entwicklungsstandes sowie der regen Weiterentwicklung neuer Funktionalitäten ausgewählt. Diese Arbeit soll die Eignung von ZFS für den Einsatz auf mehreren großen Speichersystemen mit handelsüblicher Hardware evaluieren. Darüber hinaus wird zur Erkennung von Optimierungsmöglichkeiten der Einfluss verschiedener Parameter auf die Geschwindigkeit und Leistung untersucht. Durch Benchmarks werden die Vor- und Nachteile der Funktionen, die ZFS von herkömmlichen Dateisystemen unterscheiden, ermittelt. Es treten immer häufiger Fälle auf, bei denen statt einer allgemeinen Optimierung eine Anpassung an bestimmte Anforderungen gewünscht ist. Deshalb sollen in dieser Arbeit ebenfalls Optimierungen hinsichtlich hoher Geschwindigkeit, hoher Sicherheit und paralleler Zugriffe untersucht und behandelt werden, um anschließend Empfehlungen für eine optimale Konfiguration von ZFS geben zu können. Zur Erkennung eventuell notwendiger Anpassungen der Hardwarekonfiguration wird bei jedem Benchmark, soweit möglich, der begrenzende Faktor ermittelt.

2 ZFS

ZFS ist ein im Jahre 2001 von Sun Microsystems für das Betriebssystem Solaris entwickeltes Dateisystem, welches gleichzeitig einen Logical Volume Manager (LVM) und RAID-Funktionalitäten in sich vereint [9]. Das Ziel war der Entwurf einer leicht zu handhabenden Lösung für Speichersysteme, welche vielfältige Funktionen und Erweiterungen anbietet. Während ZFS ursprünglich für „Zettabyte File System“ stand, ist es inzwischen nur noch ein Pseudo-Akronym [7]. Obwohl es hauptsächlich für den professionellen Einsatz auf Servern und in Rechenzentren konzipiert ist, kann es auch in Verbindung mit günstiger Standardhardware verwendet werden. Durch die sehr einfach gehaltene Konfiguration und Verwaltung von ZFS kann es sowohl von erfahrenen, als auch von unerfahrenen Benutzern mit wenig Aufwand eingesetzt werden.
Nachdem der Quellcode 2005 unter der CDDL-Lizenz veröffentlicht wurde, begann die Entwicklung mehrerer Derivate zur Portierung von ZFS auf andere Betriebssysteme. Als Ergebnis des „ZFS on Linux“(ZOL)-Projekts wurde 2013 die erste stabile Portierung eines Kernel-Moduls für Linux veröffentlicht, welches offiziell für den produktiven Einsatz freigegeben war. In dieser Arbeit kommt die Version 28 des ZoL-Projekts zum Einsatz.[9] Da momentan nur zwei aktuelle Bücher zum Thema ZFS existieren, die sich darüber hinaus nur auf die Portierung für BSD beziehen, ist die Arbeit großteils auf Internetquellen aufgebaut.

2.1 Begriffserklärungen

Nachfolgend ist die für das Verständnis dieser Arbeit notwendige Terminologie von ZFS (Vgl. [10]) erläutert:

vdev (virtual device) - ist ein virtuelles Gerät, auf dem die in ZFS gespeicherten Daten abgelegt werden. Es kann sich dabei um eine Festplatte, eine reguläre Datei, ein RAID-Z oder gespiegelte Festplatten handeln. Im Falle eines RAID-Z ist es nur ein Container für weitere vdevs.

RAID-Z - ist ein speziell für ZFS implementiertes Software-RAID mit dem Ziel einer hohen Datenverfügbarkeit und Geschwindigkeit. Es kann mit herkömmlichen RAIDs verglichen werden und existiert in den Varianten RAID-Z1, RAID-Z2 und RAID-Z3. Dabei steht die Zahl im Namen jeweils für die Anzahl der *vdevs*, die zur Speicherung von Pa-

ritäten genutzt werden. Bei einem RAID-Z3 können dementsprechend drei Festplatten ausfallen, ohne dass Daten verloren gehen. Wie beim RAID wird durch Striping und Verteilen der Paritäten über die Festplatten eine höhere Lese- und Schreibgeschwindigkeit erreicht. Weitere Gemeinsamkeiten und Unterschiede können dem Kapitel 6.1, S. 49 entnommen werden.

zpool - ist ein Zusammenschluss aus mehreren vdevs zu einem großen Speicherpool. Da er für die Ablage aller Daten verwendet wird, ist er ein zentraler Bestandteil jeder ZFS-Konfiguration. Zur besseren Auslastung der Hardware werden die Daten gleichmäßig über alle vdevs verteilt. Häufig wird ein zpool verkürzt als Pool bezeichnet.

Dataset - bezeichnet ZFS-Dateisysteme, Volumes und Snapshots. Datasets sind in einer Vererbungshierarchie organisiert und immer einem bestimmten zpool zugeordnet. Sie dienen der Strukturierung von Pools und ermöglichen eine Anpassung an verschiedene Anforderungen. Durch das Setzen von Parametern können die Komprimierung, der Prüfsummenalgorithmus, die logische Blockgröße und viele weitere Funktionen und Eigenschaften für jedes Dataset einzeln verändert werden. Alle Datasets teilen sich den Speicher des gesamten Pools und sind deshalb nicht als Partition mit eigenem Speicherbereich anzusehen.

zvol - ist ein Dataset, welches ein virtuelles Blockdevice enthält. Auf diesem kann wie auf einer Festplatte ein Dateisystem angelegt werden. Dies ermöglicht die Kombination der Funktionen von ZFS (z. B. Komprimierung) mit anderen Dateisystemen (z. B. BeeGFS[11]), ohne dass diese dafür angepasst sein müssen.

Copy-On-Write - beschreibt die Funktion von Dateisystemen, Blöcke nicht direkt zu überschreiben, sondern eine Kopie der neuen Daten in einem anderen freien Block abzulegen. Nachdem ein neuer Block erfolgreich abgespeichert wurde, wird der entsprechende Zeiger angepasst. Sollte das System während eines Schreibvorgangs abstürzen, entsteht somit kein inkonsistenter Zustand des Dateisystems. Ausschließlich die in dem Moment geschriebenen Daten gehen verloren, währenddessen die vorherige Version des Blocks noch unverändert ist. Darüber hinaus ermöglicht dieses Konzept das schnelle und einfache Anlegen von Snapshots.

Snapshot - hält den Zustand eines Dateisystems zu einem bestimmten Zeitpunkt fest. Werden Daten im Dateisystem verändert, so wird sowohl die aktuelle, als auch die Version des Snapshots aufgehoben. Zu einem späteren Zeitpunkt kann entweder der Zustand des Snapshots wiederhergestellt oder auf einzelne Daten des Snapshots zugegriffen werden.

Scrub - durchsucht einen Pool nach beschädigten Daten. Dabei wird die Richtigkeit aller gespeicherten Prüfsummen verifiziert, um fehlerhafte Daten zu lokalisieren. Sofern Paritätsinformationen existieren, werden diese zur Rekonstruktion der defekten Blöcke genutzt. Um den Pool vor Beschädigungen zu schützen, sollte ein Scrub mindestens einmal im Monat durchgeführt werden.

Resilvering - bezeichnet die Wiederherstellung von Daten eines RAID-Z oder gespiegelter vdevs unter Verwendung von Paritätsinformationen. Der Vorgang wird beispielsweise durchgeführt, wenn eine Festplatte nach einem Ausfall ausgetauscht wurde.

2.2 Praktische Vorteile von ZFS

Durch die Zusammenführung des Dateisystems mit einem Logical Volume Manager sowie RAID-Funktioninalitäten sind diese drei Komponenten in ZFS entsprechend gut aufeinander abgestimmt. Im Vergleich zu herkömmlichen Dateisystemen existieren einige Vorteile, die in diesem Kapitel kurz zusammengefasst und im weiteren Verlauf der Arbeit untersucht und belegt werden sollen.
So kann beispielsweise durch den Einsatz eines RAID-Z eine hohe Verfügbarkeit der Daten sichergestellt werden. Obwohl sich dessen Funktionalitäten auf den ersten Blick kaum von denen eines Hard- oder Software-RAIDs unterscheiden, zeigt sich im Falle eines Festplattenausfalls ein entscheidender Vorteil. Während herkömmliche RAIDs sowohl freie als auch belegte Blöcke rekonstruieren müssen, ist bei einem RAID-Z nur die Wiederherstellung der belegten Bereiche notwendig. Dies kann je nach Füllstand des Pools eine enorme Zeitersparnis zur Folge haben (siehe Kapitel 6.3, S. 52). Darüber hinaus verringert sich die Wahrscheinlichkeit eines weiteren Festplattenausfalls während der Wiederherstellungsphase, was bei steigenden Festplattenkapazitäten von immer größerer Bedeutung ist.[12]
Durch die Copy-On-Write Funktionalität eignet sich ZFS sehr gut zur speichereffizienten und performanten Erstellung von Snapshots. Bei der Anfertigung von Backups können diese genutzt werden, um einen konsistenten Zustand während des Sicherungszeitraums zu gewährleisten. Darüber hinaus können sie zur Sicherung und Wiederherstellung früherer

Zustände eines Datasets eingesetzt werden.

Im Gegensatz zu einer Kombination aus einem Hardware-RAID und einem einfachen Dateisystem, bietet ZFS die Möglichkeit, die Kapazität eines Pools dynamisch zu erweitern. Dabei werden im laufenden Betrieb weitere *vdevs* zum Pool hinzugefügt. Eine Verkleinerung ist jedoch nicht ohne die Zerstörung des Pools möglich.

Mithilfe von Prüfsummen verifiziert ZFS die Integrität jedes logischen Datenblocks und kann deshalb den Datenstrom während des gesamten Übertragungsvorgangs absichern. Außerdem gewährleistet dieses Verfahren, dass Daten auch nach langen Zeiträumen noch exakt so abgerufen werden können, wie sie ursprünglich abgelegt wurden.[13]

Eine effizientere Nutzung des Speichers kann entweder durch den Einsatz einer blockweisen Deduplizierung oder Komprimierung erreicht werden. Bei der Deduplizierung werden identische Blöcke anhand der berechneten Prüfsummen erkannt und nicht erneut abgespeichert. Diese Funktion kann je nach Nutzung sehr viel Speicher sparen, aber gleichzeitig auch viel CPU-Leistung und Arbeitsspeicher beanspruchen. Mithilfe der auf Dateisystemebene durchgeführten Komprimierung kann ebenfalls je nach Art der Daten eine Speicherersparnis erzielt werden. Bei Bedarf ist auch der gleichzeitige Einsatz beider Verfahren möglich.

Soll ein Pool an ein anderes System angeschlossen werden, so muss dieser lediglich am ursprünglichen System exportiert werden. Dabei werden alle zur Verwaltung notwendigen Daten auf den *vdevs* abgelegt. Am Zielsystem müssen die Festplatten nicht wieder in derselben Reihenfolge angeschlossen werden. Beim Import sucht ZFS nach allen *vdevs*, die Teil des gewünschten Pools sind und stellt diesen inklusive aller Einstellungen und Eigenschaften wieder bereit. Für einen erfolgreichen Import muss die ZFS-Version auf dem Zielsystem mindestens der des Quellsystems entsprechen.[14]

Da sowohl das Anlegen eines *zpools* als auch das Erstellen eines RAID-Z keine größeren Initialisierungsprozesse benötigt, ist dies in wenigen Sekunden möglich. Mit diesen und weiteren kleinen Vorteilen versucht ZFS auch zukünftigen Anforderungen immer größerer und schnellerer Speichermedien gerecht zu werden.

2.3 Das Testsystem

Zur Evaluierung und Optimierung von ZFS kommt ein leistungsfähiger Server als Testsystem zum Einsatz. Um störende Einflüsse zu vermeiden, befindet sich dieser nicht im produktiven Betrieb und wird exklusiv für die Benchmarks eingesetzt. Auf diese Weise sollen aussagekräftige und reproduzierbare Ergebnisse gewährleistet werden. Nach dem Abschluss aller Tests sollen der nachfolgend beschriebene und noch weitere Server mithilfe der gewonnenen Erkenntnisse optimiert und gemeinsam eingesetzt werden. Die verschiedenen Belastungsszenarien werden künstlich durch entsprechende Gestaltung der Tests erzeugt. Sie sollen möglichst genau den späteren Ansprüchen im produktiven Einsatz entsprechen und gleichzeitig für zukünftige Benchmarks reproduzierbar sein.
In den nächsten beiden Kapiteln wird genauer auf die Hard- und Softwarekonfiguration des Testsystems eingegangen.

2.3.1 Hardware

In nachfolgender Tabelle ist die Hardwarekonfiguration des Testsystems aufgelistet:

Komponente	**technische Beschreibung**
CPU	2 x Intel Xenon Haswell E5-2637v3, 4-Core, 3,50 GHz
RAM	8 x 32 GB DDR4 2133 MHz ECC (256 GB)
HDD	2 x SAS Seagate Cheetah 15K.7 300 GB (für OS) 44 x SAS 12 GBit Seagate ST6000NM0034 6 TB (= 264 TB Brutto)
Controller intern	8fach SAS/SATA HBA LSI/Avago 9207-8i (für OS)
Controller (JBOD)	8fach SAS 12 GBit HBA LSI/Avago 9300-8E (für ZFS)
Netzwerk	10 GBit Ethernet DualPort Adapter Intel X520-SR2 (2-Port SFP+/ Fiber LC)

Tabelle 1: Technische Daten des ZFS-Servers

Es handelt sich dabei um eine Konfiguration aus handelsüblicher Hardware, die bereits auf die Anforderungen von ZFS abgestimmt wurde. Da die notwendige CPU-Leistung und Größe des RAMs stark von den verwendeten ZFS-Funktionen abhängt, wird im Laufe der Arbeit jeweils die leistungsbegrenzende Hardwarekomponente ermittelt. Gegebenenfalls könnte dann durch Nachrüstung oder Anpassung der Hardwarekonfiguration bei der Beschaffung des nächsten ZFS-Servers eine Steigerung der Leistungsfähigkeit erreicht

werden.

Das System besteht aus zwei 19 Zoll Gehäusen. In einem Gehäuse, dem Storage Node, befinden sich 44 SAS-Festplatten, welche später für ZFS genutzt werden sollen. Diese sind über zwei SFF-8644 Kabel mit 4 x 12 GBit/s an das zweite Gehäuse, den Server, angeschlossen. Dieser enthält alle anderen Komponenten. Die gesamte Hardware ist in Abbildung 26, S. 68 bis 29, S. 69 dargestellt.

2.3.2 Software

Als Betriebssystem kommt auf dem Testsystem Xubuntu in Version 14.04 zum Einsatz. Aufgrund von Lizenzinkompatibilitäten zwischen Linux und ZFS war es bisher nicht möglich, ZFS in die offiziellen Paketquellen von Ubuntu aufzunehmen [15]. Trotz heftiger Kritik wurde dies in Ubuntu 16.04 dennoch getan [16]. Bei älternen Versionen ist vor der Installation ein Personal Package Archive (PPA) einzubinden, welches die notwendigen Pakete zur Verfügung stellt. Aus diesem kann anschließend das Paket *ubuntu-zfs* installiert werden. Dabei werden automatisch alle Komponenten für die entsprechende Plattform und Rechnerarchitektur kompiliert. Im Anschluss muss das installierte Kernel-Modul geladen werden.[9]

Die notwendigen Befehle sind in folgendem Listing dargestellt:

```
1 apt-add-repository ppa:zfs-native/stable
  apt-get update
  apt-get install ubuntu-zfs
  modprobe zfs
```

Nach Abschluss der Installation stehen sowohl das notwendige Kernel-Modul als auch alle für die Administration benötigten Tools zur Verfügung. Dazu gehören *zpool*, *zfs*, *zdb* und *zstreamdump*. Das Tool *zpool* wird zum Erstellen, Verwalten und Löschen von Pools genutzt und ist damit für das zentrale Element von ZFS zuständig. Mit *zfs* werden die Datasets innerhalb eines Pools verwaltet. Darüber hinaus ermöglicht es das Anlegen von Snapshots und die Anpassung von Datasets durch das Setzen von Parametern. Der Debugger von ZFS verbirgt sich hinter dem Tool *zdb* und richtet sich hauptsächlich an die Entwickler des Dateisystems. Man kann es jedoch auch zum Auslesen von Statistiken eines Pools bzw. Datasets nutzen. Durch die Möglichkeit der Simulation einiger Einstellungen können beispielsweise die Auswirkungen einer blockweisen Deduplizierung geprüft werden, bevor diese wirklich aktiviert wird. Mithilfe von *zstreamdump* können die Headerinformationen und Statistiken eines ZFS-Dumps betrachtet werden.

2.3.3 Einrichtung und Konfiguration

Da ZFS in Form eines Kernel-Moduls implementiert wurde, muss dieses vor der Verwendung geladen sein. Sollte dies nicht automatisch geschehen, kann es mit dem Kommando *modprobe zfs* nachträglich geladen werden. Um ZFS verwenden zu können, muss anschließend ein *zpool* angelegt werden. Dies erfolgt unter Verwendung des gleichnamigen Tools und ist mit folgendem Befehl möglich:

```
1 zpool create -m <MOUNTPOINT> -o ashift=12 <POOLNAME> <DEVICES>
```

Dabei müssen mindestens der Name des Pools, der zu verwendende Mountpoint sowie die gewünschten *vdevs* angegebenen werden. Darüber hinaus können nach dem Parameter *-o* von der Standardkonfiguration abweichende Optionen für den *zpool* gesetzt werden. Eine häufig genutzte Option ist *ashift*. Dabei handelt es sich um die für die *vdevs* zu verwendende physische Blockgröße (siehe Kapitel 5.1.1, S. 20). Der Wert wird als Exponent zur Basis 2 angegeben und ist besonders bei Festplatten relevant, die intern andere Sektorgrößen nutzen, als sie aus Kompatibilitätsgründen an das Betriebssystem melden. Ein Sektor bezeichnet bei Festplatten die kleinste Organisationseinheit, die gelesen bzw. geschrieben werden kann [17]. Der Wert 12 entspricht der mittlerweile standardmäßig eingesetzten Sektorgröße von 4 KiB. Während der zuvor erläuterte Befehl die *vdevs* zu einem Pool ohne Redundanzen oder Paritätsinformationen verknüpfen würde, können durch Angabe bestimmter Schlüsselwörter (siehe Tabelle 2) einzelne *vdevs* beispielsweise gespiegelt bzw. zu einem RAID-Z verbunden werden.

Schlüsselwort	**Bedeutung**
raidz1	Software-RAID mit einem vdev für Paritäten
raidz2	Software-RAID mit zwei vdev für Paritäten
raidz3	Software-RAID mit drei vdev für Paritäten
mirror	Spiegelung mehrerer vdevs
cache	Verwendung von vdevs zum Caching
log	Auslagerung des Intent Logs auf angegebene vdevs
spare	Ersatz-vdev bei Plattenausfall (raidz oder mirror)

Tabelle 2: Schlüsselwörter zur Strukturierung eines Pools

Diese zwei Befehle würden entsprechende Pools erzeugen:

```
# Anlegen eines zpools bestehend aus einem Mirror (gespiegelte vdevs)
zpool create -m <MOUNTPOINT> -o ashift=12 <POOLNAME> mirror <DEVICES>

4 # Anlegen eines zpools bestehend aus einem RAID-Z2
zpool create -m <MOUNTPOINT> -o ashift=12 <POOLNAME> raidz2 <DEVICES>
```

Eine spätere Abfrage der Struktur eines Pools ist ebenfalls mit dem Tool *zpool* unter Angabe des Parameters *status* und dem entsprechenden Poolnamen möglich:

Listing 1: Ausgabe von 'zpool status'

```
user@host:~$ zpool status <POOLNAME>
  pool: <POOLNAME>
 state: ONLINE
  scan: none requested
config:

                  NAME              STATE     READ WRITE CKSUM
                  <POOLNAME>        ONLINE       0     0     0
                    mirror-0        ONLINE       0     0     0
                      sdb           ONLINE       0     0     0
                      sdc           ONLINE       0     0     0
                    mirror-1        ONLINE       0     0     0
                      sdd           ONLINE       0     0     0
                      sde           ONLINE       0     0     0
```

Durch eine hierarchische Ausgabe der *vdevs* werden Spiegelpaare und RAID-Zs kenntlich gemacht. Alle *vdevs*, die eine Stufe unterhalb des Pools liegen, werden durch Striping verknüpft. In diesem Beispiel sind das *mirror-0* und *mirror-1*, die demnach mit einem RAID-10 verglichen werden können.
Innerhalb eines solchen Pools können anschließend Datasets angelegt werden. Diese sind ebenfalls in einer Art Vererbungshierarchie organisiert. Im Gegensatz zum Pool werden sie mit dem Tool *zfs* angelegt:

```
zfs create <POOLNAME>/<DATASETNAME>/[<DATASETNAME>]
```

Standardmäßig werden Datasets entsprechend ihres Platzes in der Hierarchie unterhalb des Mountpoints des dazugehörigen *zpools* eingebunden. Alternativ kann auch ein abweichender Pfad angegeben werden. Für jedes Dataset können verschiedene Parameter gesetzt werden. Dies kann ebenfalls mit dem Kommandozeilentool *zfs* über *set* und *get* Anweisungen durchgeführt werden [18]. In folgendem Listing ist das Setzen und Abfragen der Schlüssel-Wert-Paare für den Parameter *compression* dargestellt:

Listing 2: Setzen und Abfragen von Optionen eines Pools/Datasets

```
# Abfrage eines Parameters
user@host:~$ zfs get compression <POOLNAME>/<DATASETNAME>
NAME     PROPERTY    VALUE    SOURCE
testpool compression off      default

# Setzen eines Parameters
user@host:~$ zfs set compression=gzip <POOLNAME>/<DATASETNAME>
```

Um eine Liste aller gesetzten Optionen zu erhalten, kann bei der Abfrage als Parameter *all* übergeben werden. Bei fehlender Angabe eines Pools oder Datasets wird der entsprechende Wert für alle Pools und Datasets ausgegeben.

3 Monitoring-Tools

Unter Monitoring versteht man die Überwachung eines Systems anhand eines oder mehrerer Parameter [19]. Zur Leistungsüberwachung des Testsystems werden je nach Testszenario und dem zu testendem Parameter verschiedene Anwendungen eingesetzt. Mit diesen wird während der Benchmarks die Auslastung der Hardware überwacht und ausgewertet. Gleichzeitig werden sie genutzt, um die von den Benchmark-Tools ermittelten Lese- und Schreibgeschwindigkeiten auf Plausibilität zu überprüfen. Nach Abschluss der Tests kann anhand der Messwerte der begrenzende Faktor für die Geschwindigkeit des Pools ermittelt werden.

3.1 sysstat

Das Paket *sysstat* bietet eine Vielzahl verschiedener Monitoring-Tools zur Überwachung der Systemauslastung eines Computers. Dazu gehören *sar*, *sadf*, *mpstat*, *iostat*, *tapestat*, *pidstat*, *cifsiostat* und *sa* [20]. Die große Menge an Funktionalitäten dieser Sammlung ermöglicht eine sehr umfangreiche und detaillierte Überwachung. Da diese Möglichkeiten den für das Monitoring des Testsystems notwendigen Umfang überschreiten, wird nachfolgend nur auf die zur Ermittlung des begrenzenden Faktors notwendigen Tools eingegangen.
Mithilfe von *sar* können Daten über die Systemauslastung und -aktivität gesammelt bzw. gespeichert werden [21]. Die Auswahl der zu erhebenden Daten wird über diverse Kommandozeilenparameter durchgeführt. Durch Angabe eines Intervalls können die Abstände zwischen den einzelnen Messpunkten festgelegt werden. Das Programm sammelt so lange Daten, bis entweder eine vorher eingestellte Dauer überschritten oder es vom Benutzer beendet wird. Die Ausgabe der Messwerte kann maschinenlesbar auf dem Standardausgabestrom und gleichzeitig unter Angabe des Parameters *-o* binär in eine Datei erfolgen [21]. Der Vorteil bei der Speicherung der erfassten Werte besteht darin, dass während des Benchmarks nicht gleichzeitig die Systemauslastung ausgewertet und beobachtet werden muss. Möchte man die Ergebnisse nicht manuell auswerten, kann die Datei nach Abschluss des Tests in einem anderen Programm zur grafischen Auswertung importiert werden. Hierfür kann eine Anwendung, wie z. B. *kSar*[22] genutzt werden.
Ein Echtzeitmonitoring kann ebenfalls mit *sar* durchgeführt werden. Dabei muss als Kommandozeilenparameter der Abstand zwischen den auszugebenden Messungen in Sekunden angegeben werden. Als zweiter Kommandozeilenparameter kann optional die Anzahl der durchzuführenden Messungen folgen. Andernfalls muss die Ausführung über die Tastenkombination Strg + C gestoppt werden. Ohne Angabe weiterer Parameter wird aus-

schließlich die CPU überwacht. Ein Beispiel für die Ausgabe ist in Listing 3 zu sehen.

Listing 3: Ausgabe von 'sar 1 3'

```
user@host:~$ sar 1 3
Linux 3.13.0-85-generic (host)      05/12/2016        _x86_64_          (16 CPU)

10:02:25 AM    CPU    %user    %nice  %system  %iowait   %steal    %idle
10:02:26 AM    all     0.00     0.00    24.14     5.35     0.00    70.50
10:02:27 AM    all     0.00     0.00    23.86     5.51     0.00    70.63
10:02:28 AM    all     0.00     0.00    22.19     5.20     0.00    72.60
Average:       all     0.00     0.00    23.40     5.36     0.00    71.24
```

Über weitere Parameter können zusätzlich Statistiken zur RAM-Auslastung, der Netzwerkschnittstelle und vielen weiteren Komponenten erfasst werden. Am Ende der Messung werden zur besseren Auswertbarkeit noch die Durchschnittswerte für den gesamten Messzeitraum ausgegeben. Dies geschieht auch, wenn das Tool über die zuvor erwähnte Tastenkombination beendet wird.

Mithilfe des Tools *sadf* können die mit *sar* erfassten Statistiken in Form eines SVGs visualisiert werden. Um die Auslastung der CPU sowie des RAMs grafisch darzustellen, müssen folgende Befehle ausgeführt werden:

```
# Sammeln der Daten mit sar
sar -o /tmp/monitoring.data 1 120

# Erstellen des SVGs mit sadf
sadf -g /tmp/monitoring.data -- -ur > /tmp/monitoring.svg
```

Ein Beispiel für ein solches SVG ist in Abbildung 30, S. 70 zu finden. Die Messung wurde während eines Schreibvorgangs auf einem *zpool* durchgeführt. ZFS befand sich dabei in der Standardkonfiguration ohne weitere Anpassungen.

3.2 zpool

Mit *zpool* können nicht nur die Pools von ZFS verwaltet, sondern auch überwacht werden. Durch Angabe des Parameters *list* sowie des gewünschten Pools kann folgende Statistik über den aktuellen Zustand des Pools abgerufen werden [23]:

Listing 4: Ausgabe von 'zpool list'

```
user@host:~$ zpool list <POOLNAME>
NAME      SIZE  ALLOC   FREE  EXPANDSZ   FRAG    CAP  DEDUP  HEALTH  ALTROOT
testpool 4.53T  306G   4.23T    -         3%     6%  1.00x  ONLINE     -
```

Die Bedeutung der einzelnen Spalten kann nachfolgender Tabelle entnommen werden:

NAME	Name des Pools
SIZE	Größe des Pools
ALLOC	Belegter Speicher
FREE	Freier Speicher
EXPANDSZ	nicht genutzter Speicher, nachdem eine Festplatte durch eine größere ersetzt wurde [24]
FRAG	Fragmentierung der freien Blöcke
CAP	Belegter Speicher in Prozent
DEDUP	Deduplizierungsfaktor
HEALTH	Zustand des Pools
ALTROOT	Alternatives root des Pools (falls vorhanden)

Tabelle 3: Bedeutung der Ergebnisse von *zpool list* [2]

Darüber hinaus kann die momentane I/O-Statistik eines Pools angezeigt und überwacht werden. Durch Angabe eines Intervalls in Sekunden und des entsprechenden Pools erfolgt eine relegmäßige Ausgabe der Lese- und Schreibstatistiken. Diese ermöglicht eine Echtzeitüberwachung der Poolauslastung. In Listing 5 ist ein Beispiel für eine entsprechende Ausgabe zu sehen.

Listing 5: Ausgabe von 'zpool iostat'

```
fuser@host:~$ zpool iostat <POOLNAME> 1
              capacity     operations    bandwidth
pool       alloc   free   read  write   read  write
----------  -----  -----  -----  -----  -----  -----
testpool     307G  4.23T      3     18   366K  2.12M
testpool     307G  4.23T      8  2.67K  36.0K   322M
testpool     308G  4.23T     16  2.21K  68.0K   255M
testpool     308G  4.23T      8  3.13K  36.0K   376M
```

Die dargestellte Ausgabe zeigt im Intervall von einer Sekunde die jeweiligen Mittelwerte für den Zeitraum seit der letzten Messung an. Je größer das Intervall gewählt wird, desto

weniger fallen Geschwindigkeitsspitzen bzw. -einbrüche auf. Die Angaben repräsentieren folgende Werte:

pool	Name des Pools
alloc capacity	Belegter Speicher
free capacity	Freier Speicher
read operations	Anzahl I/O Operationen (Lesen)
write operations	Anzahl I/O Operationen (Schreiben)
read bandwidth	Lesegeschwindigkeit
write bandwidth	Schreibgeschwindigkeit

Tabelle 4: Bedeutung der Ergebnisse von *zpool iostat* [3]

Sollte die Abfrage der Werte für jedes *vdev* einzeln notwendig sein, kann zusätzlich der Parameter *-v* angegebenen werden [23]. Für noch genauere Statistiken muss der Debugger von ZFS verwendet werden.

3.3 zdb

Neben dem reinen Debugging kann *zdb* auch zum Monitoring eingesetzt werden. Da es sich jedoch eher an die Entwickler richtet, fällt die Dokumentation großteils knapp aus. Somit ist es möglich, dass Funktionen existieren, die man als Endnutzer nicht direkt finden kann. Des Weiteren sind die Ausgaben des Tools bzw. dessen Funktionsweise teilweise schwer zu verstehen.

Mit der Simulation der blockweisen Deduplizierung bietet es jedoch ein sehr hilfreiches Feature an. Dabei wird die theoretische Auswirkung der Deduplizierung auf einen bestimmten Pool berechnet. Dieser Vorgang kann je nach Größe des Pools viel Zeit in Anspruch nehmen. Am Ende werden der erreichbare Deduplizierungsfaktor sowie weitere Eckdaten des Pools ausgegeben. Anhand dieser Simulation kann entschieden werden, ob eine Aktivierung der Deduplizierung ratsam ist. Weitere Ausführungen zur Deduplizierung und deren Vor- und Nachteilen sind in Kapitel 5.4, S. 35 zu finden. Die folgende Ausgabe zeigt das Ergebnis einer solchen Simulation, die über den Parameter *-S* ausgelöst werden kann [25]:

Listing 6: Ausgabe von 'zdb -S'

```
user@host:~$ zdb -S <POOLNAME>
Simulated DDT histogram:

bucket              allocated                     referenced
------  ------------------------------  ------------------------------
refcnt  blocks  LSIZE   PSIZE   DSIZE   blocks  LSIZE   PSIZE   DSIZE
------  ------  -----   -----   -----   ------  -----   -----   -----
     1    555K  69.3G   33.3G   33.3G    555K  69.3G   33.3G   33.3G
    2M      1   128K    128K    128K   2.13M   272G    272G    272G
 Total    555K  69.3G   33.3G   33.3G   2.67M   342G    306G    306G

dedup = 9.18, compress = 1.12, copies = 1.00, dedup * compress / copies = 10.26
```

Die für die Auswertung der Simulation relevanten Werte befinden sich in der letzten Zeile der Ausgabe. Der unter *dedup* ausgegebene Wert entspricht dem berechneten Deduplizierungsfaktor. Je größer dieser ist, desto wirkungsvoller ist die Deduplizierung. Der als *compress* angegebene Komprimierungsfaktor wird nicht berechnet, sondern entspricht dem tatsächlichen Zustand des Pools. Es wird demnach keine Simulation der Komprimierung durchgeführt. Sollte eine bestimmte Anzahl an Replikas konfiguriert sein, würde der Wert *copies* der Anzahl dieser Replikas entsprechen. Aus den drei beschriebenen Werten wird zusätzlich das Verhältnis zwischen den zu speichernden Daten und dem dafür benötigten Speicherplatz unter Berücksichtigung aller Einflussgrößen berechnet und ausgegeben.

4 Benchmarking mit fio

Ziel des Benchmarkings ist die Messung der Leistung von Systemen unter reproduzierbaren und vergleichbaren Bedingungen. Da erste Tests mit *bonnie++*[26] gezeigt haben, dass es nicht die volle Geschwindigkeit des Pools erreichen kann, wird in diesem Kapitel die Eignung von *fio* zur Simulation der unterschiedlichen Lastszenarien untersucht. Während der Benchmarks wird die Auslastung des Testsystems mit den in Kapitel 3, S. 10 beschriebenen Monitoring-Tools zur Erkennung begrenzender Faktoren überwacht. Die von dem Benchmark-Tool ermittelten Geschwindigkeiten werden anschließend zum Vergleich verschiedener Einstellungen und Konfigurationen verwendet.

Neben dem reinen Benchmarking eignet sich *fio* auch zur Durchführung von spezifischen Stresstests zur Belastung der Hardware. Durch die Unterstützung diverser I/O-Engines wie *libaio* oder *posixaio* kann es sehr flexibel eingesetzt werden. Bei der Wahl einer dieser Engines wird die Art und Weise beeinflusst, in der die Ein- bzw. Ausgabe der Daten durchgeführt wird [4].

Der Ablauf von Benchmarks kann individuell über Kommandozeilenparameter oder Konfigurationsdateien definiert und angepasst werden. Der einfache und übersichtliche Aufbau dieser Konfigurationsdateien ermöglicht eine detaillierte und einfach zu reproduzierende Abbildung verschiedener Lastszenarien. Dabei können unter anderem die zeitliche Abfolge, die Datenmenge und die Anzahl gleichzeitiger Zugriffe (Jobs) konfiguriert werden. Somit können individuelle und reproduzierbare Benchmarks zur Analyse verschiedener Systemkonfigurationen erstellt werden.[27]

Abbildung 9, S. 71 zeigt eine Konfigurationsdatei, die alle Parameter enthält, mit denen die Benchmarks in dieser Arbeit durchgeführt werden. In Abbildung 10, S. 72 ist ein komplexeres Testszenario abgebildet. Dabei werden drei Clients simuliert, die zeitversetzt und mit unterschiedlichem Verhalten Daten schreiben. Durch die Möglichkeit mehrere kleine Benchmarks innerhalb einer Konfigurationsdatei samt zeitlicher Abfolge zu definieren, können sehr komplexe Lastszenarien abgebildet werden.

Listing 7 zeigt die Ausgabe der Ergebnisse eines mit *fio* durchgeführten Benchmarks. Dabei wurden alle für den Test notwendigen Parameter über die Kommandozeile übergeben. Auf den Einsatz paralleler Jobs wurde zugunsten einer übersichtlicheren Ausgabe verzichtet.

Listing 7: Ausgabe von 'fio'

```
user@host:~$ fio --rw=write --name=benchmark1 --numjobs=1 --size=50M
test.file: (g=0): rw=write, bs=4K-4K/4K-4K/4K-4K, ioengine=sync, iodepth=1
fio-2.1.3
Starting 1 process

test.file: (groupid=0, jobs=1): err= 0: pid=305074: Thu Mar 31 09:04:37 2016
  write: io=51200KB, bw=219742KB/s, iops=54935, runt= 233msec
    clat (usec): min=13, max=87, avg=15.59, stdev= 2.74
     lat (usec): min=13, max=89, avg=16.16, stdev= 2.77
    clat percentiles (usec):
     |  1.00th=[  13],  5.00th=[  13], 10.00th=[ 13], 20.00th=[ 14],
     | 30.00th=[  14], 40.00th=[ 14], 50.00th=[ 14], 60.00th=[ 14],
     | 70.00th=[  18], 80.00th=[ 19], 90.00th=[ 19], 95.00th=[ 19],
     | 99.00th=[  25], 99.50th=[ 26], 99.90th=[ 27], 99.95th=[ 29],
     | 99.99th=[  57]
    lat (usec) : 20=96.06%, 50=3.92%, 100=0.02%
  cpu          : usr=15.09%, sys=83.19%, ctx=4, majf=0, minf=25
  IO depths    : 1=100.0%, 2=0.0%, 4=0.0%, 8=0.0%, 16=0.0%, 32=0.0%, >=64=0.0%
     submit    : 0=0.0%, 4=100.0%, 8=0.0%, 16=0.0%, 32=0.0%, 64=0.0%, >=64=0.0%
     complete  : 0=0.0%, 4=100.0%, 8=0.0%, 16=0.0%, 32=0.0%, 64=0.0%, >=64=0.0%
     issued    : total=r=0/w=12800/d=0, short=r=0/w=0/d=0

Run status group 0 (all jobs):
  WRITE: io=51200KB, aggrb=219742KB/s, minb=219742KB/s, maxb=219742KB/s, mint=233msec,
        maxt=233msec
```

Dem mittleren Block ab Zeile 6 können detaillierte Informationen zu dem ausgeführten Job, wie die durchschnittliche Schreibgeschwindigkeit, die erreichten IOPs und die Laufzeit entnommen werden. Wird der Benchmark mit mehreren parallelen Jobs durchgeführt, so wird dieser Block für jeden Job einzeln ausgegeben. Die letzte Zeile der Ausgabe dient der Zusammenfassung aller Jobs und kann zur allgemeinen Auswertung des Benchmarks herangezogen werden. Die Parameter sind folgendermaßen zu interpretieren:

io	geschriebene / gelesene Datenmenge aller Jobs
aggrb	Gesamtgeschwindigkeit aller Jobs
mindb	Geschwindigkeit des langsamsten Jobs
maxb	Geschwindigkeit des schnellsten Jobs
mint	Dauer des schnellsten Jobs
maxt	Dauer des langsamsten Jobs

Tabelle 5: Bedeutung der Ergebnisse von *fio* [4]

Alle für die durchzuführenden Benchmarks relevanten Kommandozeilenparameter und ihre Bedeutung können der nachfolgenden Tabelle entnommen werden:

--name	Name des Tests und der anzulegenden Dateien
--ioengine	Zu verwendende I/O-Engine
--rw	Art des Tests (read/write/randread/randwrite)
--bs	Blockgröße
--numjobs	Anzahl paralleler Jobs
--size	Dateigröße pro Job
--refill_buffers	Schreibt immer neue Zufallsdaten in den Puffer
--randrepeat	Seed für Zufallsgenerator immer gleich (wenn Wert *1* ist)
--sync	Nutzt synchrone Schreibzugriffe (wenn Wert *1* ist)
--fallocate	Reserviert den Speicher beim Anlegen der Dateien (wenn nicht *none*)

Tabelle 6: Kommandozeilenparameter von *fio* [4]

Auch *fio* konnte durch das Schreiben eines einzelnen Datenstroms nicht die maximal mögliche Geschwindigkeit des Pools ausschöpfen. Dies liegt unter anderem daran, dass die sehr hohen Geschwindigkeiten sowohl die Benchmarktools als auch die gesamte I/O-Kette des Betriebssystems stark beanspruchen. Durch die Verwendung 10 paralleler Jobs konnte jedoch eine vollständige Auslastung des Pools erreicht werden. Da dieser aus 44 Festplatten besteht, über die alle Lese- bzw. Schreibzugriffe verteilt werden, können die 10 Datenströme näherungsweise als ein einzelner betrachtet werden. Die negativen Auswirkungen paralleler Zugriffe sind bei dieser Anzahl vernachlässigbar.
Für die Benchmarks im Rahmen dieser Arbeit wird jeweils die in Linux integrierte *libaio* sowie eine Blockgröße von 128 KiB verwendet. Dies entspricht der Standardgröße der logischen Blöcke von ZFS (siehe Kapitel 5.1.2, S. 22). Um eventuelle Caching-Effekte zu vermeiden, wurde die zu schreibende bzw. zu lesende Datenmenge auf 1000 GiB (10 x 100 GiB) festgelegt. Dies entspricht der vierfachen Größe des Arbeitsspeichers. Die Parameter *--refill_buffers* und *--randrepeat=0* sorgen dafür, dass nicht immer wieder die gleichen Daten geschrieben werden, um somit ein praxisnahes Ergebnis zu erzielen.
Da ZFS die von *fio* verwendeten Methoden zur Reservierung des Speicherplatzes beim Anlegen von Dateien nicht unterstützt, sollte der Parameter *--fallocate=none* bei allen Benchmarks angegeben werden [4]. Andernfalls wird der gesamte angeforderte Speicherplatz zur Reservierung beschrieben, was bei großen Datenmengen viel Zeit in Anspruch nimmt.

Der standardmäßig bei der Untersuchung der verschiedenen Optimierungsmöglichkeiten durchgeführte Benchmark kann mit folgendem Befehl ausgelöst werden:

```
1 fio --name=benchmark --directory=/ZFS/ --ioengine=libaio --rw=write --bs=128k --
      numjobs=10 --size=100G --refill_buffers --randrepeat=0 --fallocate=none
```

Während die meisten Tests mit sequenziellen Lese- und Schreibzugriffen durchgeführt werden, wird es an einigen Stellen notwendig sein, zufällige Zugriffe zu simulieren. Dabei werden die Blöcke einer Datei nicht in der richtigen, sondern einer zufälligen Reihenfolge gelesen bzw. geschrieben, wodurch sich die Zugriffszeiten mechanischer Festplatten stark auswirken. Zur Durchführung eines solchen Benchmarks muss für den Parameter *--rw* entweder *randread* oder *randwrite* angegeben werden.

Bei der Untersuchung des Einflusses verschiedener Blockgrößen muss *fio* über den Parameter *--bs* an die jeweilige Größe angepasst werden. Um komprimierbare bzw. dedupli-zierbare Daten zu erzeugen, ist der Parameter *--refill_ buffers* zu entfernen. Dabei wird der von *fio* genutzte Pufferinhalt zyklisch geschrieben, wodurch eine Datei entsteht, die entsprechend viele Redundanzen für eine Komprimierung bzw. Deduplizierung bietet.

Sollten die beschriebenen Anpassungen notwendig sein, um spezifische Effekte sichtbar zu machen, werden diese an entsprechender Stelle kenntlich gemacht.

5 Ansatzpunkte für allgemeine Optimierungen

In diesem Kapitel werden alle ZFS-Parameter und Optimierungsmöglichkeiten untersucht, die keinem bestimmten Lastszenario zugeordnet werden können. Sie betreffen mehrere oder alle Szenarien und sollen deshalb zentral behandelt werden. Die Tests werden jeweils ausgehend von der Standardkonfiguration mit 4 KiB physischer Blockgröße (*ashift*) durchgeführt. Ausschließlich die zu testenden Optionen werden variiert, während alle anderen Parameter unverändert bleiben. Um möglichst definierte und einheitliche Testbedingungen zu schaffen, wird der Pool nach jedem Test zerstört und neu angelegt. Zusätzlich werden zwischen den Benchmarks alle Caches des Kernels mit folgendem Befehl verworfen:

```
echo 3 > /proc/sys/vm/drop_caches
```

Alle Optimierungen werden unabhängig voneinander, d. h. jedes Mal ausgehend von der Standardkonfiguration durchgeführt. Auf diese Weise sollen ungewollte Wechselwirkungen zwischen verschiedenen Parametern vermieden werden. Um eine möglichst hohe Auslastung der gesamten Hardware zu erreichen, werden für die meisten Tests alle 44 verfügbaren Festplatten zu einem großen *zpool* ohne Redundanz verknüpft. Durch das Striping aller *vdevs* kann diese Konfiguration mit einem sehr großen RAID-0 verglichen werden. Sollten zur Verdeutlichung von bestimmten Effekten Anpassungen der Größe oder Struktur des Pools notwendig sein, so werden diese an entsprechender Stelle erwähnt.
Die Optimierungen sollen jeweils für die maximale Lese- und Schreibgeschwindigkeit eines Pools durchgeführt werden. Zur Ermittlung zuverlässiger Werte wird jede Messung dreimal, unter Bildung eines Mittelwerts druchgeführt.

5.1 Blockgrößen

Man unterscheidet bei ZFS grundsätzlich zwischen einer physischen und einer logischen Blockgröße. Die physische wird durch den Parameter *ashift* eines Pools bestimmt und allgemein als Blockgröße bezeichnet. Die logische Blockgröße unterteilt sich in die *volblocksize* und die *recordsize*. Erstere ist nur bei der Verwendung von *zvols* relevant und entspricht der Blockgröße dieser virtuellen Blockdevices.[28] Da auf Volumes in dieser Arbeit nicht weiter eingegangen wird, wird der Einfluss dieses Parameters auf die Geschwindigkeit nicht untersucht. Die *recordsize* entspricht der logischen Blockgröße von Datasets.
Nachfolgend wird auf die Bedeutung und den Einfluss der physischen Blockgröße (*ashift*), der *recordsize* sowie der Sektorgröße von Festplatten eingegangen.

5.1.1 Physische Blockgröße (ashift)

Die physische Blockgröße entspricht der Größe der Daten, die auf ein *vdev* geschrieben bzw. von ihm gelesen werden. Sie ist immer an ein bestimmtes *vdev* gebunden und wird entweder beim Erstellen eines Pools oder beim Hinzufügen eines *vdevs* gesetzt. Sobald diese an einen Pool gebunden sind, kann die physische Blockgröße nicht mehr verändert werden. Ein Pool kann jedoch aus *vdevs* mit verschiedenen Blockgrößen aufgebaut werden.[28] Aufgrund der Endgültigkeit des Wertes ist es wichtig vor der Erstellung des Pools einige Vorüberlegungen anzustellen.
Bei der Verwendung von Festplatten sollte die Blockgröße immer an die Größe der Sektoren angepasst werden. Zu beachten ist, dass aktuelle Festplatten das Advanced Format (AF) nutzen. Dabei haben die Sektoren statt der ursprünglichen 512 B eine Größe von 4 KiB [29]. Zusätzlich führen diese Festplatten meist eine Emulation von 512 B Sektoren durch, um die Kompatibilität mit älteren Betriebssystemen gewährleisten zu können. Durch die Anpassung an die Sektorgröße können überflüssige Lese- und Schreibzugriffe vermieden und somit eine Steigerung der Geschwindigkeit erreicht werden. Beim Einsatz von Festplatten mit AF sollte die Anpassung nicht an die emulierte, sondern die physische Sektorgröße von 4 KiB erfolgen.
Bei allen anderen Arten von *vdevs* muss diese Vorüberlegung analog durchgeführt werden. So sollte die Blockgröße bei der Verwendung von Dateien als *vdev* an die des darunterliegenden Dateisystems angepasst werden. Wird der Wert beim Anlegen eines Pools nicht explizit angegeben, so versucht ZFS selbstständig die richtige Blockgröße zu ermitteln. Ob diese jedoch bei Festplatten mit 512 B Emulation immer korrekt bestimmt werden kann,

ist fraglich.

Das Setzen der physischen Blockgröße erfolgt mithilfe des Tools *zpool*. Dies kann entweder beim Anlegen eines Pools oder beim Hinzufügen von neuen *vdevs* durch Angabe der Option *ashift* geschehen (siehe Kapitel 2.3.3, S. 8). Die zulässigen Werte reichen von 512 B bis 8 KiB. Um andere Einflüsse während des Benchmarks auszuschließen, wurde ein Pool angelegt, der nur aus einer einzelnen Festplatte besteht. Zur Vermeidung von parallelen Zugriffen, wurde *fio* in diesem Test nur mit einem Job ausgeführt. Um Caching-Effekten zu verhindern, wurden trotzdem 1000 GiB Daten geschrieben. Die für den Test verwendete Festplatte nutzt das AF und arbeitet mit einer physischen Sektorgröße von 4 KiB. In nachfolgendem Diagramm sind die Ergebnisse der Benchmarks zu sehen:

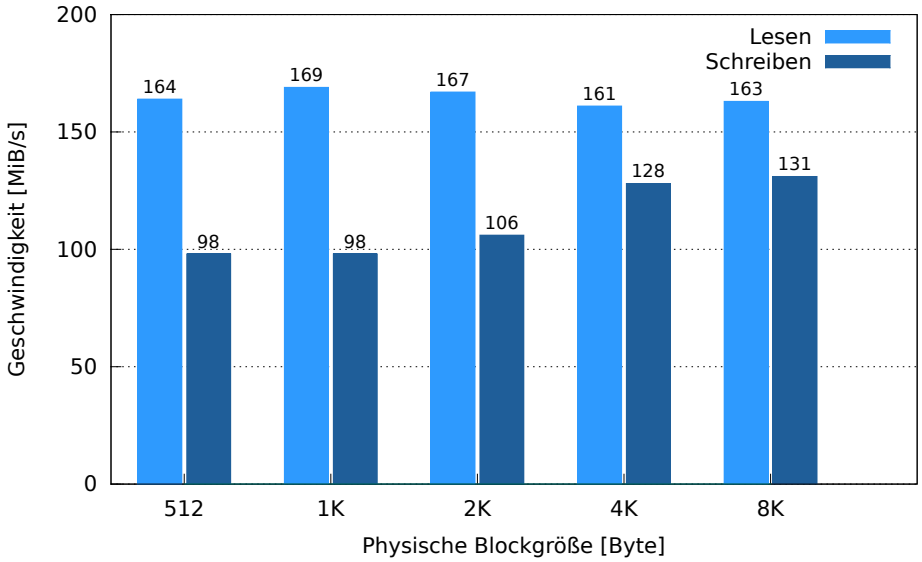

Abbildung 1: Einfluss der physischen Blockgröße

Die Lesegeschwindigkeiten liegen bei allen Blockgrößen etwa auf dem gleichen Niveau. Grund dafür sind die Caching-Funktionalitäten der Festplatten-Controller, die immer den gesamten Sektor, aus dem zuletzt gelesen wurde, zwischenspeichern. Da dieser bei der Verwendung kleiner Blockgrößen nicht mehrfach gelesen werden muss, können ähnliche Geschwindigkeiten erreicht werden. Beim Vergleich der Schreibgeschwindigkeiten ist mit steigender Blockgröße eine Zunahme zu erkennen. Während sie bei einer Blockgröße von 512 B bei 98 MiB/s liegt, steigt sie bei Anpassung an die Sektorgröße der Festplatte (4 KiB) auf 128 MiB/s an. Dieser Effekt entsteht durch das mehrfache Schreiben von Sektoren bei Verwendung zu kleiner Blockgrößen. Der Benchmark zeigt eindeutig, dass eine Anpassung der physischen Blockgröße an die Sektorgröße der Festplatten optimale Geschwindigkeiten gewährleistet. Eine weitere Erhöhung der Größe bringt keine spürbare Geschwindigkeitssteigerung.

Darüber hinaus kann den Ergebnissen die maximal mögliche Lese- bzw. Schreibgeschwindigkeit einer einzelnen Festplatte des Testsystems entnommen werden. Diese liegt bei etwa 160 bzw. 130 MiB/s und kann zum Vergleich mit verschiedenen *zpool*-Konfigurationen herangezogen werden.

5.1.2 Logische Blockgröße (recordsize)

Die logische Blockgröße von Datasets wird als *recordsize* bezeichnet. Sie ist für viele der integrierten Funktionen, wie die Komprimierung, die Berechnung der Prüfsummen und die Deduplizierung relevant. Sie kann unabhängig von der physischen Blockgröße der *vdevs* entweder für einen ganzen Pool oder einzelne Datasets gesetzt werden. Des Weiteren ist der Wert veränderbar und kann jederzeit angepasst werden. Zu interpretieren ist die *recordsize* als eine Obergrenze, nicht als feste Größe. Werden Dateien geschrieben, die kleiner sind, so wird ein kleinerer Block angelegt. Sind die Daten größer, werden sie über mehrere logische Blöcke verteilt. Diese dynamische Größe hat den entscheidenden Vorteil, dass die *recordsize* erhöht werden kann, ohne dass kleine Dateien einen ganzen Block belegen. Auf diese Weise wird der von herkömmlichen Dateisystemen bekannte Overhead vermieden. Sollte der Wert nicht explizit gesetzt werden, wird automatisch eine *recordsize* von 128 KiB verwendet. Der zulässige Wertebereich reicht von 512 B bis 1 MiB.[30]

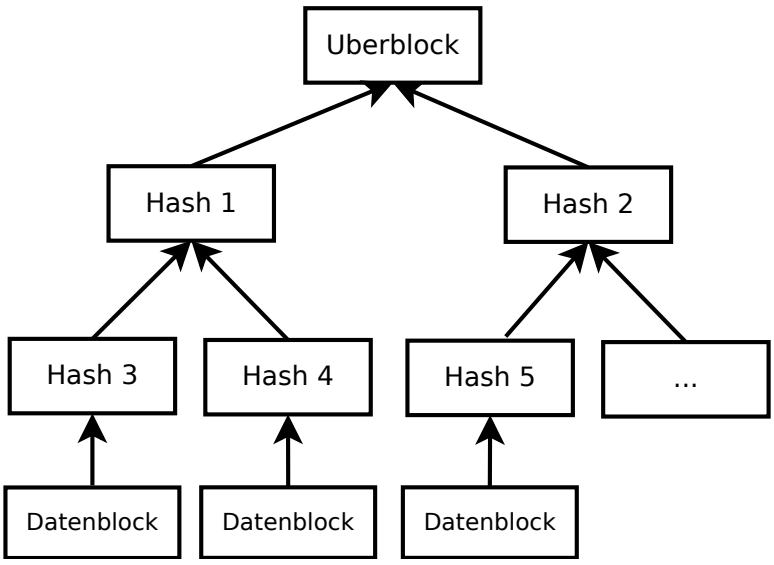

Abbildung 2: Organisation der logischen Blöcke (Merkle Tree)

Zur Sicherung der Datenintegrität speichert ZFS Prüfsummen zu jedem logischen Block ab. Diese werden anschließend durch weitere Prüfsummen abgesichert. Auf diese Weise ergibt sich ein binärer Baum, der vereinfacht in Abbildung 2 dargestellt ist und als Merkle Tree bezeichnet wird [31]. Die Blätter dieses Baumes entsprechen jeweils einem Datenblock, während alle anderen Knoten nur eine Prüfsumme enthalten.

Der Wurzelknoten wird im Falle von ZFS Uberblock genannt und beinhaltet immer die Prüfsumme des gesamten Baumes. Die Bezeichnung leitet sich von dem deutschen Wort „Überblock“ ab. Wird ein logischer Block geschrieben oder verändert, so müssen die dazugehörige Prüfsumme und alle Prüfsummen der darüberliegenden Knoten neu berechnet werden. Somit werden bei jedem Schreibvorgang mehrere Prüfsummen, inklusive der des Uberblocks, berechnet und angepasst. Um auf das Beispiel in Abbildung 2 zurückzukommen, würde Hash 1 über Hash 3 und 4 berechnet werden. Dabei werden die beiden Hashes aneinander gereiht und als Eingabe für den Hashalgorithmus verwendet. Der Baum wird dynamisch erstellt und kann bei hoher Poolbelegung sehr umfangreich werden und viele Ebenen umfassen. Dieser Aufwand kommt der Integrität der Daten zu Gute. Selbst wenn keine Redundanzen bzw. Paritätsinformationen exisitieren, können fehlerhafte Daten zuverlässig erkannt werden. Wird ein Scrub für einen Pool ausgelöst, so wird der Baum von oben nach unten durch Berechnung und Vergleich der Prüfsummen verfiziert.[31]
Eine weitere Besonderheit von ZFS ist die Copy-On-Write (COW) Funktionalität. Werden bereits gespeicherte Daten verändert, so werden sie in einem neuen Block abgelegt, ohne die vorherige Version zu überschreiben. Anschließend werden die entsprechenden Zeiger angepasst. Dies ermöglicht unter anderem das schnelle Anlegen von Snapshots und macht Dateisystemüberprüfungen überflüssig.[31]
Der Einfluss der logischen Blockgröße auf die Geschwindigkeit anderer Funktionen wird in den nachfolgenden Kapiteln untersucht. Um den Einfluss auf die Geschwindigkeit der Standardkonfiguration von ZFS zu prüfen, wurden verschiedene Werte für die *recordsize* getestet. Um vergleichbare Ergebnisse zu erhalten, wurde die von *fio* verwendete Blockgröße jeweils über den Parameter *--bs* an die zu testende angepasst. Da die Ergebnisse mit steigender *recordsize* stark schwanken, musste die Anzahl der Testdurchläufe von drei auf zehn erhöht werden. Der Einfluss auf die Leistung von ZFS kann Abbildung 3 entnommen werden.

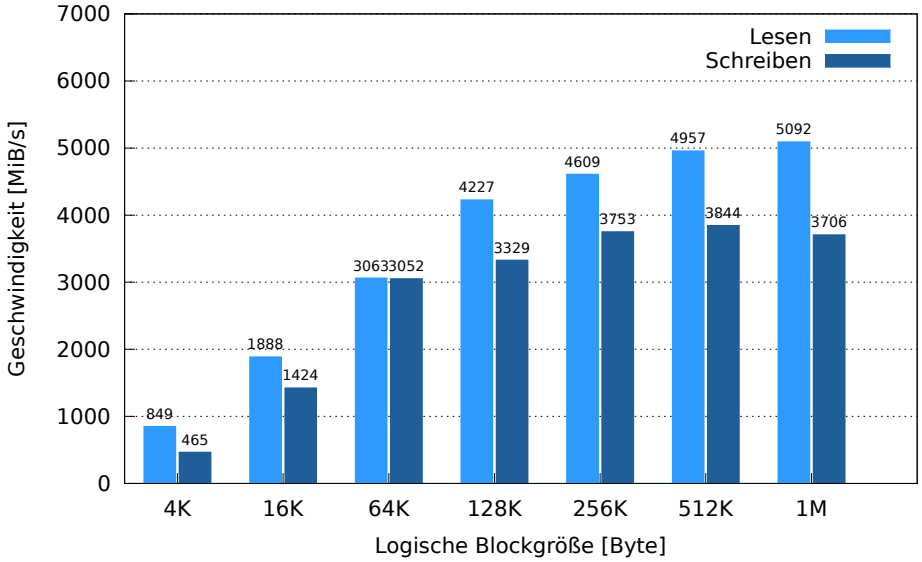

Abbildung 3: Einfluss der logischen Blockgröße

Mit zunehmender logischer Blockgröße werden höhere Lese- und Schreibgeschwindigkeiten erreicht. Nur bei einer Größe von 1 MiB sinkt die Schreibgeschwindigkeit wieder um ca. 100 MiB/s. Bei Nutzung einer *recordsize*, die unterhalb der Standardgröße von 128 KiB liegt, sind größere Geschwindigkeitsverluste festzustellen. Bei einer *recordsize* über 128 KiB kann die Lesegeschwindigkeit um bis zu 20 % und die Schreibgeschwindigkeit um bis zu 15 % gesteigert werden. Optimale Resultate können insbesondere bei einer Größe von 512 KiB bzw. 1 MiB erzielt werden.

Der Grund für die höheren Geschwindigkeiten liegt unter anderem im Merkle Tree. Durch die größeren Blöcke müssen weniger Prüfsummen berechnet und angepasst werden. Darüber hinaus müssen weniger Metadaten zu den Blöcken abgelegt werden. Die Geschwindigkeitsvorteile können jedoch nur beim Schreiben und Lesen von großen Dateien erreicht werden, da für kleinere Dateien auch kleinere Blöcke angelegt werden. Im Falle von kleinen Dateien hat eine große *recordsize* weder einen positiven, noch einen negativen Einfluss auf die Geschwindigkeit. Deshalb kann die Optimierung nicht nur eingesetzt werden, wenn große Dateien auf dem Pool abgelegt werden sollen.

Durch Erhöhung der *recordsize* kann bei Größen von 512 KiB bzw. 1 MiB eine spürbare Steigerung der Geschwindigkeit erreicht werden. Für eine Reduzierung der logischen Blockgröße konnte kein praxisrelevanter Grund gefunden werden. Bei einer großen *recordsize* kann aufgrund der dynamischen Größe gleichzeitig von den Vorteilen kleinerer Blöcke profitiert werden.

5.2 Prüfsummen

Einen weiteren grundlegenden Bestandteil von ZFS stellen die Prüfsummen dar, welche für verschiedene Funktionen benötigt werden. Beim Schreiben werden sie für den jeweiligen logischen Block sowie für alle darüber liegenden Knoten im Merkle Tree (siehe Kapitel 5.1.2, S. 22) berechnet und anschließend in diesem abgespeichert. Bei jedem Lesevorgang verifiziert ZFS die Integrität der Daten anhand der zuvor abgelegten Prüfsummen. Wird eine Integritätskontrolle des Pools (Scrub) durchgeführt, werden alle Prüfsummen des Merkle Trees berechnet und verglichen. Sollten diese nicht übereinstimmen, können die Daten anhand von Paritätsinformationen, soweit vorhanden, wiederhergestellt werden.[32] Bei einem aus 44 Festplatten bestehenden Pool dauerte ein Scrub von 100 TiB Daten etwa fünf Stunden.

Die blockweise Deduplizierung greift ebenfalls auf die berechneten Prüfsummen zurück und nutzt diese zur schnellen Suche von Duplikaten. Da für diesen Zweck komplexere und vor allem kollisionsfreie Prüfsummen benötigt werden, bietet ZFS die Wahl zwischen den Algorithmen Fletcher2, Fletcher4 und SHA256 [18]. Der Algorithmus muss jeweils für den Parameter *checksum* eines Datasets bzw. Pools gesetzt werden. Darüber hinaus können die Prüfsummen auch ohne Angabe eines Algorithmus aktiviert bzw. deaktiviert werden. Der momentan verwendete Standard von ZFS ist Fletcher4. Von der Deaktivierung der Prüfsummenbildung wird strikt abgeraten, da die Integrität der Daten sonst nicht gewährleistet werden kann [33].

Der Fletcher-Algorithmus eignet sich sehr gut, um mit wenig Rechenaufwand Prüfsummen zu berechnen, die zur Erkennung von Bitfehlern genutzt werden können. Die jeweilige Zahl hinter dem Namen des Algorithmus steht für die Länge des Hashes in Bit.[34] Da es sich um keinen kryptologisch sicheren Hash handelt, kann es dazu kommen, dass verschiedene Eingangsdaten zum gleichen Hash führen. Dies ist einerseits auf die Einfachheit des Algorithmus und andererseits auf die kurze Hashlänge von 2 bzw. 4 Bit zurückzuführen. Aus diesem Grund wird beim Einsatz der Deduplizierung immer automatisch der Secure Hash Algorithm 2 (SHA-2) mit 256 Bit langen Hashes verwendet (siehe Kapitel 5.4, S. 35). Dabei handelt es sich um eine kryptologische Hashfunktion, bei der Kollisionen sehr selten auftreten [35]. Da die höhere Sicherheit mit einem wesentlich größeren Rechenaufwand verbunden ist, sollte SHA256 nicht für die reine Fehlererkennung ohne Deduplizierung eingesetzt werden. Diese Empfehlung könnte jedoch bald überholt sein, da neueste CPUs teilweise eine Hardwarebeschleunigung für SHA256 anbieten [36]. Wenn diese von ZFS genutzt wird, ist eine deutliche Geschwindigkeitssteigerung bei der Berech-

nung der Prüfsummen zu erwarten.

In Abbildung 4 sind die Lese- und die Schreibgeschwindigkeit in Abhängigkeit von den verschiedenen Hashalgorithmen dargestellt.

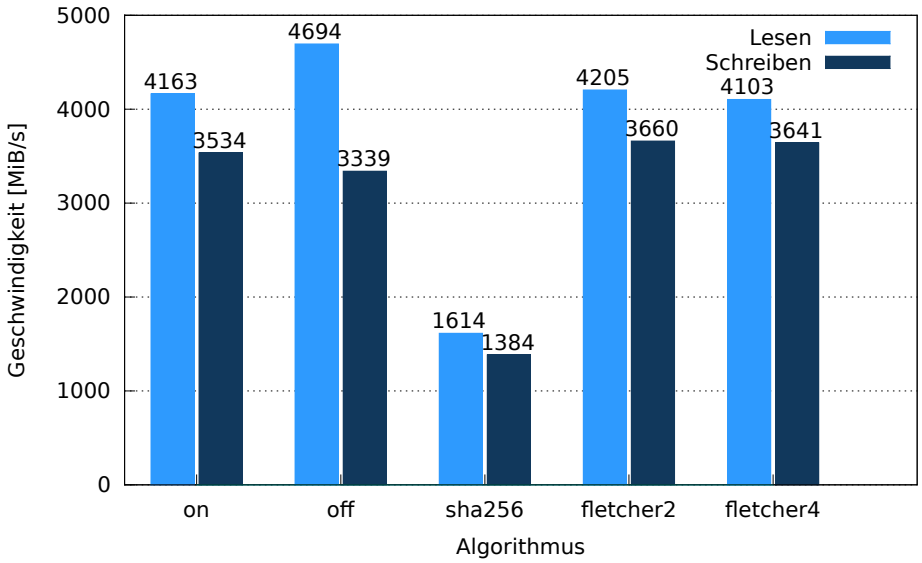

Abbildung 4: Vergleich der Hashalgorithmen hinsichtlich der Geschwindigkeit

Die zwei Versionen des Fletcher-Algorithmus erreichten die maximale Geschwindigkeit des Pools, ohne den Prozessor vollständig auszulasten. Um dennoch einen Vergleich vornehmen zu können, wurde zusätzlich die Auslastung der CPU erfasst (siehe Abbildung 5). Dabei wurde die durch die Wartezeit für die Ein- und Ausgabe verbrauchte Leistung (iowait) herausgerechnet, da diese bei schnelleren Speichermedien nicht entstehen würde und somit nicht durch den Algorithmus verbraucht wurde.

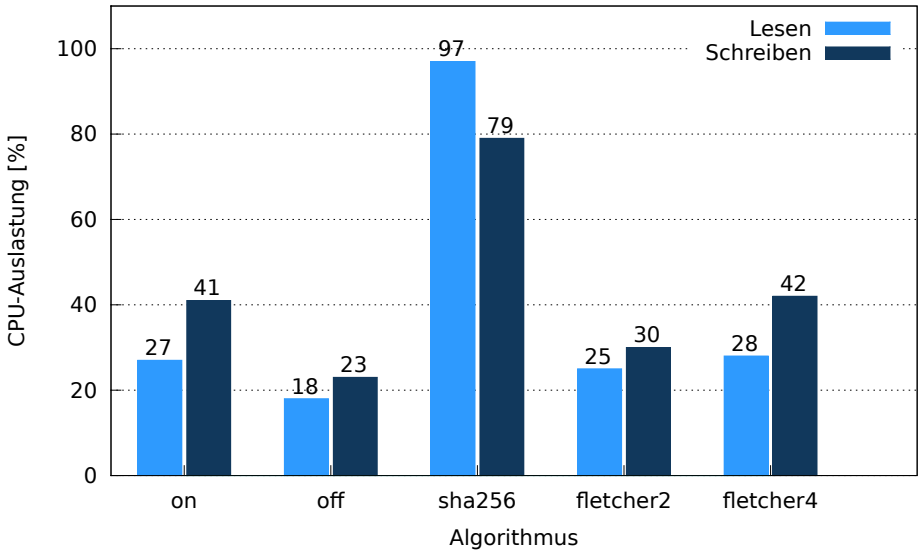

Abbildung 5: Vergleich der Hashalgorithmen hinsichtlich der CPU-Auslastung

Bei der Lese- und Schreibgeschwindigkeit sind keine Unterschiede zwischen Fletcher2 und Fletcher4 erkennbar. Schaut man auf die Auslastung der CPU, so schneidet Fletcher4 mit 42% etwas schlechter, als Fletcher2 mit 30% ab. Anhand der CPU-Auslastung kann auch bestätigt werden, dass es sich bei dem momentan für den Parameter *on* verwendeten Standardalgorithmus um Fletcher4 handelt.
SHA256 erreicht im Vergleich nur reichlich ein Drittel der Schreib- bzw. Lesegeschwindigkeit des Fletcher-Algorithmus. Beim Lesen ist die CPU dabei vollständig und beim Schreiben zu ca. 79% ausgelastet. Durch die geringere Auslastung ist die Schreibgeschwindigkeit etwas niedriger, als die Lesegeschwindigkeit. Die hohe Auslastung der CPU beim Lesen von Daten entsteht durch die Integritätskontrolle von ZFS, bei der die Prüfsummen der Blöcke berechnet und mit den abgespeicherten verglichen werden. Weshalb beim Schreiben mit SHA256 nicht die gesamte CPU-Leistung zur Steigerung der Geschwindigkeit verwendet wird, ist jedoch unklar.
Die Abschaltung der Prüfsummenbildung spart im Vergleich zu Fletcher2 nur etwa ein Drittel der Rechenleistung. Die Schreibgeschwindigkeit kann dadurch nicht erhöht werden und liegt sogar unter der Geschwindigkeit bei aktivierter Prüfsummenbildung. Der einzige spürbare Vorteil ist eine Steigerung der Lesegeschwindigkeit um 13%. Bedenkt man jedoch, dass dadurch die Datenintegrität nicht mehr überprüft werden kann, so wiegen die Vorteile die Nachteile nicht auf.
Die Ergebnisse untermauern die Empfehlung, bei deaktivierter Deduplizierung einen der Fletcher-Algorithmen zu nutzen, um zumindest die Integritätskontrolle zu gewährleisten. Zu SHA256 sollte aufgrund des enormen Rechenaufwands nur im Falle einer Deduplizierung gegriffen werden. Für die reine Fehlererkennung ist dieser Hashalgorithmus bei normalen Anforderungen nicht notwendig. Von der vollständigen Deaktivierung der Prüfsummenbildung ist auch im Zuge einer Geschwindigkeitsoptimierung abzuraten, da die Vorteile im Vergleich zu den Risiken sehr gering bis nicht vorhanden sind.

5.3 Komprimierungsverfahren

Zur effizienteren Nutzung des verfügbaren Speicherplatzes bietet ZFS die Möglichkeit, eine Komprimierung auf Dateisystemebene durchzuführen. Dabei wird jeder logische Block komprimiert, um mehr Daten auf dem vorhandenen Speicherplatz ablegen zu können. Die Komprimierung kann entweder für einen ganzen Pool oder einzelne Datasets aktiviert werden. In der Standardkonfiguration von ZFS ist die Komprimierung nicht aktiviert. Zur Anpassung an verschiedene Anforderungen stehen die Komprimierungsalgorithmen LZJB, GZIP[1-9], ZLE und LZ4 zur Wahl [18]. Darüber hinaus kann die Komprimierung durch Angabe der Werte *on* oder *off* ein- bzw. ausgeschaltet werden. Wird die Komprimierung ohne Angabe eines Algorithmus aktiviert, so wird der aktuelle Standardalgorithmus von ZFS verwendet. Die entsprechenden Werte müssen für den Parameter *compression* eines Pools oder Datasets gesetzt werden.

Bei GZIP handelt es sich in Linux-Umgebungen um einen der populärsten Komprimierungsalgorithmen. Er baut auf den Deflate-Algorithmus auf und ist für nahezu jedes Betriebssystem frei verfügbar. Er erreicht sehr gute Komprimierungsverhältnisse und kann unter Verwendung verschiedener Komprimierungsstufen eingesetzt werden. Diese reichen von 1 bis 9, wobei 9 das beste Komprimierungsverhältnis und 1 die höchste Komprimierungsgeschwindigkeit bei schlechtestem Komprimierungsverhältnis bietet. Exemplarisch untersucht werden die Stufen 1, 6 und 9.[37]

Sowohl ZLE als auch LZJB sind Algorithmen, die speziell für ZFS implementiert wurden. In einem anderem Kontext finden sie kaum oder keine Verwendung. Bei ZLE handelt es sich um einen Algorithmus, der ausschließlich Folgen von Nullen sucht und ersetzt. Dadurch benötigt er kaum Rechenleistung, erzielt aber infolge dessen nur selten hohe Komprimierungsverhältnisse. Aus der Funktionsweise leitet sich der Name „Zero Length Encoding“ (ZLE) ab [28]. LZJB ist universeller einsetzbar und ermöglicht bessere Komprimierungsverhältnisse als ZLE. Der Name steht für „Lempel Ziv Jeff Bonwick“ und ergibt sich aus der Familie von Komprimierungsalgorithmen, zu der er gehört (Lempel Ziv) und dem Namen seines Entwicklers (Jeff Bonwick).[38]

Ursprünglich wurde LZJB von ZFS als Standardalgorithmus eingesetzt. In neueren Versionen wurde LZ4 eingeführt und zum neuen Standard erklärt. Die Gründe für diese Entscheidung werden sich im Laufe der Benchmarks zeigen. LZ4 ist ein für hohe Geschwindigkeiten optimierter Komprimierungsalgorithmus, der gleichzeitig gute Komprimierungsverhältnisse erreichen soll. Aus diesem Grund wird er in der Regel für On-the-fly-Komprimierung eingesetzt.[39] Ein besonderes Feature ist die Analyse des voraussichtlichen Komprimie-

rungsverhältnisses durch die Komprimierung des ersten logischen Blocks einer Datei. Sollte das Verhältnis unter 1,125 liegen, werden die Daten unkomprimiert abgespeichert [40]. Auf diese Weise kann insbesondere auf leistungsschwachen Systemen Rechenleistung gespart werden.
Um die verschiedenen Algorithmen und deren Eignung für unterschiedliche Einsatzwecke beurteilen zu können, wird sowohl die Geschwindigkeit als auch das Komprimierungsverhältnis untersucht.

5.3.1 Geschwindigkeit

Da alle Benchmarks mit nicht komprimierbaren Daten durchgeführt wurden, konnte bei den meisten Algorithmen das Worst-Case-Szenario gemessen werden. Da LZ4 die Komprimierung bei solchen Daten abbricht, konnte in diesem Fall nur die reine Schreibgeschwindigkeit des Pools gemessen werden. Aus diesem Grund wurde der Test mit LZ4 zusätzlich ohne die Option *--refill_buffers*, also mit komprimierbaren Daten durchgeführt. Dieser zusätzliche Benchmark ist durch die Abkürzung „komp.“ im nachfolgenden Diagramm kenntlich gemacht:

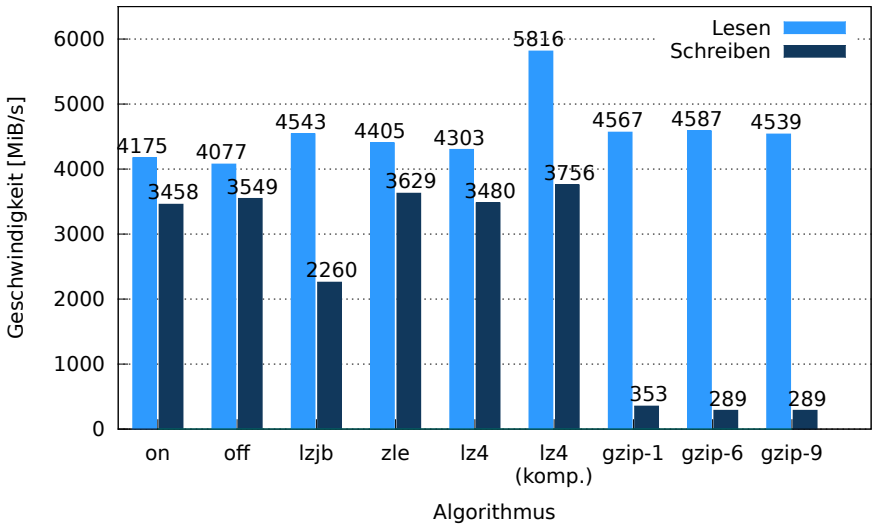

Abbildung 6: Vergleich der Komprimierungsgeschwindigkeit

Bei der Verwendung nicht komprimierbarer Daten erreichen alle Algorithmen etwa die gleiche Lesegeschwindigkeit. Diese liegt sogar leicht über der des Pools ohne Komprimierung. Da das Komprimierungsverhältnis stets bei 1,0 liegt, kann diese Erhöhung jedoch nicht durch die Komprimierung der Daten erklärt werden. Dadurch muss auch bei aktivierter Komprimierung die gleiche Datenmenge von den Festplatten gelesen werden. Beim Lesen betrug die CPU-Auslastung bei allen Algorithmen rund 50 %. Im Vergleich zur Auslastung von 30 % bei deaktivierter Komprimierung, hält sich der zusätzliche Rechenaufwand

in Grenzen.

Unter Verwendung von komprimierbaren Daten kann mit LZ4 eine wesentlich höhere Lesegeschwindigkeit erreicht werden. Diese lässt sich durch die Komprimierung der Daten mit dem Faktor 2 erklären. Auf diese Weise muss nur die Hälfte der Daten von den Festplatten gelesen werden. Dennoch kann nicht die doppelte Geschwindigkeit erreicht werden, da bei diesem Benchmark die CPU vollständig ausgelastet war und dadurch die Geschwindigkeit begrenzte. Die hohe Auslastung wird sowohl durch die Dekomprimierung als auch die Bildung der Prüfsummen verursacht.

Sowohl LZ4 als auch ZLE schöpfen die volle Schreibgeschwindigkeit des Pools aus. Bei ZLE ist dies durch die Funktionsweise des Algorithmus bedingt, der mit wenig Rechenaufwand nach Folgen von Nullen sucht und diese ersetzt. Bei LZ4 sorgt der frühzeitige Abbruch bei nicht komprimierbaren Daten für dieses Verhalten. Durch die Verwendung von komprimierbaren Daten kann der Algorithmus eine leichte Erhöhung der Schreibgeschwindigkeit erreichen. Sowohl bei LZ4 (komp.), LZJB als auch GZIP wurde die Komprimierungsgeschwindigkeit jeweils durch die CPU begrenzt. LZJB schafft im Vergleich zur deaktivierten Komprimierung nur ca. zwei Drittel der Geschwindigkeit und liegt damit im Mittelfeld der Testergebnisse. Hinsichtlich der Komprimierungsgeschwindigkeit schneidet GZIP mit Abstand am schlechtesten ab. Unter Verwendung der Komprimierungsstufe 1 wird nur noch rund ein Zehntel der Schreibgeschwindigkeit eines Pools mit deaktivierter Komprimierung erreicht. Im Vergleich dazu sind die Stufen 6 und 9 nochmals langsamer, während zwischen den beiden kein Unterschied festgestellt werden kann. Aufgrund der Parallelisierung können die Schreibgeschwindigkeiten mithilfe einer CPU mit höherer Taktfrequenz oder mehr Kernen erhöht werden. Im Hinblick auf die Geschwindigkeit konnte nachgewiesen werden, dass der von ZFS standardmäßig (bei Angabe von *on*) eingesetzte Komprimierungsalgorithmus LZ4 ist.

Da jeder logische Block einzeln komprimiert wird, wurden zusätzlich Benchmarks zur Ermittlung des Zusammenhangs zwischen Geschwindigkeit und *recordsize* durchgeführt. Diese wurden exemplarisch mit GZIP-9 für nicht komprimierbare Daten durchgeführt, um das Verhalten bei möglichst hohen Anforderungen an die Leistungsfähigkeit des Systems zu untersuchen. Die von *fio* verwendete Blockgröße wurde jeweils über den Parameter *--bs* an die zu testende *recordsize* angepasst. Die Ergebnisse können Abbildung 7 entnommen werden.

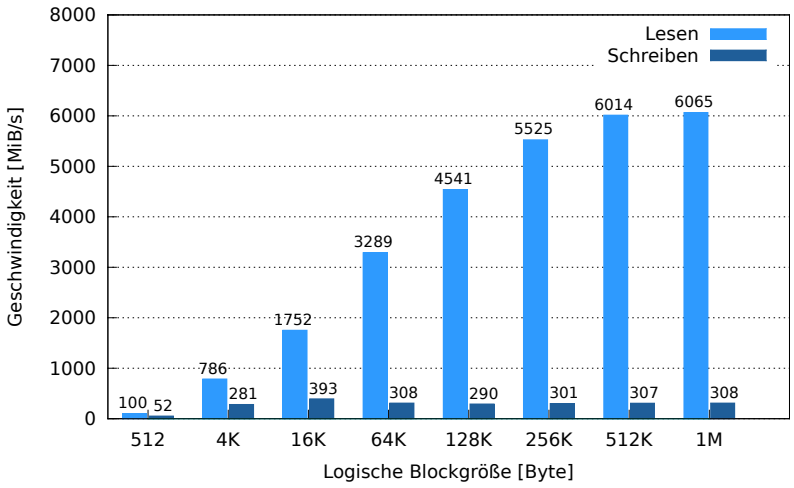

Abbildung 7: Abhängigkeit der Geschwindigkeit von GZIP-9 von der *recordsize*

Die Lesegeschwindigkeiten zeigen das gleiche Verhalten, wie bereits bei der Untersuchung des Einflusses der *recordsize* (Kapitel 5.1.2, S. 22) ohne Komprimierung festgestellt wurde. Auch bei diesem Benchmark sind sie, wie schon beim Vergleich der Algorithmen, wieder etwas höher als die eines Pools mit deaktivierter Komprimierung. Die beste Komprimierungsgeschwindigkeit kann bei einer Blockgröße von 16 KiB erreicht werden. Bei dieser Größe ist die Leserate hingegen relativ gering. Bei einer weiteren Erhöhung der *recordsize* sinkt sie auf etwa 300 MiB/s und bliebt anschließend nahezu konstant. Nur bei einer Größe von 512 B sind die Ergebnisse wesentlich schlechter.
Die Empfehlung aus Kapitel 5.1.2, S. 22, eine *recordsize* von 512 KiB oder 1 MiB einzusetzen, kann auf die Optimierung der Lese- und Schreibgeschwindigkeiten beim Einsatz von Komprimierung übertragen werden.

5.3.2 Komprimierngsverhältnis

Zum Vergleich der Komprimierungsverhältnisse der verschiedenen Algorithmen wurden zwei unterschiedlich gut komprimierbare Datensätze verwendet. Als Beispiel für gut komprimierbare Daten dienen die ungepackten Quellen des Linux-Kernels in Version 3.18.30[41]. Diese bestehen aus vielen kleinen Textdateien, die ein gutes Komprimierungsverhalten zeigen. Eine Reihe von unkomprimierten MRT-Aufnahmen mit einer Gesamtgröße von 25 GiB wurde als Beispiel für schwer komprimierbare Daten gewählt. Dabei handelt es sich um Daten, die später im produktiven Einsatz auf dem System gespeichert werden sollen.
Zur Ermittlung des Komprimierungsverhältnisses wurde der jeweilige Datensatz in einen leeren Pool kopiert. Mithilfe des Tools *zfs* wurde anschließend der Wert für das Attribut *refcompressratio* ausgelesen. Dieser spiegelt das Verhältnis zwischen der Größe der zu

speichernden Daten und dem tatsächlich für die komprimierten Daten benötigten Speicherplatz wider. Bei einem Komprimierungsverhältnis von 1 wird keine Speicherersparnis erzielt. Je größer der Wert ist, desto stärker können die Daten komprimiert werden.

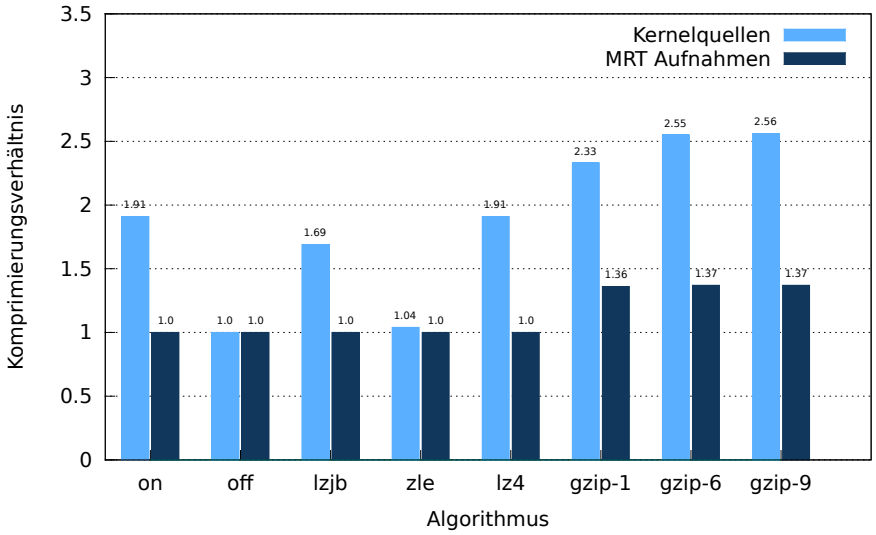

Abbildung 8: Vergleich der Komprimierungsverhältnisse

In Abbildung 8 sind die ermittelten Komprimierungsverhältnisse dargestellt. ZLE liefert bei beiden Datensätzen die schlechtesten Ergebnisse. Durch die sehr spezielle Arbeitsweise dieses Algorithmus kann er nur in den seltensten Fällen hohe Komprimierungsraten erreichen. Betrachtet man die MRT-Aufnahmen, so können diese ausschließlich von GZIP komprimiert werden. LZ4 bricht infolge der Komprimierbarkeitsanalyse ab, während LZJB trotz Komprimierung keine Speicherersparnis erzielen kann. Da LZ4 eine unnötige Belastung der CPU vermeidet, ist es für solche Fälle am Besten geeignet. Mit GZIP kann für die MRT-Aufnahmen ein gutes Ergebnis erzielt werden.

Die Kernelquellen können mit jedem Algorithmus komprimiert werden, wobei der Unterschied zur unkomprimierten Größe bei ZLE sehr gering ist. Auch für diesen Datensatz erzielt GZIP die besten Werte, gefolgt von LZ4, welches im Vergleich zu LZJB ebenfalls ein besseres Ergebnis erreicht. Während GZIP mit der Komprimierungsstufe 6 eine stärkere Komprimierung als mit der Stufe 1 erzielt, kann zwischen den Stufen 6 und 9 kaum ein Unterschied festgestellt werden. Beim Vergleich der Komprimierungsverhältnisse wird erneut deutlich, dass es sich beim Standardalgorithmus für die Komprimierung um LZ4 handelt.

Da jeder logische Block einzeln komprimiert wird und Muster nur innerhalb eines Blockes gesucht und ersetzt werden können, wurde zusätzlich die Abhängigkeit des Komprimierungsverhältnisses von der logischen Blockgröße (siehe Kapitel 5.1.2, S. 22) untersucht.

Die Messungen wurden wieder exemplarisch für GZIP-9 durchgeführt, um eventuelle Einflüsse möglichst deutlich sichtbar zu machen. Die erreichten Komprimierungsraten können Abbildung 9 entnommen werden.

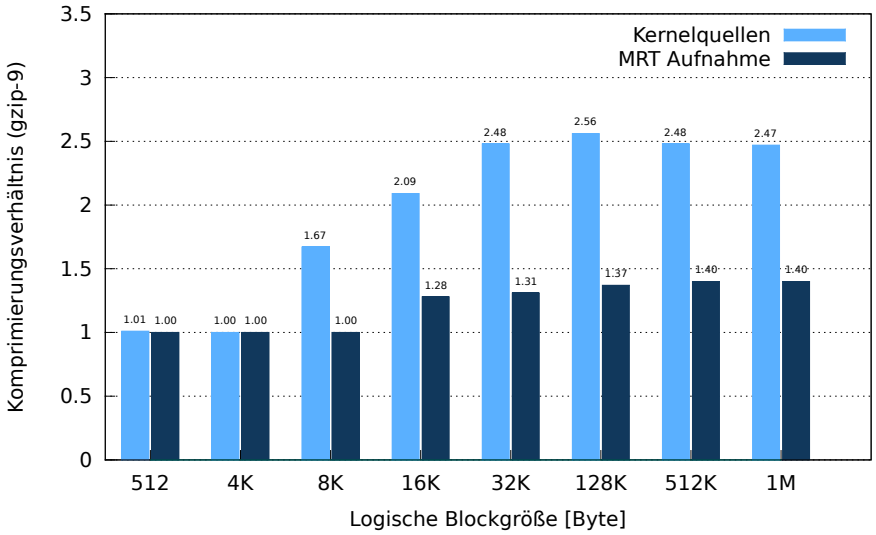

Abbildung 9: Abhängigkeit des Komprimierngsverhältnisses von GZIP-9 von der *recordsize*

Eine Komprimierung der MRT-Aufnahmen ist ab einer *recordsize* von 16 KiB möglich, währenddessen die Kernelquellen schon ab einer Größe von 8 KiB komprimiert werden können. Unterhalb des genannten Bereichs ist keine Speicherersparnis möglich. Bis zu einer Größe von 128 KiB können mit steigender *recordsize* höhere Komprimierungsverhältnisse erzielt werden. Bei einer weiteren Erhöhung bis 1 MiB treten keine Veränderungen mehr auf. Da das Optimum bereits bei der standardmäßig eingesetzten Größe von 128 KiB erreicht ist, kann selbst durch die Anpassung der *recordsize* keine stärkere Komprimierung mehr erreicht werden. Sollte eine Verringerung der logischen Blockgröße notwendig sein, sollten jedoch auch die negativen Auswirkungen berücksichtigt werden.
Durch Komprimierung der gleichen Datensätze mit dem Linux-Tool *gzip* wurden dieselben Werte wie bei der durch ZFS auf Dateisystemebene durchgeführten Komprimierung erreicht. Noch höhere Komprimierungsverhältnisse können nur durch vorheriges Zusammenfassen der Dateien zu einem Tarball erreicht werden. Im *zpool* würde dementsprechend nicht für jede kleine Datei ein einzelner logischer Block angelegt werden, vielmehr erfolgt eine Verteilung des Tarballs über mehrere große Blöcke. Da die Blöcke, in denen Muster gesucht und ersetzt werden können dadurch größer wären, könnte die Komprimierung effizienter durchgeführt werden. Weitere Informationen zur dynamischen Größe logischer Blöcke können Kapitel 5.1.2, S. 22 entnommen werden.

5.3.3 Bewertung

LZ4 kann im Vergleich zu LZJB sowohl in Sachen Komprimierungsgeschwindigkeit, als auch mit dem Komprimierungsverhältnis überzeugen. Aus diesem Grund kann kein praxisrelevantes Einsatzszenario für LZJB genannt werden. ZLE kann ebenfalls nur in sehr spezifischen Fällen gewinnbringend eingesetzt werden, da lange Folgen von Nullen nur bei den wenigsten Datensätzen vorkommen. Da GZIP aufgrund der starken CPU-Anforderungen für viele Einsatzbereiche ungeeignet ist, kann die Entscheidung der ZFS-Entwickler LZ4 als Standardalgorithmus einzusetzen nachvollzogen werden.
Liegt der Fokus bei der Komprimierung auf einem möglichst geringen Geschwindigkeitsverlust, so sollte LZ4 zum Einsatz kommen. Es beeinflusste im Benchmark weder die Lese- noch die Schreibgeschwindigkeit und erreichte darüber hinaus gute Komprimierungsverhältnisse. Das Feature des frühzeitigen Abbruchs bringt sowohl Vor- als auch Nachteile mit sich. Wird eine schwache CPU eingesetzt, kann auf diese Weise wertvolle Rechenzeit gespart werden. Bei schwer komprimierbaren Daten (z. B. mit Faktor 1,1) kann der frühzeitige Abbruch unter Umständen zugunsten eines geringeren Speicherverbrauchs ungewollt sein. Vor allem da die Komprimierung je nach Poolgröße und Leistungsfähigkeit der Hardware nicht die Geschwindigkeit beeinflusst.
GZIP kann eingesetzt werden, wenn eine effiziente Speichernutzung im Mittelpunkt steht. Der Algorithmus kann durch die bekannt guten Komprimierungsverhältnisse überzeugen und eignet sich deshalb optimal für diesen Einsatz. Die hohen Anforderungen an die CPU sorgen jedoch auch bei nicht komprimierbaren Daten für geringe Schreibgeschwindigkeiten, da derselbe Rechenaufwand betrieben werden muss. Dafür kann auch bei schwer komprimierbaren Daten, wie den MRT-Aufnahmen, eine gute Komprimierungsrate erreicht werden.
Alles in allem bietet ZFS für verschiedene Anforderungen die passenden Algorithmen. Da für jedes Dataset separat ein Algorithmus gewählt werden kann, können auch Teile eines Pools an verschiedene Anforderungen angepasst werden. Da LZ4 verhältnismäßig wenig Leistung in Anspruch nimmt, den Speicherverbrauch jedoch spürbar verringern kann, sollte es für jeden Pool eingesetzt werden, insofern keine anderen Argumente dagegen sprechen. Darüber hinaus kann auf diese Weise die Leserate leicht gesteigert werden. Hinsichtlich der Geschwindigkeit sollte die *recordsize* entsprechend der Empfehlungen in Kapitel 5.1.2, S. 22 auf 512 KiB oder 1 MiB gesetzt werden. Dies gewährleistet zudem ein gutes Komprimierungsverhältnis.

5.4 Deduplizierung

Neben der Komprimierung bietet ZFS mit der blockweisen Deduplizierung eine weitere Möglichkeit zur effizienten Speichernutzung. Sie arbeitet auf Ebene der logischen Blöcke und vermeidet, dass Duplikate, d. h. identische Blöcke mehrfach gespeichert werden. Insbesondere für Pools, in denen mehrfach dieselben Daten oder Dateien mit sich wiederholenden Mustern abgelegt sind, kann der Speicherverbrauch dadurch stark reduziert werden. Die Erkennung redundanter Blöcke wird mithilfe der in Kapitel 5.2, S. 25 beschriebenen Prüfsummen durchgeführt. Da Fletcher2 und Fletcher4 aufgrund ihrer geringen Kollisionsfreiheit für diesen Zweck ungeeignet sind, kommt bei aktivierter Deduplizierung immer SHA256 zum Einsatz [42].

Zum Vergleich der Prüfsummen neuer Blöcke mit denen aller bereits existierenden, wird im Arbeitsspeicher eine sogenannte Dedup Table (DDT) aufgebaut. Diese enthält alle Prüfsummen der bereits geschriebenen Blöcke und kann je nach Größe und Belegung des Pools sehr viel Speicher in Anspruch nehmen. Sollte sie nicht mehr in den vorhandenen Arbeitsspeicher passen und somit teilweise auf eine Festplatte ausgelagert werden müssen, ist aufgrund der langsameren Zugriffszeiten mit enormen Geschwindigkeitseinbußen zu rechnen. In vielen Quellen sind deshalb Empfehlungen von bis zu 5 GB freiem Arbeitsspeicher pro TB Poolgröße zu finden.[43]

Sogar die 256 GB RAM des Testsystems reichen nicht aus, um diesen Anforderungen für den ca. 200 TB großen Pool gerecht zu werden. Um die Deduplizierung dennoch gewinnbringend einsetzen zu können, kann sie für jedes Dataset einzeln und nur für geeignete Daten aktiviert werden. Aufgrund des hohen Arbeitsspeicherbedarfs sollte vor der Aktivierung eine Simulation, wie in Kaptitel 3.3, S. 14 beschrieben, durchgeführt werden. Anhand der erzielten Ergebnisse kann anschließend der Nutzen mit den zu erwartenden Kosten bzw. Geschwindigkeitsverlusten verglichen werden. Sollte die Deduplizierung trotz eines zu kleinen Arbeitsspeichers verwendet werden, kann durch den Einsatz einer SSD als Cache eine starke Beschleunigung erzielt werden (siehe Kapitel 5.6.1, S. 42).

Für Daten, die gleichzeitig gut komprimierbar und deduplizierbar sind, ist auch eine Kombination beider Funktionen möglich. Dabei wird jeder logische Block zuerst komprimiert und anschließend dedupliziert. Da bei der Komprimierung identischer Daten immer das gleiche Komprimat entsteht, summieren sich die Effekte der Funktionen.

Die Aktivierung der Deduplizierung erfolgt über Anpassung des Parameters *dedup* eines Pools oder Datasets. Zur Auswahl stehen folgende Optionen [18]:

- **off** - Deduplizierung ist deaktiviert
- **on** - Deduplizierung ist aktiviert
- **verify** - der Inhalt logischer Blöcke mit identischen Prüfsummen wird zur Vermeidung von Fehlern zusätzlich byteweise verglichen
- **sha256** - Deduplizierung wird basierend auf SHA256-Prüfsummen durchgeführt
- **sha256,verify** - Deduplizierung wird basierend auf SHA256-Prüfsummen durchgeführt und der Inhalt der logischen Blöcke wird verglichen

Da bei aktivierter Deduplizierung immer SHA256 zum Einsatz kommt und alle anderen Prüfsummenparameter überschrieben werden, sind einige der Optionen theoretisch nicht von Nöten. Sowohl *on* und *sha256*, als auch *verify* und *sha256,verify* bewirken jeweils das gleiche Verhalten [42]. Der Vollständigkeit halber wurde der Benchmark dennoch für alle zuvor genannten Einstellungen durchgeführt. Abbildung 10 können die Ergebnisse für schlecht deduplizierbare Daten entnommen werden.

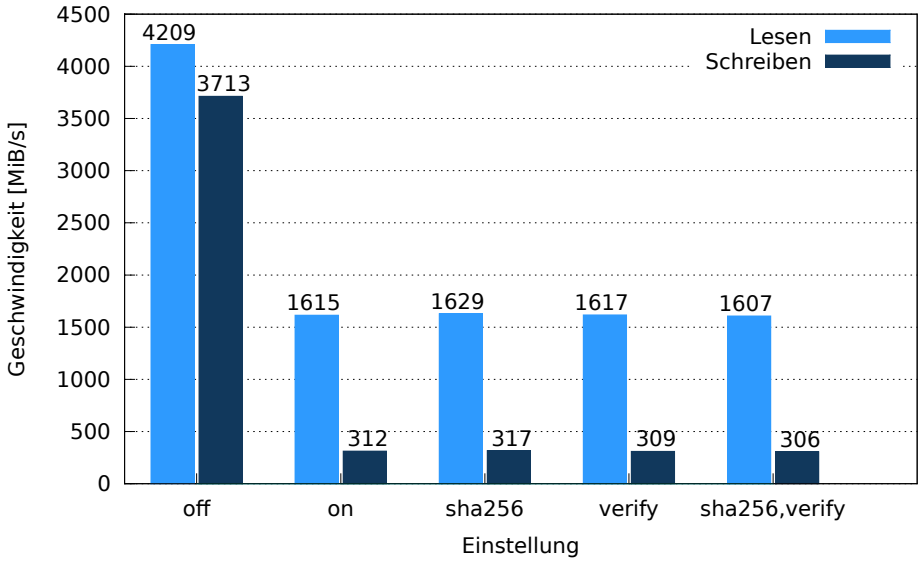

Abbildung 10: Geschwindigkeit bei Deduplizierung mit Faktor 1,5

Bei allen vier Einstellungen konnte jeweils ein Deduplizierungsverhältnis von 1,5 erreicht werden. Untereinander können kaum Leistungsunterschiede festgestellt werden. Im Vergleich zur Standardkonfiguration sinkt die Schreibgeschwindigkeit bei aktivierter Deduplizierung von rund 3700 MiB/s auf 300 MiB/s. Weil die CPU beim Schreiben immer nur

kurzzeitig durch die Berechnung der SHA256-Prüfsummen ausgelastet war, muss die Geschwindigkeit durch die Latenzen bei der Suche nach Duplikaten begrenzt worden sein. Auch beim Lesen der Daten bricht die Geschwindigkeit deutlich ein. Da das gleiche Verhalten bereits bei der Betrachtung der Prüfsummen in Kapitel 5.2, S. 25 festgestellt wurde, können die 1600 MiB/s durch den Einsatz des SHA256-Algortihmus erklärt werden. Auch in diesem Fall kann durch aktuelle CPUs mit Hardwarebeschleunigung für SHA256 eine Steigerung der Geschwindigkeit erreicht werden.
Aufgrund des hohen Arbeitsspeicherbedarfs wird die Deduplizierung meist nur für Datasets mit geeigneten Inhalten verwendet. Deshalb wurde zusätzlich eine Messung mit gut deduplizierbaren Daten (Faktor 11) durchgeführt. Im Gegensatz zum vorherigen Test wurde *fio* dabei ohne den Parameter *--refill_buffers* aufgerufen. Die Ergebnisse können folgender Abbildung entnommen werden:

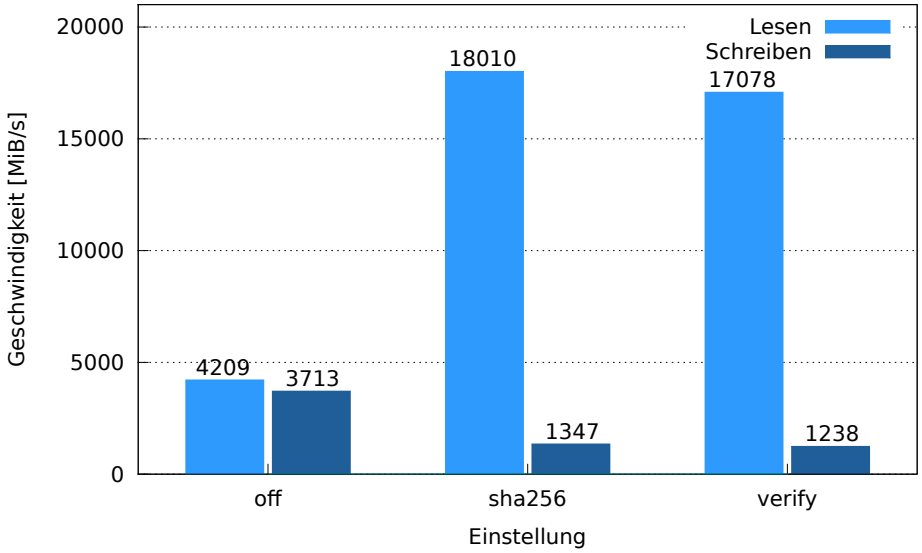

Abbildung 11: Geschwindigkeit bei Deduplizierung mit Faktor 11

Im Vergleich zur deaktivierten Deduplizierung sinkt die Schreibgeschwindigkeit bei dieser Art von Daten nur auf etwa ein Drittel. Darüber hinaus sind erstmals Unterschiede zwischen der reinen Verwendung von Prüfsummen und der byteweisen Verifikation der logischen Blöcke erkennbar. Da das Verifizieren der Blöcke nur 8 % langsamer ist, kann die Funktion zur Vermeidung von Fehlern ohne große Geschwindigkeitsverluste aktiviert werden. Der Anstieg der Schreibgeschwindigkeit tritt auf, da die DDT bei deduplizierbaren Blöcken immer nur bis zu der entsprechenden Prüfsumme und nicht vollständig durchsucht werden muss.

Da der Arbeitsspeicher des Testsystems relativ groß ist, können beim Lesen viele der dedupliziertenBlöcke im Cache gehalten werden (siehe Kapitel 5.6.1, S. 42). Aufgrunddessen müssen wesentlich weniger Daten gelesen und weniger Prüfsummen zur Integritätskontrolle gebildet werden. Deshalb ist die Lesegeschwindigkeit im Vergleich zur Standardkonfiguration ca. viermal höher. Diese hängt jedoch sehr stark von der Größe des Arbeitsspeichers ab. Die Steigerung tritt nur auf, wenn deduplizierte Blöcke mehrfach hintereinander oder gleichzeitig gelesen werden. Auch die Leserate ist bei aktivierter Verifikation der Blöcke etwas niedriger.

Aufgrund der Latenzen beim Prüfsummenvergleich ist bei steigender Auslastung des Pools und einer größeren DDT mit noch niedrigeren Schreibgeschwindigkeiten zu rechnen. Aus diesem Grund und aufgrund des hohen Arbeitsspeicherbedarfs sollte die Deduplizierung nur für Datasets aktiviert werden, bei denen eine starke Speicherersparnis zu erwarten ist. Für alle anderen Fälle bringt der Einsatz von Komprimierung meist mehr Vorteile mit sich. Letztendlich müssen die Vor- und Nachteile für jeden Anwendungsfall einzeln abgewogen werden.

5.5 Poolkonfiguration

Um verschiedensten Anforderungen gerecht zu werden, bietet ZFS die Möglichkeit Pools unterschiedlich zu strukturieren. Diese Konfigurationen unterscheiden sich hauptsächlich in der Art und Anzahl der abgespeicherten Paritätsinformationen, die letztendlich eine höhere Verfügbarkeit der gespeicherten Daten gewährleisten. Die Strukturierung kann durch Angabe bestimmter Schlüsselwörter beim Erstellen eines Pools vorgenommen werden (siehe Kapitel 2.3.3, S. 8).
Sollen Festplattenausfälle kompensiert werden können, müssen diese entweder gespiegelt sein oder zu einem RAID-Z gehören. Der Verlust an nutzbarem Speicher hängt dabei von dem gewählten RAID-Z-Level und dessen Größe bzw. der Anzahl der gespiegelten *vdevs* ab. Darüber hinaus variieren auch die Anforderungen an die CPU und somit die Lese- und Schreibgeschwindigkeiten. Um einen Überblick über die Leistungsfähigkeit verschiedener Poolkonfigurationen zu erhalten, werden die drei verfügbaren RAID-Z-Level und ein Pool aus Spiegelpaaren mit einem Pool ohne Redundanzen verglichen. Nachfolgend sind die getesteten Konfigurationen aufgelistet:

- **Striping:** 20 Festplatten ohne Paritäten
- **Spiegelpaare:** 20+20 Festplatten paarweise gespiegelt; Spiegelpaare sind durch Striping verbunden
- **RAID-Z1:** 20+1 Festplatten im RAID-Z1-Verbund
- **RAID-Z2:** 20+2 Festplatten im RAID-Z2-Verbund
- **RAID-Z3:** 20+3 Festplatten im RAID-Z3-Verbund

Für eine bessere Vergleichbarkeit wurden die Pools so strukturiert, dass der nutzbare Speicher jeweils identisch ist. Die Ergebnisse des Benchmarks sind in Abbildung 12 zu sehen.
Die drei RAID-Z Konfigurationen zeigen mit zunehmender Anzahl an Paritätsinformationen eine schlechtere Schreibgeschwindigkeit. Der begrenzende Faktor war bei allen Varianten die CPU-Geschwindigkeit. Eine Ursache für die verhältnismäßig schlechte Schreibgeschwindigkeit ist die Größe der erstellten RAID-Zs. Empfehlungen zufolge sollte ein RAID-Z aus maximal sieben bis acht *vdevs* bestehen, um optimale Geschwindigkeiten zu erreichen [44]. In Kapitel 7.1, S. 55 sind Ergebnisse von weiteren Tests mit verschiedenen RAID-Z Konfigurationen zu finden, die die Unterschiede zwischen Pools mit einem

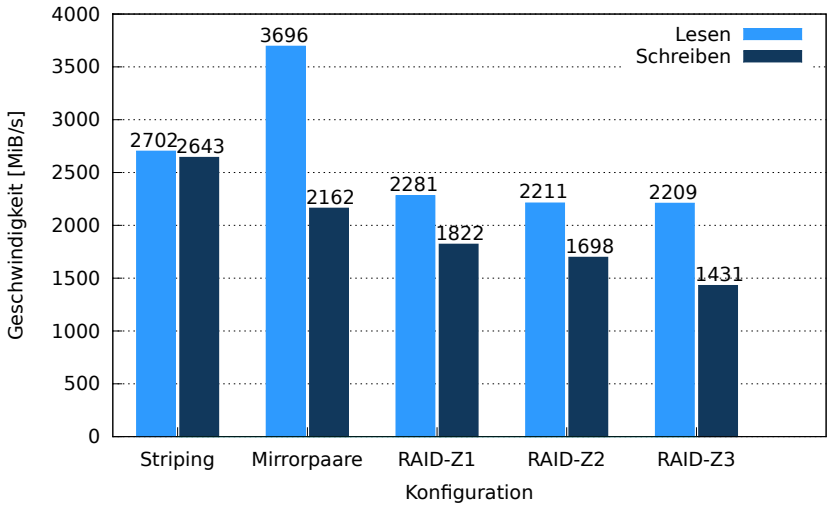

Abbildung 12: Abhängigkeit der Geschwindigkeit von der Poolkonfiguration

großen RAID-Z bzw. mehreren kleinen RAID-Zs zeigen. Da die Integrität der Daten ausschließlich anhand der Prüfsummen bestätigt wird, liegen die Lesegeschwindigkeiten der verschiedenen RAID-Z-Level auf demselben Niveau. Die Paritäten werden beim Lesen nur im Falle einer fehlerhaften Prüfsumme verwendet. Die dennoch etwas niedrigere Leserate, als beim reinen Striping, ist der dynamischen Breite der Stripes eines RAID-Z geschuldet, wodurch deren exakte Position immer erst ermittelt werden muss (siehe Kapitel 6.1, S. 49).

Die Schreibgeschwindigkeit der Spiegelpaare ist etwas niedriger, als die des Pools ohne Paritäten. Dies liegt vermutlich einerseits daran, dass immer maximal die Geschwindigkeit der langsamsten Festplatte eines Paares erreicht werden kann und andererseits an dem Overhead, der durch die doppelte Abspeicherung entsteht. Da alle Daten redundant auf zwei verschiedenen Festplatten abgelegt werden, kann im Vergleich zum Pool ohne Paritäten eine 37 % höhere Lesegeschwindigkeit erzielt werden. Diese Erhöhung wird erreicht, indem später benötigte Daten vorausschauend vom Spiegelpartner gelesen werden. Da die Daten gleichmäßig über die Spiegelpaare verteilt werden und das vorausschauende Lesen damit nicht ganz trivial ist, wurde nicht die theoretisch mögliche Verdopplung der Geschwindigkeit erreicht.

Ob ein linearer Zusammenhang zwischen der Anzahl der Festplatten und der Lese- bzw. Schreibgeschwindigkeit beim Striping ohne Paritäten besteht, wird in Kapitel 7.3, S. 60 untersucht. Welche Poolkonfiguration letztendlich eingesetzt werden sollte, muss je nach Anwendungsfall entschieden werden. Alles in allem bietet ZFS viele Möglichkeiten die Strukturierung des Pools an hohe Sicherheits- bzw. Geschwindigkeitsanforderungen anzupassen.

5.6 Einsatz von SSDs als Cache- bzw. Log-Devices

ZFS bietet die Möglichkeit durch den Einsatz von SSDs oder anderen schnellen Speichermedien die Lese- und Schreibgeschwindigkeiten in bestimmten Fällen zu steigern. Durch Angabe der Schlüsselwörter *cache* und *log* können sie wie in Kapitel 2.3.3, S. 8 beschrieben als Cache- bzw. Log-Device zum Pool hinzugefügt werden.
Auf den Nutzen und die Funktionsweise dieser Mechanismen wird in den nachfolgenden Kapiteln eingegangen. Um den Einfluss auf den Pool bei einfachen Nutzungsszenarien zu ermitteln, wurden mit *fio* sequenzielle und zufällige Lese- und Schreibtests durchgeführt. Die Ergebnisse können Abbildung 13 entnommen werden, wobei folgende SSDs zum Einsatz kamen:

- **Cache:** SAS 12Gbit SSD HGST Ultrastar1600 800GB
  - Lesen: 396 MiB/s | Schreiben: 374 MiB/s
- **Log:** SAS 12Gbit SSD HGST Ultrastar800e 100GB
  - Lesen: 386 MiB/s | Schreiben: 367 MiB/s

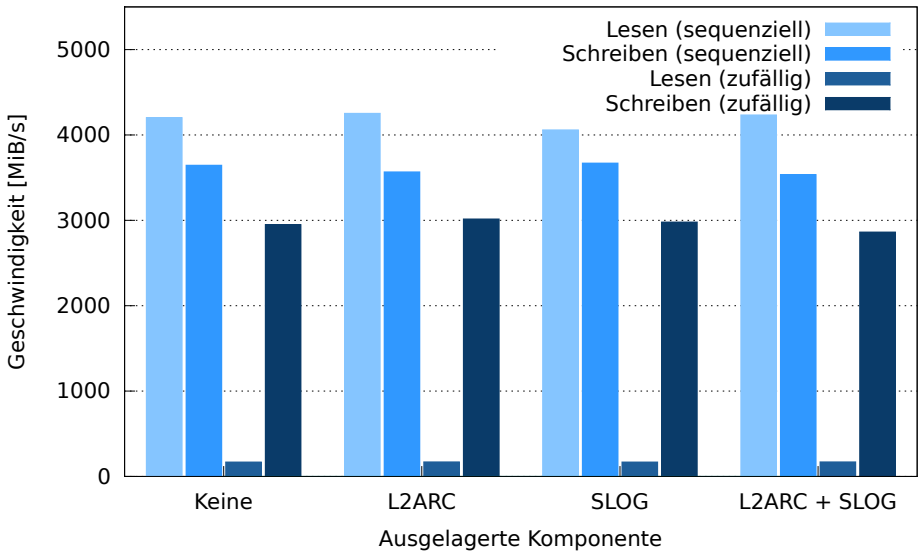

Abbildung 13: Einfluss von SSDs als Cache- bzw. Log-Device

Bei keinem der vier Benchmarks kann durch den Einsatz eines Cache- oder Log-Devices eine Veränderung der Leistung festgestellt werden. Gleichzeitig kann den Ergebnissen das Verhalten von ZFS bei zufälligen Zugriffen entnommen werden. Die Schreibgeschwindigkeit sinkt dabei aufgrund interner Optimierungen (siehe *Transaktionsgruppen* Kapitel 5.6.2, S. 43) nur um etwa 25 % im Vergleich zum sequenziellen Schreiben. Durch die sehr

hohen Latenzen der Festplatten erreicht die zufällige Lesegeschwindigkeit nur 4 % der sequenziellen.
Somit können SSDs nicht zur Optimierung von ZFS im Allgemeinen eingesetzt werden. Benchmarks für Szenarien in denen deutliche Unterschiede festzustellen sind, folgen in den nächsten Kapiteln.

5.6.1 Caching (ARC / L2ARC)

Zur Optimierung von Lesezugriffen verwendet ZFS standardmäßig einen im RAM abgelegten Adjustable Replacement Cache (ARC). Dabei werden die Daten nach zwei verschiedenen Kriterien ausgewählt und im Arbeitsspeicher gehalten, um sie anschließend mit geringer Latenz ausliefern zu können. Die Auswahl erfolgt einerseits nach einer „Most Frequently Used"-Liste und andererseits nach einer „Most Recently Used"-Liste. Dadurch werden sowohl häufig, als auch kürzlich gelesene Blöcke im Cache abgelegt. Blöcke die bereits einmal im Cache waren, werden in sogenannten „ghost"-Listen referenziert, damit sie bei erneuten Zugriffen bevorzugt wieder in den ARC aufgenommen werden können.[45]
Da die Größe des Arbeitsspeichers begrenzt ist, wurde ein Level 2 ARC (L2ARC) implementiert, welcher optional aktiviert und auf eine SSD ausgelagert werden kann. Sobald der ARC vollständig gefüllt ist, werden alle weiteren Daten in den L2ARC ausgelagert. Da hierbei meist sehr schnelle SSDs mit geringen Zugriffszeiten zum Einsatz kommen, kann insbesondere bei sich wiederholenden zufälligen Lesezugriffen eine deutliche Geschwindigkeitssteigerung erzielt werden. Bei sequenziellen Lesevorgängen ist dies überwiegend nicht möglich, da die Lesegeschwindigkeit des ganzen Pools häufig wesentlich höher ist, als die einer als L2ARC eingesetzten SSD.[46]
Zur Untersuchung des beschriebenen Effekts wurden mehrere Benchmarks durchgeführt, bei denen eine 1 TiB große Datei zufällig eingelesen wird. Die Veränderung der Geschwindigkeit in Abhängigkeit von der Anzahl der Durchläufe kann Abbildung 14 entnommen werden.

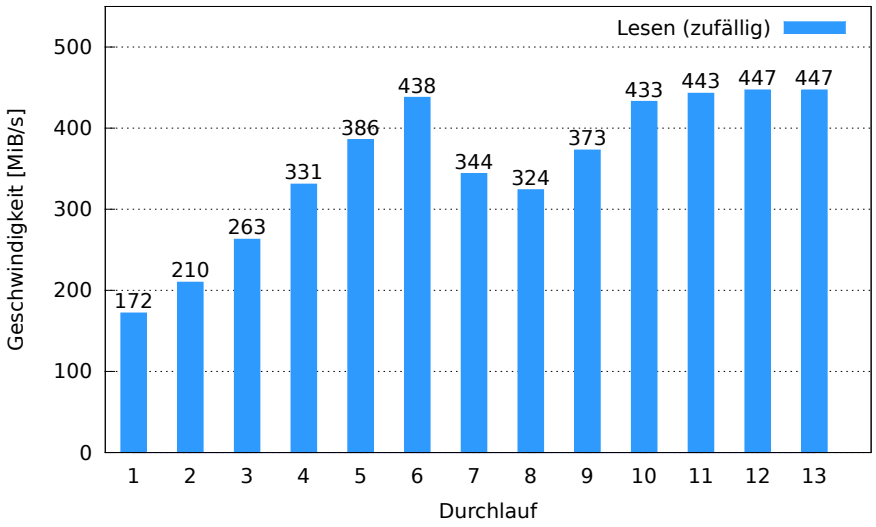

Abbildung 14: Wiederholtes zufälliges Lesen mit aktiviertem L2ARC

Beim ersten Einlesen der Datei kann aufgrund des noch leeren Caches sowie der hohen Zugriffszeiten der Festplatten nur eine Geschwindigkeit von 170 MiB/s erreicht werden. Mit zunehmender Anzahl an Durchläufen nähert sie sich einem Grenzwert von ca. 450 MiB/s. Bei den Wiederholungen sieben bis neun sinkt die Geschwindigkeit nochmals bevor sie konstant bleibt. Da die Vorhersage der zum Caching geeigneten Blöcke schwierig ist, handelt es sich dabei um ein sehr gutes Ergebnis. Letztendlich kann die Geschwindigkeit durch den Einsatz eines L2ARC fast verdreifacht werden.
Vor allem bei Datenbanksystemen können die Latenzen durch die zweite Stufe des Cachings verringert werden. Die Kosten für eine entsprechend schnelle SSD müssen grundsätzlich für jedes Einsatzszenario einzeln mit der zu erwartenden Beschleunigung des Pools abgewogen werden. Eine allgemeingültige Optimierung kann durch den Einsatz des L2ARC nicht erreicht werden.

5.6.2 Logging (ZIL / SLOG)

Zur Geschwindigkeitsoptimierung und Minimierung der Fragmentierung fasst ZFS alle Daten vor dem Schreiben zu sogenannten Transaction Groups (TXG) zusammen. Zur Bildung dieser Transaktionsgruppen werden zu schreibende Daten standardmäßig bis zu fünf Sekunden im Arbeitsspeicher zurückgehalten, bevor sie auf dem persistenten Speicher des Pools (Festplatten) abgelegt werden. Sollte das System in diesem Zeitraum beispielsweise aufgrund eines Stromausfalls abstürzen, wären sie verloren. Zur Gewährleistung der Transaktionssicherheit werden die Daten bei synchronen Schreibzugriffen gleichzeitig in den ZFS Intent Log (ZIL) geschrieben. Dabei handelt es sich um einen persistenten Speicher, der im Falle eines Stromausfalls zur Wiederherstellung der letzten TXG genutzt

wird. Da dieser standardmäßig Teil des Pools ist, müssen doppelt so viele Schreibzugriffe durchgeführt werden, wodurch sich die Geschwindigkeit enorm verschlechtert.[47]
Bei der Verwendung des ZIL beschränkt sich ZFS ohne weitere Anpassungen auf synchrone Schreibvorgänge, da die dabei geschriebenen Daten meist sehr wichtig sind. Bei dieser Art von Zugriffen wird im ausführenden Programm auf den erfolgreichen Abschluss des Speichervorgangs gewartet, bevor die eigentliche Programmausführung fortgesetzt wird. Da die Ausführung asynchroner Schreibzugriffe dem Kernel überlassen wird, während die Anwendung bereits weiterarbeitet, handelt es sich hierbei häufig um weniger wichtige Daten. Deshalb wird der ZIL zugunsten eines höheren Datendurchsatzes in diesem Fall nicht verwendet. Die Transaktionssicherheit kann für jedes Dataset einzeln durch Anpassung des Parameters *sync* aktiviert bzw. deaktiviert werden.[48]
Folgende Einstellungen können getroffen werden [18]:

- **standard** - nur synchrone Schreibvorgänge werden mithilfe des ZIL abgesichert
- **always** - jeder Schreibvorgang findet unter Verwendung des ZIL statt
- **disabled** - der ZIL wird deaktiviert

Somit kann die Transaktionssicherheit je nach Wichtigkeit der Daten angepasst bzw. deaktiviert werden. Sollte eine Deaktivierung nicht möglich sein, so kann durch die Konfiguration eines Separate Intent Logs (SLOG) eine deutliche Steigerung der Schreibgeschwindigkeit erzielt werden. Dabei wird unter Angabe des Schlüsselwortes *log* eine SSD oder eine batteriegepufferte RAM-Disk zum Pool hinzugefügt, auf die der ZIL anschließend ausgelagert wird. Dadurch müssen keine mehrfachen Schreibzugriffe auf dem Pool durchgeführt werden und die Geschwindigkeit wird ausschließlich durch das Cache-Device begrenzt.[48]
Um die Geschwindigkeit bei der Verwendung eines ZIL bzw. SLOG zu untersuchen, wurden verschiedene Benchmarks mit synchronen Schreibzugriffen durchgeführt. Mit *fio* ist dies durch Angabe des Parameters *--sync=1* möglich. Die Ergebnisse sind in Abbildung 15 zu sehen.

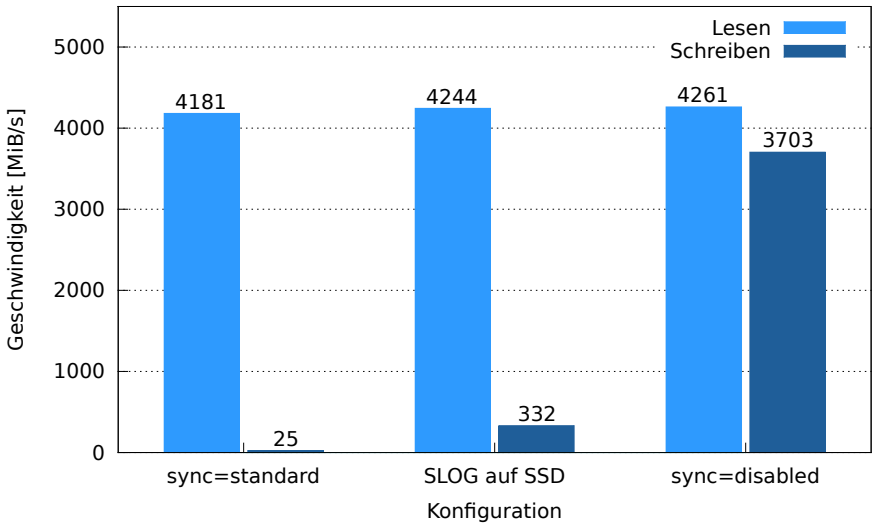

Abbildung 15: Synchrone Schreibzugriffe mit und ohne SLOG

Da der ZIL nur bei Schreibvorgängen zum Einsatz kommt, kann kein Einfluss auf die Lesegeschwindigkeit festgestellt werden. Bei Verwendung des ZIL liegt die Geschwindigkeit aufgrund der zusätzlichen Belastung des Pools nur bei 25 MiB/s. Unter Verwendung einer SSD als SLOG kann sie auf 332 MiB/s gesteigert werden. Da hierbei die Schreibgeschwindigkeit der SSD den begrenzenden Faktor darstellt, könnte sie durch die Verwendung einer batteriegepufferte RAM-Disk nochmals enorm erhöht werden. Bei abgeschalteter Transaktionssicherheit sind keine Leistungsunterschiede zwischen synchronen und asynchronen Zugriffen zu erkennen. Sollte ein Server an eine unterbrechungsfreie Stromversorgung (USV) angeschlossen sein, kann der ZIL meist deaktiviert werden, da ein Stromausfall nicht sofort zum Absturz des Systems führt.
Da der ZIL lediglich bei einem Systemabsturz zur Wiederherstellung der Daten der letzten TXG verwendet wird und die damit verbunden Geschwindigkeitsverluste sehr hoch sind, sollte er nur aktiviert werden, wenn ein Verlust der Daten der letzten paar Sekunden nicht akzeptabel ist. Zur Steigerung der Schreibgeschwindigkeit sollte in diesem Fall eine SSD als SLOG zum Einsatz kommen. Prinzipiell bietet ZFS mit dem ZIL eine gute Möglichkeit, das System bei synchronen oder wahlweise auch asynchronen Schreibvorgängen bei einem Stromausfall vor Datenverlust zu schützen.

5.7 Hohe Belegung des Pools

Aufgrund des Copy-On-Write (COW), der dynamischen *recordsize* und weiterer Funktionen entsteht mit steigendem Füllstand des Pools eine relativ hohe Fragmentierung des noch freien Speicherplatzes. Dies kann ab einer Belegung von mehr als 80 % zu starken Geschwindigkeitseinbußen führen. Aus diesem Grund wird in vielen Quellen empfohlen, den Pool zu maximal 80 % zu füllen [49].

Um die Richtigkeit dieser Behauptung zu überprüfen, werden zwei verschiedene Fälle untersucht. Dabei wurde der Pool einmal durch sequenzielles Schreiben von Daten bzw. durch zufällige Schreibzugriffe gefüllt. Die erreichten Lese- und Schreibgeschwindigkeiten in Abhängigkeit von der Poolbelegung können Abbildung 16 bzw. 17 entnommen werden. Aufgrund des COW war beim zufälligen Beschreiben des Pools eine hohe Fragmentierung des freien und des belegten Speichers zu erwarten. Deshalb wurde diese jeweils für den zu 96 % ausgelasteten Pool ermittelt und verglichen. Mithilfe folgender Befehle können die zwei Eigenschaften ausgelesen bzw. berechnet werden:

```
# Ermittlung der Fragmentierung des freien Speichers
zpool list <POOLNAME>

# Ermittlung der Fragmentierung des belegten Speichers
zdb -ddddd <POOLNAME> | gawk --non-decimal-data -f /zfsscripts/fragments.awk
```

Zur Ermittlung der Fragmentierung des belegten Speichers muss zusätzlich das Tool *gawk* installiert werden. Das dabei zu verwendende Skript *fragments.awk* ist in Listing 8, S. 71 im Anhang zu finden. Für die Benchmarks kam ein kleinerer, aus neun Festplatten bestehender Pool zum Einsatz.

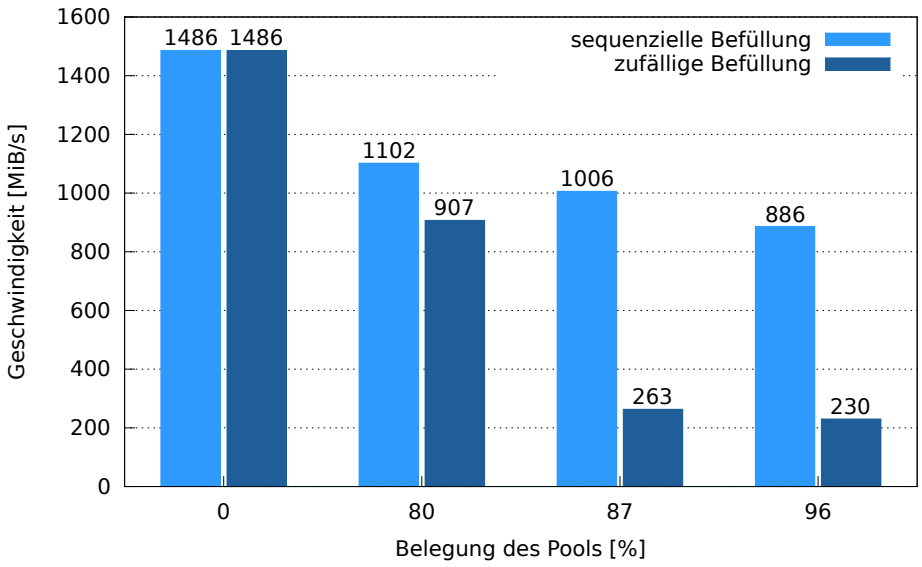

Abbildung 16: Einfluss der Poolbelegung auf die sequenzielle Lesegeschwindigkeit

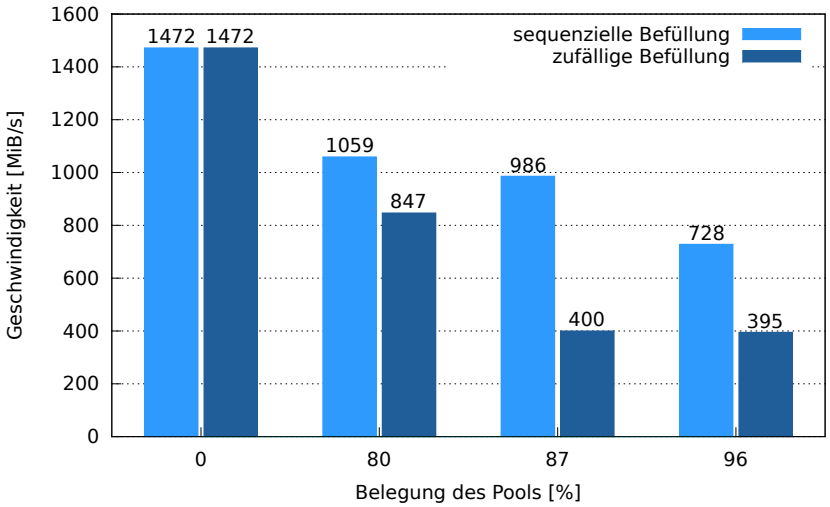

Abbildung 17: Einfluss der Poolbelegung auf die sequenzielle Schreibgeschwindigkeit

Trotz der starken Geschwindigkeitsdifferenzen können die beiden Testfälle auch bei hoher Belegung des Pools nicht anhand der Fragmentierung unterschieden werden. Der belegte Speicher war dabei jeweils zu etwa 90 % und der freie Speicher zu etwa 60 % fragmentiert. Bei der sequenziellen Befüllung des Pools sinkt die Geschwindigkeit mit zunehmender Belegung langsam, aber kontinuierlich. Die Halbierung des Datendurchsatzes bei vollem Speicher kann auf den Aufbau der verwendeten Festplatten zurückgeführt werden. Da die Spuren von außen nach innen kleiner werden und sie standardmäßig in dieser Reihenfolge beschrieben werden, wird bei einer fast vollen Festplatte nur etwa die halbe Geschwindigkeit erreicht.

Bei der Befüllung des Pools mithilfe zufälliger Schreibzugriffe ist die Geschwindigkeit zwar bereits bei einer Speicherbelegung von 80 % niedriger, aber der eigentliche Einbruch tritt erst bei noch höheren Belegungen auf. Dabei sinkt die Leserate im Vergleich zum ersten Test auf rund ein Viertel und die Schreibgeschwindigkeit auf ca. die Hälfte. Die Ergebnisse bei einem zu 87 % bzw. zu 96 % belegten Pool sind nahezu identisch.

Eine mögliche Ursache für diesen Einbruch ist auf Ebene der Metaslabs zu finden. Dabei handelt es sich um rund 200 gleichgroße Speicherbereiche, die auf jedem *vdev* angelegt werden. Auf diese Weise sollen kleinere Verwaltungseinheiten geschaffen und die Listen mit den Adressen freier und belegter Blöcke (Space Maps[50]) klein gehalten werden. Da pro Festplatte nur in einen der Metaslabs geschrieben wird, muss dieser anhand von bestimmten Merkmalen ausgewählt werden. Zur Optimierung der Geschwindigkeit werden der noch freie Speicher sowie die Position auf der Festplatte (innere oder äußere Spur) als Kriterien herangezogen.[51]

Die zu beschreibenden Blöcke werden innerhalb der Metaslabs standardmäßig nach dem

„First Fit“-Prinzip ausgewählt. Dabei wird immer ausgehend von der letzten Schreibposition der nächste freie Speicherbereich verwendet. Sobald der freie Speicher eines Metaslabs auf unter 4 % fällt, wird hingegen die „Best Fit“-Methode eingesetzt. Dabei wird zur Minimierung des Verschnitts nach dem kleinstmöglichen Speicherbereich gesucht, in den die zu schreibenden Daten passen. Da die Suche bei diesem Verfahren mehr Zeit in Anspruch nimmt und immer erst zur entsprechenden Stelle im Metaslab gesprungen werden muss, verringert sich sowohl die Schreib- als auch die Lesegeschwindigkeit.[52]
Darüber hinaus sorgt das COW beim Überschreiben der vor dem Benchmark angelegten Datei dafür, dass nicht wieder dieselben physischen Blöcke belegt werden. Dadurch entstehen viele über die Metaslabs verteilte Lücken, die bei hoher Belegung des Pools zum Ablegen von Daten gesucht werden müssen. Somit ist kein sequenzielles Schreiben innerhalb der Metaslabs möglich und die Schreibgeschwindigkeit verringert sich nochmals unabhängig davon, welche Methode zur Auswahl eines geeigneten Blocks zum Einsatz kommt.
Weil die Daten beim sequenziellen Schreiben sehr gleichmäßig über die Metaslabs verteilt werden können und keine Lücken entstehen, kommt es in diesem Fall vermutlich nicht zu einem Einbruch der Geschwindigkeit. Da beim zufälligen Schreiben eine zuvor angelegte Datei in anderer Reihenfolge wieder überschrieben wird, ist das gleichmäßige Verteilen der Daten wesentlich schwieriger. Intern werden dabei Daten wieder gelöscht und durch das COW umverteilt. Deshalb ist es denkbar, dass der freie Speicher einiger Metaslabs bei diesem Szenario schon eher unter 4 % sinkt, wodurch das „Best Fit“-Verfahren zum Einsatz kommt und die Geschwindigkeit einbricht. Es handelt sich dabei jedoch nur um die plausibelste Erklärung, es kann nicht ausgeschlossen werden, dass weitere Faktoren dieses Verhalten bedingt haben.
Im produktiven Betrieb werden Daten häufig verändert und gelöscht, weshalb das erste Szenario nur selten vorliegt. Aus diesem Grund ist bei einer Speicherbelegung von mehr als 80 % mit einem deutlichen Geschwindigkeitseinbruch zu rechnen. Dass die Angabe dieser Grenze als relativer und nicht als absoluter Wert erfolgt, kann unter anderem dadurch erklärt werden, dass ZFS auch intern bei der Verwaltung der Metaslabs den freien und belegten Speicher ins Verhältnis setzt. Dementsprechend ist die zu Anfang erwähnte Empfehlung plausibel. Da die verschiedenen Pools nicht anhand der Fragmentierung unterschieden werden können, ist es in der Praxis schwierig, herauszufinden, in welchem Zustand sich dieser momentan befindet. Sind sehr hohe Datendurchsatzraten notwendig, sollte der Pool maximal zu 80 % gefüllt werden.

6 Vergleich eines RAID-Z mit einem Software-RAID

Mit dem RAID-Z wurde für ZFS ein eigenständiges Software-RAID entwickelt. Es ist stark an die Optimierung der in ZFS integrierten Funktionen angepasst und auf diese angewiesen. Darüber hinaus bietet es im Vergleich zu herkömmlichen Soft- und Hardware-RAIDs einige Vorteile. Diese sollen nachfolgend kurz dargestellt und durch Messungen belegt werden. Der Vergleich wird dabei zu dem in den Linux-Kernel integrierten Software-RAID vorgenommen.

6.1 Funktionsvergleich

Allgemeine Informationen zum RAID-Z und dessen drei Varianten sind in Kapitel 2.1, S. 2 zu finden. Während ein RAID-Z1 mit einem RAID 5 und ein RAID-Z2 mit einem RAID 6 verglichen werden kann, existiert zum RAID-Z3 kein Äquivalent unter den herkömmlichen RAID-Leveln. Obwohl die Ziele des RAID-Z und eines gewöhnlichen RAIDs übereinstimmen, unterscheidet sich ihre Funktionsweise zum Teil sehr stark. Die größte Besonderheit des RAID-Z ist die dynamische Breite der Stripes, welche nicht immer die gleiche Anzahl physischer Blöcke umfassen müssen. Ein Beispiel dafür ist in Abbildung 18 vereinfacht dargestellt.

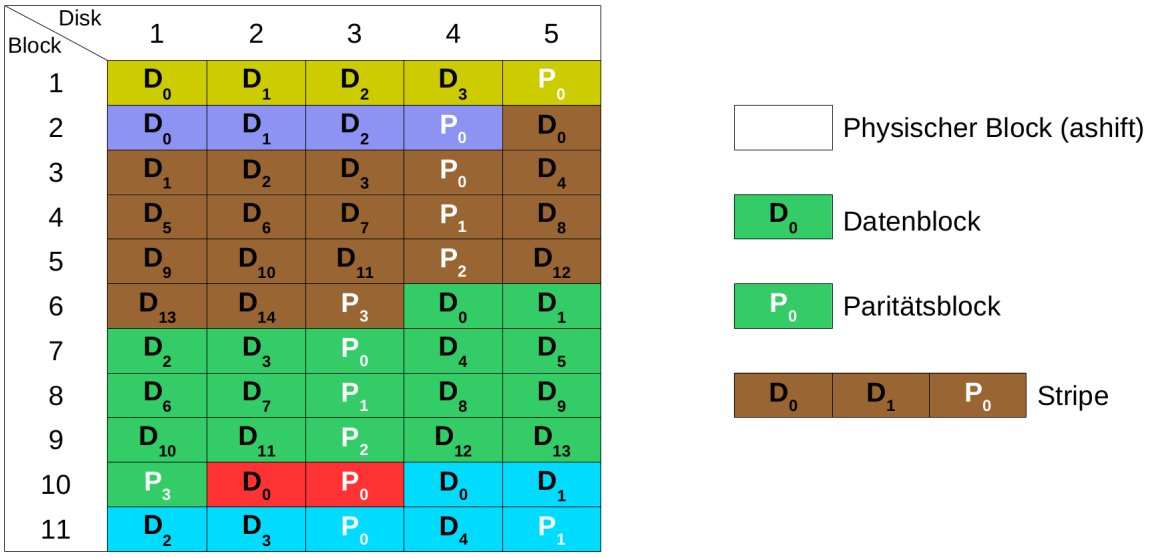

Abbildung 18: Dynamische Größe der Stripes eines RAID-Z1

Ein Viereck steht in der Darstellung für einen physischen Block, dessen Größe über den Parameter *ashift* beim Anlegen eines Pools eingestellt werden kann. Alle Blöcke einer Farbe entsprechen jeweils einem Stripe. Weil jeder logische Block in einem eigenen Stripe abgelegt wird, hängt deren Breite stark von der *recordsize* ab. Da Stripes immer im Ganzen gelesen und geschrieben werden, wird verhindert, dass beim Ablegen oder Abrufen von

Dateien nicht benötigte Daten bewegt werden müssen. Sollte die Anzahl der physischen Blöcke, auf die ein logischer Block aufgeteilt werden muss, die Anzahl der Festplatten des RAID-Z überschreiten, so werden mehrere Blöcke jeder Festplatte verwendet und zusätzliche Paritäten abgespeichert. Beispiele dafür sind die braun bzw. grün markierten Blöcke in der Abbildung. In Bezug auf ZFS werden diese weiterhin als ein einziger Stripe bezeichnet. Sollten sehr kleine Daten im Pool gespeichert werden, so entstehen kleinere Stripes, wie die in der Abbildung rot bzw. lila markierten. Beim Anlegen vieler kleiner Stripes wird mehr Speicher für die Paritätsinformationen benötigt, weshalb der nutzbare Speicher nie exakt vorhergesagt werden kann. Dementsprechend kann beispielsweise bei einem RAID-Z1 je nach Anzahl der Festplatten und Größe der logischen Blöcke wesentlich mehr als die Speicherkapazität einer einzelnen Festplatte für die Paritäten notwendig sein. Da die logischen Blöcke meist wesentlich größer als die physischen sind, wirkt sich dieser Effekt jedoch nicht so stark aus.[53]

Die Stripes beginnen durch ihre dynamische Größe nicht immer an der gleichen Position und können deshalb nur anhand der von ZFS verwalteten Metadaten lokalisiert werden. Beim Resilvering (Wiederherstellen) einer ausgefallenen Festplatte muss dementsprechend zusätzlich die Struktur des RAID-Z rekonstruiert werden. Beim Lesen wird auf die Parität nur im Falle eines Fehlers beim Prüfsummenvergleich zurückgegriffen.

Durch die Kombination von Prüfsummen und Paritäten kann sogar ein RAID-Z1 einzelne Bitfehler beheben. Da eine Paritätsinformation nicht ausreicht, um den fehlerhaften Block zu identifizieren, wird zusätzlich die Prüfsumme verwendet. Dabei ermittelt das RAID-Z1 durch Probieren, welcher Block des Stripes wiederhergestellt werden muss, damit die Prüfsummen wieder übereinstimmen. Dies ist im Vergleich zu einem RAID 5 ein wesentlicher Vorteil, da dieses nicht in der Lage ist einzelne Bitfehler zu korrigieren.[54]

Im Gegensatz zum RAID-Z werden die Paritäten beim RAID 5 und RAID 6 zyklisch über alle Festplatten verteilt. Die Breite der Stripes entspricht immer der Anzahl der Festplatten.

Ein weiterer Vorteil des RAID-Z zeigt sich beim Anlegen. Während herkömmliche RAIDs einen umfangreichen Initialisierungsprozess samt Berechnung aller Paritäten durchführen, ist dies bei einem RAID-Z nicht notwendig. Auch beim Resilvering werden nur die belegten Blöcke des entsprechenden *vdevs* rekonstruiert [55]. Weitere RAID-Level wie RAID-0 oder RAID-1 können in ZFS durch die Verwendung eines Pools ohne Paritäten bzw. den Einsatz von gespiegelten Festplattenpaaren abgebildet werden.

6.2 Geschwindigkeitsvergleich

Zum Vergleich der Geschwindigkeiten werden ein RAID-Z2 und ein RAID 6, bestehend aus jeweils acht Festplatten, verwendet. Da sie beide zwei Festplatten für Paritätsinformationen reservieren und häufig eingesetzt werden, sind sie sehr gut für einen direkten Vergleich geeignet. Das RAID 6 wird mit dem in den Linux-Kernel integrierten Software-RAID realisiert. Um dieses nutzen zu können, muss das Paket *mdadm* installiert sein. Anschließend kann das RAID mit folgendem Befehl angelegt werden:

```
mdadm --create /dev/md0 --level=6 --raid-devices=8 /dev/sdj /dev/sdk /dev/sdl /dev/
    sdm /dev/sdn /dev/sdo /dev/sdp /dev/sdq
```

Hinweise zum Erstellen eines RAID-Z sind in Kapitel 2.3.3, S. 8 zu finden.

Beim Anlegen der RAIDs zeigt sich bereits der erste Unterschied. Während das RAID-Z2 nach ca. zwei Sekunden vollständig erstellt und initialisiert ist, dauert dieser Vorgang beim RAID 6 etwa 18 Stunden. Letzteres kann zwar bereits vor Abschluss des Initialisierungsprozesses verwendet werden, jedoch kann sich die Dauer des Vorgangs dadurch stark verlängern. Die lange Initialisierung des RAID 6 wird durch die Herstellung der Paritäten für jeden Stripe verursacht.

Die Messung der Lese- und Schreibgeschwindigkeiten erfolgte für beide RAIDs nach Abschluss der Initialisierung. Die Benchmarks wurden jeweils für den intakten Zustand sowie den Fall eines Festplattenausfalls (degraded RAID) durchgeführt. Da der Pool bei diesem Test aus acht Festplatten aufgebaut ist, wurde *fio* nur unter Verwendung eines Jobs ausgeführt. Dennoch kann in diesem Fall die volle Geschwindigkeit des Pools erreicht und gleichzeitig die negativen Einflüsse paralleler Zugriffe vermieden werden. Die Ergebnisse der Benchmarks können folgender Abbildung entnommen werden:

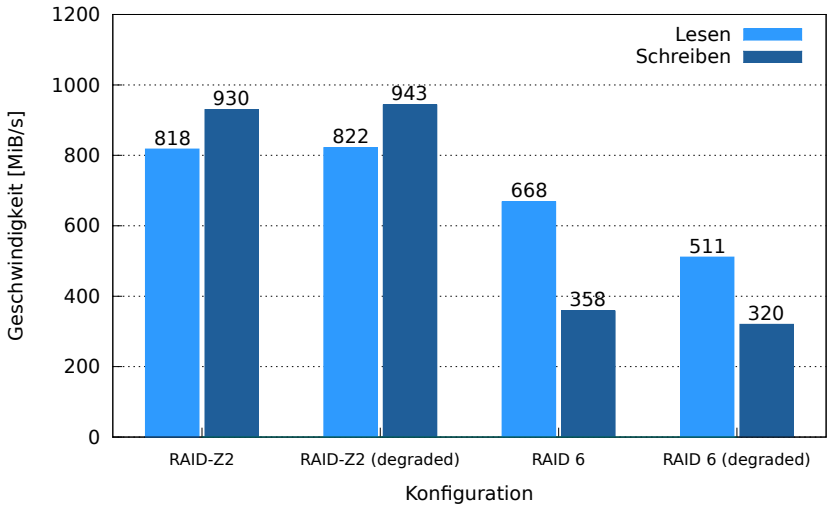

Abbildung 19: Geschwindigkeitsvergleich zwischen RAID 6 und RAID-Z2

Das RAID-Z2 erreicht im Vergleich zum RAID 6 eine ca. 18 % höhere Lese- und 62 % höhere Schreibgeschwindigkeit. Da die CPU-Auslastung jeweils bei ca. 5 % lag, wurde die Geschwindigkeit bei keinem der beiden RAIDs von der Rechenleistung begrenzt. Der durch das Entfernen einer Festplatte simulierte Ausfall zeigt keinen Einfluss auf die Geschwindigkeiten des RAID-Z2. Während die Schreibgeschwindigkeit des RAID 6 ebenfalls kaum beeinflusst wird, verringert sich die Lesegeschwindigkeit um 26 %.
Damit ist das RAID-Z2 dem RAID 6 sowohl bei den Lese- und Schreibgeschwindigkeiten als auch bei der Dauer des Initialisierungsprozesses deutlich überlegen. Ein Hardware-RAID könnte unter Umständen noch höhere Geschwindigkeiten erreichen, weshalb die Aussage nur für das getestete Software-RAID zutrifft.

6.3 Vergleich des Wiederherstellungsverhaltens

Da im Falle eines Festplattenausfalls die Daten der defekten Festplatte auf einer neuen wiederhergestellt werden müssen und die Dauer dieses Vorgangs mitunter von entscheidender Bedeutung ist, soll auch dieses Verhalten anhand von Messungen und Benchmarks untersucht werden. Der Austausch kann dabei mit folgenden Kommandos durchgeführt werden:

```
# RAID-Z2
zpool replace <POOLNAME> <ALTES DEVICE> <NEUES DEVICE>

# RAID 6
mdadm --manage <RAID-DEVICE> --add <NEUES DEVICE>
```

Die Rekonstruktion der Daten beginnt in beiden Fällen sofort nach dem Hinzufügen der neuen Festplatte. Ohne gleichzeitige Benutzung der RAIDs konnten dabei folgende Wiederherstellungszeiten ermittelt werden:

- **leeres RAID 6:** 18 Stunden
- **volles RAID 6:** 18 Stunden
- **leeres RAID-Z2:** 2 Sekunden
- **volles RAID-Z2:** 30 Stunden

Sowohl die Wiederherstellung des leeren als auch des vollen RAID 6 dauert genauso lange, wie das zuvor getestete Anlegen. Somit hat der Füllstand im Falle eines Festplattenausfalls weder einen positiven noch einen negativen Einfluss. Beim RAID-Z2 ist die Abhängigkeit zwischen Speicherbelegung und Dauer des Resilverings deutlich zu erkennen. Während die Rekonstruktion eines leeren RAID-Z2 in zwei Sekunden abgeschlossen ist, dauert sie

bei einem sehr hohen Füllstand, mit 30 Stunden, nahezu doppelt so lange, wie die eines RAID 6. Somit kann das RAID-Z2 nur bis zu einer Belegung von reichlich 50 % durch eine kürzere Wiederherstellungszeit überzeugen. Grund dafür ist die dynamische Breite der Stripes, wodurch die Rekonstruktion der RAID-Struktur wesentlich aufwendiger ist und mehr Zeit in Anspruch nimmt. Bevor die eigentliche Wiederherstellung der Daten beginnt, muss anhand von Metadaten erst die Position und Größe aller Stripes berechnet werden.

Da RAIDs auch während eines Wiederherstellungsprozesses hohe Geschwindigkeiten erreichen müssen, wurden auch für diesen Fall Benchmarks durchgeführt. Vor Beginn der Tests wurde das RAID jeweils zu 50 % gefüllt, eine Festplatte entfernt und die Wiederherstellung gestartet. Die Ergebnisse im Vergleich zu einem intakten RAID können folgender Abbildung entnommen werden:

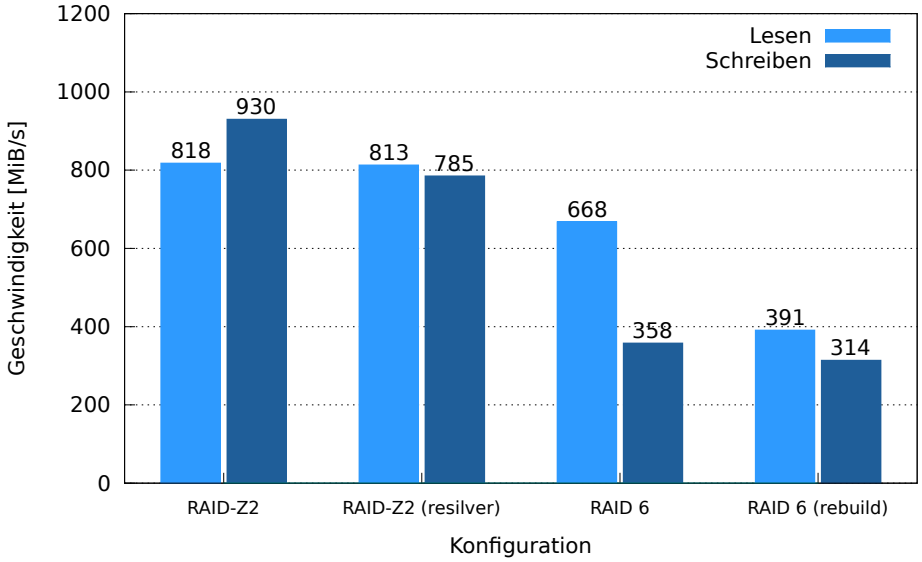

Abbildung 20: Geschwindigkeiten bei der Wiederherstellung eines RAID-Z2 bzw. RAID 6

Beim RAID-Z2 ist während der Wiederherstellung eine um 16 % niedrigere Schreibgeschwindigkeit festzustellen. Beim RAID 6 sinkt nur die Leserate um ca. 41 %, was immer noch akzeptabel, aber bereits eine spürbare Beeinträchtigung ist. Dafür, dass die Benchmarks während einer Wiederherstellung durchgeführt wurden, konnten beide gute Ergebnisse erzielen. Es ist jedoch zu beachten, dass sich die Dauer der Wiederherstellung durch starke Benutzung deutlich erhöhen kann.

Hinsichtlich des Wiederherstellungsverhaltens wiesen sowohl das RAID-Z2 als auch das RAID 6 Vor- und Nachteile auf.

7 Lastszenarien und ihre Optimierung

In diesem Kapitel werden Konfigurationen untersucht, die nicht der Verbesserung der Leistung im Allgemeinen dienen, sondern für bestimmte Anwendungsszenarien eine Optimierung bewirken können. Der Einfluss dieser Anpassungen wird ebenfalls anhand von Benchmarks untersucht und belegt, welche mit *fio* und den bereits in Kapitel 4, S. 15 erwähnten Parametern durchgeführt werden. Sollten abweichende Einstellungen zur Simulation eines Szenarios notwendig sein, werden diese an entsprechender Stelle kenntlich gemacht. Zum Vergleich werden erneut die maximale Lese- und Schreibgeschwindigkeit herangezogen. Die bereits im allgemeinen Teil dieser Arbeit (Kapitel 5, S. 19) behandelten Optimierungsmöglichkeiten gelten auch für die nachfolgenden Lastszenarien und werden deshalb nicht erneut angeführt. Folgende Lastszenarien sollen untersucht und optimiert werden:

- **hohe Datenverfügbarkeit und -sicherheit** - die Verfügbarkeit der Daten und ihr Schutz vor Zerstörung stehen an erster Stelle. Niedrigere Geschwindigkeiten werden für eine höhere Sicherheit hingenommen.
  - Beispiel: Speicher für wichtige Daten
- **parallele Zugriffe** - mehrere Nutzer bzw. Clients arbeiten gleichzeitig auf dem Pool. Die Struktur des Pools sowie dessen Geschwindigkeiten sollen für möglichst viele parallele Zugriffe optimiert werden. Die Sicherheit der Daten wird bei diesem Szenario nicht betrachtet.
  - Beispiel: Server für Home-Verzeichnisse
- **maximale Geschwindigkeit** - der Pool soll möglichst hohe Lese- und Schreibgeschwindigkeiten erreichen, während die Sicherheit der Daten bzw. ihre Verfügbarkeit eine untergeordnete Rolle einnehmen. Mögliche Datenverluste werden zur Erhöhung der Geschwindigkeit in Kauf genommen.
  - Beispiel: Zwischenspeicher für Berechnungen

Mit diesen drei Szenarien sollten die am häufigsten auftretenden Anwendungsfälle abgedeckt sein. Aufgrund der weiten Verbreitung dieser Szenarien bietet ZFS diverse Möglichkeiten, entsprechende Anpassungen durchzuführen. Diese werden nachfolgend getestet und diskutiert.

7.1 Hohe Datensicherheit und -verfügbarkeit

Die wohl am häufigsten an ein Speichersystem gestellten Anforderungen sind eine hohe Sicherheit und Verfügbarkeit der Daten. Deshalb wird nachfolgend ein Szenario untersucht, bei dem der Schutz der Daten vor Verlust bzw. längerer Nichtverfügbarkeit im Vordergrund steht. Geschwindigkeitseinbußen werden für die Erreichung dieser Ziele in Kauf genommen. Sollte eine Optimierung der Geschwindigkeit bei gleichem Sicherheits- oder Verfügbarkeitslevel möglich sein, so wird diese dennoch berücksichtigt.
Grundsätzlich müssen dabei die Anforderungen Datensicherheit und -verfügbarkeit separat betrachtet werden. Datensicherheit bezeichnet den Schutz von Informationen vor Verfälschung oder Zerstörung. Dies kann unter anderem durch die Erstellung von Backups auf unabhängigen Systemen gewährleistet werden und garantiert nicht gleichzeitig eine hohe Verfügbarkeit der Daten. Beim Ausfall des primären Speichersystems kann es während der Wiederherstellung zu längeren Ausfallzeiten kommen.[56]
Die Gewährleistung einer hohen Datenverfügbarkeit umfasst alle Maßnahmen zur Vermeidung von Ausfällen bzw. zur Minimierung von Ausfallzeiten. Dies beinhaltet jedoch nicht den vollständigen Schutz der Daten vor Verlust. Obwohl die Verfügbarkeit als Teilgebiet der Datensicherheit gesehen werden kann, werden sie zur besseren Anschaulichkeit im Nachfolgenden unabhängig voneinander betrachtet.[57]
Eine hohe Verfügbarkeit der Daten kann unter ZFS mithilfe der verschiedenen RAID-Z-Level bzw. gespiegelten *vdevs* erreicht werden. Einen Pool aus Spiegelpaaren aufzubauen bringt den Vorteil, dass der Rechenaufwand beim Schreiben von Daten sehr gering ist und die Wiederherstellung (Resilvering) einer ausgefallenen Festplatten sehr schnell geht. Dieser Effekt entsteht durch das reine Kopieren der Daten, ohne die Berechnung von Paritäten. Dadurch werden jedoch 50 % der Speicherkapazität für die Redundanz verbraucht. Sollte eine Festplatte defekt sein, so dürfen während der Wiederherstellung nur Festplatten anderer Spiegelpaare ausfallen, ohne einen Datenverlust zu verursachen.
Im Vergleich dazu können bei einem RAID-Z2 zwei bzw. bei einem RAID-Z3 drei Festplatten ausfallen. Dies ist in Zeiten von sehr großen Festplatten und entsprechend langandauernden Wiederherstellungsprozessen ein wichtiges Feature. Darüber hinaus wird je nach Größe der RAID-Zs im Vergleich zu Spiegelpaaren anteilig weniger Speicherplatz für die Paritäten verbraucht. Da RAID-Zs für eine Vielzahl von Anwendungsfällen mehr Vorteile mit sich bringen, werden diese für das zu untersuchende Szenario eingesetzt und optimiert.
Da von sehr großen RAID-Zs abgeraten wird (siehe Kapitel 5.5, S. 39), wurden verschie-

dene Poolkonfigurationen bestehend aus unterschiedlich großen RAID-Z2s getestet. Zur besseren Vergleichbarkeit wurden zusätzlich die Werte eines Pools ohne Redundanz gemessen. Die nutzbare Speicherkapazität entspricht bei allen Konfigurationen der von 24 Festplatten. Die Ergebnisse können Abbildung 21 entnommen werden. Zur Ermittlung des begrenzenden Faktors wurde zusätzlich die CPU-Auslastung in Abbildung 22 erfasst.

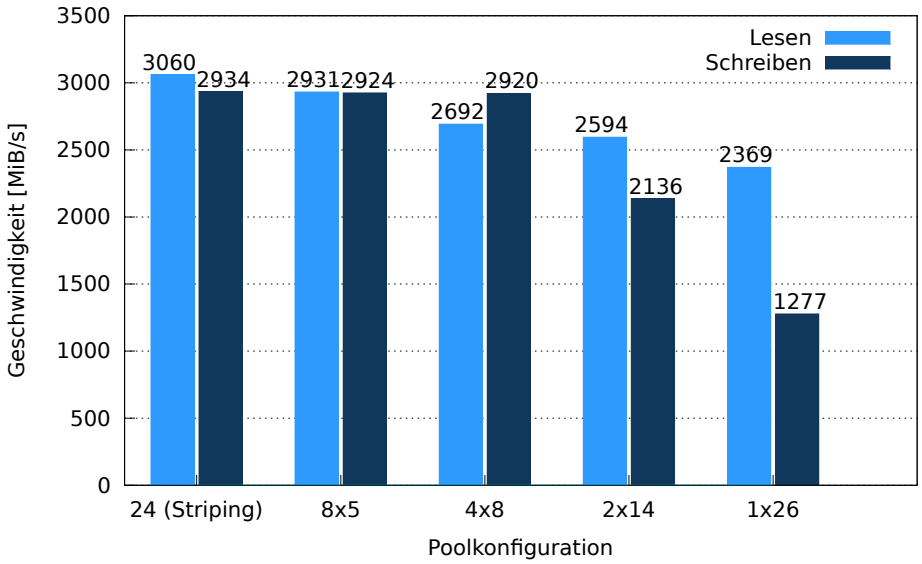

Abbildung 21: Vergleich verschiedener RAID-Z2 Konfigurationen (Geschwindigkeit)

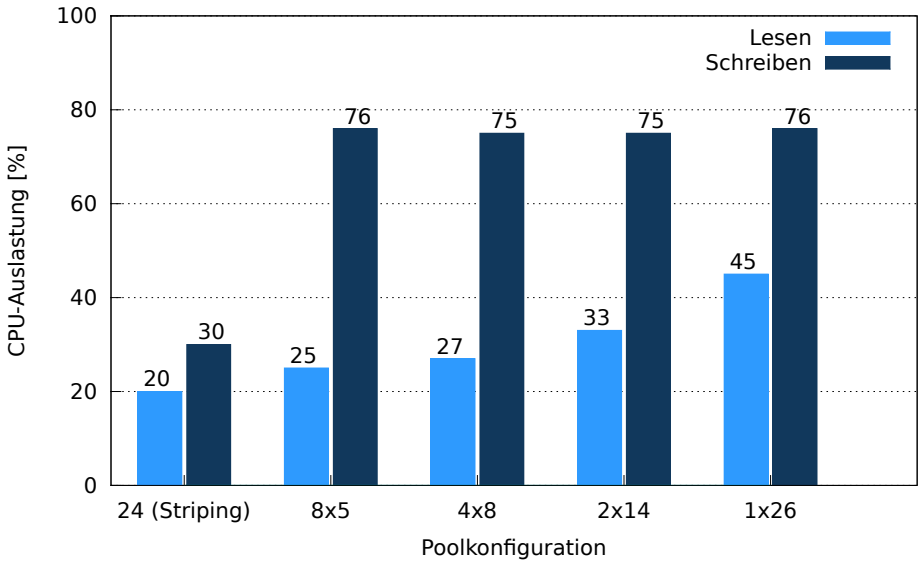

Abbildung 22: Vergleich verschiedener RAID-Z2 Konfigurationen (CPU-Auslastung)

Mit steigender Größe weist das RAID-Z2 schlechtere Schreibgeschwindigkeiten auf. Konfigurationen mit mehreren kleinen RAID-Z2s erreichen deutlich höhere Geschwindigkeiten. Die CPU-Auslastung liegt bei allen Konfigurationen bei ca. 75 %. Tests mit noch größeren Pools zeigten, dass es sich dabei nicht um den begrenzenden Faktor handelt. Da die Geschwindigkeit der verwendeten Festplatten ebenfalls als Engpass ausgeschlossen werden

kann, scheint ZFS nicht für große RAID-Zs optimiert zu sein. Die Lesegeschwindigkeiten sinken hingegen mit zunehmender Größe nur leicht, während die CPU stärker beansprucht wird.

Bis zu einer Größe von acht *vdevs* pro RAID-Z2 können keine spürbaren Geschwindigkeitsverluste im Vergleich zum Pool ohne Paritätsinformationen festgestellt werden. Aus diesem Grund sollten RAID-Zs zugunsten höherer Geschwindigkeiten und niedrigerer CPU-Auslastungen möglichst klein gehalten werden. Neben der Geschwindigkeitssteigerung bieten mehrere kleine RAID-Zs noch weitere Vorteile. Bei der Wiederherstellung einer ausgefallenen Festplatte wird ausschließlich die Leistung des betroffenen RAID-Z und nicht die der anderen bzw. des gesamten Pools beeinflusst. Darüber hinaus können im Vergleich zu einem großen RAID-Z theoretisch mehr Festplatten ausfallen, ohne dass Daten verloren gehen. Dies setzt jedoch voraus, dass diese nicht zum gleichen Verbund gehören. Ein Beispiel für einen Befehl mit dem ein großer Pool aus mehreren RAID-Z2s erstellt werden kann, ist in Listing 11, S. 72 zu finden.

Zum automatischen Austausch defekter Festplatten können beim Anlegen eines Pools sogenannte Hot Spares konfiguriert werden. Diese ersetzen im Falle eines Festplattenausfalls das entsprechende *vdev* und werden automatisch wiederhergestellt.

Soll neben der hohen Verfügbarkeit auch der Schutz der Daten vor Verlust gewährleistet werden, müssen regelmäßige Backups angefertigt werden. Dabei kann entweder auf gängige Backup-Lösungen oder die von ZFS angebotenen Funktionen zurückgegriffen werden. Das Tool *zfs* bietet unter Verwendung der Parameter *send* und *receive* die Möglichkeit einen Export bzw. Import von Daten durchzuführen. Auf diese Weise können ein ganzer Pool oder einzelne Snapshots auf einen anderen *zpool* übertragen werden.[58] Zur Ablage der Backups auf einem unabhängigen System, kann der Transfer auch über das Netzwerk durchgeführt werden.

Mithilfe von Backups und RAID-Zs kann eine hohe Verfügbarkeit der Daten und ein guter Schutz vor deren Zerstörung eingerichtet werden. Sollten noch höhere Anforderungen an die Verfügbarkeit bestehen, so kann statt des getesteten RAID-Z2 auf ein RAID-Z3 zurückgegriffen werden. Zur zuverlässigeren Erkennung von beschädigten Blöcken kann der Prüfsummenalgorithmus auf SHA256 gesetzt werden, wodurch sich die notwendige Rechenleistung jedoch erhöht.

7.2 Konkurrierende Zugriffe

Häufig werden große Speichersysteme von vielen Benutzern gleichzeitig verwendet und müssen dabei immer noch annehmbare Geschwindigkeiten liefern. Neben einem Pool, in dem von beliebigen Nutzern Daten abgelegt werden können, wäre weiterhin ein Szenario denkbar, bei dem eine große Anzahl von Home-Verzeichnissen auf einem ZFS-Server abgelegt werden soll. Um dabei ein flüssiges Arbeiten für jeden einzelnen Nutzer zu gewährleisten, darf die Performance des Systems auch bei einer großen Anzahl gleichzeitiger Lese- und Schreibvorgänge nicht einbrechen. Aus diesem Grund wird in diesem Kapitel der Einfluss konkurrierender Zugriffe auf die Gesamtgeschwindigkeit des Pools untersucht. Da mit *fio* maximal 1024 parallele Jobs simuliert werden können, wurden verschiedene Benchmarks mit bis zu 1000 konkurrierenden Lese- und Schreibzugriffen durchgeführt [4]. Damit sollte sowohl das Nutzungsverhalten kleiner als auch mittelgroßer IT-Infrastrukturen abgedeckt sein. Neben sequenziellen wurden auch zufällige Lese- und Schreibvorgänge getestet. Die Ergebnisse können folgender Abbildung entnommen werden:

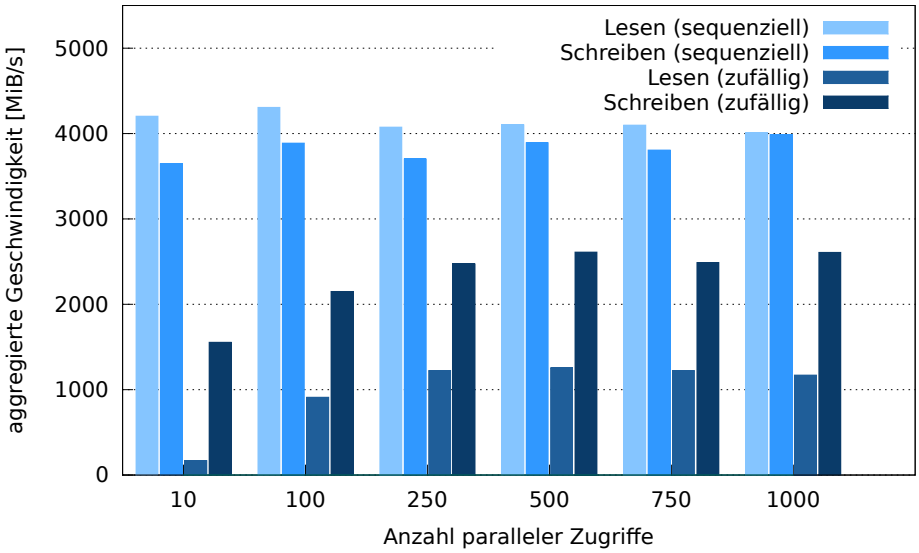

Abbildung 23: Einfluss paralleler Zugriffe auf die Geschwindigkeit

Die aggregierten sequenziellen Lese- und Schreibgeschwindigkeiten schwanken zwar je nach Anzahl paralleler Jobs ein wenig, bleiben aber relativ konstant und brechen nicht ein. Sowohl beim zufälligen Lesen als auch Schreiben kann die Geschwindigkeit bis zu einer Anzahl von 250 parallelen Zugriffen sogar erhöht werden. Beim Lesen kann eine Steigerung von 170 MiB/s auf 1200 MiB/s und beim Schreiben von 1600 MiB/s auf 2600 MiB/s erreicht werden. Da der Pool aus vielen Festplatten aufgebaut ist, verteilen sich die Latenzzeiten mit zunehmender Anzahl paralleler Zugriffe besser. Ab 250 parallelen Jobs

verändert sich die Geschwindigkeit nicht mehr.
Aufgrund der in Kapitel 5.6.2, S. 43 beschriebenen Transaktionsgruppen tritt beim Schreiben kein nennenswerter Einbruch der Datenraten auf. Durch die vorherige Sortierung der Daten im Arbeitsspeicher verhalten sich auch konkurrierende Zugriffe nahezu wie ein einzelner Datenstrom. Durch diese und weitere interne Optimierungen konnte ZFS bei diesem Senario auch ohne von der Standardkonfiguration abweichende Einstellungen sehr gut abschneiden. Ob die Geschwindigkeit bei einer noch größeren Anzahl paralleler Jobs einbrechen würde, kann aufgrund der Limitierungen von *fio* nicht überprüft werden.
Angesichts der sehr guten Benchmarkergebnisse ist für dieses Szenario keine Optimierung von ZFS notwendig. Durch interne Mechanismen kann die Leistung bei der Parallelisierung zufälliger Zugriffe sogar erhöht werden. ZFS ist somit sehr gut für das beschriebene Szenario geeignet.

7.3 Maximale Geschwindigkeit bei geringer Sicherheit

Aufgrund der immer weiter steigenden CPU-Geschwindigkeiten werden zunehmend schnellere Speichersysteme benötigt. Deshalb wird nachfolgend ein Szenario untersucht, bei dem die Maximierung des Datendurchsatzes im Mittelpunkt steht. Datensicherheit und Verfügbarkeit sind zwar nicht ungewollt, aber haben im Vergleich zur Geschwindigkeit eine niedrigere Priorität. Die gespeicherten Daten sind entweder reproduzierbar oder es handelt sich um eine Arbeitskopie, von der auf einem anderen System ein Backup existiert. Ein Verlust der Daten wird deshalb als annehmbares Risiko angesehen. Denkbar wäre beispielsweise ein Einsatz als Zwischenspeicher für Berechnungen, die große Datenmengen erzeugen oder verarbeiten.

Um für diesen Zweck optimale Geschwindigkeiten zu erreichen, sollte ein Pool ohne Paritäten eingesetzt werden. Die Berechnung und Speicherung dieser Paritäten würde die Leistung ausschließlich zum Negativen beeinflussen. Durch das Striping werden die Daten über die verschiedenen *vdevs* verteilt und eine hohe Geschwindigkeit erreicht. Darüber hinaus kann der Anteil der nutzbaren Speicherkapazität auf diese Weise erhöht werden. Bei der beschriebenen Konfiguration handelt es sich um jene, die bereits beim Benchmarking der allgemeinen Optimierungen in Kapitel 5, S. 19 zum Einsatz kam.

Da von zu großen Pools meist abgeraten wird, wurde der Zusammenhang zwischen Geschwindigkeit und Anzahl der *vdevs* (Festplatten) untersucht. Die Ergebnisse können nachfolgender Abbildung entnommen werden:

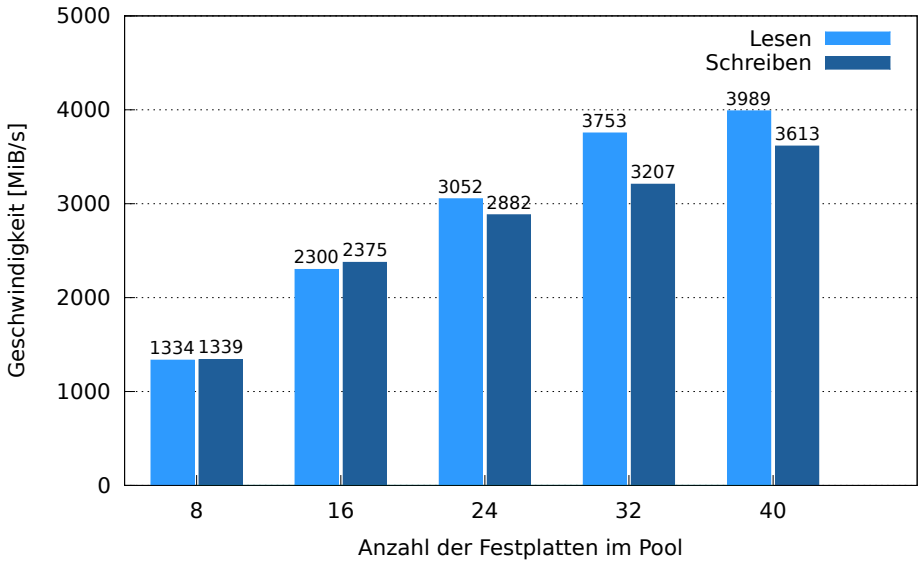

Abbildung 24: Geschwindigkeiten verschiedener Poolgrößen

Da der beschriebene Zusammenhang aus den absoluten Geschwindigkeiten der verschiedenen Poolgrößen nur schwer abgeleitet werden kann, sind diese zusätzlich auf jeweils ein

vdev heruntergerechnet in Abbildung 25 dargestellt.

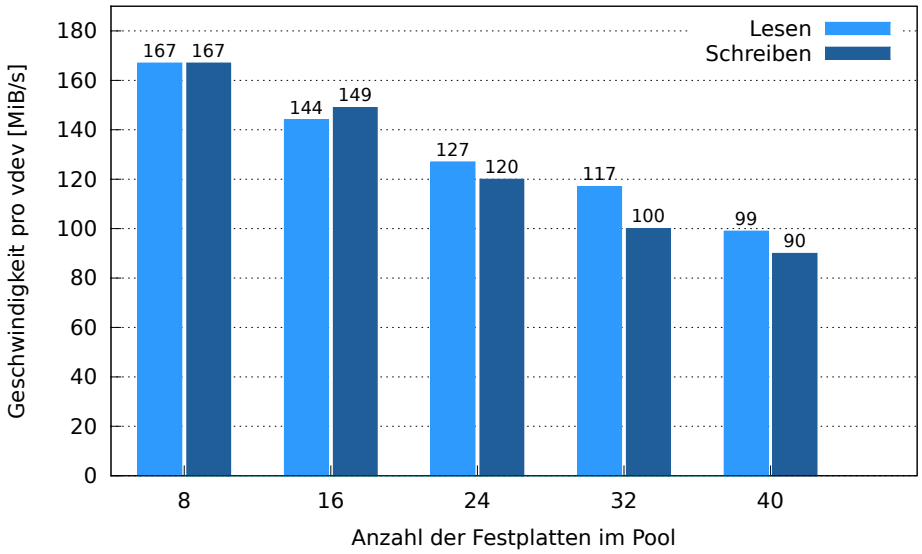

Abbildung 25: Geschwindigkeiten verschiedener Poolgrößen (pro vdev)

Die Benchmarks zeigen, dass die Geschwindigkeit nicht wie zu erwarten linear zur Anzahl der *vdevs* steigt. Mit zunehmender Größe des Pools nimmt die Transferrate pro Festplatte kontinuierlich und gleichmäßig ab. Während die Schreibgeschwindigkeit bis zu einer Größe von 24 *vdevs* jeweils nahezu identisch bzw. sogar höher als die Leserate ist, sinkt diese anschließend schneller.

Der Zusammenhang zwischen Geschwindigkeit und Poolgröße zeigt, dass mit mehreren kleinen Pools insgesamt eine höhere Performance als mit einem großen erreicht werden kann. Prinzipiell sollte ein Pool nicht aus mehr als 20 *vdevs* aufgebaut werden, wobei gespiegelte Festplatten und RAID-Zs jeweils als ein *vdev* zu sehen sind. Da die Geschwindigkeiten bis zu dieser Größe in etwa linear skalieren, sollten größere Pools nur zum Einsatz kommen, wenn sehr große zusammenhängende Speicherbereiche benötigt werden. Ansonsten sollten sie nur so groß wie nötig und so klein wie möglich gestaltet werden, um optimale Geschwindigkeiten zu erreichen. Die Verwendung von RAID-Zs ist eine gute Möglichkeit, um gleichzeitig eine hohe Verfügbarkeit der Daten zu gewährleisten und die Anzahl an *vdevs*, aus denen sich ein Pool zusammenssetzt, zu verringern.

Mithilfe eines Pools ohne Paritäten kann unter Verwendung weniger Festplatten eine hohe Geschwindigkeit erreicht werden. Dieser sollte aber wie bereits erwähnt nur für das beschriebene Szenario eingesetzt und nicht zu groß gestaltet werden. Prinzipiell ist die Wahrscheinlichkeit eines Datenverlusts bei dieser Konfiguration sehr hoch.

8 Fazit und Ausblick

ZFS konnte aufgrund vielseitiger Funktionen und Einstellungsmöglichkeiten als umfangreiche und flexible Lösung für Speichersysteme überzeugen. Trotz dieser Komplexität erreichte es für die meisten Einsatzzwecke bereits mit der Standardkonfiguration sehr gute Geschwindigkeiten. Darüber hinaus erwies sich das Anlegen, Verwalten und Anpassen von Pools und Datasets als sehr intuitiv und leicht nachvollziehbar. Mit diesen Eigenschaften bringt ZFS alles mit, was für eine erfolgreiche Etablierung als Speichersystem in kleinen und großen IT-Umgebungen notwendig ist.

Eine zusätzliche Optimierung des Datendurchsatzes kann je nach Anforderung durch Anpassung verschiedener Parameter, wie der *recordsize* oder einer entsprechenden Strukturierung des Pools erreicht werden. Durch Funktionen wie Komprimierung oder Deduplizierung kann ZFS für verschiedene Einsatzszenarien abgestimmt werden. Aufgrund der performanten RAID-Zs müssen sich dabei eine hohe Geschwindigkeit und Verfügbarkeit der Daten nicht ausschließen. Da ein *zpool* auch nachträglich erweitert werden kann, stellt der steigende Speicherbedarf kein Problem dar.

Letztendlich muss die passende Konfiguration jedoch immer in Abhängigkeit von den gegebenen Rahmenbedingungen ermittelt werden. Dabei können die Ergebnisse dieser Arbeit zur besseren Einschätzung der Vor- und Nachteile verschiedener Einstellungen herangezogen werden. Bei der Variation von Parametern sollten jedoch immer auch die negativen Einflüsse berücksichtigt werden. So sollte die *recordsize* z. B. nie zu niedrig eingestellt bzw. ein Pool aus zu vielen *vdevs* zusammengesetzt werden.

Da Pools mithilfe von Datasets an verschiedene Anforderungen angepasst werden können, müssen nicht alle Optimierungsentscheidungen für den gesamten Pool getroffen werden. Dies ermöglicht eine vielfältige und flexible Nutzung der angebotenen Funktionen. Darüber hinaus können die meisten Optionen auch nach dem Anlegen eines Pools aktiviert bzw. deaktiviert werden. Auf diese Weise müssen bei der Einrichtung einer ZFS-Umgebung nur wenige Entscheidungen sofort und endgültig getroffen werden.

Beim Einsatz von Funktionen wie der Komprimierung sollte bei der Beschaffung eines für ZFS optimierten Servers insbesondere auf eine leistungsfähige CPU geachtet werden. Ein verhältnismäßig großer Arbeitsspeicher wird hingegen nur bei der Deduplizierung bzw. dem Caching von Daten benötigt. Da das Kernel-Modul des „ZFS on Linux“-Projekts erst seit Kurzem für den Produktivbetrieb freigegeben ist, kann auch in Bezug auf die Software noch mit einigen Verbesserungen gerechnet werden.

Nachdem im Rahmen dieser Arbeit der Großteil aller Optimierungsmöglichkeiten analysiert wurde, steht nur noch die Integration von ZFS in die bestehende IT-Infrastruktur aus. Um auch über das Netzwerk auf den bereitgestellten Speicher zugreifen zu können, sind weitere Benchmarks und Optimierungen in Kombination mit NFS notwendig. Dabei müssen weitere Probleme gelöst werden, damit nicht z. B. die Netzwerkanbindung einen Engpass für die Geschwindigkeit bildet.
Alles in allem bietet ZFS ein breites Spektrum an Anpassungsmöglichkeiten für verschiedene Einsatzszenarien und kann deshalb sehr flexibel eingesetzt werden. Durch die ständige Weiterentwicklung ist auch in Zukunft mit weiteren Funktionalitäten und Verbesserungen zu rechnen. Man kann daher sagen, dass ZFS eine zukunftssichere Lösung zur Verwaltung großer Speichersysteme ist.

Literaturverzeichnis

[1] ZFS Fragmentation, Skript. http://www.guillermomolina.com.ar/en/articles/solaris/125-zfs-fragmentation, 2 June 2016.

[2] Displaying Information About ZFS Storage Pools. http://docs.oracle.com/cd/E19253-01/819-5461/gamml/index.html, 1 April 2016.

[3] Oracle Solaris ZFS Administration Guide. http://docs.oracle.com/cd/E19253-01/819-5461/gammt/index.html, 1 April 2016.

[4] fio, Manpage. http://linux.die.net/man/1/fio, 2 June 2016.

[5] Software-defined storage. https://en.wikipedia.org/wiki/Software-defined_storage, 6 June 2016.

[6] Btrfs. https://de.wikipedia.org/wiki/Btrfs, 6 June 2016.

[7] ZFS (Dateisystem). https://de.wikipedia.org/wiki/ZFS_%28Dateisystem%29, 23 May 2016.

[8] ReFS. https://de.wikipedia.org/wiki/ReFS, 6 June 2016.

[9] ZFS on Linux. https://wiki.ubuntuusers.de/ZFS_on_Linux/, 30 March 2016.

[10] ZFS-Eigenschaften und Terminologie. https://www.freebsd.org/doc/de_DE.ISO8859-1/books/handbook/zfs-term.html, 8 June 2016.

[11] BeeGFS. https://de.wikipedia.org/wiki/BeeGFS, 6 June 2016.

[12] RAID-Z. https://de.wikipedia.org/wiki/RAID#RAID-Z_im_Dateisystem_ZFS, 1 April 2016.

[13] ZFS Dateisystem. https://www.thomas-krenn.com/de/wiki/ZFS_%28Dateisystem%29, 1 April 2016.

[14] Migrating ZFS Storage Pools. http://docs.oracle.com/cd/E19253-01/819-5461/gbchy/index.html, 4 April 2016.

[15] GPL Violations Related to Combining ZFS and Linux. https://sfconservancy.org/blog/2016/feb/25/zfs-and-linux/, 11 April 2016.

[16] Ubuntu 16.04 LTS Xenial Xerus mit Snap und ZFS. http://www.computerbase.de/2016-04/linux-ubuntu-16.04-lts-xenial-xerus-mit-snap-und-zfs/, 6 June 2016.

[17] Festplattenlaufwerk. https://de.wikipedia.org/wiki/Festplattenlaufwerk#Speichern_und_Lesen_von_Byte-organisierten_Daten, 19 April 2016.

[18] zfs, Manpage. https://www.freebsd.org/cgi/man.cgi?zfs%288%29, 2 June 2016.

[19] Monitoring. http://www.it-administrator.de/themen/netzwerkmanagement/grundlagen/111034.html, 2 June 2016.

[20] Sysstat. http://sebastien.godard.pagesperso-orange.fr/, 1 April 2016.

[21] sar manual page. http://sebastien.godard.pagesperso-orange.fr/man_sar.html, 1 April 2016.

[22] ksar : a sar grapher. https://sourceforge.net/projects/ksar/, 1 April 2016.

[23] zpool, Manpage. https://www.freebsd.org/cgi/man.cgi?zpool%288%29, 2 June 2016.

[24] ZFS Device Replacement Enhancements. http://docs.oracle.com/cd/E19253-01/819-5461/githb/index.html, 1 April 2016.

[25] zdb, Manpage. https://www.freebsd.org/cgi/man.cgi?query=zdb&manpath=FreeBSD+9.1-RELEASE, 2 June 2016.

[26] Bonnie++. https://en.wikipedia.org/wiki/Bonnie%2B%2B, 31 March 2016.

[27] fio. http://freecode.com/projects/fio, 31 March 2016.

[28] Performance tuning. http://open-zfs.org/wiki/Performance_tuning, 12 April 2016.

[29] Advanced Format. https://en.wikipedia.org/wiki/Advanced_Format, 19 April 2016.

[30] Tuning ZFS recordsize. https://blogs.oracle.com/roch/entry/tuning_zfs_recordsize, 2 June 2016.

[31] ZFS Administration. https://pthree.org/2012/12/14/zfs-administration-part-ix-copy-on-write/, 2 May 2016.

[32] Prüfsummen und Daten mit Selbstheilungsfunktion. http://docs.oracle.com/cd/E19253-01/820-2313/gaypb/, 11 May 2016.

[33] Schnellstartanleitung. https://www.freebsd.org/doc/de_DE.ISO8859-1/books/handbook/zfs-quickstart.html, 2 June 2016.

[34] Fletcher's checksum. https://en.wikipedia.org/wiki/Fletcher's_checksum, 14 April 2016.

[35] SHA-2. https://de.wikipedia.org/wiki/SHA-2, 14 April 2016.

[36] Fast SHA-256 Implementations on Intel Architecture Processors. http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/sha-256-implementations-paper.pdf, 7 June 2016.

[37] gzip. https://de.wikipedia.org/wiki/Gzip, 12 April 2016.

[38] lzjb. https://en.wikipedia.org/wiki/LZJB, 12 April 2016.

[39] LZ4. https://de.wikipedia.org/wiki/LZ4, 11 April 2016.

[40] LZ4 Compression. http://wiki.illumos.org/display/illumos/LZ4+Compression, 12 April 2016.

[41] linux-3.18.30.tar.xz. https://cdn.kernel.org/pub/linux/kernel/v3.x/linux-3.18.30.tar.xz, 21 April 2016.

[42] The dedup Property. https://docs.oracle.com/cd/E19120-01/open.solaris/817-2271/gjhav/index.html, 26 May 2016.

[43] ZFS: To Dedupe or not to Dedupe... http://constantin.glez.de/blog/2011/07/zfs-dedupe-or-not-dedupe, 26 May 2016.

[44] ZFS Administration, Part II- RAIDZ. https://pthree.org/2012/12/05/zfs-administration-part-ii-raidz/, 11 May 2016.

[45] Persistent L2ARC. http://wiki.illumos.org/display/illumos/Persistent+L2ARC, 8 June 2016.

[46] Michael W Lucas / Allan Jude. *FreeBSD Mastery: ZFS*. Tilted Windmill Press, ISBN-13: 978-0692452356, Seite 34 - 36, 2015.

[47] ZFS fundamentals: transaction groups. http://blog.delphix.com/ahl/2012/zfs-fundamentals-transaction-groups/, 2 June 2016.

[48] To SLOG or not to SLOG. https://www.ixsystems.com/blog/o-slog-not-slog-best-configure-zfs-intent-log/, 2 June 2016.

[49] Recommended Storage Pool Practices. https://docs.oracle.com/cd/E26505_01/html/E37384/zfspools-4.html, 12 May 2016.

[50] Space Maps. https://blogs.oracle.com/bonwick/en/entry/space_maps, 9 June 2016.

[51] Allan Jude / Michael W Lucas. *FreeBSD Mastery - Advanced ZFS*. Tilted Windmill Press, ISBN-13: 978-0692452356, Seite 206 - 209, 2016.

[52] Doubling Exchange Performance. https://blogs.oracle.com/roch/entry/doubling_exchange_performance, 3 June 2016.

[53] ZFS RAIDZ stripe width. http://blog.delphix.com/matt/2014/06/06/zfs-stripe-width/, 2 June 2016.

[54] Software RAID using ZFS. https://en.wikipedia.org/wiki/ZFS#RAID-Z, 2 June 2016.

[55] ZFS: RAID-Z Resilvering. http://milek.blogspot.de/2014/12/zfs-raid-z-resilvering.html, 2 June 2016.

[56] Datensicherheit. http://wirtschaftslexikon.gabler.de/Definition/datensicherheit.html, 2 June 2016.

[57] Hochverfügbarkeit. https://de.wikipedia.org/wiki/Hochverf%C3%BCgbarkeit, 2 June 2016.

[58] Senden und Empfangen von ZFS-Daten. http://docs.oracle.com/cd/E19253-01/820-2313/gbchx/index.html, 13 May 2016.

A Abbildungen

Abbildung 26: Vorderseite des Servers

Abbildung 27: Rückseite des Servers

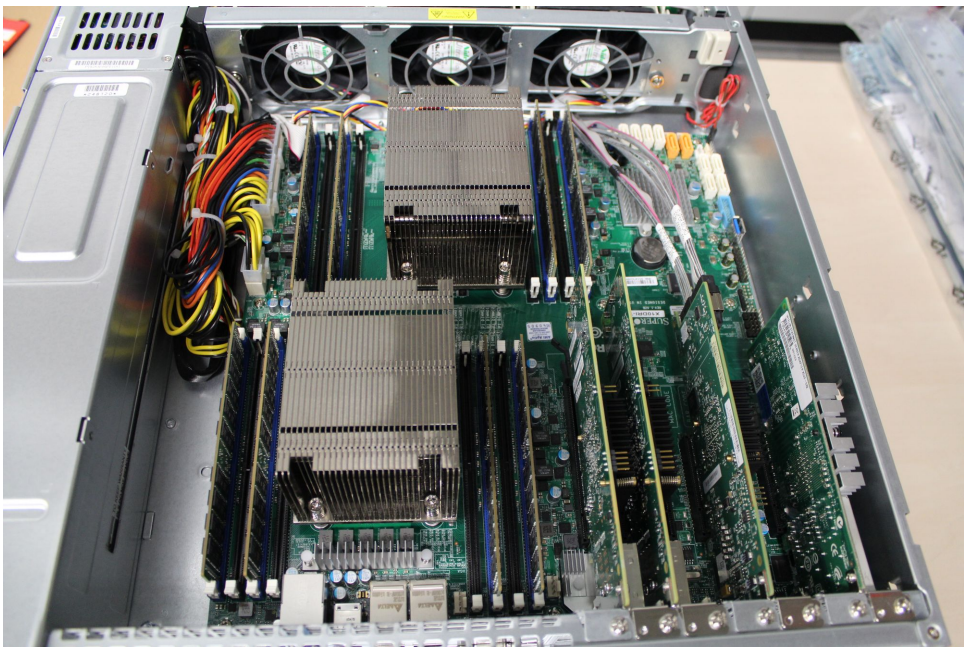

Abbildung 28: Hardware des Servers

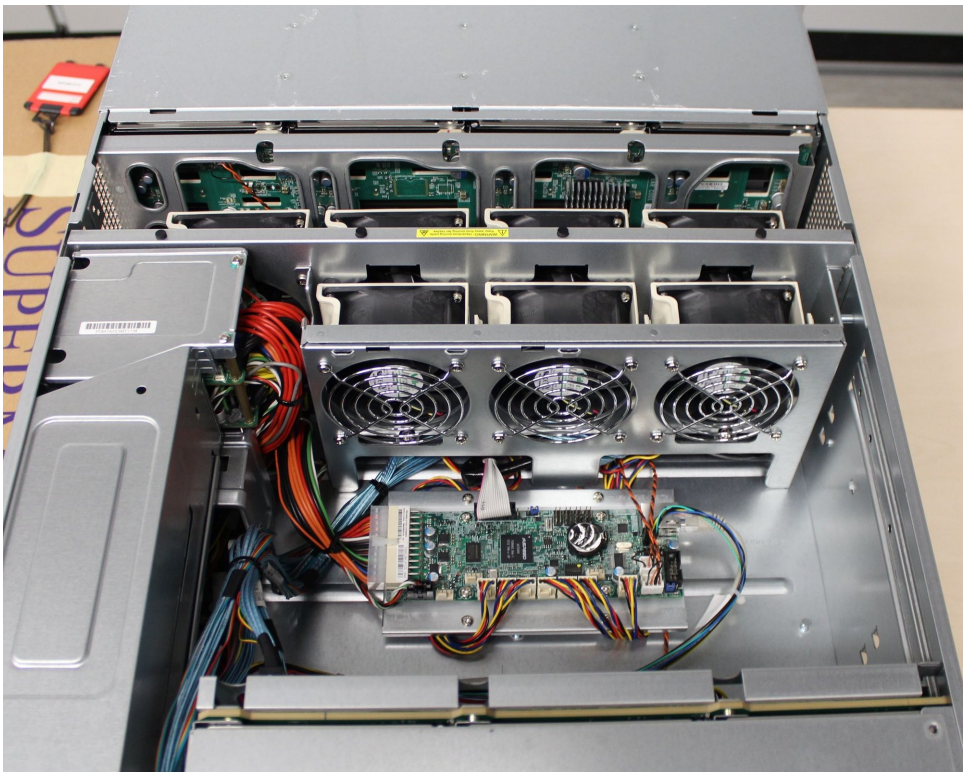

Abbildung 29: Hardware des Storage-Nodes

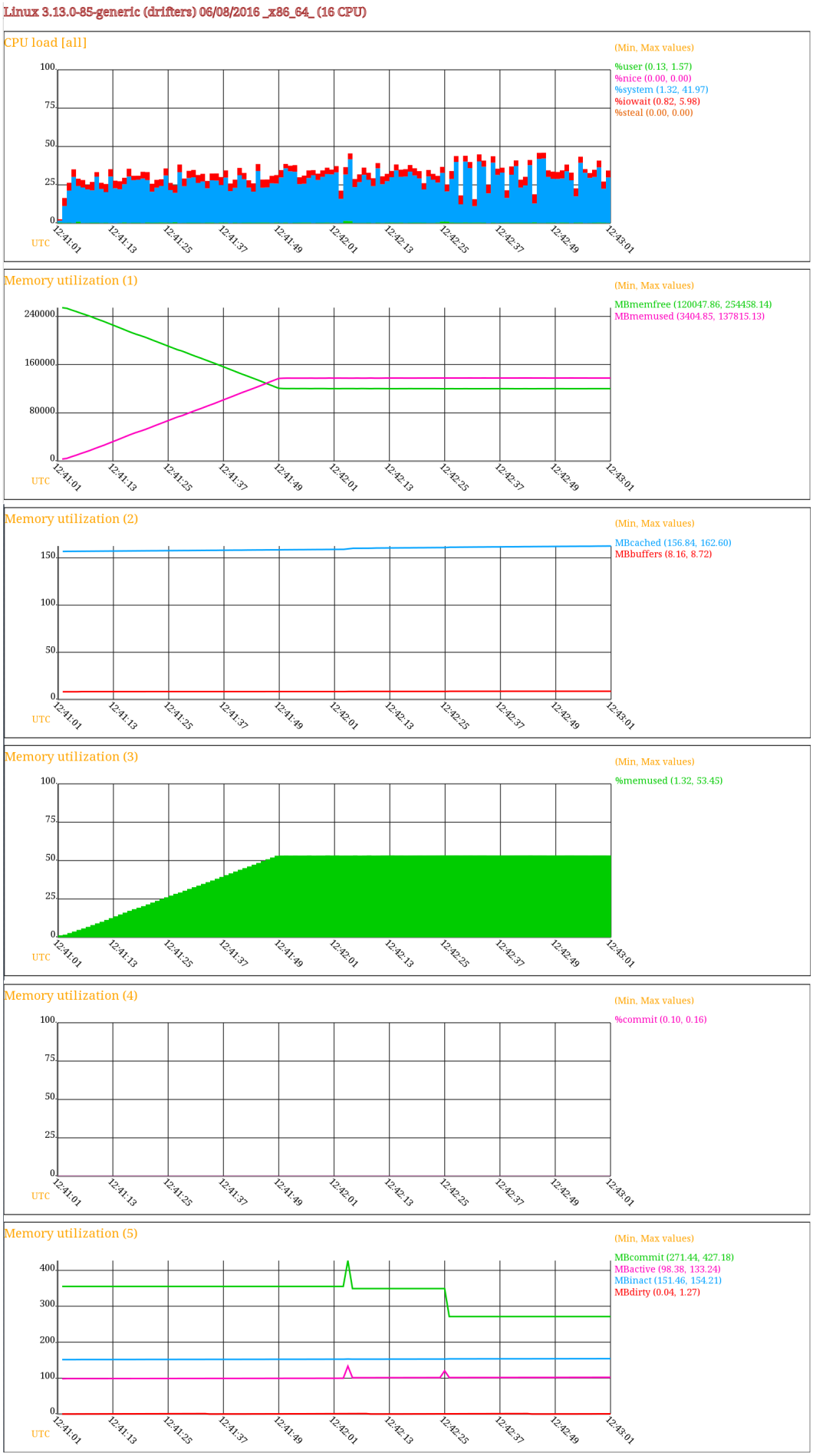

Abbildung 30: Mit *sadf* erstelltes SVG

B Listings

Listing 8: Skript - fragments.awk [1]

```
/Indirect blocks/{
file_number++;
next_block = 0;
}
/L0/{
split($3,fields,":");
this_block = ("0x"fields[2])+0;
this_block_size = ("0x"fields[3])+0;
total_blocks++;
if( next_block != 0 ) {
if( next_block == this_block )
not_fragmented++;
else
fragmented++;
}
next_block = this_block + this_block_size;
}
END{
printf("There are %d files.\n", file_number );
total_fragment_blocks = fragmented + not_fragmented;
printf("There are %d blocks and %d fragment blocks.\n", total_blocks,
     total_fragment_blocks );
printf("There are %d fragmented blocks (%2.2f%%).\n", fragmented, fragmented*100.0/
     total_fragment_blocks );
printf("There are %d contiguous blocks (%2.2f%%).\n", not_fragmented, not_fragmented
     *100.0/total_fragment_blocks );
}
```

Listing 9: Fio-Konfigurationsdatei für den Standardbenchmark

```
[global]
ioengine=libaio
bs=128k
directory=/ZFS/
refill_buffers
randrepeat=0
fallocate=none

[benchmark1]
name=benchmark1
rw=write
numjobs=10
size=100G
```

Listing 10: Fio-Konfigurationsdatei mit mehreren Clients

```
[global]
ioengine=libaio
bs=128k
directory=/ZFS/
refill_buffers
randrepeat=0
fallocate=none

[client1]
startdelay=0
name=benchmark1
rw=write
numjobs=1
size=1000G


[client2]
startdelay=30
name=benchmark2
rw=write
numjobs=5
size=100G


[client3]
startdelay=60
name=benchmark3
rw=write
numjobs=10
size=10G
```

Listing 11: Anlegen eines Pools aus mehreren RAID-Z2

```
zpool create -m <MOUNTPOINT> <POOLNAME> raidz2 /dev/sda /dev/sdb /dev/sdc /dev/sdd \
    /dev/sde raidz2 /dev/sdf /dev/sdg /dev/sdh /dev/sdi /dev/sdj raidz2 /dev/sdk \
    /dev/sdl /dev/sdm /dev/sdn /dev/sdo raidz2 /dev/sdp /dev/sdq /dev/sdr /dev/sds \
    /dev/sdt raidz2 /dev/sdu /dev/sdv /dev/sdw /dev/sdx /dev/sdy raidz2 /dev/sdz \
    /dev/sdaa /dev/sdab /dev/sdac /dev/sdad raidz2 /dev/sdae /dev/sdaf /dev/sdag \
    /dev/sdah /dev/sdai raidz2 /dev/sdaj /dev/sdak /dev/sdal /dev/sdam /dev/sdan
```

Selbstständigkeitserklärung

Ich versichere, dass ich die vorliegende Arbeit ohne fremde Hilfe selbstständig verfasst und nur die angegebenen Quellen und Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht. Die Arbeit wurd bisher in gleicher oder ähnlicher Form weder veröffentlicht, noch einer anderen Prüfungsbehörde vorgelegt.