ZFS on Linux Performance Evaluation

Norbert Schramm 6146299

Masterprojekt
Fachbereich Informatik
Universität Hamburg

30. März 2016

Inhaltsverzeichnis

1. Einleitung

Zur Speicherung von Daten auf einem Computer ist ein Dateisystem notwendig, dass die Daten korrekt speichert, verwaltet und das wieder abrufen ermöglicht. Hierbei kann man in die verfügbaren Dateisysteme in zwei Gruppen unterteilen, die klassischen und die modernen Dateisysteme. Zu den Klassischen zählen bekannte und weit verbreitete Dateisysteme, wie ext4 unter Linux oder NTFS unter Windows. Sie sind seit Jahren etabliert und ermöglichen es, Daten zuverlässig zu speichern. Darüber hinaus sind in der jüngeren Zeit allerdings neue Herausforderungen hinzugekommen, die ein modernes Dateisystem erfüllen muss. Unter Linux gibt es hier zum Beispiel im Zusammenhang mit RAID-Verbünden Probleme, wie etwa das langsame wiederherstellen größerer Plattenverbünde oder das Write-Hole-Problem, welches zur Dateninkonsistenz führen kann.

Moderne Dateisysteme, wie ZFS besitzen einen erweiterten Funktionsumfang, um die Möglichkeiten des Dateisystems zu erweitern oder die Datensicherheit zu erhöhen (siehe Kapitel 1.2. In diesem Fall beschäftigen wir uns mit dem Dateisystem ZFS. Da dieses ursprünglich von Sun Microsystems für ihr Betriebssystem Solaris entwickelte Dateisystem erst auf andere Betriebssysteme portiert werden musste, kann es zu Leistungsunterschieden kommen. Zusätzlich kommt hinzu, dass der Code von ZFS unter der *Common Development and Distribution License* (CDDL) von Sun Microsystems veröffentlicht wurde. Diese ist inkompatibel zu der von Linux verwendeten *GNU General Public License* (GPL), weshalb eine direkt in die Kernelsourcen integrierte Linux-Implementierung nicht möglich ist. Das Projekt *ZFS on Linux* (ZoL) baute daher den Funktionsumfang in Form eines Kernelmoduls für Linux nach. Seit April 2013 gilt diese Version als stable release und hat somit das experimentelle Stadium hinter sich gelassen und zur produktiven Nutzung freigegeben.

Im Kapitel 2 führe ich daher einen Vergleich von drei unterschiedlichen unixoiden Betriebssystemen (Illumos, FreeBSD, Ubuntu) durch, die ZFS unterstützen, um zu untersuchen, wie performant die Portierung von ZoL im Vergleich zur originalen Version ist.

Im Kapitel 3 wird mit Hilfe des Programms *wrstat* die ZoL-Version unter Ubuntu genauer betrachtet.

Das Netzwerkdateisystem Lustre basiert auf dem lokalen Dateisystem ldiskfs, welches eine Weiterentwicklung von ext4 ist. Seit dem stable release von ZoL gibt es unter Lustre auch die Möglichkeit, ZFS als lokales Dateisystem zu verwenden. Diese Kombination wird in Kapitel 4 kurz betrachtet.

1.1. ZFS Entwicklung und Geschichte

Der Name ZFS stand ursprünglich für den Namen *Zettabyte File System* und wurde von Sun Microsystems seit 2001 als closed source Dateisystem für das hauseigene Betriebssystem Solaris entwickelt [3]. Zusammen mit der Umwandlung von Solaris in OpenSolaris in ein Opensource-Projekt unter der ebenfalls von Sun entwickelten CDDL wurde ZFS 2005 veröffentlicht. Nach der Übernahme von Sun durch Oracle wurden die Arbeiten an OpenSolaris 2010 eingestellt. Zur Weiterentwicklung wurde der ZFS-fork OpenZFS ge-

gründet, dessen Mitglieder unter anderem aus Entwicklern der Betriebssysteme Illumos, FreeBSD oder einiger Linux-Distributionen bestehen.

Bereits 2006 gab es die erste FUSE-Implementierung, um ZFS unter Linux nutzen zu können. Da ZFS allerdings eher für den Einsatz in leistungsstarken Umgebungen entwickelt wurde und etwas mehr Rechenleistung der CPU benötigt, als klassische Dateisysteme, verbreitete sich diese Lösung nicht sehr stark. Größter Nachteil von FUSE-Dateisystemen ist der verringerte Datendurchsatz, da die Implementierung im Userspace erfolgt.

Pawel Jakub Dawidek portierte 2008 mit Hilfe von Sun-Entwicklern ZFS in das Betriebssystem FreeBSD, welches ZFS in der Version 7.0 als experimentelles Dateisystem und in Version 8.0 als stabile Variante nutzt. Diese Version ist direkter Bestandteil des Betriebssystems.

Nachdem die Arbeiten an Solaris 2010 eingestellt wurden, nahm Illumos als neue Referenzplattform für ZFS seinen Platz ein. Viele Abwandlungen von ZFS auf anderen Betriebssystemen stammen von der unter Illumos genutzten ZFS-Version ab.

Das Projekt ZFS on Linux umgeht das Problem der unterschiedlichen Lizenzen (CDDL, GPL) und pflegt eine eigene Portierung von ZFS. Diese kann einfach heruntergeladen und das Linux-Dateisystem damit gepatched werden, um ZFS direkt als performantes Kernel-Modul nutzen zu können. Seit der Version 0.6.1 ist ZoL für den produktiven Einsatz freigegeben [1]. Laut der Aussage eines Hauptentwicklers aus dem Jahr 2014 liegt ZoL 0.6.3 effektiv nur noch nur noch 18 Funktionen hinter der Referenz zurück"[10] wovon es für neun jedoch bereits Workarounds gibt und die restlichen neun mit der folgenden Version 0.6.4 implementiert werden sollen. Die aktuelle Versionnummer ist 0.6.5.6, womit davon ausgegangen werden kann, dass ZoL im Funktionsumfang der Illumos-Version in nichts nachsteht. Dennoch war es bei meinen Messungen in einem Versuch noch nicht möglich, einen unter Illumos erstellten ZFS-Pool unter Linux zu importieren. Das Projekt OpenZFS hat unter anderem diese Interoperabilität zum Ziel. Die Version 1.0 soll erreicht werden, wenn die ioctl-Schnittstelle */dev/zfs* fertig entwickelt und stabil ist.

Im laufe der Entwicklung wurden im originalen ZFS weitere Funktionen hinzugefügt, wie verschiedene Kompressionsalgorithmen und komplexere Partitäts-modi für größere Arrays (siehe folgendes Kapitel).

1.2. Funktionen von ZFS

Dieses Kapitel beinhaltet eine kurze Auflistung und Erklärung einiger besonderer Funktionen, die für ZFS besonders sind und es von klassischen Dateisystemen unterscheiden.

Ein wesentliches Merkmal von ZFS sind **128bit** lange Pointer, mit denen Dateien adressiert werden. Dies ermöglicht es, eine rechnerische Menge von 2^{128} Bytes gespeichert werden. Da die meisten Betriebssysteme jedoch aktuell 64bit-Systeme sind, ist der aktuell verfügbare Adressraum auf 2^{64} Bytes beschränkt, was 16 EiB entspricht. Bei späteren Implementationen ist eine Erweiterung auf 128 Bit ohne Probleme möglich. Bisher werden die ersten 64 Bit einfach mit 0 dargestellt. Die maximale Dateigröße beträgt ebenfalls 2^{64} Byte, die maximale Anzahl möglicher Dateien im Dateisystem und maximale Anzahl an Ordnern ist auf 2^{48} begrenzt. Laut Marketingaussagen des Chefentwicklers Jeff Bonwick

würde das vollständige Füllen eines 128 Bit großen Speicherpools mehr Energie benötigen als für das Verdampfen der Weltmeere von Nöten wäre[2]. Diese Aussauge untermauert auch die Philosophie von ZFS, ein vorausschauendes und langlebiges Dateisystem zu werden.

Die **Datenintegrität** spielt bei ZFS ebenfalls eine Große Rolle. Für jede Datei wird ein Hashwert erstellt und auf dem selben Speichermedium abgelegt. Von jedem Ordner und dem Ordner darüber werden ebenfalls Hashes berechnet, sodass es am oberen Ende einen Master-Hash gibt, der für die Datenintegrität aller unter ihm gespeicherten Daten garantiert. Fehlerhafte Daten können so erkannt werden und bei redundanter Speicherung selbstständig behoben werden. Daher spricht man bei ZFS gelegentlich auch von einem selbstheilenden Dateisystem. Der Overhead zur Berechnung der Prüfsummen und der dafür zusätzlich benötigte Speicherplatz sind nur minimal, sodass die IO-Leistung von ZFS nur minimal schlechter ist, als von einem ext4-Dateisystem [8]. Die Garantie, zu jedem Zeitpunkt (auch nach einem Stromausfall) ein konsistentes Dateisystem zu haben, macht Tests, wie *fsck* in klassischen Dateisystemen überflüssig.

Zur Erhöhung der Redundanz in einem Array aus mehreren Festplatten werden häufig **RAID-Verbünde** genutzt. Gerade bei großen Verbünden, wie RAID-5 oder RAID-6 mit einer bzw. zwei Paritätsfestplatten, dauert das erstellen und wiederherstellen eines Arrays zum Teil sehr lange, da für jedes Bit eine Parität berechnet werden muss. Dies ist dem Umstand geschuldet, dass Volumemanager (z.B. *mdadm*) und Dateisystem nicht kommunizieren. ZFS kombiniert die Funktionen beider und arbeitet dadruch viel effizienter. So muss ein RAID-5 äquivalent (unter ZFS *raidz* genannt) nicht initialisiert werden, sondern kann direkt genutzt werden. Die Paritäten für geschriebene Daten werden erst während des Schreibvorgangs berechnet. Bei einer Wiederherstellung nach einem Ausfall müssen so auch nur vorhandene Daten wiederhergestellt werden und nicht sämtliche Blöcke des Arrays. In der aktuellen Version beherrscht ZFS mit dem Modus *raidz3* sogar eine 3-fache Parität. Andere RAID-Modi, wie 0,1,10 werden ebenfalls äquivalent unterstützt.

Durch die Unterteilung in **Storage-Pools** können Unterverzeichnisse mit unterschiedlichen Eigenschaften (Kompression, Deduplikation) ausgestattet werden. Des weiteren können verschiedene RAID-Konfigurationen zu einem Pool zusammengesetzt werden, zum Beispiel einer raidz und ein raidz2 zu einem Storage-Pool. Zusätzlich können zu einem Pool auch noch schnelle Speichermedien als Read-Cache (L2ARC) oder Write-Log (ZIL) zugeordnet werden, um die Leistung des Pools zu erhöhen.

Ebenfalls zur Datensicherheit trägt die **Copy on Write** Funktionalität von ZFS bei. Wird eine Datei gelesen, modifiziert und gespeichert, wird die neue Version an eine freie Stelle des Dateisystems geschrieben. Erst nach dem erfolgreichen Schreiben wir die alte Datei als inaktiv markiert und der Pointer auf die neue Datei umgeschwenkt. Somit kann eine Datei durch einen Systemabsturz nicht zerstört werden. Durch CoW wird ebenfalls das vom RAID-5 bekannte Write-Hole-Problem behoben, bei dem eine Datei trotz Parität zerstört werden kann, wenn das System während des Schreibvorgangs abstürzt.

Hand in Hand mit CoW gehen auch *Snapshots*. Diese ermöglichen eine platzsparende Versionierung des Dateisystems. Wird ein Snapshot erstellt, werden alle Daten eingefroren und nur neu geschriebene oder modifizierte Dateien benötigen weiteren Speicherplatz auf dem Pool.

Mit der Einführung von **Deduplikation** können bei ähnlichen Daten große Mengen an Speicherplatz gespart werden. Hierbei wird überprüft, ob identische Datenblöcke bereits auf der Festplatte liegen. Ist dies der Fall, so wird für die neue Datei lediglich ein referenzierende Pointer auf die bereits bestehende Datei geschrieben und der Speicherplatz komplett eingespart. Da dieses Verfahren jedoch sehr viel RAM benötigt (rund 5 GB freien Hauptspeicher pro TB Speicherplatz [16]), ist es nur bedingt sinnvoll. Pro Datenblock werden 320 Byte RAM benötigt. Nehmen wir an, dass 5 TB Daten eine mittlere Blockgröße von 64 kb haben, so entspricht dies 78125000 Blöcken. Multipliziert mit 320 Byte ergibt das rund 25 GB für die Dedup-Tabelle.

Durch transparente **Kompression** durch das Dateisystem können je nach zu speicherndem Datentyp große Mengen an Speicherplatz eingespart werden. Wird eine Datei auf die Festplatte geschrieben, werden sie vom Prozessor komprimiert und dann auf den Datenträger geschrieben. Dies ermöglicht bei gut komprimierbaren Daten nicht nur einen verminderten Speicherplatzbedarf sondern auch Datenraten, die höher sind als die physikalischen Lese-/Schreibraten der Festplatte, siehe Kapitel 2.2.

2. ZFS im Vergleich

Da der Funktionsumfang von allen getesteten ZFS-Versionen gleich ist, kann die Performance der Funktionen direkt mit einander vergleichen werden. Schwerpunkt lag hierbei neben dem eigentlichen Vergleich der unterschiedlichen Betriebssystemversionen auch die Untersuchung der Funktionen Kompression und Deduplikation.

Da der erste Test auf zu alter Hardware stattfand, habe ich mit insgesamt drei verschiedenen Computern die Tests auf allen drei Betriebssystemen ausgeführt, um so gleichzeitig eine Abhängigkeit der Leistung von ZFS in Abhängigkeit von der verwendeten Hardware zeigen zu können. Trotz einer unterschiedlichen Festplatte in einem der drei Systeme ist dennoch eine Tendenz zur Abhängigkeit vom Prozessor zu erkennen.

2.1. verwendete Komponenten

2.1.1. Hardware

Insgesamt wurden drei Computer verwendet. Ein Rack-Server Dell R710, eine Workstation von Fujitsu-Siemens-Computers und ein normaler Desktop-Computer. Die Systeme sind der Leistung nach aufsteigend aufgezählt.

System 1: Die Workstation von FSC hat einen Intel Xeon E3110 Prozessor mit 2x3.0 GHz Takt. Der Prozessor stammt von Intels Core2-Architektur ab und ist seit Q1'08 verfügbar. Er ist damit der älteste getestete Prozessor. Es sind 8 GB RAM verbaut. Für die getestete Festplatte wurde eine Western Digital Black 1 TB verwendet, welche direkt an einem SATA-Port am Server angeschlossen war.

System 2: Der Dell-Server hat einen Intel Xeon X5677-Prozessor mit 4 Kernen und 3.46 GHz Taktung. Die CPU stammt aus der Nehalem-Architektur und ist somit der zweiäl-

teste Prozessor (Veröffentlicht: Q1'10). Hier waren 32 GB RAM verbaut. Die Festplatte war eine 136 GB 15K SAS-Festplatte, welche über den integrierten RAID-Controller PERC6/i als RAID-0 mit einer Festplatte durchgereicht wurde.

System 3: Das modernste System im Test basiert auf einem Intel Corei-5 2500k mit 4x3.3 GHz Takt. Er stammt aus der Sandy Bridge-Serie von Intel und ist somit eine Generation neuer, als das vorherige Modell (Veröffentlicht: Q1'11). Als Arbeitsspeicher wurden wieder 8 GB RAM verwendet. Es wurde die gleiche WD Black 1 TB Festplatte an einem SATA-Port verwendet, wie im ersten System.

2.1.2. Betriebssysteme

Für einen Vergleich wurden drei unixoide Betriebssysteme verwendet:

OpenIndiana: OpenIndiana ist ein auf Illumos basierendes Betriebssystem. Da Illumos nach der Einstellung der Entwicklung von OpenSolaris als neue Referenzplattform für ZFS gilt, ist OpenIndiana ein Betriebssystem aus Solaris-Basis, für das ZFS ursprünglich entwickelt wurde. Genutzt wurde die Version oi_151a8 von OpenIndiana auf Basis von SunOS 5.11 von Juli 2013. ZFS ist als Bestandteil von OpenIndiana bereits vorinstalliert. Im folgenden die Ausgabe des Befehls *uname -a*

```
SunOS openindiana 5.11 oi_151a8 i86pc i386 i86pc Solaris
```

FreeBSD: Als Vertreter der Portierungen mit direktem Einbau in das Betriebssystem wurde FreeBSD gewählt. Die sehr offene BSD-Lizenz ermöglichte es, ZFS direkt zu implementieren, weshalb ZFS seit Version 8.0 fester Bestandteil des Betriebssystems ist. FreeBSD gilt als ein sehr performantes und stabiles Serverbetriebssystem, besonders im Bereich der Paketverarbeitung im Netzwerk. ZFS ist als Bestandteil von FeeBSD bereits vorinstalliert. Genutzt wurde die aktuelle Version 10.2 mit der folgenden Ausgabe von *uname -a*

```
FreeBSD freebsd 10.2-RELEASE FreeBSD 10.2-RELEASE #0 r286666: Wed Aug 12
15:26:37 UTC 2015 rootreleng1.nyi.freebsd.org:/usr/obj/usr/src/sys/GENERIC
amd64
```

Ubuntu: Für den Test von ZFS on Linux wurde Ubuntu verwendet. Es ist einer der bekanntesten Vertreter von Linux und basiert auf Debian. ZoL kann direkt über Paketquellen einfach installiert werden. In der kommenden Version 16.04 ist es trotz Lizenzproblemen geplant, ZFS direkt mit aufzunehmen. Verwendet wurde Ubuntu 14.04.3 LTS. Der Kernel wurde mit dem Parameter CONFIG_LOCK_STAT=y"bereits neu kompiliert, um für die Tests im nächsten Kapitel mit wrstat vorbeireitet zu sein. Auf die Leistung von ZFS gegenüber dem originalen Kernel hat dies bei dieser Testmessung keinen Einfluss.

```
Linux ubuntu 3.13.11-ckt33 #1 SMP Tue Mar 8 20:38:30 CET 2016 x86_64 x86_64
x86_64 GNU/Linux
```

Da ZFS unter Ubuntu nicht vorinstalliert ist, musste ich etwas Aufwand betreiben, um ZFS unter dem modifizierten Kernel lauffähig zu bekommen. Das vollständige Listing für die Installation befindet sich im Anhang unter A.1 Als erstes wurden sowohl die Solaris Porting Layer (SPL) als auch die ZFS-Sources aus dem Git-Repository von ZFSonLinux heruntergeladen und laut Listing installiert 5. Danach wurden die benötigten Pakete für das Kernel-Patching runtergeladen und installiert 6. Hier wurde in der Kernel-config die Zeile CONFIG_LOCK_STAT=yëinkommentiert bzw. editiert und anschließend neu kompiliert. Nach einem Neustart wurde der neue Kernel geladen.

Um ZFS unter diesem lauffähig zu bekommen, musste SPL erneut installiert werden. Für die Installation von ZFS waren noch zwei Änderungen notwendig. Ein scheinbar bekanntes Problem ist, dass das Kompilieren von ZFS versagt, sobald der Kernel modifiziert wurde [15]. Deshalb muss die Datei *META* editiert werden und in einem nicht ganz sauberen Workaround die Lizenz von *CDDL* auf *GPL* geändert werden.

Des Weiteren muss eine Quelldatei *include/linux/vfs_ compat.h* von ZFS editiert werden (siehe Listing 7), da das kompilieren sonst fehlschlägt. Bei einem originalen Kernel würde diese Schleife nicht ausgeführt werden. Da der Kernel aber modifiziert wird, muss die Funktionalität der Schleife invertiert werden, da es sich im Wesentlichen noch um den originalen Kernel handelt. Nun ist die Installation von ZFS ebenfalls möglich und auf dem modifizierten Kernel lauffähig.

Auf allen Systemen wies der Befehl *zdb* die Version 5000 von ZFS aus.

2.1.3. Benchmarks

Für den Vergleich wurde hauptsächlich das Programm bonnie++ verwendet, welches auf allen drei Plattformen verfügbar war. Die Benchmarks mit dem Programm iozone konnten auf Grund von Zeitmangel nur auf dem Server 2 (Dell) ausgeführt werden und wurde deshalb nicht in die Auswertung aufgenommen.

Bonnie++ ist ein Festplattenbenchmark, welcher sich im wesentlichen in vier Schritte einteilen lässt [4]:

- *Write:* sequentielles Schreiben einer Datei auf die Festplatte mit dem `write`-Befehl
- *ReWrite:* jede BUFSIZ der Datei wird mir `read` gelesen, modifiziert und mit `write` geschrieben, wofür ebenfalls ein `lseek` notwendig ist. Da kein neuer Speicherplatz verbraucht wird, wird der Cache des Dateisystems so getestet
- *Read:* sequentielles auslesen der Datei mit `read`
- *File Creation Tests:* erstellen, ändern und entfernen einer großen Zahl von Verzeichnissen ohne Inhalt

Die ersten drei Punkte bilden den Schwerpunkt meiner Analyse. Die komplette Datei mit allen Messwerten aus bonnie++ befindet sich im Anhang des Projekts. Eine Auswertung mit *hexdump* der geschriebenen Daten von bonnie++ ergab, dass es sich um

einfache Muster handelt, die größtenteils aus Nullen bestehen. Daher ist die Kompressionsrate übermäßig stark (ca um den Faktor 130x) und nicht die Festplatte, sondern die Kompression auf der CPU der Bottleneck. Die Ergebnisse dieses Benchmarks sind somit nicht realistisch. Ein Test zur Kompressionsrate von wirklichkeitsnahen Beispieldaten wurde im Kapitel 3.3 durchgeführt.

Die Installation (als root) wurde wie folgt durchgeführt:

```
pkgadd -d http://get.opencsw.org/now
/opt/csw/bin/pkgutil -U
/opt/csw/bin/pkgutil -y -i bonnie++
/usr/sbin/pkgchk -L CSWbonnie++ # list files
```

Listing 1: Installation auf OpenIndiana

```
pkg install bonnie++
```

Listing 2: Installation auf FreeBSD

```
apt-get install bonnie++
```

Listing 3: Installation auf Ubuntu

Bonnie++ wurde mit folgenden Parametern aufgerufen:

```
bonnie++ -d /tank/ -u root -x 1 -z 0 -q -n 1024 » bench.csv
```

- *-d /tank/*: Zielverzeichnis, in dem getestet werden soll
- *-u root*: Wird das Programm als root ausgeführt, muss diese Option explizit gesetzt werden
- *-x 1*: Anzahl der Durchläufe
- *-z 0*: Der Seed des Benchmarks ist 0, damit bei jedem Durchlauf die selben Bedingungen mit dem selben Seed herschen
- *-q*: Quiet, keine grafische Ausgabe während des Tests
- *-n 1024*: Anzahl n der n*1024 zu erstellenden Dateien für den File Creation Test
- *» bench.csv*: Speichern der Messergebnisse in eine csv-Datei

Für jede einzelne Messung mit bonnie++ wurde der ZFS-Pool neu erzeugt und nach der Messung wieder gelöscht. Das folgende Listing 4 zeigt den Ausschnitt für einen Durchlauf unter Ubuntu. Zuerst wird der Pool erstellt und ein Eintrag in die bench.csv gemacht,

was gerade getestet wird. Dann werden die jeweiligen Einstellungen vorgenommen, bonnie++ ausgeführt und der Pool wieder zerstört.

```
zpool create tank /dev/sdb
echo "atime=on dedup=off compression=off" >> bench.csv
bonnie++ -d /tank/ -u root -x 1 -z 0 -q -n 1024 >> bench.csv
iozone -Mce -a -s 1g -f /tank/file -i0 -i1 -i2 -b out1.xls
zpool destroy tank
```

Listing 4: Installation auf FreeBSD

Insgesamt wurden 15 Durchläufe gemacht mit folgenden Einstellungen am ZFS-Pool:

- keine Modifikationen
- deaktivieren der atime für schnelleres lesen (bei allen folgenden Messungen ebenfalls deaktiviert)
- Deduplikation
- LZ4-Kompression
- LZJB-Kompression
- GZip-Kompression der Stufen 1-9
- Deduplikation & LZJB-Kompression

2.2. Messergebnisse

Im Wesentlichen werde ich mich auf die Messergebnisse des neusten Prozessors beschränken, da dieser in der Praxis die höchste Relevanz hat.

Die Abbildung 1 zeigt einen neu erstellten Pool ohne zusätzliche Modifikationen an.

Zu sehen ist ein relativ gleichmäßiges Ergebnis, in dem alle drei Betriebssysteme in etwa gleichauf liegen. Die Schreib- und Leserate entspricht ungefähr der maximal möglichen Datenrate der Festplatte. Lediglich FreeBSD sticht in der Leseleistung heraus und schafft über 20 MB/s mehr Durchsatz. Gründe für die Unterschiede sind möglicherweise das unterschiedliche Handling der Daten durch das jeweilige Betriebssystem, da der jüngste gemeinsame Vorfahre aller drei Systeme Unix aus den 80er Jahren ist.

Ein sehr einfaches Mittel zur möglichen Steigerung der Leseleistung eines Betriebssystems ist das deaktivieren der *atime*. Bei einem Lesezugriff auf eine Datei wird neben dem reinen Lesezugriff auch ein Schreibzugriff nötig um den Wert atime mit der aktuellen Zeit zu aktualisieren. Diese Information nutzen jedoch nur wenige Anwendungen, wie zum Beispiel Mailprogramme. Zusätzlich verlangsamt der Schreibprozess in der Theorie die Leserate. In Abbildung 2 ist daher ein Vergleich der Leseleistung mit aktiviertem und deaktivierten atime-Parameter.

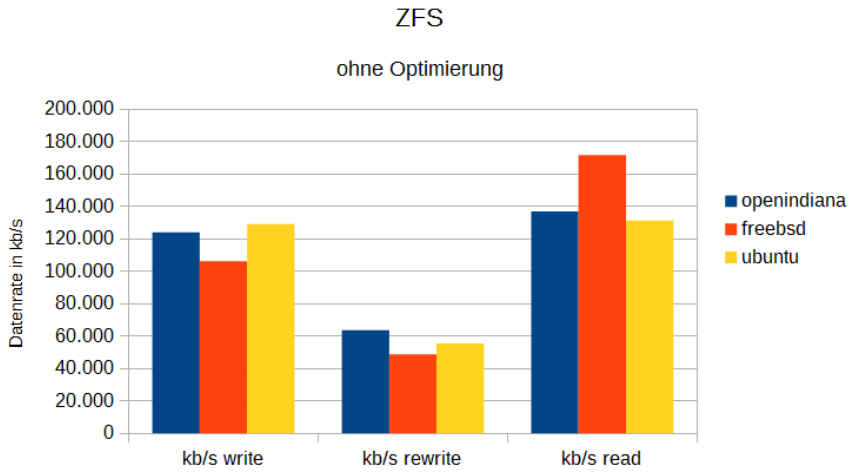

Abbildung 1: System 3, compression=off, dedup=off, atime=on

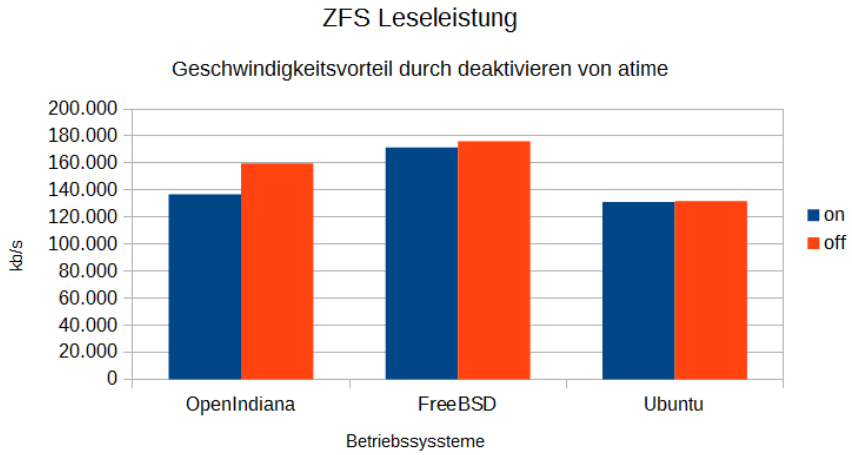

Abbildung 2: System 3, compression=off, dedup=off, atime=off

Hier fällt auf, dass OpenIndiana durch diese Änderung sofort einen Leistungsgewinn von über 20 MB/s verzeichnen kann. Auch bei FreeBSD ist ein leichter Leistungsgewinn zu sehen. Ubuntu zieht aus dieser Optimierung keinen Vorteil. Das könnte daran liegen, dass seit Ubuntu 10.04 standardmäßig der Wert *relatime* verwendet wird, welcher unnötige Schreibzugriffe auf den atime-Wert verhindert. Für die Schreibleistung bringt diese Optimierung keine Vorteile.

Das aktivieren der Kompression kann - je nach Art der Daten - einen enormen Geschwindigkeitsvorteil bringen. In Abbildung 3 ist die LZ4-Kompression aktiviert. Dieser Kompressions-Algorithmus wird für die Verwendung von ZFS als sog. *Immer-an-Feature* beworben. Hintergrund ist, dass der Algorithmus eine relativ gute Datenkompression bei relativ geringem CPU-Overhead bietet.

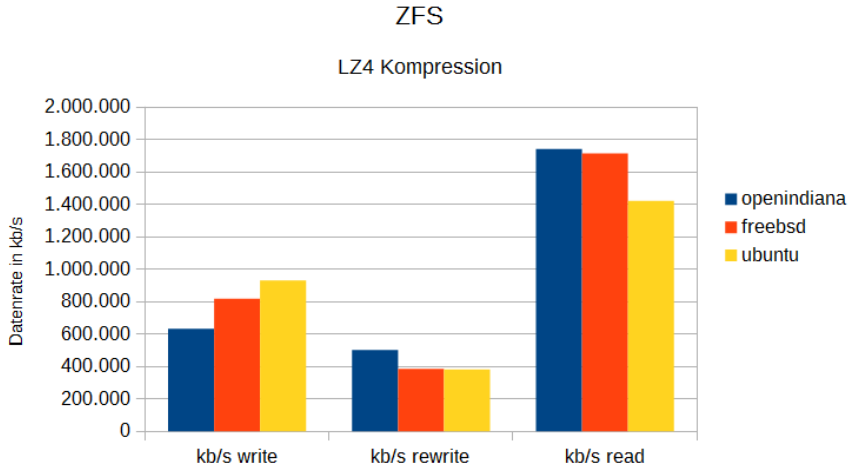

Abbildung 3: System 3, compression=lz4, dedup=off, atime=off

Zusätzlich zur Kompression durch LZ4 wird bei aktivierter Komprimierung noch eine sog. Zero-Length-Elimination vorgenommen. Das bedeutet, dass lange Kolonnen aus Nullen in der Datei bereits vor der Eigentlichen Kompression zusammengefasst werden. Dies und die gute Komprimierbarkeit der Testdaten von bonnie++ sorgen für einen deutlich höheren Durchsatz, als der maximal mögliche physikalische Durchsatz der Festplatte. Die Abbildung zeigt, dass beim schreiben Ubuntu den höchsten Durchsatz erzielt, während OpenIndiana beim lesen die schnellsten Werte erzielt. Auch der rewrite-Prozess wird dadurch beschleunigt.

Die Deduplikation arbeitet auf Block-Basis. Da während des Schreibvorgangs die Hashes der Blöcke berechnet werden müssen, sinkt entsprechend die Schreibrate, wie in Abbildung 4 zu sehen. Die Leseleistung entspricht im wesentlichen der eines Pools ohne Deduplikation, da der Leseprozess keinerlei Rechenleistung erfordert. Auch hier sind alle drei Betriebssysteme im wesentlichen gleichauf, wobei die Schreibleistung von OpenIndiana mit Abstand die wenigsten Einbrüche zu verzeichnen hat.

In der folgenden Abbildung 5 wurden sowohl Kompression und Deduplikation als auch

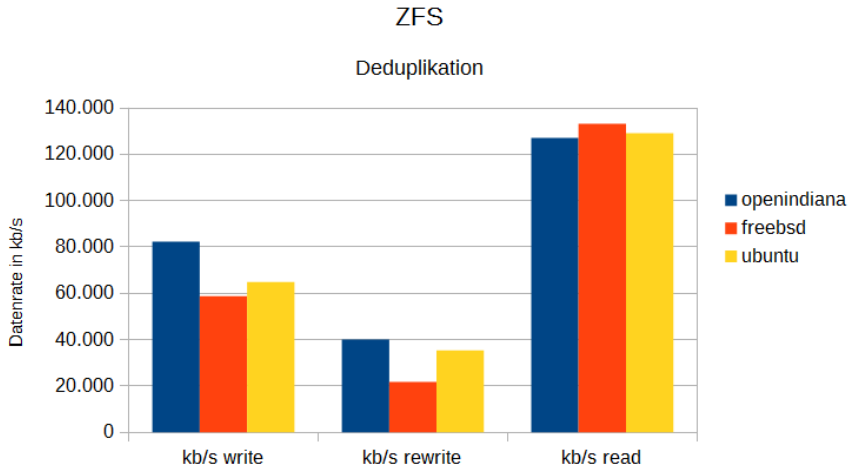

Abbildung 4: System 3, compression=off, dedup=on, atime=off

atime=off verwendet. Im direkten Vergleich zur darüber liegenden Abbildung 4 sieht man, dass die Verhältnisse in etwa gleich bleiben, kombiniert mit einer höheren Datenrate durch die Kompression. Ebenfalls zu sehen ist, dass in jedem der drei Tests ein anderes Betriebssystem das Feld anführt, jedoch alle drei insgesamt nahezu gleichauf liegen.

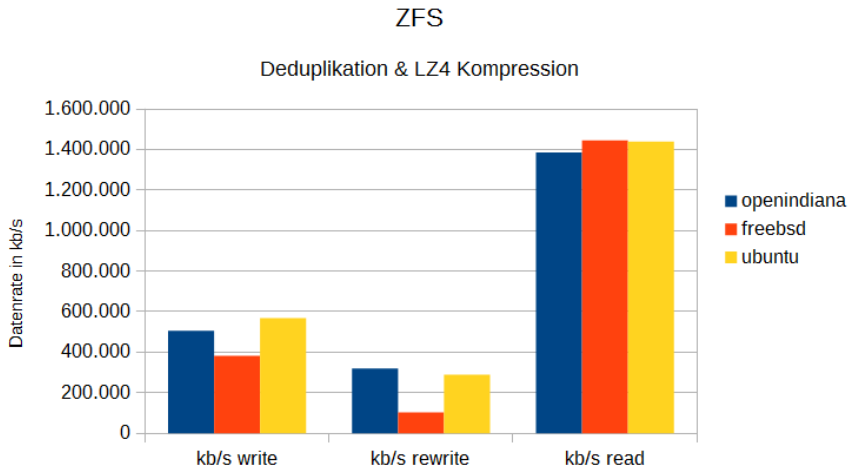

Abbildung 5: System 3, compression=lz4, dedup=on, atime=off

Da ZFS verschiedene Kompressions-Algorithmen anbietet, werden diese in den kommenden drei Abbildungen für Write (6), ReWrite (7) und Read (8) dargestellt. Den Anfang machen der bereits oben dargestellte LZ4 und LZJB-Algorithmus, welcher gleichzeitig der aktuelle Standard für Kompression unter ZFS ist. Danach folgen die verschiedenen Stufen von Gzip (1 entspricht der kleinsten Kompression). In allen Fällen war während

des Benchmarks der Prozessor der Flaschenhals. Während die Festplatte reell nur wenige MB/s Durchsatz hatte, war ein Kern des Prozessors durchgehend auf 100% Auslastung.

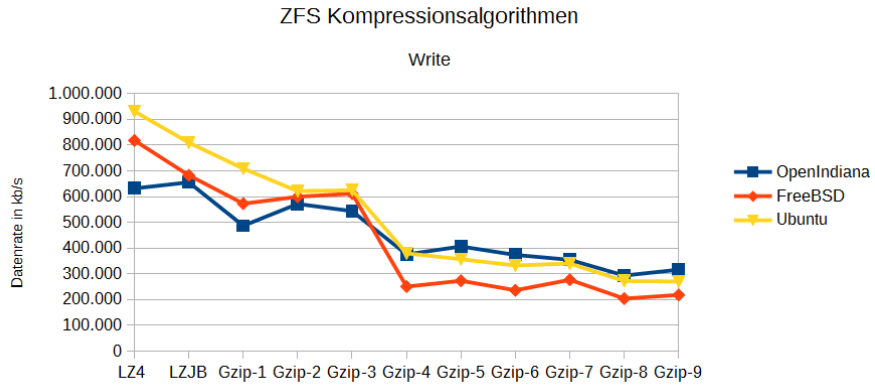

Abbildung 6: System 3, Schreibrate

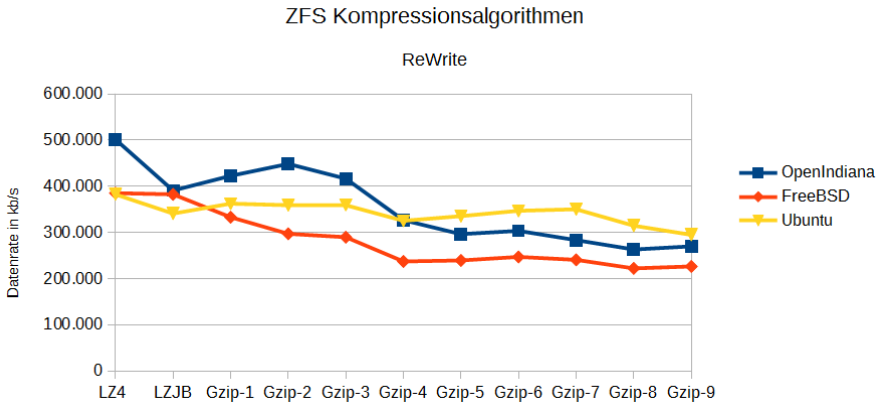

Abbildung 7: System 3, ReWrite-Rate

Zu sehen ist, dass alle drei Systeme sich ähnlich verhalten und leistungstechnisch nah bei einander liegen. In der Schreibrate ist Ubuntu fast immer das schnellste System oder zumindest gleichauf mit OpenIndiana. Einzige Ausreißer sind die GZip-Stufen 1 bis 3 unter OpenIndiana und FreeBSD in der Leseleistung. In den höheren GZip-Stufen hat Ubuntu die beste Leserate und FreeBSD eine deutlich schlechtere.

Vergleich unterschiedlicher Hardware

Als Ergänzung dazu setze ich zum direkten Vergleich der oben durchgeführten Benchmarks auf aktueller Hardware noch ein paar ausgewählte Vergleiche mit älterer Hardware, um die Abhängigkeit von ZFS im allgemeinen und ZFS on Linux im speziellen von

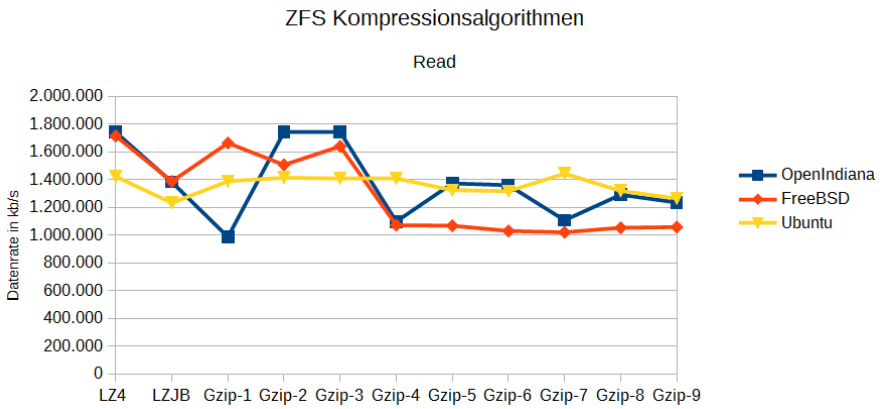

Abbildung 8: System 3, Leserate

leistungsstarker Hardware zu zeigen.

Dazu habe ich in Abbildung 9 den Test auf einem LZ4-komprimierten Pool mit einem drei Jahre älteren Xeon E3110-Dualcore-Prozessor mit alter Front-Side-Bus-Technologie ausgewählt.

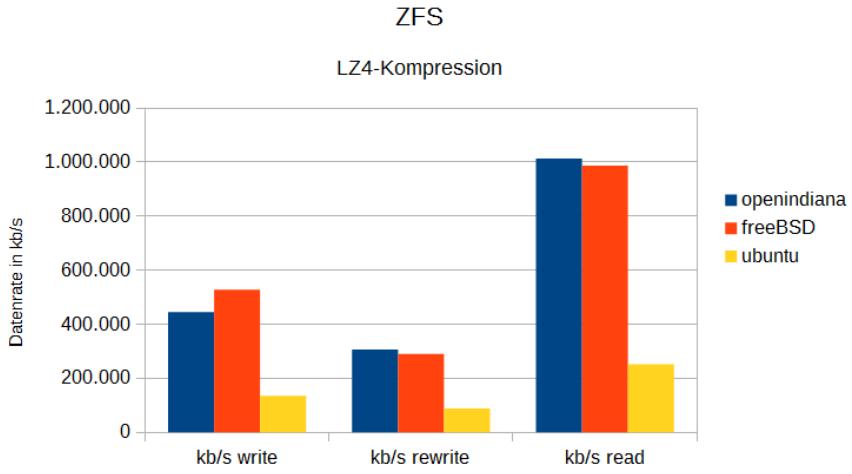

Abbildung 9: System 1, compression=lz4, dedup=off, atime=off

Als erstes fällt auf, dass der Prozessor nur rund 66% des Durchsatzes schafft (Die Anzahl der Kerne spielt in diesem Fall keine Rolle, da nur ein Kern den Datenstrom komprimiert). Das Verhältnis der Datenrate bei OpenIndiana und FreeBSD ist zwischen dem Xeon E3110 und dem neueren Corei5-2500k nahezu identisch. Anders sieht es allerdings bei Ubuntu aus, welches trotz identischer Festplatte und Software eine deutlich schlechtere Leistung aufweist. Einen Grund hierfür konnte ich leider nicht identifizieren, jedoch vermute ich unter ZFS on Linux eine intensivere Nutzung neuerer Funktionen

aktueller Prozessoren.

Als Vergleich von allen drei Hardwareplattformen kann leider nur ein Kompressions-Benchmark herangezogen werden, da in System 2 eine andere Festplatte verwendet wurde. Da diese bei einem Kompressionstest jedoch nicht der limitierende Faktor ist, sondern der Prozessor, ist dennoch ein Vergleich möglich.

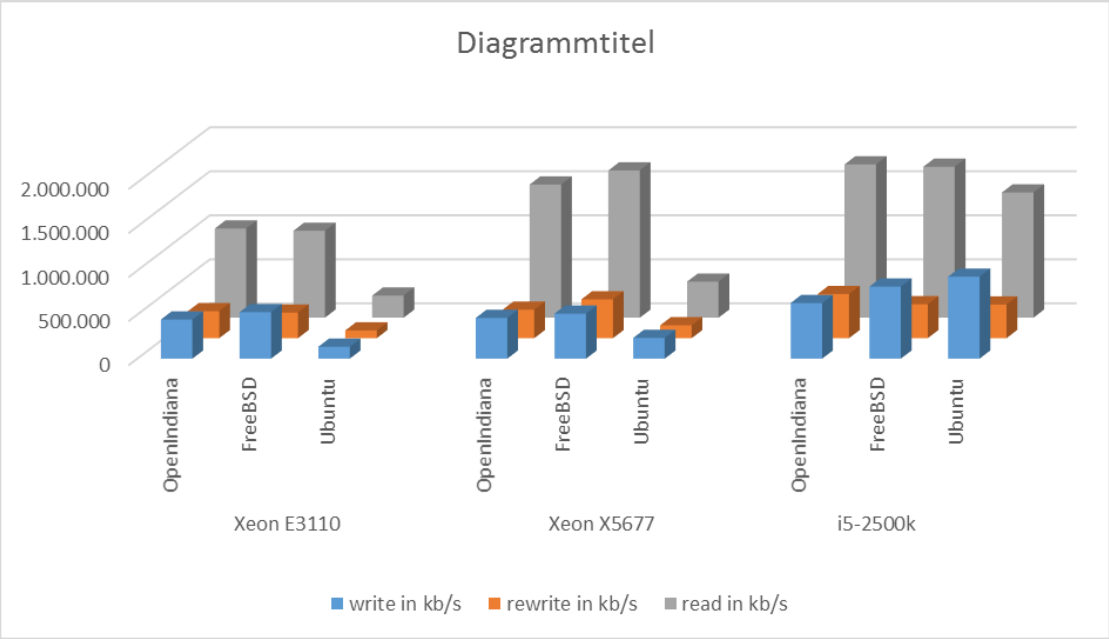

Abbildung 10: Vergleich von 3 Prozessoren mit LZ4 Kompression

In Abbildung 10 sind alle drei Prozessorgenerationen mit allen drei Betriebssystemen dargestellt. Die Prozessoren sind dabei ihrem Alter bzw. Leistung nach sortiert. Ähnlich wie bereits oben in Abbildung 9 für den E3110-Prozessor zeigt sich auch für den neueren Xeon X5677 ein Leistungseinbruch unter Ubuntu. Ebenso ist eine kontinuierliche Steigerung des Durchsatzes mit modernerer Hardware zu erkennen.

3. ZFS on Linux

3.1. Messaufbau

Im folgenden Kapitel wird ein kurzer Einblick in das Betriebssystem gegeben, welche Prozesse während eines Benchmarks auf einem ZFS-Dateisystem wieviel Kapazitäten verwenden.

Für eine Analyse von ZoL nutzen wir das bereits in Kapitel 2.1.1 beschriebene System 3 mit einem Corei5-2500k Prozessor und das in Kapitel 2.1.2 beschriebene Ubuntu 14.04.

Für die Nutzung von wrstat [6] wurde der Kernel bereits vorbereitet und der LOCK_STAT-Parameter aktiviert.

Als Analyseprogramm nutze ich wrstat [6]. Dieses sammelt während des Aufrufs eines Programms Daten über verschiedene Methoden, wie das /proc-Verzeichnis oder das Programm oprofile. Da letzteres Zugriff auf das normalerweise in Linux nicht unkomprimiert vorhandene *vmlinux* benötigt, muss mit dem in Listing 11 gezeigten Vorgang erst der Debugkernel runtergeladen werden, bevor im zweiten Schritt wrstat mit seinen Abhängigkeiten installiert wird. Die Konfiguration des Programms erfolgt über die Datei wrstat.config, welche in der Doku des Programms erklärt wird. Der vollständige Aufruf von bonnie++ mit wrstat lautet wir folgt:

```
~/wrstat/wrstat-run ./wr-results/durchlauf-01/ bonnie++ -u root -d /tank/
-x 1 -z 0 -q
```

In den vier Durchläufen wurden in ZFS folgende Einstellungen verwendet:

- ZFS ohne Optimierungen (Pure)
- LZJB-Kompression
- Deduplikation
- Deduplikation & LZJB-Kompression

3.2. Messergebnisse

Betrachten wir für den ersten Durchlauf ohne Optimierungen zunächst die Auslastung des Hauptspeichers in Abbildung 11.

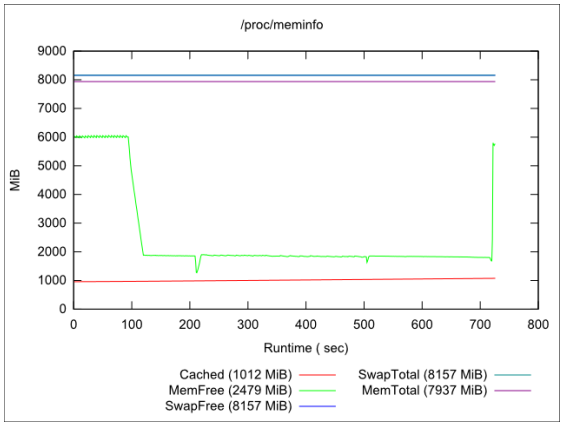

Abbildung 11: ZFS Pure: Hauptspeicherauslastung

Bonnie++ beginnt mit einem Write-Vorgang, der sich in das schreiben einzelner Dateien (bis ca. 100 Sekunden) und dem sequentiellen Schreiben unterteilt (bis ca. 200 Sekunden). Man sieht deutlich die Auswirkungen des ARC (Adjustable Replacement Cache), welcher ein eigenständiger Cache für das ZFS-Dateisystem ist. Neben dem schreiben der

Daten auf die Festplatte werden alle Daten in diesen Cache geschrieben, der sich im Hauptspeicher befindet. Dieser besteht aus zwei Bereichen (Most Frequently Used Files (MFU), Most Recently Used Files MRU)), welche dynamisch ihr Größenverhältnis anpassen. Da unserer Schreibvorgang doppelt so groß ist, wie der zur Verfügung stehende RAM, ist dieser Cache erschöpft. Da der ARC nur einen fest definierten Anteil vom gesamten RAM verwendet, bleibt ein gewisser Teil von rund 2000 MB frei.

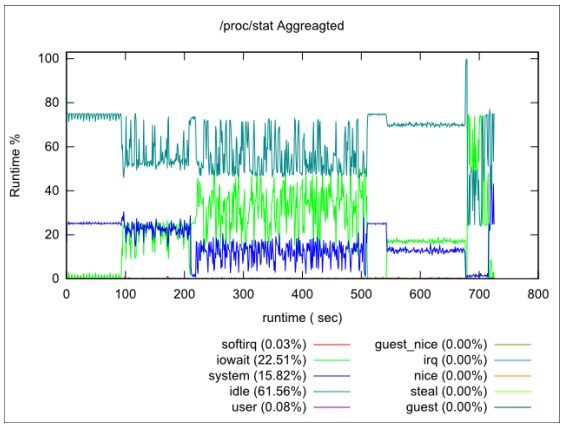

Abbildung 12: ZFS Pure: Prozessorzustände

Abbildung 12 zeigt die Prozessorauslastung bzw. seine Zustände. Hier sehr man sehr gut, wie der gefüllte ARC zur Folge hat, dass die CPU deutlich häufiger auf IO warten muss (der iowait-Wert steigt bei Sekunde 100 von Null auf 20% an).

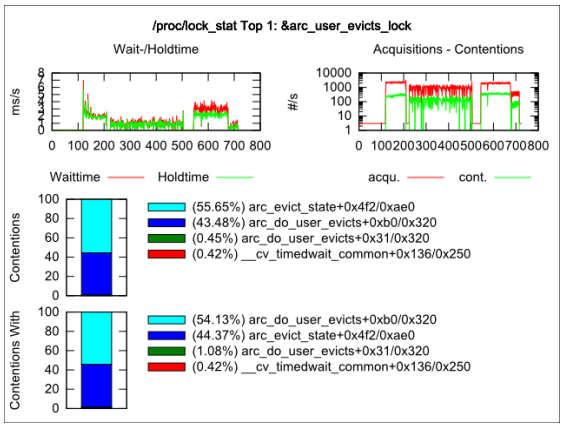

Abbildung 13: ZFS Pure: Lockverlauf des Verdrängungsprozesses des ARC

Ist der ARC voll, muss entschieden werden, welche Dateien verdrängt werden. Da der Cache sequentiell mit Daten vollgeschrieben wird, welche nie gelesen werden, spielt der MRU eine größere Rolle und verdrängt den Platz für MFU. In diesem Fall verhält sich ZFS wie der Cache von ext4, welches ausschließlich auf dem MRU-Prinzip basiert.

Abbildung 13 zeigt den Prozess, welcher die meisten Locks während der Messung beanspruchte. Ab dem Zeitpunkt des vollständig gefüllte RAMs werden Locks erstellt. Da die Datei doppelt so groß wie der RAM ist und sie danach modifiziert und anschließend gelesen wird, ist dieser Prozess bis zum Ende des Benchmarks aktiv. Da dieser Benchmark nur mit einem Datenstrom arbeitet, sind die Auswirkungen auf die Leistung eher gering. Für viele parallel arbeitende Prozesse werden die beim Zugriff/Modifikation des ARC häufig genutzten Locks schnell zu einem Problem, sodass die Leistung mit steigender Anzahl paralleler Threads wieder zu sinken beginnt [13].

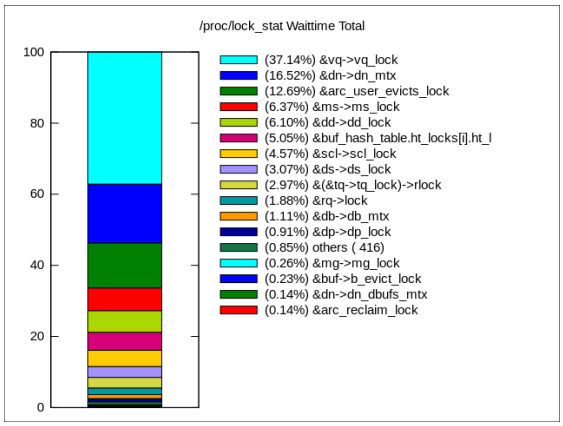

Abbildung 14: ZFS Kompression: Prozesse mit hoechster Wartezeit auf Locks

Mit aktiver Kompression sorgen ebenfalls Bestandteile von ZFS für die meisten Sperren (Abb. 14). Auf Platz 1 liegt vq_lock, welches ein Bestandteil der vdev_queue ist (Abb. 15. Diese ist der ZFS i/o scheduler und unterteilt IOs in 5 Klassen: sync read, sync write, async read, async write, und scrub/resilver. Die aktivste Zeit ist während der rewrite-Phase.

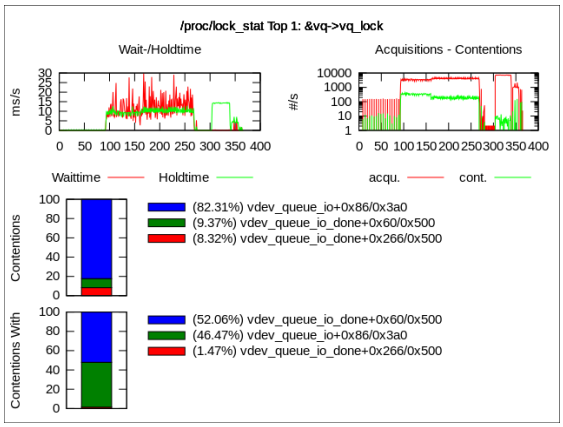

Abbildung 15: ZFS Kompression: Lock_Stat Top 1: vdev_queue

Auf Platz 2 (Abb. 16 ist die Funktion dn_mtx, welche Bestandteil von dnode_sync in ZFS ist. Dieser dnode-buffer ist verantwortlich für das schreiben von Blöcken. Wird eine Datei im RAM modifiziert, wird ein dnode_sync ausgeführt, um sie zu schreiben. Ein dnode ist ein Objekt, zum Beispiel Metadaten. Entsprechend seiner schreibenden Funktion ist es im write und rewrite-Teil aktiv.

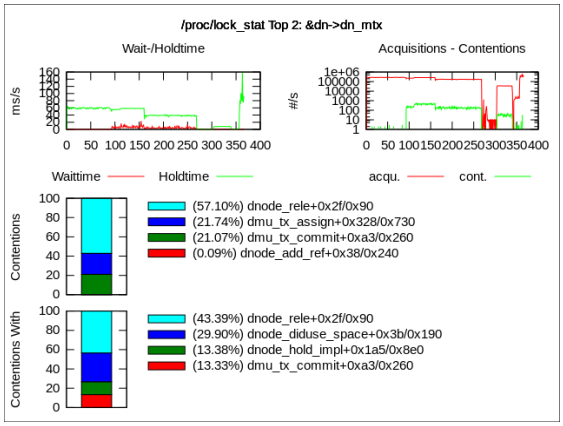

Abbildung 16: ZFS Kompression: Lock_Stat Top 2: dn_mtx

Der aus dem ersten Durchlauf bekannte ARC belegt Platz drei (Abb. 17.

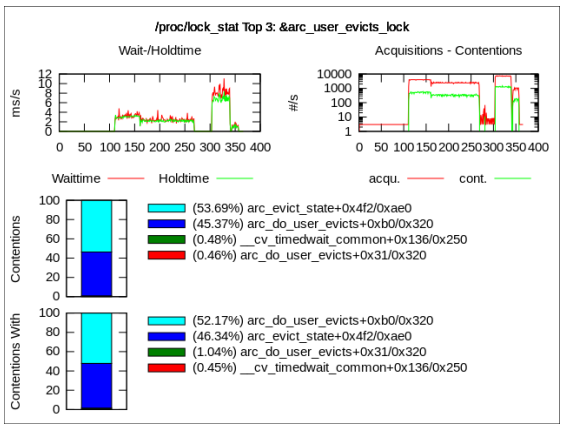

Abbildung 17: ZFS Kompression: Lock_Stat Top 3: arc_user_evicts_block

Somit sind sind über 60% aller Waittimes durch ZFS verursacht.

Insgesamt ist eine höhere System-Auslastung in Abbildung 18 zu sehen. Addiert man die System-Teile auf, so erhält man ca. 100%, was der Vollauslastung eines Kerns entspricht. Grund hierfür ist die CPU, welche die Daten komprimiert. Dieser Prozess ist für den sequentiellen Datenstrom von bonnie++ auf einen Thread begrenzt.

Bei der Deduplikation erhöht sich die iowait-Zeit des Prozessors deutlich, wie in Abb. 19 zu sehen. Anders als bei den beiden vorangegangenen Durchläufen, werden bei der

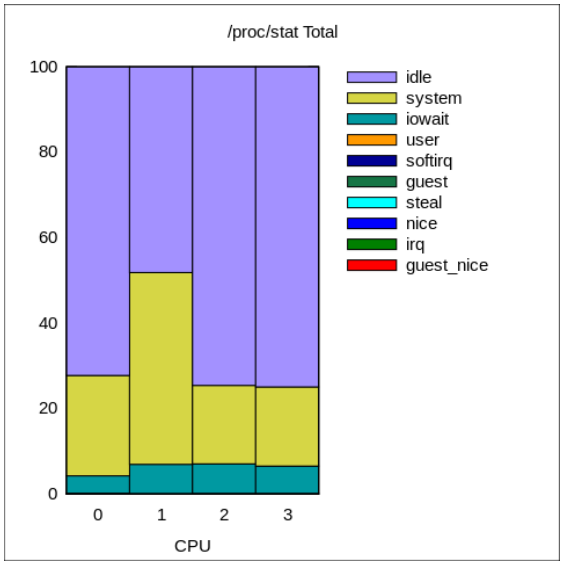

Abbildung 18: ZFS Kompression: Prozessorauslastung

Deduplikation zuerst Hashwerte von jedem Datenblock berechnet, welche mit der DDT (DedupTable) abgeglichen werden müssen. Erst dann wird entschieden, ob ein Block geschrieben wird oder nur eine Referenz auf einen bereits bestehenden Block erstellt wird.

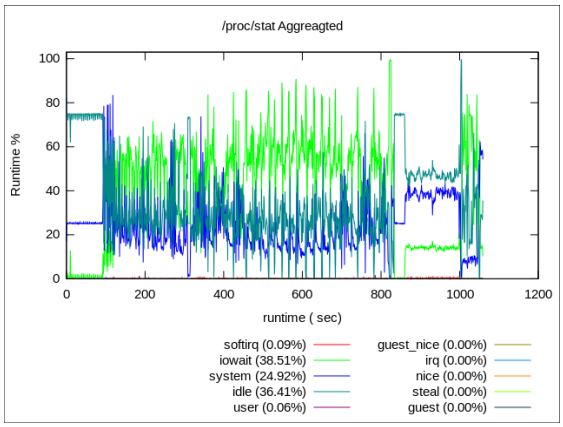

Abbildung 19: ZFS Deduplikation: Prozessorauslastung

Die meisten Locks werden durch Metadatenarbeit durch dnode_sync verursacht (Abb. 20).

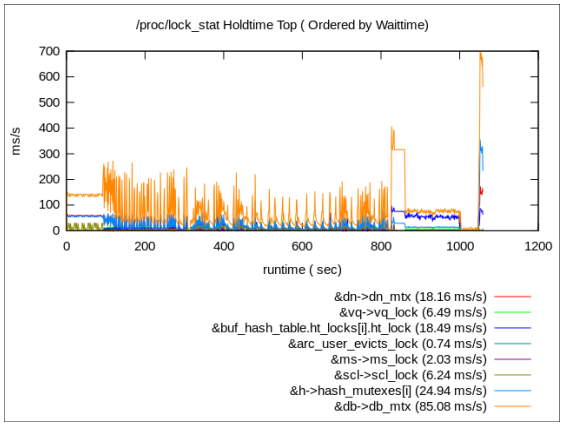

Abbildung 20: ZFS Deduplikation: Haltezeiten für Locks

Zum Abschluss spiegelt sich auch in der Kombination aus Deduplikation und Kompression eine Mischung aus den beiden vorangegangenen Tests wider. So ist der idle-Wert der CPU wieder von 35% bei Dedup auf jetzt knapp 50% angestiegen (Abb. 21. Dies ist eine Folge der verringerten Schreiblast durch die vorangegangene Kompression.

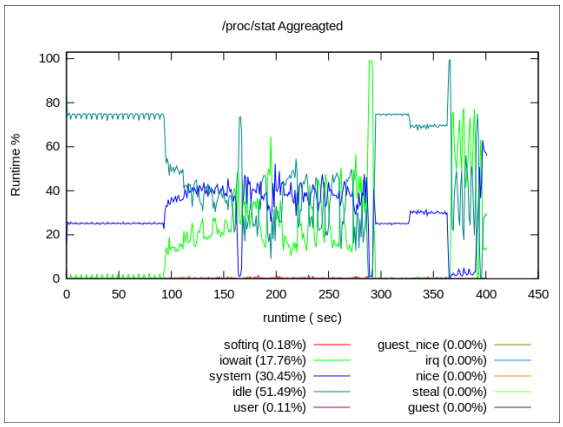

Abbildung 21: ZFS Deduplikation & Kompression: Prozessorauslastung

Während in der Auflistung der Haltezeiten in Abb. 22 der ARC mit unter 2 ms/s kaum eine Rolle spielt, ist die Deduplikation mit über 200 ms/s für den Großteil der Locks verantwortlich.

Die im vorangegangenen Kapitel gezeigten sehr hohen Datenraten sind eine Folge der guten Kompression durch den Prozessor. Die Abbildung 23 zeigt die tatsächliche Auslastung der physikalischen Festplatte an. Da wrstat für die Festplattenauslastung keine Graphen für die vorangegangenen Durchläufe erzeugt hat, ist ein direkter Vergleich leider

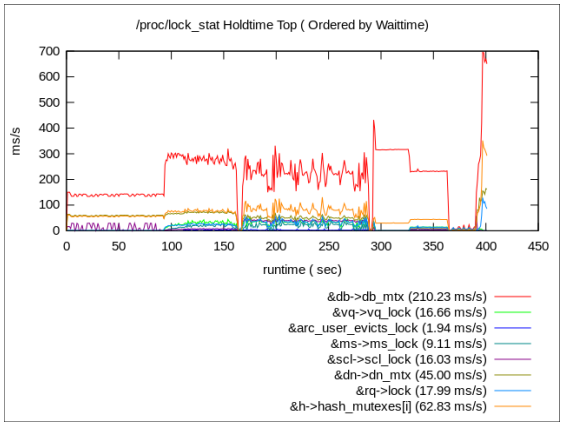

Abbildung 22: ZFS Deduplikation & Kompression: Haltezeiten für Locks

nicht möglich. Jedoch ist aus Beobachtungen die Auslastung bei reiner Kompression bzw. Deduplikation ähnlich hoch. Lediglich im ersten Test mit dem unmodifizierten ZFS liegt die Auslastung wie im vorherigen Kapitel in Abbildung 1 auf dem Niveau des tatsächlich möglichen Durchsatz.

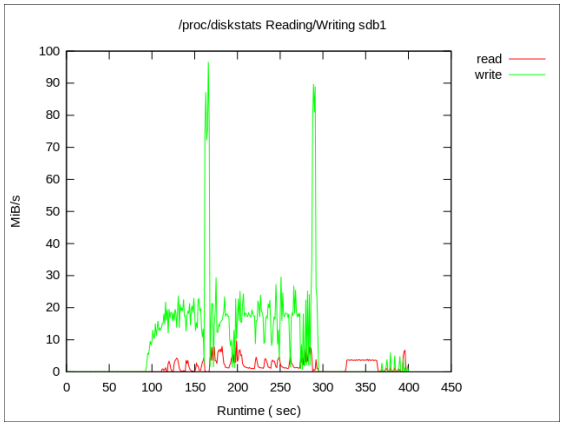

Abbildung 23: ZFS Deduplikation & Kompression: Datenrate der getesteten Festplatte

3.3. Kompression und Deduplikation in der Praxis

Da gerade die Kompression in den Tests mit bonnie++ sehr stark von der Realität abweichende Kompressionsraten ermöglicht, habe ich mit rund 900 GB Testdaten deren Komprimierbarkeit und die Fähigkeit der Deduplikation überprüft. Zum Einsatz kamen dafür Daten des Max-Planck-Instituts von einem Ozeanmodell (MPIOM). Der Großteil der Daten sind NetCDF und GRIB-Dateien, welche üblicherweise bereits vorkomprimiert sind. Neben der Kompression und Deuplizierbarkeit wurde ebenfalls die Laufzeit des Kopiervorgangs mit *time* gemessen, welche sich als nicht unerheblich herausgestellt hat.

Die kompletten Resultate sind Anhang unter 15 zu finden.

Die erste Messung ohne beide Funktionen ergab eine Laufzeit von 217,5 Minuten, wobei eine mittlere Datenrate von 74.88 MB/s erreicht wurde (siehe Tabelle 1. Die Datenrate entspricht der Ausgabe des Kopierbefehls *rsync*. Datenquelle war ein mit 1 GBit angebundener Dateiserver via NFS.

Tabelle 1: Messung der Laufzeit für verschiedene Optimierungen

Optimierung	geschriebene Daten	Laufzeit	Datenrate	Dedup/Comp.
default	978621 MB	217m30.527s	74.88 MB/s	-
LZ4	627305 MB	227m10.272s	71.70 MB/s	c = 1.56x
GZIP	469593 MB	223m58.709s	72.72 MB/s	c = 2.08x
Dedup	978622 MB	491m28.573s	33.14 MB/s	d = 1.04x
Dedup + LZ4	627311 MB	550m51.974s	29,57 MB/s	c = 1.54, d = 1.26

Bei dem zu Grunde liegenden Datensatz war der GZIP-Algorithmus sowohl bei Kompressionsrate, als auch Laufzeit dem LZ4-Algorithmus überlegen, was jedoch eher einen Ausnahmefall darstellt. Eine eher schlechte Leistung erzielte die Deduplikation. Während die Laufzeit sich auf mehr als die doppelte Zeit erhöht, ist ihr Platzgewinn mit einem Faktor von 1.04 eher gering. Während des Kopiervorgangs entstanden häufiger Leerlaufzeiten, in denen weder Daten geschrieben, noch über das Netzwerk übertragen wurden.

Eine genauere Analyse der DedupTable (DDT) ergab, dass der zur Verfügung stehende RAM von 12 GB für eine Datenmenge von rund 900 GB rausreichen sollte. ZFS on Linux nutzt standardmäßig nur maximal 50% des zur Verfügung stehenden RAMS. Daraus ergibt sich eine für den ARC nutzbare Kapazität von 6 GB. Die Metadaten, zu denen die DDT gehört, belegen wiederum nur maximal 25% des ARC, was 1.5 GB entspricht. Der Befehl *zdb -b lustre-ost0* gibt mit dem Wert "bp count"die Anzahl der geschriebenen Blöcke aus, in diesem Fall 2738958 Blöcke. Für jeden Block werden 320 Byte als Hash-Wert in die DDT geschrieben. Somit ergibt sich eine DDT-Größe von rund 835 MB. Eine mögliche Erklärung ist, dass noch weitere Metadaten von ZFS mit der DDT um die 1.5 GB konkurrieren und somit für die DDT auf die langsame Festplatte ausgewichen werden muss, was den Vorgang der Deduplizierung enorm verlängert.

Teilt man die Anzahl der geschriebenen Bytes durch den Wer "bp countßo erhält man die von ZFS geschriebene durchschnittliche Blockgröße, in diesem Fall sind das 121K, was nahe der Maximalgröße von 128K liegt. Bei großen Blöcken ist die Chance geringer, einen ähnlichen Block mit selbem Aufbau zu finden, was die Möglichkeiten der Deduplizierung weiter einschränkt.

Eine mögliche Optimierungsmaßnahme ist die manuelle Erhöhung des vom RAM nutzbaren Teils für den ARC auf 80 oder 90%. Da der ARC so entworfen ist, dass er keinem anderen Systemprozess Speicher wegnimmt, würde sich bei erhöhtem RAM-Verbrauch durch andere Prozesse der ARC automatisch anpassen und verkleinern.

4. Lustre on ZFS

4.1. Lustre

Lustre ist ein POSIX-kompatibles, verteiltes Dateisystem, welches auf Standardhardware aufbaut und als Open Source Projekt in der Top100 der größten Supercomputer das mit rund 60% am häufigsten verwendete Dateisystem ist [7]. Eine der aktuell größten Implementationen ist das Dateisystem *Spider* des zweitgrößten Supercomputers Titan. Auf rund 26.000 Clients wird insgesamt eine Speicherkapazität von 32 PB Speicher mit einer parallelen Zugriffsgeschwindigkeit von rund 1 TB/s erreicht [12].

Luste besteht aus (wie in Abb. 24) drei Komponenten:

- Management-Einheit (MGS)
- (mehrere) Metadatenserver (MDS) mit mehreren Targets (MDT)
- Objektspeicherserver (OSS) mit einem oder mehreren Speichertargets (OST)

Die Verbindung untereinander kann über normalen Ethernet-Verbindungen oder Infiniband erfolgen.

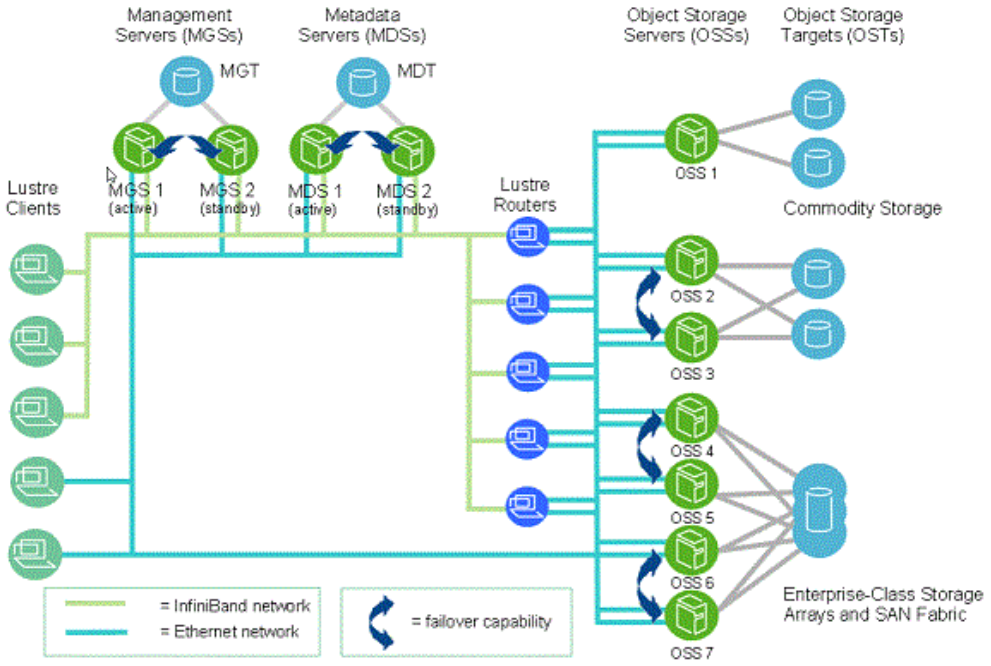

Abbildung 24: Lustre grundlegender Aufbau, Quelle: `http://lustre.org/about/`

Im Gegensatz zu anderen verteilen Dateisystemen, wie GPFS werden die Metadaten nur auf einigen wenigen Servern vorgehalten, welche dadurch auf den parallelen Zugriff

auf viele kleine Dateien optimiert werden können. Die eigentlichen Daten werden dann auf die eigentlichen Fileserver (OSS) verteilt. Hierbei kann ebenfalls festgelegt werden, ob eine Datei nur auf einem Server gespeichert werden soll oder über Mehrere verteilt. Zur Steigerung der Ausfallsicherheit und Datenintegrität können auch mehrere OSS auf dasselbe OST zugreifen (analog dazu MDS/MDT), siehe Abb. 24. Als OST können einzelne Festplatten oder Speicherverbünde, wie RAID eingesetzt werden.

Als lokales Dateisystem auf einem OST wird typischerweise das hauseigene *ldiskfs* eingesetzt, welches eine Weiterentwicklung des ext4-Dateisystems von Linux ist. Wenige Wochen nach dem stable-release von ZFS on Linux wurde ZFS als Alternative zu ldiskfs in der Version 2.4 von Lustre als zweites mögliches Dateisystem aufgenommen, um von dessen Vorteilen profitieren zu können. So kann neben den oben beschriebenen besseren Datenintegrität von ZFS gegenüber klassischen RAID-Verbünden und Dateisystemen auch je nach zu speichernden Daten eine erhebliche Einsparung von Speicherplatz durch die transparente Datenkompression in ZFS ermöglicht werden, was zu effizienterer Nutzung der vorhandenen Hardware und dem einsparen von Kosten führt. ZFS unterstützt schnelle Kompressionsalgorithmen, wie LZ4, welche mit nur wenig CPU-Overhead auskommen und somit auch auf bisher mit ldiskfs betriebenen Servern zum Einsatz kommen kann.

Auch Metadatenserver können von ZFS profitieren. Hier finden vorrangig Zugriffe mit kleinen Blockgrößen statt. Werden Daten häufig abgefragt, werden sie von ZFS im sog. Adjustable Replacement Cache (ARC) abgelegt. Dieser ist eine Mischung aus dem unter Linux verwendeten MRU-Cache-Algorithmus (Most Recent Used) und dem MFU-Cache-Algorithmus (Most Frequently Used). Diese Aufteilung ermöglicht sowohl ein cachen häufig zugegriffener, als auch neu zugegriffener Dateien. Als Erweiterung es ARC kann auch ein weiteres schnelles Speichermedium, wie eine SSD, als L2ARC genutzt werden. Für das schreiben kleiner Datenblöcke kann ebenfalls auf einer SSD ein Write-Log (das sog. SLOG) genutzt werden. Das darauf befindliche ZIL (ZFS Intend Log) kann beim häufigen schreiben kleiner Datenblöcke (Default: 64 KB) einen Geschwindigkeitszuwachs bringen [14]. Ist kein separates SLOG-Device vorhanden, wird das ZIL mit in den ZFS-Pool geschrieben, was für Leistungsprobleme beim schreiben sorgen kann (siehe Kapitel 4 bzw. [11]).

4.2. Messaufbau

Lustre wurde auf einem Setup aus drei baugleichen Servern installiert. Jeder Server hat einen Xeon X5560 Prozessor mit 4 Kernen und 8 Threads, 12 GB RAM und je einer 500 GB Systemfestplatte und einer für Lustre/ZFS freien 2 TB Festplatte. Die Verbindung untereinander erfolgt mit einem 1 GBit-Uplink an einen gemeinsamen Switch. Als Betriebssystem wurde CentOS Linux release 7.2.1511 (Core) verwendet mit folgender Kernel-Version:

```
Linux nehalem1.cluster 3.10.0-327.4.5.el7.x86_64 #1 SMP Mon Jan 25 22:07:14
UTC 2016 x86_64 x86_64 x86_64 GNU/Linux
```

Die Installation von Lustre und ZFS erfolgte wie im Listing 12. Wichtig hierbei ist die

Deaktivierung von SELinux am Ende des Prozesses.

Das erstellen der MGS, MDS und OSS wurde wie in Listing 13 durchgeführt. Da die Management-Einheit und der Metadatenserver in diesem kleinen Testsetup nur wenig Leistung benötigen, wurden sie in 1 GB große ZFS-pools in eine Datei in /tmp auf nehalem1 gespeichert. Jeder der drei Server hat die freie 2 TB Festplatte als OST0 bis OST2 als Speicherplatz verwendet. Damit der Prozess Lustre beim Start die jeweiligen Pools auch einbinden kann, ist die Konfiguration der Datei /etc/ldev.conf wichtig. Die Datei wird geparst und alle Einträge mit dem eigenen Hostnamen gemountet. Die anderen Einträge werden ignoriert. Zum Schluss wird Lustre gestartet und eingehängt. Der Befehl *lfs setstripe -c -1* sorgt dafür, dass eingehende Dateien gleichmäßig auf alle OSTs verteilt geschrieben werden.

4.2.1. verwendete Benchmarks

Als Benchmark kamen bonnie++ für einen einfachen Test von einem Host (nehalem1) aus zum Einsatz, sowie das parallel von allen drei Hosts ausgeführte Programm ior. Die Installation von ior erfolgte wie im Listing 14. Da es parallel ausgeführt wird, muss zuerst MPICH installiert werden. Zum laden der MPI-Pfade müssen bei jedem Login in die Shell die folgenden zwei Befehle ausgeführt werden:

```
export PATH=/lustre/software/mpich3/bin:$PATH
export LD_LIBRARY_PATH=/lustre/software/mpich3/lib:${LD_LIBRARY_PATH}
```

Der Aufruf von bonnie++ erfolgte ähnlich wie in den vorangegangenen Messungen. Es wurde lediglich auf den Parameter -n 1024 verzichtet.

```
bonnie++ -d /mnt/lustre/client/ -u root -x 1 -z 0 -q » bench.csv
```

Ior wurde parallel mit Hilfe von *mpiexec* aufgerufen:

```
mpiexec -hosts=nehalem1,nehalem2,nehalem3 -np 6 /lustre/software/ior/bin/ior
-v -F -t 1m -b 24g -o /mnt/lustre/client/testfile » ior-out.txt
```

- *-hosts*: Auflistung aller genutzten Hosts
- *-np 6*: Anzahl der ausgeführten Instanzen von ior (zwei Pro Host)
- *-v*: Verbose
- *-F*: Zugriffsart *file-per-process*
- *-t 1m*: transferSize 1 Megabyte
- *-b 24g*: Dateigröße für den Test (entspricht doppelter Menge des RAM eines Hosts)
- *-o /mnt/lustre/client/testfile*: Testpfad auf das Lustre-Laufwerk

4.3. Ergebnisse

Da bonnie++ auf einem Host ausgeführt wird, aber auf drei Hosts die Daten gespeichert werden, teilen sich die Schreibzugriffe auf eine lokale Festplatte auf und zwei über das Netzwerk angebundene OSTs. Da diese nur über 1 GBit Netzwerkuplink erreichbar sind, ist die Datenrate hier durch das Netzwerk begrenzt. Abbildung 25 zeigt die verschiedenen Ausführungen von bonnie++, während die Einstellungen für ZFS nach jedem Durchlauf verändert wurden (siehe X-Achse).

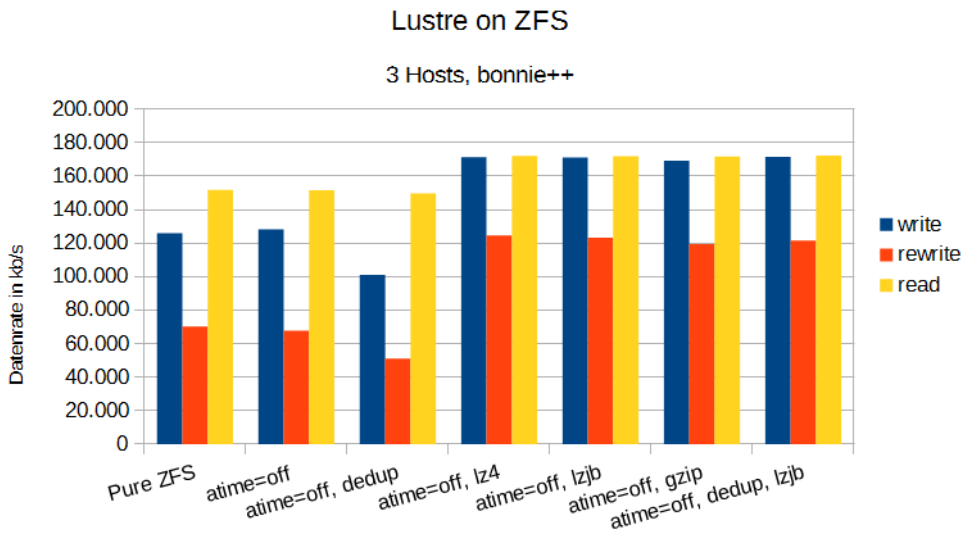

Abbildung 25: Bonnie++ Benchmark auf Lustre auf ZFS-Basis

Da die Daten gleichmäßig auf alle drei OSTs geschrieben werden, sind die Zugriffe auf das lokale Laufwerk schneller abgearbeitet, als die entfernten OSTs. Daher ist auch unabhängig von Kompression und Deduplikation schnell ein maximales Durchsatzlimit erreicht. Das Problem an dieser Stelle ist, dass die Daten unkomprimiert über das Transportnetz gesendet werden und erst auf der lokalen Festplatte durch ZFS komprimiert werden können.

Abbildung 26 zeigt Die Auslastung von Netzwerk und Festplatte auf dem Host, auf dem der Benchmark ausgeführt wird. Hier sieht man die Aufteilung der write, rewrite und read-Teile des Benchmarks und den anschließenden Test zum erstellen und löschen von Dateien. Während die lokale Festplatte dank Kompression in allen Teilen nur sehr wenig tatsächlichen Durchsatz hat (es werden nur die bereits komprimierten Daten geschrieben), ist das Netzwerk der limitierende Faktor. Mit dem LZ4-Algorithmus kann ZFS die Testdateien von bonnie++ um den Faktor 115x verkleinern.

Im Vergleich dazu sieht man in Abbildung 27 das hier wahrscheinlich Lustre der bremsende Faktor ist, da weder die Festplatte voll ausgelastet ist, noch die Netzwerkverbindung gesättigt ist (außer zum Ende des Lesetests).

Die folgende Abbildung 28 zeigt analog zu dem bonnie++ Benchmark das Verhalten

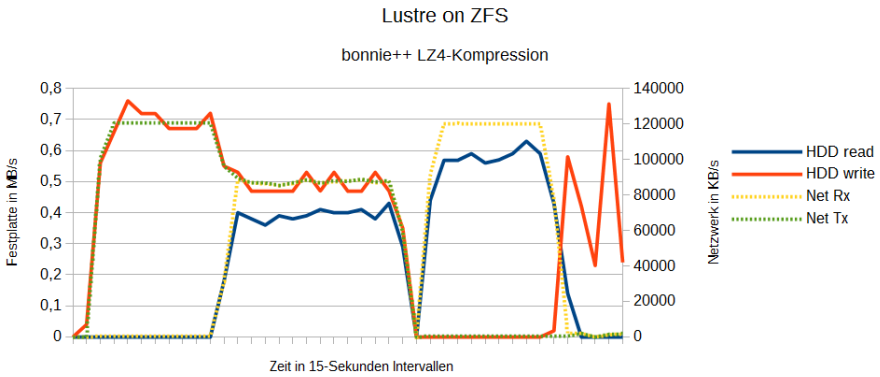

Abbildung 26: Netzwerk- und Festplattenauslastung auf nehalem1 während eines bonnie++ Durchlaufs mit aktivierter Kompression

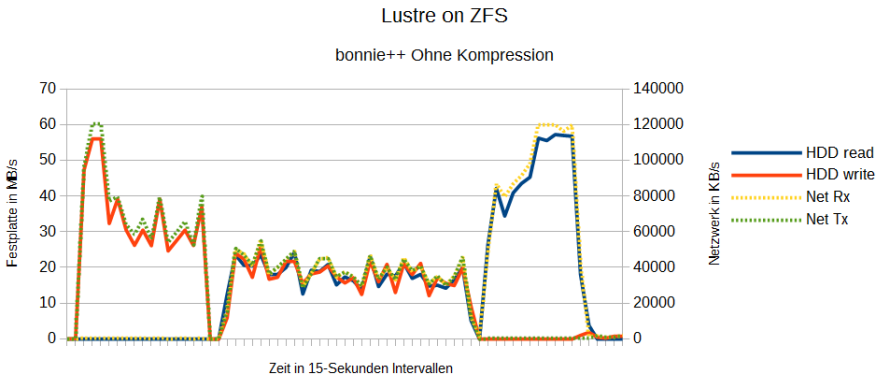

Abbildung 27: Netzwerk- und Festplattenauslastung auf nehalem1 während eines bonnie++ Durchlaufs ohne Kompression

des Systems mit verschiedenen ZFS-Parametern für Kompression und Deduplikation. Wie erwartet ist der Durchsatz von LZ4 am höchsten, gefolgt von LZJB und GZip. Mit ior wurden von allen drei Hosts aus jeweils zwei ior-Instanzen mit einer Dateigröße von 24 GB geschrieben bzw. gelesen, was 144 GB Gesamtdaten entspricht. Die Testdaten von ior lassen sich von ZFS mit dem LZ4-Algorithmus nur um den Faktor 3.9 komprimieren, weshalb diese Messwerte ein für die Praxis realistischeres Ergebnis zur Folge haben (900 GB Testdaten auf dem wr-Cluster konnten mit LZ4 um den Faktor 1.51, mit GZip-6 um 2,03 komprimiert werden 15.

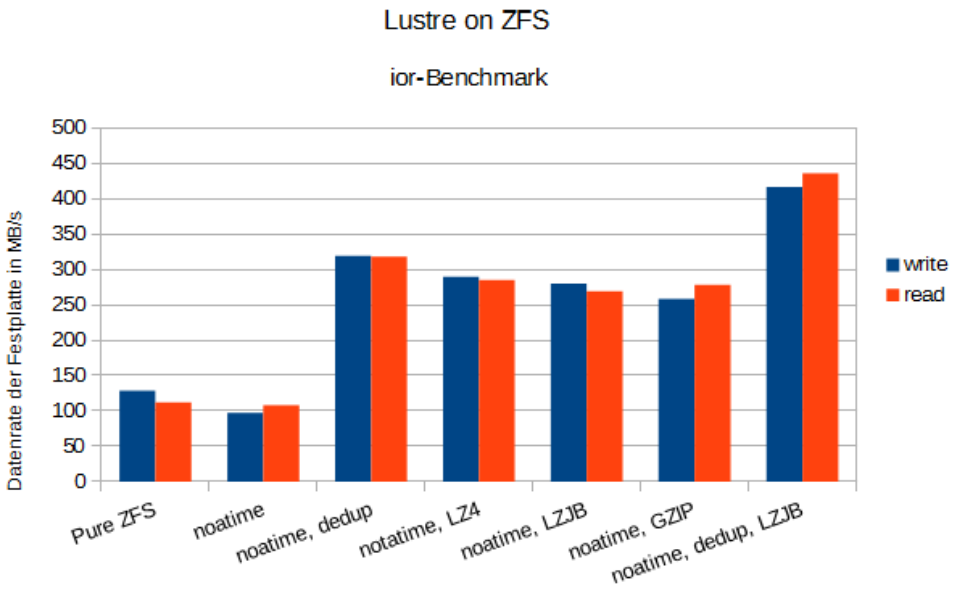

Abbildung 28: Ior-Benchmark auf Lustre auf ZFS-Basis

Für die Auslastung der Festplatte und des Netzwerks habe ich parallel zum bonnie++-Durchlauf die Auslastung der beiden Komponenten mit und ohne LZ4-Kompression mit dem Analyseprogramm *sar* aufgezeichnet separat aufgezeichnet und in einer Grafik zusammengefasst. Abbildung 29 zeigt den Verlauf bei aktiver Kompression. Ior führt zuerst einen Schreibtest aller Daten und anschließend einen Lesetest durch, was auch an der Zweiteilung des Graphen in der Mitte zu erkennen ist, an der die blaue Linie (HDD read) und die rote Linie (HDD write) sich abwechseln.

Die anfängliche Schreibrate der Festplatte (rote Linie) zeigt, dass durch die schlechtere Komprimierbarkeit der Testdaten eine erhöhte Schreiblast vorhanden ist. Außerdem müssen neben den zwei lokalen Schreibprozessen auch die vier über das Netzwerk ankommenden Datenströme geschrieben werden. Wie die starken Zacken der Kurven andeuten, gab es keine gleichmäßige Auslastung der Netzwerkverbindung und insgesamt gleich es eher einem Burst-Traffic. Es gibt ebenfalls einen direkten Zusammenhang zwischen Auslastung der Festplatte und der Netzwerkverbindung. Laut dem Linux-Programm *iotop* ist während einer Phase mit geringer Datenrate die IO-Auslastung des Prozesses *txg_sync*

sehr hoch. Ein voller ARC-Cache von ZFS kann eine mögliche Ursache sein. [5] Abhilfe könnte mehr Hauptspeicher im jeweiligen System oder ein schneller L2ARC sein, zum Beispiel auf einer SSD.

Abbildung 30 zeigt den selben Ablauf ohne Kompression. Hier treten die Bursts in Netzwerk und Festplatte schon auf, bevor der ARC den vollständigen RAM belegt hat. Durch die Glättung durch das 20-Sekunden-Messintervall fallen diese jedoch nur durch kleinere Zacken besonders in der ersten Hälfte des Graphen auf. Eine mögliche Erklärung ist die stärkere Auslastung der Festplatte. Da diese sechs Datenströme gleichzeitig verarbeiten muss und durch fehlende Kompression die vollen Daten geschrieben werden müssen, ist das Netzwerk erst während der Lesephase wirklich nennenswert ausgelastet, während der Prozessor auf Grund fehlender Kompression keine Arbeit zu verrichten hat. Während der Lesephase hatte die CPU eine iowait-Zeit von rund 30%.

5. Zusammenfassung

Vor knapp drei Jahren wurde die ZFS-Portierung ZFS on Linux als stable-release veröffentlicht und für die Verwendung in Produktivsystemen freigegeben. Zu diesem Zeitpunkt gab es noch eine kleine Anzahl an Funktionen, die ZoL im Gegensatz zum originalen ZFS unter Illumos noch nicht beinhaltete. Mit der aktuellen Version soll dieser Unterschied aufgehoben sein. Es gibt jedoch auch Funktionen, wie *Improve N-way mirror read performance*, die ausschließlich in FreeBSD implementiert sind [9]. Leistungstechnisch steht die Linux-Version dem Original gleichauf, was die Messergebnisse in Kapitel 2 ebenfalls untermauern. Dort ist zu sehen, dass die ZFS-Versionen in unterschiedlichen Bereichen verschieden stark sind und kein eindeutiger Erster hervorgeht.

Eine genauere Betrachtung zeigt, dass ZoL sich sehr ähnlich wie sein Vorbild verhält und die Arbeiten auf dem ARC mit sehr vielen Locks verrichtet. Diese kann bei starker paralleler Belastung zu Performanceeinbrüchen führen.

Kapitel 4 zeigt, dass ZFS auch als Basis für Lustre eine gute Alternative zu ldiskfs darstellt, da es wartungsärmer ist als ein Verbund aus Dateisystem, Volumemanager und RAID-Verbund und dank transparenter Kompression eine (je nach Nutzdaten) deutlich effizientere Auslastung der Hardware ermöglicht. Notwendig hierfür ist lediglich ein durchschnittlicher Prozessor und möglichst viel RAM, den ZFS mit ARC als Cache nutzt.

Seit der Veröffentlichung ist die Nutzung von ZoL stetig angestiegen. Bereits kurze Zeit später konnte das parallele Dateisystem Lustre auch ZFS als lokales Dateisystem nutzen und auch kommerzielle Produkte, wie SoftNAS oder die Virtualisierungsplattform Proxmox, nutzen ZoL als Basis.

Die leichte Verwaltung der Speicherpools, ebenso wie Snapshots, begünstigen das Zusammenspiel von ZFS mit Linux-Containerisierung, wie Docker.

Nicht zuletzt die Funktionen, wie transparente Kompression, Deduplikation, Copy-on-Write und eine hohe Datenintegrität machen ZFS zu einem modernen Dateisystem, welches klassische Dateisysteme und RAID-Verbünde ablösen könnte.

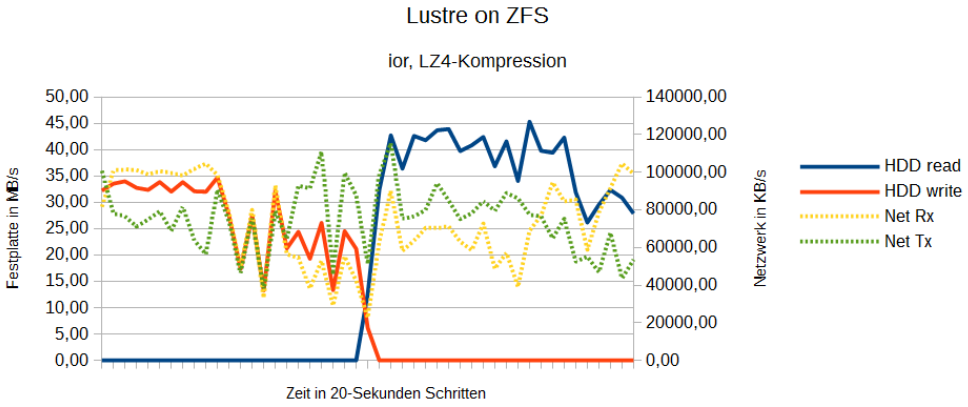

Abbildung 29: Netzwerk- und Festplattenauslastung auf nehalem1 während eines ior Durchlaufs mit Kompression

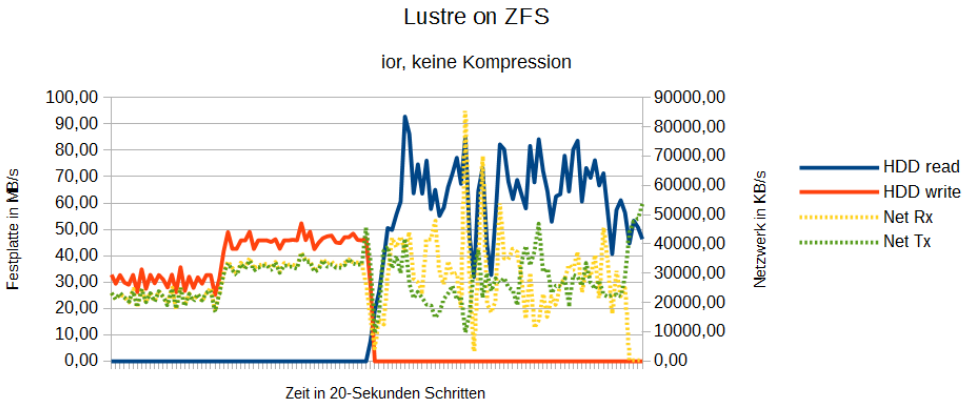

Abbildung 30: Netzwerk- und Festplattenauslastung auf nehalem1 während eines ior Durchlaufs ohne Kompression

A. Scripte

A.1. Installation von ZFS unter Linux mit modifiziertem Kernel

```
sudo apt-get install build-essential gawk alien fakeroot \
        linux-headers-$(uname -r) autoconf libtool zlib1g-dev \
        uuid-dev libblkid-dev libselinux-dev git libattr1-dev
git clone https://github.com/zfsonlinux/spl.git
git clone https://github.com/zfsonlinux/zfs.git
cd spl
./autogen.sh
./configure
make deb-utils deb-kmod
sudo dpkg -i *.deb
cd ../zfs
./autogen.sh
./configure
make deb-utils deb-kmod
sudo dpkg -i *.deb
```

Listing 5: Installation von ZFS unter Ubuntu

```
sudo apt-get install linux-source build-essential kernel-package
mkdir kernel
cd kernel
tar xvjf /usr/src/linux-*tar.bz2
cd linux-source*
cp /boot/config-`uname -r` .config
nano .config
    # Zeile CONFIG_LOCK_STAT=y einkommentieren, mit =y ergaenzen
yes "" | make oldconfig
make-kpkg --initrd buildpackage
cd ..
sudo dpkg -i linux-image-3.5.7.7_3.5.7.7-10.00.Custom_i386.deb
sudo init 6
```

Listing 6: Linux Kernel patchen

```
cd spl
./autogen.sh
./configure
make deb-utils deb-kmod
sudo dpkg -i *.deb
cd ../zfs
nano META
```

```
        #Change License from CDDL to GPL, needed for compiling \
        ZFS with CONFIG_LOCK_STAT=y in Kernel
nano include/linux/vfs_compat.h
        #In Zeile Zeile 128:
        #ifndef HAVE_SET_NLINK  ->      #ifdef HAVE_SET_NLINK
./autogen.sh
./configure
make deb-utils deb-kmod
sudo dpkg -i *.deb
```

Listing 7: erneute Installation von ZFS für den modifizierten Kernel

A.2. Benchmark-scripte für Bonnie++

```
#!/bin/bash
zpool create tank c3t1d0
echo "atime=on dedup=off compression=off" >> bench.csv
/opt/csw/sbin/bonnie++ -d /tank/ -u root -x 1 -z 0 -q -n 1024 >>
    bench.csv
zpool destroy tank
echo "##########################1/15"
zpool create tank c3t1d0
zfs set atime=off tank
echo "atime=off dedup=off compression=off" >> bench.csv
/opt/csw/sbin/bonnie++ -d /tank/ -u root -x 1 -z 0 -q -n 1024 >>
    bench.csv
zpool destroy tank
echo "##########################2/15"
zpool create tank c3t1d0
zfs set atime=off tank
zfs set dedup=on tank
echo "atime=off dedup=on compression=off" >> bench.csv
/opt/csw/sbin/bonnie++ -d /tank/ -u root -x 1 -z 0 -q -n 1024 >>
    bench.csv
zpool destroy tank
echo "##########################3/15"
zpool create tank c3t1d0
zfs set atime=off tank
zfs set compression=lz4 tank
echo "atime=off dedup=off compression=lz4" >> bench.csv
/opt/csw/sbin/bonnie++ -d /tank/ -u root -x 1 -z 0 -q -n 1024 >>
    bench.csv
zpool destroy tank
```

```
echo "############################4/15"
zpool create tank c3t1d0
zfs set atime=off tank
zfs set compression=lzjb tank
echo "atime=off␣dedup=off␣compression=lzjb" >> bench.csv
/opt/csw/sbin/bonnie++ -d /tank/ -u root -x 1 -z 0 -q -n 1024 >>
    bench.csv
zpool destroy tank
echo "############################5/15"
zpool create tank c3t1d0
zfs set atime=off tank
zfs set compression=gzip-1 tank
echo "atime=off␣dedup=off␣compression=gzip-1" >> bench.csv
/opt/csw/sbin/bonnie++ -d /tank/ -u root -x 1 -z 0 -q -n 1024 >>
    bench.csv
zpool destroy tank
echo "############################6/15"
zpool create tank c3t1d0
zfs set atime=off tank
zfs set compression=gzip-2 tank
echo "atime=off␣dedup=off␣compression=gzip-2" >> bench.csv
/opt/csw/sbin/bonnie++ -d /tank/ -u root -x 1 -z 0 -q -n 1024 >>
    bench.csv
zpool destroy tank
echo "############################7/15"
zpool create tank c3t1d0
zfs set atime=off tank
zfs set compression=gzip-3 tank
echo "atime=off␣dedup=off␣compression=gzip-3" >> bench.csv
/opt/csw/sbin/bonnie++ -d /tank/ -u root -x 1 -z 0 -q -n 1024 >>
    bench.csv
zpool destroy tank
echo "############################8/15"
zpool create tank c3t1d0
zfs set atime=off tank
zfs set compression=gzip-4 tank
echo "atime=off␣dedup=off␣compression=gzip-4" >> bench.csv
/opt/csw/sbin/bonnie++ -d /tank/ -u root -x 1 -z 0 -q -n 1024 >>
    bench.csv
zpool destroy tank
echo "############################9/15"
zpool create tank c3t1d0
zfs set atime=off tank
zfs set compression=gzip-5 tank
```

```
echo "atime=off␣dedup=off␣compression=gzip-5" >> bench.csv
/opt/csw/sbin/bonnie++ -d /tank/ -u root -x 1 -z 0 -q -n 1024 >>
    bench.csv
zpool destroy tank
echo "############################10/15"
zpool create tank c3t1d0
zfs set atime=off tank
zfs set compression=gzip-6 tank
echo "atime=off␣dedup=off␣compression=gzip-6" >> bench.csv
/opt/csw/sbin/bonnie++ -d /tank/ -u root -x 1 -z 0 -q -n 1024 >>
    bench.csv
zpool destroy tank
echo "############################11/15"
zpool create tank c3t1d0
zfs set atime=off tank
zfs set compression=gzip-7 tank
echo "atime=off␣dedup=off␣compression=gzip-7" >> bench.csv
/opt/csw/sbin/bonnie++ -d /tank/ -u root -x 1 -z 0 -q -n 1024 >>
    bench.csv
zpool destroy tank
echo "############################12/15"
zpool create tank c3t1d0
zfs set atime=off tank
zfs set compression=gzip-8 tank
echo "atime=off␣dedup=off␣compression=gzip-8" >> bench.csv
/opt/csw/sbin/bonnie++ -d /tank/ -u root -x 1 -z 0 -q -n 1024 >>
    bench.csv
zpool destroy tank
echo "############################13/15"
zpool create tank c3t1d0
zfs set atime=off tank
zfs set compression=gzip-9 tank
echo "atime=off␣dedup=off␣compression=gzip-9" >> bench.csv
/opt/csw/sbin/bonnie++ -d /tank/ -u root -x 1 -z 0 -q -n 1024 >>
    bench.csv
zpool destroy tank
echo "############################14/15"
zpool create tank c3t1d0
zfs set atime=off tank
zfs set compression=on tank
zfs set dedup=on tank
echo "atime=off␣dedup=on␣compression=on" >> bench.csv
/opt/csw/sbin/bonnie++ -d /tank/ -u root -x 1 -z 0 -q -n 1024 >>
    bench.csv
```

```
zpool destroy tank
echo "finish"
```

Listing 8: Benchmarkscript für OpenIndiana

```
#!/bin/bash
zpool create tank ada1
echo "atime=on␣dedup=off␣compression=off" >> bench.csv
bonnie++ -d /tank/ -u root -x 1 -z 0 -q -n 1024 >> bench.csv
zpool destroy tank
echo "##########################1/15␣done"
zpool create tank ada1
zfs set atime=off tank
echo "atime=off␣dedup=off␣compression=off" >> bench.csv
bonnie++ -d /tank/ -u root -x 1 -z 0 -q -n 1024 >> bench.csv
zpool destroy tank
echo "##########################2/15␣done"
zpool create tank ada1
zfs set atime=off tank
zfs set dedup=on tank
echo "atime=off␣dedup=on␣compression=off" >> bench.csv
bonnie++ -d /tank/ -u root -x 1 -z 0 -q -n 1024 >> bench.csv
zpool destroy tank
echo "##########################3/15␣done"
zpool create tank ada1
zfs set atime=off tank
zfs set compression=lz4 tank
echo "atime=off␣dedup=off␣compression=lz4" >> bench.csv
bonnie++ -d /tank/ -u root -x 1 -z 0 -q -n 1024 >> bench.csv
zpool destroy tank
echo "##########################4/15␣done"
zpool create tank ada1
zfs set atime=off tank
zfs set compression=lzjb tank
echo "atime=off␣dedup=off␣compression=lzjb" >> bench.csv
bonnie++ -d /tank/ -u root -x 1 -z 0 -q -n 1024 >> bench.csv
zpool destroy tank
echo "##########################5/15␣done"
zpool create tank ada1
zfs set atime=off tank
zfs set compression=gzip-1 tank
echo "atime=off␣dedup=off␣compression=gzip-1" >> bench.csv
bonnie++ -d /tank/ -u root -x 1 -z 0 -q -n 1024 >> bench.csv
zpool destroy tank
```

```
echo "##########################6/15 done"
zpool create tank ada1
zfs set atime=off tank
zfs set compression=gzip-2 tank
echo "atime=off dedup=off compression=gzip-2" >> bench.csv
bonnie++ -d /tank/ -u root -x 1 -z 0 -q -n 1024 >> bench.csv
zpool destroy tank
echo "##########################7/15 done"
zpool create tank ada1
zfs set atime=off tank
zfs set compression=gzip-3 tank
echo "atime=off dedup=off compression=gzip-3" >> bench.csv
bonnie++ -d /tank/ -u root -x 1 -z 0 -q -n 1024 >> bench.csv
zpool destroy tank
echo "##########################8/15 done"
zpool create tank ada1
zfs set atime=off tank
zfs set compression=gzip-4 tank
echo "atime=off dedup=off compression=gzip-4" >> bench.csv
bonnie++ -d /tank/ -u root -x 1 -z 0 -q -n 1024 >> bench.csv
zpool destroy tank
echo "##########################9/15 done"
zpool create tank ada1
zfs set atime=off tank
zfs set compression=gzip-5 tank
echo "atime=off dedup=off compression=gzip-5" >> bench.csv
bonnie++ -d /tank/ -u root -x 1 -z 0 -q -n 1024 >> bench.csv
zpool destroy tank
echo "##########################10/15 done"
zpool create tank ada1
zfs set atime=off tank
zfs set compression=gzip-6 tank
echo "atime=off dedup=off compression=gzip-6" >> bench.csv
bonnie++ -d /tank/ -u root -x 1 -z 0 -q -n 1024 >> bench.csv
zpool destroy tank
echo "##########################11/15 done"
zpool create tank ada1
zfs set atime=off tank
zfs set compression=gzip-7 tank
echo "atime=off dedup=off compression=gzip-7" >> bench.csv
bonnie++ -d /tank/ -u root -x 1 -z 0 -q -n 1024 >> bench.csv
zpool destroy tank
echo "##########################12/15 done"
zpool create tank ada1
```

```
zfs set atime=off tank
zfs set compression=gzip-8 tank
echo "atime=off dedup=off compression=gzip-8" >> bench.csv
bonnie++ -d /tank/ -u root -x 1 -z 0 -q -n 1024 >> bench.csv
zpool destroy tank
echo "############################13/15 done"
zpool create tank ada1
zfs set atime=off tank
zfs set compression=gzip-9 tank
echo "atime=off dedup=off compression=gzip-9" >> bench.csv
bonnie++ -d /tank/ -u root -x 1 -z 0 -q -n 1024 >> bench.csv
zpool destroy tank
echo "############################14/15 done"
zpool create tank ada1
zfs set atime=off tank
zfs set compression=on tank
zfs set dedup=on tank
echo "atime=off dedup=on compression=on" >> bench.csv
bonnie++ -d /tank/ -u root -x 1 -z 0 -q -n 1024 >> bench.csv
zpool destroy tank
echo "finish"
```

Listing 9: Benchmarkscript für FreeBSD

```
#!/bin/bash
zpool create tank /dev/sdb
echo "atime=on dedup=off compression=off" >> bench.csv
bonnie++ -d /tank/ -u root -x 1 -z 0 -q -n 1024 >> bench.csv
iozone -Mce -a -s 1g -f /tank/file -i0 -i1 -i2 -b out1.xls
zpool destroy tank
echo "############################1/16 done"
zpool create tank /dev/sdb
zfs set atime=off tank
echo "atime=off dedup=off compression=off" >> bench.csv
bonnie++ -d /tank/ -u root -x 1 -z 0 -q -n 1024 >> bench.csv
iozone -Mce -a -s 1g -f /tank/file -i0 -i1 -i2 -b out2.xls
zpool destroy tank
echo "############################2/16 done"
zpool create tank /dev/sdb
zfs set atime=off tank
zfs set compression=on tank
echo "atime=off dedup=off compression=on" >> bench.csv
bonnie++ -d /tank/ -u root -x 1 -z 0 -q -n 1024 >> bench.csv
```

```
iozone -Mce -a -s 1g -f /tank/file -i0 -i1 -i2 -b out3.xls
zpool destroy tank
echo "############################3/16 done"
zpool create tank /dev/sdb
zfs set atime=off tank
zfs set dedup=on tank
echo "atime=off dedup=on compression=off" >> bench.csv
bonnie++ -d /tank/ -u root -x 1 -z 0 -q -n 1024 >> bench.csv
iozone -Mce -a -s 1g -f /tank/file -i0 -i1 -i2 -b out4.xls
zpool destroy tank
echo "############################4/16 done"
zpool create tank /dev/sdb
zfs set atime=off tank
zfs set compression=lz4 tank
echo "atime=off dedup=off compression=lz4" >> bench.csv
bonnie++ -d /tank/ -u root -x 1 -z 0 -q -n 1024 >> bench.csv
iozone -Mce -a -s 1g -f /tank/file -i0 -i1 -i2 -b out5.xls
zpool destroy tank
echo "############################5/16 done"
zpool create tank /dev/sdb
zfs set atime=off tank
zfs set compression=lzjb tank
echo "atime=off dedup=off compression=lzjb" >> bench.csv
bonnie++ -d /tank/ -u root -x 1 -z 0 -q -n 1024 >> bench.csv
iozone -Mce -a -s 1g -f /tank/file -i0 -i1 -i2 -b out6.xls
zpool destroy tank
echo "############################6/16 done"
zpool create tank /dev/sdb
zfs set atime=off tank
zfs set compression=gzip-1 tank
echo "atime=off dedup=off compression=gzip-1" >> bench.csv
bonnie++ -d /tank/ -u root -x 1 -z 0 -q -n 1024 >> bench.csv
iozone -Mce -a -s 1g -f /tank/file -i0 -i1 -i2 -b out7.xls
zpool destroy tank
echo "############################7/16 done"
zpool create tank /dev/sdb
zfs set atime=off tank
zfs set compression=gzip-2 tank
echo "atime=off dedup=off compression=gzip-2" >> bench.csv
bonnie++ -d /tank/ -u root -x 1 -z 0 -q -n 1024 >> bench.csv
iozone -Mce -a -s 1g -f /tank/file -i0 -i1 -i2 -b out8.xls
zpool destroy tank
echo "############################8/16 done"
zpool create tank /dev/sdb
```

```
zfs set atime=off tank
zfs set compression=gzip-3 tank
echo "atime=off dedup=off compression=gzip-3" >> bench.csv
bonnie++ -d /tank/ -u root -x 1 -z 0 -q -n 1024 >> bench.csv
iozone -Mce -a -s 1g -f /tank/file -i0 -i1 -i2 -b out9.xls
zpool destroy tank
echo "############################9/16 done"
zpool create tank /dev/sdb
zfs set atime=off tank
zfs set compression=gzip-4 tank
echo "atime=off dedup=off compression=gzip-4" >> bench.csv
bonnie++ -d /tank/ -u root -x 1 -z 0 -q -n 1024 >> bench.csv
iozone -Mce -a -s 1g -f /tank/file -i0 -i1 -i2 -b outa.xls
zpool destroy tank
echo "############################10/16 done"
zpool create tank /dev/sdb
zfs set atime=off tank
zfs set compression=gzip-5 tank
echo "atime=off dedup=off compression=gzip-5" >> bench.csv
bonnie++ -d /tank/ -u root -x 1 -z 0 -q -n 1024 >> bench.csv
iozone -Mce -a -s 1g -f /tank/file -i0 -i1 -i2 -b outb.xls
zpool destroy tank
echo "############################11/16 done"
zpool create tank /dev/sdb
zfs set atime=off tank
zfs set compression=gzip-6 tank
echo "atime=off dedup=off compression=gzip-6" >> bench.csv
bonnie++ -d /tank/ -u root -x 1 -z 0 -q -n 1024 >> bench.csv
iozone -Mce -a -s 1g -f /tank/file -i0 -i1 -i2 -b outc.xls
zpool destroy tank
echo "############################12/16 done"
zpool create tank /dev/sdb
zfs set atime=off tank
zfs set compression=gzip-7 tank
echo "atime=off dedup=off compression=gzip-7" >> bench.csv
bonnie++ -d /tank/ -u root -x 1 -z 0 -q -n 1024 >> bench.csv
iozone -Mce -a -s 1g -f /tank/file -i0 -i1 -i2 -b outd.xls
zpool destroy tank
echo "############################13/16 done"
zpool create tank /dev/sdb
zfs set atime=off tank
zfs set compression=gzip-8 tank
echo "atime=off dedup=off compression=gzip-8" >> bench.csv
bonnie++ -d /tank/ -u root -x 1 -z 0 -q -n 1024 >> bench.csv
```

```
iozone -Mce -a -s 1g -f /tank/file -i0 -i1 -i2 -b oute.xls
zpool destroy tank
echo "############################14/16 done"
zpool create tank /dev/sdb
zfs set atime=off tank
zfs set compression=gzip-9 tank
echo "atime=off dedup=off compression=gzip-9" >> bench.csv
bonnie++ -d /tank/ -u root -x 1 -z 0 -q -n 1024 >> bench.csv
iozone -Mce -a -s 1g -f /tank/file -i0 -i1 -i2 -b outf.xls
zpool destroy tank
echo "############################15/16 done"
zpool create tank /dev/sdb
zfs set atime=off tank
zfs set compression=on tank
zfs set dedup=on tank
echo "atime=off dedup=on compression=on" >> bench.csv
bonnie++ -d /tank/ -u root -x 1 -z 0 -q -n 1024 >> bench.csv
iozone -Mce -a -s 1g -f /tank/file -i0 -i1 -i2 -b outg.xls
zpool destroy tank
echo "finish"
```

Listing 10: Benchmarkscript für Ubuntu

```
#Debug-Kernel installieren fuer oprofile
codename=$(lsb_release -c | awk  '{print $2}')
sudo tee /etc/apt/sources.list.d/ddebs.list << EOF
deb http://ddebs.ubuntu.com/ ${codename}       main restricted
    universe multiverse
deb http://ddebs.ubuntu.com/ ${codename}-security main
    restricted universe multiverse
deb http://ddebs.ubuntu.com/ ${codename}-updates  main
    restricted universe multiverse
deb http://ddebs.ubuntu.com/ ${codename}-proposed main
    restricted universe multiverse
EOF

sudo apt-key adv --keyserver keyserver.ubuntu.com --recv-keys
   ECDCAD72428D7C01
sudo apt-get update
sudo apt-get install linux-image-$(uname -r)-dbgsym

#wrstat und Abhaengigkeiten installieren
apt-get install python-dev git
git clone https://github.com/bolek42/wrstat/
```

```
ln -s /home/stoertebeker/wrstat/wrstat-run /usr/local/bin/wrstat
cp wrstat/wrstat.config.template wrstat/wrstat.config
nano wrstat/wrstat.config #Pfade fuer vmlinux und binaries fuer
    eigenen Kernel ergaenzen
        oprofile_vmlinux /usr/lib/debug/boot/vmlinux-3.13.0-68-
            generic
        oprofile_missing_binaries /lib/modules/3.13.0-68-generic
            /kernel/
install http://sourceforge.net/projects/numpy/files/ (download,
    danach: python setup.py install)
install http://gnuplot-py.sourceforge.net/ (download, danach:
    python setup.py install)
sudo apt-get install gnuplot-x11
```

Listing 11: Installation von wrstat und Abhaenigkeiten

```
yum -y groupinstall "Development Tools"
yum -y install xmlto asciidoc elfutils-libelf-devel zlib-devel
    binutils-devel newt-devel python-devel hmaccalc perl-
    ExtUtils-Embed bison elfutils-devel audit-libs-devel
yum -y install quilt libselinux-devel
git clone git://git.hpdd.intel.com/fs/lustre-release.git
cd lustre-release/
sh ./autogen.sh
yum -y upgrade
reboot
cd lustre-release/
yum -y install python-docutils
./configure --disable-ldiskfs
make rpms
yum -y localinstall --nogpgcheck https://download.fedoraproject.
    org/pub/epel/7/x86_64/e/epel-release-7-5.noarch.rpm
yum -y localinstall --nogpgcheck http://archive.zfsonlinux.org/
    epel/zfs-release.el7.noarch.rpm
yum -y install zfs-dkms libzfs2-devel
cd /var/lib/dkms/spl/0.6.5.3/source/
./configure
make
cd /var/lib/dkms/zfs/0.6.5.3/source/
yum -y install libuuid-devel
./configure
make
cd ~/lustre-release/
```

```
./configure --disable-ldiskfs
make rpms
yum -y install  sg3_utils expect attr lsof
mkdir client
mv *client*.rpm client
rpm -Uvh *.x86_64.rpm
```

Listing 12: Installation von Lustre on ZFS auf CentOS 7

```
#nehalem1.cluster:
mkfs.lustre --mgs --backfstype=zfs --device-size 1310720 lustre-
    mgs/mgs /tmp/lustre-mgs
mkfs.lustre --mdt --backfstype=zfs --device-size 1310720 --index
    =0 --fsname=zfslus --mgsnode=nehalem1@tcp lustre-mdt0/mdt0 /
    tmp/lustre-mdt0
mkfs.lustre --ost --backfstype=zfs --index=0 --fsname=zfslus --
    mgsnode=nehalem1@tcp lustre-ost0/ost0 /dev/sdb
#nehalem2.cluster:
mkfs.lustre --ost --backfstype=zfs --index=1 --fsname=zfslus --
    mgsnode=nehalem1@tcp lustre-ost1/ost0 /dev/sdb
#nehalem3.cluster:
mkfs.lustre --ost --backfstype=zfs --index=2 --fsname=zfslus --
    mgsnode=nehalem1@tcp lustre-ost2/ost0 /dev/sdb

#auf allen drei Nodes:
nano /etc/ldev.conf
-----
nehalem1.cluster - mgs      zfs:lustre-mgs/mgs
nehalem1.cluster - mdt0     zfs:lustre-mdt0/mdt0
nehalem1.cluster - ost0     zfs:lustre-ost0/ost0
nehalem2.cluster - ost1     zfs:lustre-ost1/ost0
nehalem3.cluster - ost2     zfs:lustre-ost2/ost0
-----

systemctl start lustre
mkdir -p /mnt/lustre/client
mount -t lustre nehalem1.cluster:/zfslus /mnt/lustre/client
lfs setstripe -c -1 /mnt/lustre/client/
```

Listing 13: Einrichtung der Targets fuer Lustre

```
yum install gcc gcc-gfortran gcc-c++
mkdir /lustre/software
```

```
cd /lustre/software/
wget http://www.mpich.org/static/downloads/3.2/mpich-3.2.tar.gz
tar xzf mpich-3.2.tar.gz
mkdir /lustre/software/mpich3
cd mpich-3.2
./configure --prefix=/lustre/software/mpich3/
make
make install

cd /lustre/software/
yum -y install git
git clone https://github.com/chaos/ior.git
mv ior ior_src
mkdir /lustre/software/ior
cd ior_src/
./bootstrap
./configure --prefix=/lustre/software/ior/
make
make install
```

Listing 14: Installation von MPICH und ior

```
##########default:
sent 1024786032326 bytes  received 110338 bytes  78518648.64
    bytes/sec
total size is 1024660531722  speedup is 1.00

real    217m30.527s
user    133m42.304s
sys     68m23.729s

du -s -BM       978621M
 Total disk usage:    0.9 TiB  Apparent size:    0.9 TiB  Items:
     5902

 ##########LZ4
 sent 1024786032326 bytes  received 110338 bytes  75183312.62
     bytes/sec
total size is 1024660531722  speedup is 1.00

real    227m10.272s
user    133m11.690s
sys     68m3.047s
```

```
du -s -BM   627305M
 Total disk usage: 612.6 GiB  Apparent size:   0.9 TiB  Items:
    5902

 ########GZIP
 sent 1024786032326 bytes  received 110338 bytes  76257479.83
    bytes/sec
total size is 1024660531722  speedup is 1.00

real    223m58.709s
user    111m13.986s
sys     60m11.878s

du -s -BM   469593M
 Total disk usage: 458.6 GiB  Apparent size:   0.9 TiB  Items:
    5902

########Dedup
sent 1024786032326 bytes  received 110338 bytes  34750882.27
   bytes/sec
total size is 1024660531722  speedup is 1.00

real    491m28.573s
user    133m56.572s
sys     68m37.262s

du -s -BM 978622M
 Total disk usage:   0.9 TiB  Apparent size:   0.9 TiB  Items:
    5902
Dedup-Faktor: 1.22
######### Dedup + LZ4
sent 1024786032326 bytes  received 110338 bytes  31004799.72
   bytes/sec
total size is 1024660531722  speedup is 1.00

real    550m51.974s
user    128m27.895s
sys     67m45.197s

du -s -BM 627311M
Total disk usage: 612.6 GiB  Apparent size:   0.9 TiB  Items:
   5902
Kompression: 1,54
```

```
Dedup-Faktor:  1.26
```

Listing 15: Kompressionstest mit Beispieldaten aus dem wr-Cluster

Literatur

[1] *admin-magazin.de: ZFS für Linux einsatzbereit*. 2106. URL: http://www.admin-magazin.de/News/ZFS-fuer-Linux-einsatzbereit.

[2] *DE Wikipedia: ZFS*. 2016. URL: https://de.wikipedia.org/wiki/ZFS_%28Dateisystem%29.

[3] *EN Wikipedia: ZFS*. 2016. URL: https://en.wikipedia.org/wiki/ZFS.

[4] *Explaining Bonnie++*. 2016. URL: http://www.orangefs.org/trac/orangefs/wiki/ExplainBonnie.

[5] *Github: Stuck txg_sync process, and IO speed issues when ARC is full*. 2012. URL: https://github.com/zfsonlinux/zfs/issues/3235.

[6] *https://github.com/bolek42/wrstat*. 2016. URL: https://github.com/bolek42/wrstat.

[7] *Lustre Version 2.4 Released*. 2016. URL: http://opensfs.org/press-releases/lustre-file-system-version-2-4-released/.

[8] Hajo Möller. *Leistungsanalyse von ZFS on Linux*. 2016.

[9] *OpenZFS: Features*. 2016. URL: http://open-zfs.org/wiki/Features.

[10] *pro-linux.de: Update, aktueller Stand von ZoL*. 2104. URL: http://www.pro-linux.de/news/1/21509/aktualisierung-der-stand-von-zfs-fuer-linux.html.

[11] *Slow performance due to txg_sync for ZFS 0.6.3 on Ubuntu 14.04*. URL: http://serverfault.com/questions/661336/slow-performance-due-to-txg-sync-for-zfs-0-6-3-on-ubuntu-14-04.

[12] *Spider: The center wide file system*. 2016. URL: https://www.olcf.ornl.gov/kb_articles/spider-the-center-wide-lustre-file-system/.

[13] Prakash Surya. *OpenZFS: Reducing ARC Lock Contention*. 2015.

[14] *ZFS Administration, Part III- The ZFS Intent Log*. 2012. URL: https://pthree.org/2012/12/06/zfs-administration-part-iii-the-zfs-intent-log/.

[15] *ZFS Compile Workaround*. 2016. URL: https://github.com/zfsonlinux/zfs/issues/2824.

[16] *ZFS Tuning Guide*. 2016. URL: https://wiki.freebsd.org/ZFSTuningGuide#Deduplication.