Leistungsanalyse von ZFS on Linux
Projekt Parallelrechnerevaluation

Hajo Möller

28. März 2013
WS 2012/13

Inhaltsverzeichnis

Zusammenfassung

Diese Arbeit untersucht anhand des Dateisystem-Benchmarks *Bonnie++ 1.96*[1] die Leistung der nativen Implementierung des modernen Dateisystems ZFS unter Linux, *ZFS on Linux*[2] und stellt Vergleiche bezüglich verschiedener grundlegender Speichermedien, aktivierten Optionen wie Komprimierung, Deduplikation und Spiegelung dar. Schließlich wird ein kurzer Vergleich mit etablierten Journaling-Dateisystemen anhand der Leistung von *ext4* auf denselben Speichermedien gezogen. Die Arbeit schließt mit einem Ausblick auf in Zukunft zu erwartende Ergebnisse.

1 Einführung

Die Grundlage dieser Arbeit sind vormals wahrgenommene, deutliche Leistungseinbußen von *ZFS on Linux* - Metadatenoperationen (wie die Erstellung von Dateien) waren auf ZFS unerwartet deutlich langsamer als auf konkurrierenden Dateisystemen wie *ext4* und bewegten sich bei etwa 1000 bis 5000 Operationen pro Sekunde. Da dies von mir nicht reproduziert werden konnte habe ich mich auf die allgemeine Analyse von ZFS unter Linux konzentriert. Da der ursprünglich benutzte, rein Metadaten testende Benchmark mdtest[3] zwar auf dem Testsystem kompiliert, jedoch nicht mehr auf einem ZFS-Dateisystem zu Laufen gebracht werden konnte[4], habe ich auf den Benchmark *Bonnie++* zurückgegriffen.

1.1 ZFS

ZFS ist ein transaktionsbasiertes Copy-On-Write-Dateisystem mit eingeschlossenem Volume Manager, welches seit 2001 von Sun Microsystems (jetzt Oracle Corporation) für das Betriebssystem *Solaris 10* entwickelt wird. ZFS kann nicht direkt innerhalb des Linux-Quelltexts eingeschlossen werden, da ZFS unter der *CDDL*[5] und damit einer nicht mit der von Linux genutzten *GPL 2.0*[6] kompatiblen Lizenz steht.

[1] `http://www.coker.com.au/bonnie++/`

[2] `http://zfsonlinux.org/`

[3] `http://sourceforge.net/projects/mdtest/`

[4] Fehlermeldung von mdtest:

```
Command line used: /root/mdtest-1.8.4/mdtest -b 3 -z 1 -I 10 -i 10
03/20/2013 15:07:57: Process 0(): FAILED in show_file_system_size, unable
to statfs() file system: No such file or directory
```

[5] `https://oss.oracle.com/licenses/CDDL`

[6] `http://www.gnu.org/licenses/old-licenses/gpl-2.0.html`

ZFS besitzt einige Features, die es nicht nur für den High-Performance-Computing-Bereich zu einem sehr interessanten Dateisystem machen. Durch die bereits erwähnte Copy-On-Write-Eigenschaft ist das Dateisystem gerade bei parallelen Zugriffen konsistent und performant. Weitere Features sind die Möglichkeiten, die zu schreibenden Daten online, also während des Schreibvorgangs, zu komprimieren und/oder zu deduplizieren und so zum Teil wesentlich mehr Daten ablegen zu können als es der rohe Speicherplatz zuließe. Dies geht jedoch zu Lasten der CPU und Arbeitsspeicher, wobei gerade die Deduplizierung spürbar Ressourcen verbraucht.
Seit 2008[1] wird hauptsächlich durch das *Lawrence Livermore National Laboratory* eine native Implementierung von ZFS als Linux-Kernelmodul, *ZFS on Linux*, abseits vom Linux-Kernel-Quelltext entwickelt, welche nicht durch den CDDL/GPL-Konflikt behindert wird. Die von *ZFS on Linux* erzeugten Dateisysteme sind bis zur Version `0.6.0-rc14` vom 1. Februar 2013 vollständig kompatibel zu der aktuellen Solaris-Implementierung. Seit `0.6.0-rc14` wird intern eine zpool-Version von 5000 genutzt, um Kompatibilität mit dem OpenSolaris-Fork *illumos*[7] wiederherzustellen.

1.2 bonnie++

Bonnie++[8] ist ein auf *Bonnie*[9] basierender, inzwischen vollständig neu entwickelter Geschwindigkeitstest (Benchmark) für Dateisysteme und darunterliegende Speichermedien.
Bonnie++ testet bei Standardeinstellungen auf folgende Weise[2], es werden dazu Dateien mit einer Gesamtgröße von $2 \cdot$ `RAM` erzeugt um zu verhindern, dass die Daten vollständig gecached werden:

1. *Sequential Output - Sequentielles Schreiben*
   (a) *Per-Character*, die Dateien werden erstellt, dann per `putc()` Zeichen für Zeichen geschrieben bis die Gesamtgröße der doppelten Größe des im System verbauten Arbeitsspeichers beträgt.
   (b) *Block*, die Dateien werden nun per `write` in ganzen Blöcken des Dateisystems geschrieben.
   (c) *Rewrite*, wobei die Dateien blockweise gelesen, die Blockdaten geändert und erneut geschrieben werden. Die Dateigröße ändert sich nicht, die Dateien werden nur vollständig überschrieben.

[7] `http://illumos.org`
[8] `http://www.coker.com.au/bonnie++/`
[9] `http://www.textuality.com/bonnie/`

2. *Sequential Input - Sequentielles Lesen*

   (a) *Per-Character*, die vorher erstellten Dateien werden nun zeichenweise per `getc()` eingelesen.

   (b) *Block*, die Dateien werden in ganzen Blöcken, also so schnell wie möglich, eingelesen.

3. *Random Seeks - Zufällige Spurwechsel*. Es werden 3 parallele Prozesse gestartet, welche je 8000 zufällige `lseek()`-Operationen, also Spurwechsel, auf den Dateien durchführen, dort einen Block lesen und in 10% der Fällen die Blockdaten ändern und den Block zurückschreiben.

Für diese Arbeit wurden die Standardeinstellungen weitestgehend beibehalten, es wurde lediglich auf die Per-Character-Tests verzichtet, da unsere Daten üblicherweise blockweise statt zeichenweise gelesen und geschrieben werden.

2 Leistungsmessung

2.1 Testaufbau

Bei dem eingesetzten Testrechner handelt es sich um einen Rechner mit einem Prozessor der Intel Sandy-Bridge-Generation mit 16 GB Arbeitsspeicher, dem als Massenspeicher drei baugleiche, herkömmliche Festplatten von WD mit einer Größe von je 2 TB und einer nativen Sektorgröße von 4096 Bytes sowie eine SSD mit 256 GB Speicherplatz zur Verfügung standen.
Als Betriebssystem kommt *Ubuntu 12.04 LTS* mit einem Kernel in Version `3.5.0-25`, also dem Backport von *Ubuntu 12.10*, zum Einsatz. Die verwendete Version von *ZFS on Linux* stammt aus dem PPA *ZFS Stable Releases for Ubuntu*[10] und entspricht dem Release Candidate `0.6.0-rc14`, da die stabile Version `0.6.1` zum Zeitpunkt dieser Untersuchung noch nicht veröffentlicht war.
Die Speichermedien sind folgendermaßen konfiguriert:

- */dev/sda*: 2 TB Festplatte, vollständig für das System genutzt
- */dev/sdb*: 2 TB Festplatte, frei
- */dev/sdc*: 2 TB Festplatte, frei
- */dev/sdd*: 256 GB SSD, frei

[10] `https://launchpad.net/~zfs--native/+archive/stable`

2.2 Konfiguration des ZFS

Das jeweils getestete ZFS-Dateisystem, der sogenannte Pool, wurde mit verschiedenen Konfigurationen eingerichtet, wobei für die Daten der vollständig verfügbare Speicherplatz zur Verfügung stand indem das ZFS auf den ganzen Devices (`/dev/sdb, ...`) und nicht auf Partitionen (`/dev/sdb1, ...`) angelegt wurde. Der ZFS-Pool *tank* wurde auf einer Festplatte, zwei gespiegelten Festplatten und anschließend auf der SSD abgelegt, wobei jeweils ein Testlauf mit einer manuell festgelegten minimalen Blockgröße von 4096 Bytes und einer mit einer minimalen Blockgröße von 8192 Bytes durchgeführt wurde:

```
# 1. Auf /dev/sdb mit minimaler Blockgroesse von 4096 Bytes (-o ashift=12):
zpool create -o ashift=12 tank /dev/sdb

# 2. Auf /dev/sdb mit minimaler Blockgroesse von 8192 Bytes (-o ashift=13):
zpool create -o ashift=13 tank /dev/sdb

# 3. Auf /dev/sdb und /dev/sdc als gespiegelter Verbund, aehnlich RAID-1:
zpool create -o ashift=12 tank mirror /dev/sdb /dev/sdc

# 4. wie 3., mit minimaler Blockgroesse von 8192 Bytes:
zpool create -o ashift=13 tank mirror /dev/sdb /dev/sdc

# 5. Auf /dev/sdd, der SSD, mit 4096 Bytes Blockgroesse:
zpool create -o ashift=12 tank /dev/sdd

# 6. wie 5., mit 8192 Bytes Blockgroesse:
zpool create -o ashift=13 tank /dev/sdd
```

Anschließend wurden in dieser Reihenfolge

1. keine Änderungen vorgenommen,
2. Komprimierung aktiviert,
3. Deduplizierung bei abgeschalteter Komprimierung aktiviert,
4. Komprimierung als auch Duplizierung aktiviert.

```
# 2.
zfs set compression=on tank

# 3.
zfs set compression=off tank
zfs set dedup=on tank

# 4.
zfs set compression=on tank
```

Ein ZFS kann, wie noch gezeigt wird, die Leistung deutlich verbessern, wenn man ein dediziertes, schnelles Laufwerk für das *ZIL (ZFS Intent Log)*, dem Transaktionsjournal und somit Schreibcache des Dateisystems hinzufügt. Empfohlen sind mindestens zwei physische Laufwerke, die gespiegelt zu einem logischen Laufwerk als *SLOG* an den bestehenden Pool angehängt werden. Da das ZIL nur für anstehende Transaktionen genutzt wird genügt es, kleine Partitionen auf SSDs als SLOG zu nutzen.[3]
Ferner kann, gerade bei wenig verfügbarem Arbeitsspeicher (weniger als 1-2 GB je TB rohen Speicherplatz), die Performanz des ZFS durch den Einsatz eines schnellen Cache-Laufwerks deutlich verbessert werden. Das ZFS-System reserviert sich bis zu 87,5% des verfügbaren Arbeitsspeichers als Cache, den sogenannten *ARC (Adjustable Replacement Cache)*.
Dies ist jedoch gerade bei wenig verfügbarem RAM und aktivierter Deduplizierung sehr kritisch, da der *ARC* die vollständige Deduplizierungstabelle beinhalten muss. Um den Nachteil von wenig verbautem Arbeitsspeicher auszugleichen kann man den *ARC* um eine weitere Ebene, den *L2ARC (Level 2 ARC)*, erweitern indem man ein schnelles Speichermedium als Cache-Laufwerk hinzufügt.[4]
Für diese Arbeit wurde dazu die SSD, sofern sie nicht selbst als Grundlage des Pools benutzt wird, so umpartitioniert:

```
# /dev/sdd1 wird als primaere Partition mit einer Groesse von 4 GB angelegt,
# /dev/sdd2 bekommt, ebenfalls als primaere Partition, den restlichen
    Speicherplatz:
parted -s -- /dev/sdd unit MiB mklabel gpt mkpart primary 1 4097 mkpart
    primary 4097 -0
```

Jetzt kann `/dev/sdd1` als *SLOG* und `/dev/sdd2` als zusätzliches Cache-Device für den *L2ARC* eingesetzt werden:

```
# 1. /dev/sdd1 als SLOG mit 4096 Byte-Sektoren:
zpool add -o ashift=12 tank log /dev/sdd1

# 2. wie 1., mit minimaler Blockgroesse von 8192 Bytes:
zpool add -o ashift=13 tank log /dev/sdd1

# 3. /dev/sdd2 als L2ARC bzw. Cache-Device, hier kann keine minimale
    Blockgroesse vorgegeben werden:
zpool add tank cache /dev/sdd2
```

Nach jedem Testdurchlauf wurde der für den Test erstellte Pool per `zpool destroy tank` zerstört, damit die Festplatten und die SSD für den nächsten Test korrekt eingerichtet werden können.
So entstanden 26 verschiedene Grundkonfigurationen, für die jeweils 4 Tests (ohne Komprimierung/Deduplizierung, je einzeln aktiviert, beide aktiv) durch-

geführt wurden. Ein Testlauf dauerte im Schnitt gut 30 Minuten, so dass der Rechner insgesamt $26 \cdot 4 \cdot 30$ Minuten = 52 Stunden mit der Durchführung des Benchmarks beschäftigt war.

2.3 Durchführung

Die Leistungsmessung erfolgte mit *Bonnie++ 1.96* auf der unter „Testaufbau“ beschriebenen Hardware unter verschiedenen Konfigurationen.
Gemessen wurde die sequentielle Ein- und Ausgabe sowie die für vorgenommene Spurwechsel und wenige Änderungen benötigte Zeit, wobei die Kommandozeilenparameter für *Bonnie++* wie folgt waren:

```
1 bonnie++ -d /tank/ -u root -n 128 -x 1 -q -z 0 -f 2>/dev/null
```

Die angegebenen Parameter bedeuten:

- `-d /tank/`: Benutze das Verzeichnis `/tank/`, d.h. das vorher erstellte ZFS-Dateisystem, für den Benchmark.
- `-u root`: Führe den Test als Benutzer `root` aus, muss zwingend angegeben werden da *bonnie++* sonst die Ausführung als `root` verweigert.
- `-n 128`: Erstelle $128 \cdot 1024$ Dateien, statt nur 1024 Dateien. Ohne diese Angabe sind die Metadatenoperationen zu schnell abgeschlossen (unter 500 ms) um zuverlässig gemessen werden zu können.
- `-x 1`: Führe die Messung einmalig durch.
- `-q`: Gib nur die nötigsten Informationen aus, um ausschließlich die Messergebnisse als CSV-Daten auszugeben.
- `-z 0`: Setzt den Startwert des Zufallsgenerators auf „0“, um bei wiederholtem Aufruf exakt die selben Messungen durchzuführen.
- `-f`: Deaktiviert die Per-Character-Tests.

Die so ausgegebenen CSV-Daten wurden anschließend gesammelt und ausgewertet[11], wobei in dieser Arbeit nur Auszüge dargestellt werden.

[11] Die bei Konfiguration von mehreren Durchläufen (`-x 2`, ...) durch *Bonnie++* ausgegebenen CSV-Headerdaten sind fehlerhaft, statt „`[...],name,file_size,[...]`“ muss es „`[...],name,concurrency,seed,file_size,[...]`“ lauten. Dieser Bug ist in *Bonnie++* Version 1.97 behoben.

3 Auswertung

Die aufbereiteten Daten sind an dieses Dokument angehängt und befinden sich in „Ergebnisse des Benchmarks“, hier folgt eine kurze Auswertung besonders bemerkenswerter Ergebnisse.

3.1 Blockweises sequentielles Schreiben

Betrachten wir als Erstes die sequentielle, blockweise Erstellung von Dateien, dabei ist die minimale Blockgröße 4096 Bytes:

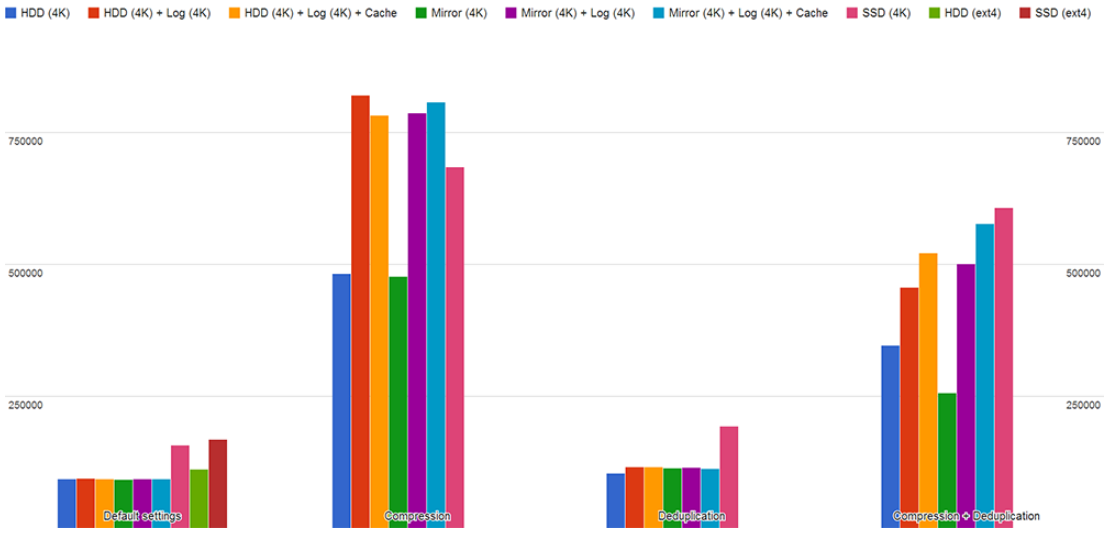

Abbildung 1: Schreibgeschwindigkeit, in x * 1000 Blöcken / Sekunde

Wie wir sehen können liegt die Geschwindigkeit von *ZFS* allgemein leicht unter der von *ext4*, unabhängig vom darunterliegenden Speichermedium.
Weiterhin ist sichtbar, dass bei deaktivierter Komprimierung und Deduplizierung eventuell verfügbare Log- und Cache-Devices keinen Einfluss auf die Schreibgeschwindigkeiten haben.
Interessanter sind die Balken bei aktivierter Komprimierung, wo wir eine Steigerung um einen Faktor von mindestens 5 sehen. Wenn zusätzlich zur Komprimierung auch eine SSD als SLOG bereitgestellt wird steigt die Geschwindigkeitszunahme sogar auf Faktor 8.
Bei Aktivierung von der Deduplizierung lassen sich hingegen nur leichte Performancegewinne beobachten.
Bemerkenswert ist außerdem, dass bei aktivierter Komprimierung und Deduplizierung ohne SLOG ein Mirror deutlich (etwa 20%) langsamer als eine einzelne Festplatte ist. Sobald die SSD als SLOG hinzugefügt wird kehrt

sich der Nachteil um und der Mirror ist etwa 20% schneller als die einzelne Festplatte.
Es hat sich herausgestellt, dass eine vorgegebene minimale Blockgröße von 8192 Bytes statt 4096 Bytes sich nur bei gleichzeitig aktivierter Komprimierung auswirkt:

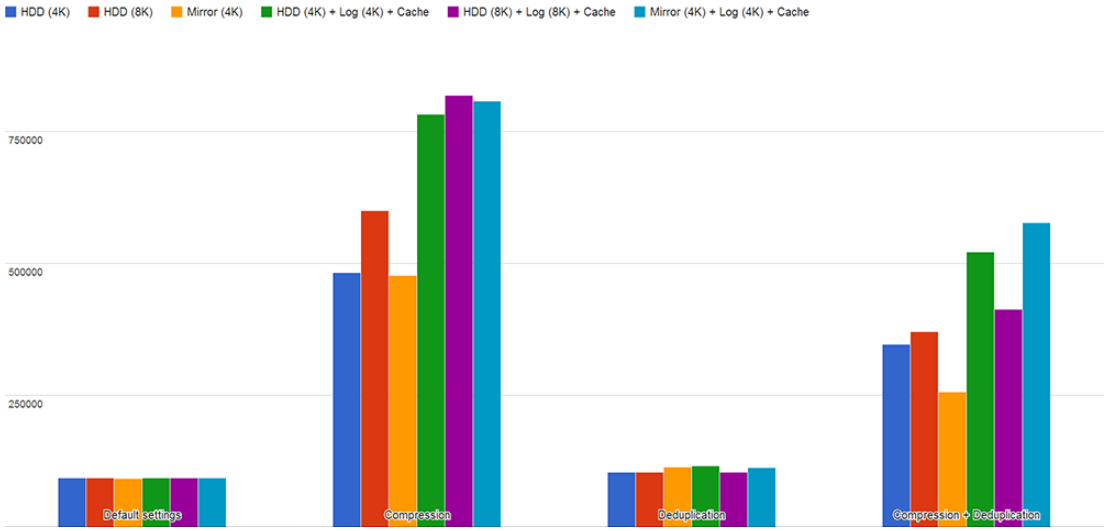

Abbildung 2: Schreibgeschwindigkeit, in x * 1000 Blöcken / Sekunde

Da die CPU-Auslastung in beiden Fällen deutlich unter 100% lag lässt dies darauf schließen, dass der verwendete Kompressionsalgorithmus auf größere Blockgrößen optimiert ist.

3.2 Blockweises sequentielles Lesen

Beim sequentiellen Lesen können wir sehen, dass die Geschwindigkeit von *ZFS* ohne Komprimierung oder Deduplizierung leicht unter der von *ext4* liegt. Weiterhin bleibt die Geschwindigkeit eines Mirrors stets unter der eines ZFS auf einer SSD:

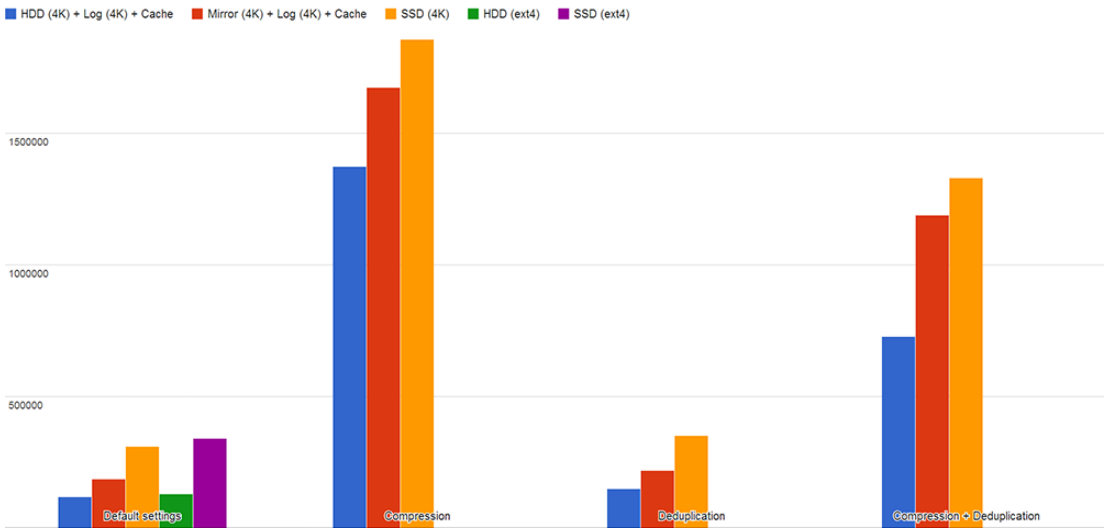

Abbildung 3: Lesegeschwindigkeit, in x * 1000 Blöcken / Sekunde

Sobald die Deduplizierung aktiviert wird ist ein *ZFS* beim sequentiellen Lesen leicht schneller als ein *ext4*, noch eine deutlichere Steigerung lässt sich bei gleichzeitiger Aktivierung der Komprimierung beobachten.
Es bleibt zu bemerken, dass ein ZFS ohne Deduplizierung aber mit Komprimierung über zehn Mal so schnell ist wie ein ZFS ohne Komprimierung oder Deduplizierung oder ein ext4.
Auch ist der Vorteil eines Mirrors gegenüber einzelner Platten sehr deutlich erkennbar, da beide Festplatten die selben Daten enthalten kann das ZFS die angeforderten Daten möglichst effizient von beiden Platten gleichzeitig lesen und den SSD-Cache füllen, so dass fast reine SSD-Lesegeschwindigkeiten erreicht werden können.

3.3 Erstellen von Dateien in zufälliger Reihenfolge

Da kein nennenswerter Unterschied der Leistung zwischen sequentiellem und zufälligem Erstellen von Dateien gemessen wurde zeige ich nur die Daten für randomisiertes Erstellen. Die weiteren Messergebnisse finden sich im Anhang unter Ergebnisse des Benchmarks.
Die Erstellung von leeren Dateien ist eine reine Metadaten-Operation, die theoretisch rein von der Leistungsfähigkeit des Prozessors abhängig ist. Unter ZFS war der Prozessor tatsächlich zu fast 100% ausgelastet, unter ext4 jedoch nur zu etwa 50%.
Komprimierung und Deduplizierung sollten die Ergebnisse nicht beeinflussen.

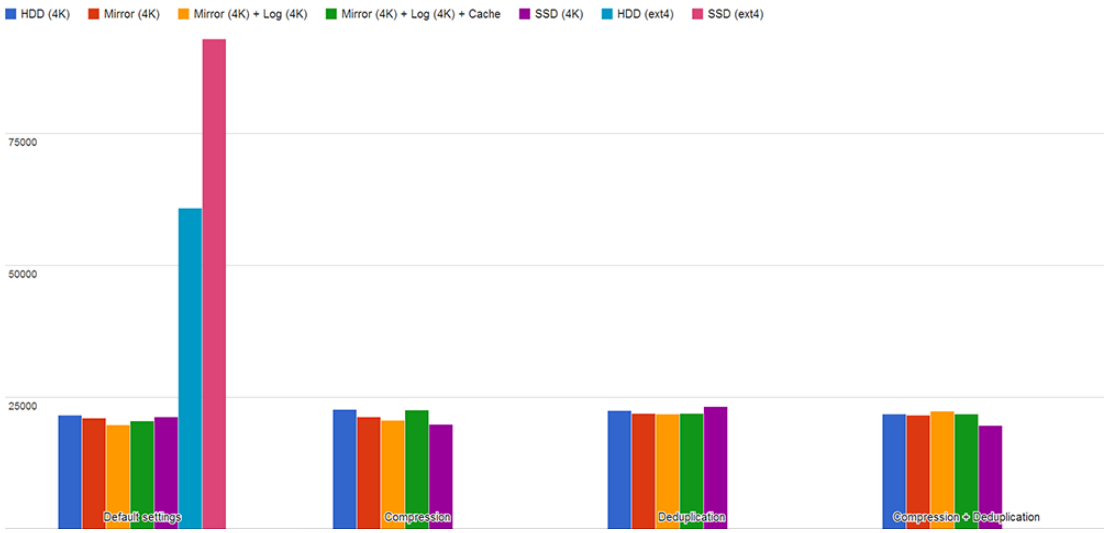

Abbildung 4: Erstellte Dateien je Sekunde

Die Vermutung, dass Deduplizierung und Komprimierung sich nicht bemerkbar machen kann bestätigt werden. Überraschend ist jedoch, dass auch das unter dem ZFS liegende Speichermedium keine Rolle spielt. Es lässt sich kein messbarer Leistungsunterschied zwischen einer einzelnen Festplatte, einem gespiegelten Verbund und einer SSD feststellen.
Da *ZFS on Linux* nach eigener Aussage[5] noch nicht vollständig auf Performance optimiert wurde, sollten die vorliegenden Ergebnisse nicht als endgültig angesehen werden. Unsere Erfahrung zeigt, dass sich die Leistung von ZFS langsam aber stetig verbessert. Die gemessenen Werte von etwa 20000 erstellten Dateien pro Sekunde sind deutlich besser als die berichteten und ursprünglich angenommenen 1000 bis 5000. Da die CPU voll ausgelastet ist lässt sich sagen, dass ein zukünftiger, weiterer Performancegewinn möglich und wahrscheinlich ist.

4 Abschließende Beurteilung

Wie wir gesehen haben ist *ZFS* ein für die mitgebrachten Features sehr performantes Dateisystem.
Ohne aktivierte Komprimierung und Deduplizierung ist *ZFS* beim Lesen und Schreiben nur leicht langsamer als *ext4*, bietet aber durch stetiges Checksumming eine wesentlich höheren Schutz vor Datenverlust durch Hardwareschäden.
Sobald die sehr effiziente und ressourcenschonende Komprimierung angeschaltet wird lassen sich deutliche Performancegewinne sowohl beim Lesen als auch beim Schreiben erzielen, so dass ein *ZFS* bis zu der zehnfachen Geschwindigkeit eines *ext4s* erreichen kann.
Das stark beanspruchende Deduplizieren verschafft hingegen im Benchmark nur leichte Performanceänderungen, spart aber deutlich rohen Speicherplatz indem Blöcke mit der selben SHA256-Checksumme nur ein Mal abgelegt und dann intern in der Deduplizierungstabelle referenziert werden. Legt man viele Dateien mit sehr ähnlichen Inhalten ab ist ein größerer Gewinn zu erwarten. Reicht jedoch der Arbeitsspeicher nicht aus, um die Deduplizierungstabelle komplett zu halten lassen sich erfahrungsgemäß drastische Einbußen beobachten, so sank bei einem unabhängigen Test auf einem Rechner mit entsprechend wenig RAM die Schreibgeschwindigkeit von 60 MB/s auf unter 5 MB/s.
Da inzwischen das Solaris-Tracingwerkzeug DTrace auf Linux portiert wurde[12] und sich viele DTrace-Skripte zur Messung von ZFS im Internet finden lassen[13], sowie das Projekt *ZFS on Linux* gerade eine neue, stabile Version von ZFS für Linux veröffentlich hat ist es nur noch eine Frage der Zeit, bis große Fortschritte in der Perfomance gemacht werden.

[12] `https://github.com/dtrace4linux/linux`

[13] Einige Beispiele sind unter `https://github.com/siebenmann/cks-dtrace/` verfügbar

Anhang

A Skript `benchmark.sh`

Da, wie in „Konfiguration des ZFS“ erklärt, der gesamte Testablauf 52 Stunden in Anspruch nahm wurde das Skript zur Sicherheit auf dem Rechner in einem `screen`-Prozess gestartet:

```
1 screen ./benchmark.sh
```

Das Bashskript `benchmark.sh`.

B Ergebnisse des Benchmarks

Die von *Bonnie++* erzeugten CSV-Daten sind aufbereitet und ergänzt an dieses Dokument angehängt:

- `Ergebnisse.ods` im OpenDocument Spreadsheet-Format.
- `Ergebnisse.xlsx` im Microsoft Excel-Format.
- `Ergebnisse.csv` im CSV-Format.

C Eingebundene Bilder in hoher Qualität

Die in diesem Dokument eingebundenen Bilder sind hier nochmal in hoher Qualität angehängt:

- `put_block-4k.png` Blockweises sequentielles Schreiben mit Mindestblockgröße 4096 Bytes.
- `put_block-8k.png` Blockweises sequentielles Schreiben mit Mindestblockgröße 8192 Bytes.
- `get_block.png` Blockweises sequentielles Lesen.
- `create.png` Dateierstellung in zufälliger Reihenfolge.

Literatur

[1] *ZFS on Linux: ChangeLog*; https://github.com/zfsonlinux/zfs/blob/34dc7c2f2553220ebc6e29ca195fb6d57155f95f/ChangeLog; ChangeLog

[2] *Bonnie++ Documentation*; http://www.coker.com.au/bonnie++/readme.html

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

[4] *ZFS Administration, Part IV- The Adjustable Replacement Cache*; http://pthree.org/2012/12/07/zfs-administration-part-iv-the-adjustable-replacement-cache/

[5] *ZFS on Linux FAQ: What's going on with performance?*; http://zfsonlinux.org/faq.html#PerformanceConsideration