

Vergleich von Linux- Dateisystemen

Projekt für das Internet- Praktikum: Analyse von Systemperformanz
im WS04/05, TU München

Heike Lupold, Diana Tanasescu, Hubert Eichner, Stefan Förster

Inhaltsverzeichnis

Inhaltsverzeichnis.....	2
1 Einleitung	3
2 Tests	3
2.1 Dateisysteme	3
2.1.1 XFS.....	3
2.1.2 JFS	3
2.1.3 ReiserFS.....	3
2.1.4 ext3.....	3
2.2 Betrachtete Faktoren	4
2.3 Feste Parameter	4
2.4 Ablauf	4
3 Resultate	7
3.1 Ergebnisüberblick	7
3.2 recls1	8
3.3 recls2	9
3.4 read1	9
3.5 read2	10
3.6 read3	10
3.7 write	11
3.8 create filetree	11
3.9 delete	12
4 Schlussfolgerung.....	13
Anhang : Mittelwerte der Ergebnisse	14

1 Einleitung

In dem nachfolgenden Projekt sollen gängige Dateisysteme für das Betriebssystem Linux getestet werden. Dabei sollen die Dateisysteme insbesondere verglichen werden, wie sie die gängigen Anforderungen erfüllen. Diese sind:

- ein schnelles Recovery nach einem Absturz, welches durch das so genannte "Journaling" von Daten erreicht wird,
- die Stabilität in allen Anwendungsfällen und
- die Skalierbarkeit.

In diesem Projekt werden nur Journaling Dateisysteme getestet, daher wird das alte Standard-Dateisystem „ext2“ nicht beachtet.

Journaling Dateisysteme bieten im Vergleich zu einfachen Dateisystemen den Vorteil, dass geänderte Dateien in einem Journal erfasst werden und die geänderten Dateien bis zur Beendigung des Schreibvorgangs ihre Gültigkeit behalten. Nach einem Systemabsturz muss daher nicht das gesamte Betriebssystem überprüft und repariert werden, sondern lediglich die zu dem Zeitpunkt bearbeiteten Dateien. Dadurch wird eine hohe Zeitersparnis beim Hochfahren ermöglicht.

2 Tests

2.1 Dateisysteme

Folgende Journaling Dateisysteme für Linux werden im Rahmen des Projekts getestet:

2.1.1 XFS

Das Dateisystem XFS wurde ursprünglich von der Firma SGI für deren Betriebssystem IRIX entwickelt. Seit Anfang 2001 existiert auch eine Portierung für Linux. Das XFS wird seit der Kernel Version 2.6. standardmäßig implementiert.

2.1.2 JFS

Das Journaled File System wurde ursprünglich im Jahr 2000 von IBM für das eigene Betriebssystem OS/2 entwickelt um besonders bei Dateiservern einen hohen Datendurchsatz sicher zu stellen. Seit 2002 ist das JFS auch für Linux verfügbar.

2.1.3 ReiserFS

ReiserFS ist ein maßgeblich von Hans Reiser mitgestaltetes Filesystem, dessen größte Stärke in der platzsparenden Speicherung vieler kleiner Dateien besteht. ReiserFS ist das erste Journaling Dateisystem, das im Standard-Linux-Kernel implementiert wurde.

2.1.4 ext3

Das „Third Extended Filesystem“ verfügt im Gegensatz zu dem Standard-Dateisystem ext2 über die Journaling Funktion und stellt mittlerweile eine ernstzunehmende Konkurrenz für die oben genannten Filesysteme dar.

2.2 Betrachtete Faktoren

Bei den betrachteten Dateisystemen werden Eigenschaften wie Stabilität und Skalierbarkeit als selbstverständlich angesehen. Daher erfolgt der Performancevergleich der Dateisysteme beim Bearbeiten von

- Metadaten (Dateigröße, Zugriffsrechte etc.),
- Nutzdaten (eigentliche Dateiinhalte),
- mit und ohne parallele Zugriffe.

Als Vergleichsgröße dient die Ausführungszeit der einzelnen Tests.

Die Faktoren bei der Testausführung sind

- das getestete Filesystem (vier Niveaus),
- Parallelität (zwei Niveaus),
- Meta- und/oder Nutzdaten (drei Niveaus).

Die Ausführungszeiten dieser Tests werden zusätzlich noch nach den jeweils ausgeführten Systemaufrufen aufgeschlüsselt und verglichen.

2.3 Feste Parameter

Der für die Tests verwendete Rechner ist ein Athlon XP 2600+ mit Debian unstable auf einem 2.6.9-Betriebssystemkern.

Die verwendete Festplatte ist eine 200GB Hitachi HDS722525VLAT80 Deskstar 7K250, welche von uns im UDAM5-Modus betrieben wird.

Die Tests werden ohne dateisystemspezifische Optimierungsmaßnahmen durchgeführt.

2.4 Ablauf

Die Steuerung des Testablaufs übernimmt ein Perlskript als Framework. Dieses führt die Einzeltests vollautomatisch durch, falls es beim Hochfahren gestartet wird und der Rechner nach einem Absturz wieder hochgefahren wird.

Das Skript führt ein Logbuch („journal.log“), in dem vermerkt wird, auf welchem Stand sich das Skript mit welchem Einzeltest befindet.

Jeder dieser Einzeltests besteht aus zwei oder drei einzelnen Schritten:

Der erste Schritt ist das Anlegen des gewünschten Filesystems.

Der zweite Schritt ist optional und wird als "test preparation" bezeichnet. z.B. das Anlegen eines Dateibaumes.

Der dritte Schritt stellt den eigentlichen Test dar.

Im Logbuch werden sowohl Beginn und Abschluss dieser einzelnen Schritte vermerkt, als auch der aktuelle, gerade bearbeitete Test.

Für den Fall, dass eine dieser drei Phasen fehl schlägt oder durch einen Absturz unterbrochen wird (dieser kann durch den aktuellen Test bedingt sein), merkt das Skript dies und springt zum nächsten Test.

Weiterhin dient das Logbuch dazu, nach einem Reboot die Tests an der richtigen Stelle fortführen zu können. Ein Reboot wird standardmäßig nach dem Abschluss der Testvorbereitung bzw. nach der eigentlichen Testdurchführung ausgeführt, um die Caches des Dateisystems zu leeren, die ansonsten die Ergebnisse verfälschen können.

Das Perlskript schlägt fehl, wenn ein neuer Testdurchlauf gestartet werden soll, aber das alte Journal noch nicht entfernt wurde.

Die eigentlichen Testanweisungen erwartet das Skript standardmäßig in der Datei „control.ctl“, welche Zeile für Zeile Anweisungen der Form
<Filesystem> <Testname> <Wiederholungen des Tests>
enthält.

Des Weiteren erwartet das Perlskript drei Arten von Skripten:

- Skript zum Anlegen von Dateisystemen "filesystem-<name>.sh"
- optionales Vorbereitungsskript "test-<name>-prepare.sh"
- eigentliches Testskript "test-<name>-perform.sh".

Die Ergebnisse des Testdurchlaufs werden in die Ergebnisdatei „results.log“ geschrieben im Format:

<testname> <filesystem> <duration>.

Die Zeitmessung erfolgt mit dem Perl-Modul Time::HiRes.

Da nicht alle Faktorkombinationen für den realen Einsatz relevant sind, werden manche in den Tests ausgelassen, dagegen werden andere doppelt ausgeführt. Dies geschieht auch, um gegebenenfalls Unterschiede in der Implementierung der einzelnen Systemaufrufe festzustellen. Insgesamt werden folgende acht Tests ausgeführt:

Testname: *create-filetree*

Dieser Test legt mit dem Unix-Befehl tar einen Verzeichnisbaum auf der gemounteten Partition an:

```
tar xf filetree.tar
```

Die am häufigsten verwendeten syscalls sind hierbei write, read, open, close, chown32 und utime. Die Dateien werden in der Reihenfolge, wie sie im Archiv abgelegt sind, erstellt.

Testname: *delete*

Dieser Test löscht einen zuvor angelegten Dateibaum rekursiv mit dem Befehl
rm -rf *

Die am häufigsten zum Einsatz kommenden syscalls sind unlink, getdents64, lstat64, open, close, chdir, fstat64 sowie fcntl64.

Testname: *read1*

Dieser Test listet sämtliche gefundenen Dateien im Verzeichnisbaum auf und liest sie parallel hierzu auf. Der verwendete Befehl ist

```
( find . -type f -print0 | xargs -0 cat ) > /dev/null
```

Die wichtigstens syscalls sind hierbei read, write, lstat64, getdents64, open, close und fstat64. Aufgrund der parallelen Natur des Tests wird ein Teil der Zeit in den syscalls wait4() und waitpid() verbracht, was eine anschließende Normierung der gesammelten Testdaten notwendig macht.

Der Unix-Befehl find sortiert die von getdents64 zurückgegebenen Einträge nicht lexikographisch.

Testname: read2

Der Test read2 entspricht dem Testablauf von read1, das Auslesen der gefundenen Dateien erfolgt jedoch nicht parallel zur Suche, sondern nach Abschluss der Suche. Der hierzu verwendete Befehl ist

```
cat $(find . -type f 2>/dev/null) > /dev/null
```

Abgesehen von wait4() und waitpid() werden die gleichen syscalls wie bei Test read1 verwendet, allerdings konnte der Test aufgrund der sehr lang werdenden Parameterliste für cat nicht getraced werden.

Testname: read3

Dieser Test liest eine sehr große Datei aus (im Gegensatz zu den bisherigen Tests, welche auf einem zuvor erstellten Dateibaum operierten).

Im Detail wird die für die Erstellung des Dateibaumes verwendete tar-Datei mit cat ausgelesen:

```
cp /export/experiment/data/filetree.tar /mnt  
(find . -type f -print0 | xargs -0 cat) > /dev/null
```

Die am häufigsten verwendeten syscalls sind read() und write(). Der Test erfolgt quasi-parallel, da nur eine Datei gefunden wird.

Testname: recls1

Bei diesem Test wird der Dateibaum durchwandert und alle Dateien gefunden. Der hierzu verwendete Befehl ist

```
find . > /dev/null
```

Die am häufigsten verwendeten syscalls sind getdents64(), lstat64(), chdir(), open(), close(), fstat64(), fcntl64 sowie write().

Testname: recls2

Dieser Test durchwandert den Dateibaum ebenfalls rekursiv, jedoch unter Verwendung des Befehls

```
ls -aR . > /dev/null
```

Im Gegensatz zu recls1 sortiert ls die gefundenen Einträge lexikographisch vor der Ausgabe. Ein Faktor bei diesem Test ist daher die Fähigkeit des Dateisystems, Dateien lexikographisch zu speichern.

Testname: write

Dieser Test legt eine große Datei auf dem Dateisystem an. Hierzu wurde lediglich das bereits oben angesprochene Archiv unter Benutzung des Befehls

```
cp /export/experiment/data/filetree.tar /mnt
```

auf die Zielpartition kopiert. Die beiden am häufigstens verwendeten syscalls sind read() und write().

Nach der Ausführung der Tests werden die Systemaufrufe mit dem Shell-Befehl

```
strace -c -o <dateiname> -f <befehl>
```

ermittelt.

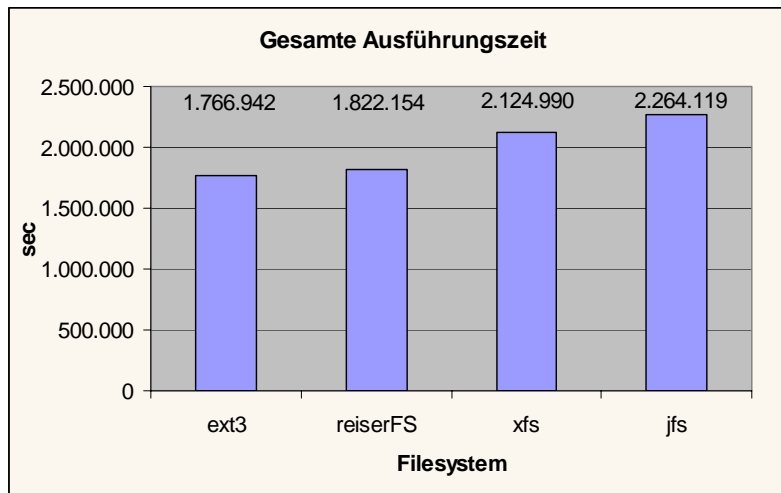
Die Option -c erstellt statistische Angaben über die Anzahl der Aufrufe jedes einzelnen syscalls sowie die Zeit, die insgesamt in einem bestimmten syscall verbracht wurde. Diese Angaben werden bei den folgenden Bewertungen der Resultate verwendet.

Allerdings erhöht das Anwenden des strace-Befehls die Ausführungszeit, da zusätzliche Arbeit zur Überwachung der Systemaufrufe für den Kern entsteht. Es ändert sich dabei aber nur die Gesamtausführungszeit, die prozentualen Angaben bleiben konstant.

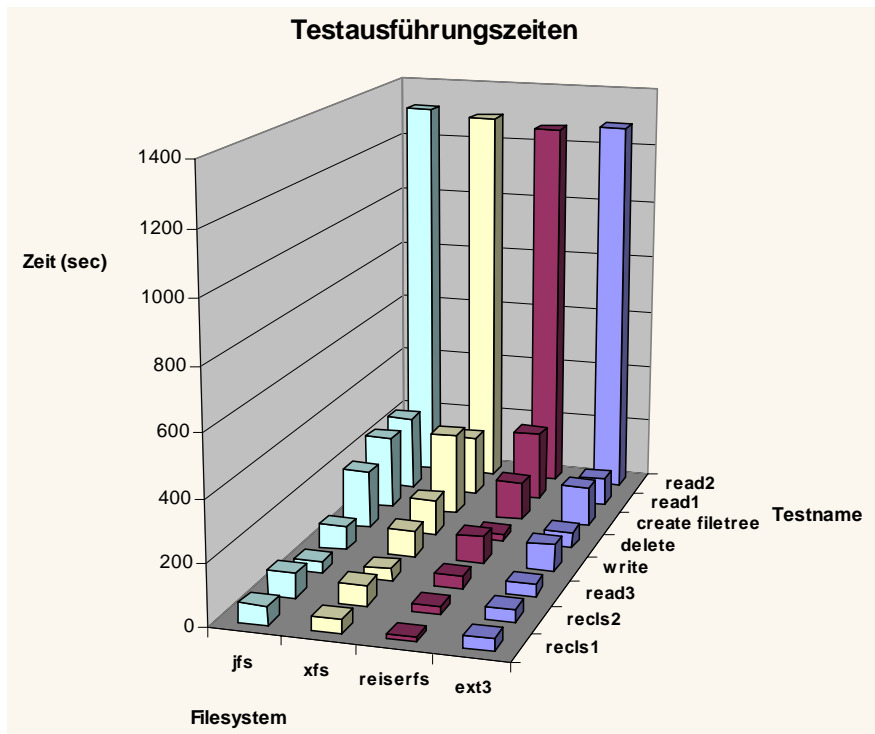
3 Resultate

3.1 Ergebnisüberblick

Den Testergebnissen nach weist das ext3 Filesystem die schnellsten Ausführungszeiten auf.



ReiserFS ist nur geringfügig langsamer, weist aber einen großen Vorsprung vor XFS auf. Schlusslicht ist JFS.

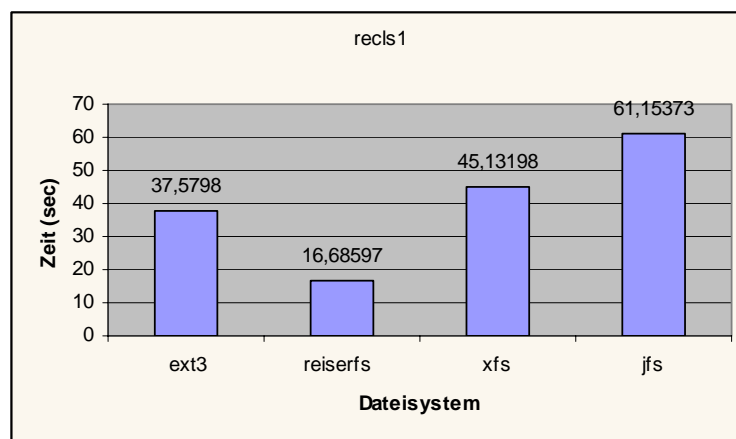


Beim Aufschlüsseln der Ausführungszeiten nach den jeweiligen Tests fällt auf, dass der Test read2 den Großteil der Ausführungszeit beansprucht.

Der write-Test wird im Vergleich zu create-filetree und read1 recht schnell ausgeführt.

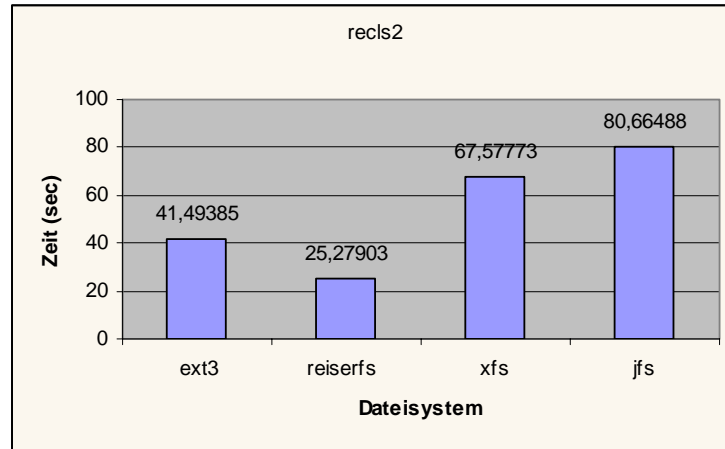
Besondere Stärken der Dateisysteme kristallisieren sich anhand einzelner Diagramme mit prozentualen Angaben zur Zeitaufteilung der Tests heraus. So ist ReiserFS das schnellste Dateisystem bei den Tests create-filetree, recls1, recls2 und delete. ext3 führt bei read2 und JFS ist bei write am schnellsten.

3.2 recls1



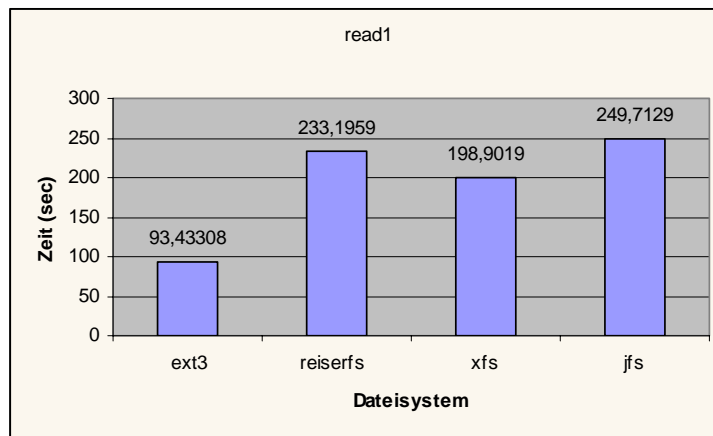
ReiserFS führt den Test recls1 mit deutlichem Unterschied am schnellsten aus. Die unter reiserfs verwendete Baumstruktur unter besserer Ausnutzung der zur Verfügung stehenden Blöcke dürfte der Hauptgrund hierfür sein.

3.3 recl2



Auch der Test recl2 wird von ReiserFS am schnellsten ausgeführt.

3.4 read1



Beim Test schneidet ext3 mit Abstand am besten ab. Die relativ schlechte Ausführungszeit von reiserFS liegt in der Art begründet, wie der Dateibaum angelegt wird.

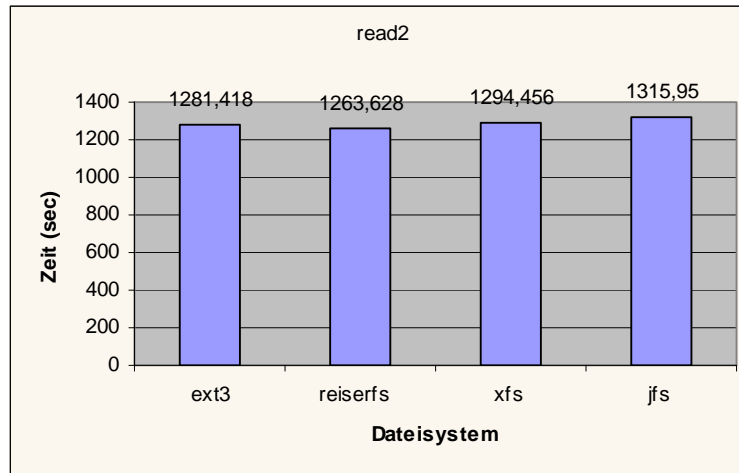
ReiserFS verwendet zur Speicherung der Daten einen B+-Baum, welcher als Schlüssel einen Hash über den Dateinamen verwendet. Die im Archiv enthaltenen Dateien wurden in lexikographischer Reihenfolge angelegt, der Befehl find gibt die Dateien jedoch dem Hashkey nach geordnet zurück. Daher liegen die bei diesem Test ausgelesenen Dateien nicht nebeneinander auf der Festplatte. Diese Einschränkung alleine erklärt jedoch nicht das schlechte Abschneiden; der Grund ist vielmehr, dass aufgrund des parallelen Charakters des Tests der Lesekopf auf der Festplatte ständig zwischen Metadateneinträgen und den eigentlichen Dateiinhalten hin- und herspringen muss, was sich aufgrund eben erklärter Problematik bei reiserFS stärker auswirkt.

Eine Möglichkeit, dieses Problem zu umgehen, wäre gewesen, den erstellten Verzeichnisbaum auf der Zielpartition mit

```
cp -R alter_baum neuer_baum
```

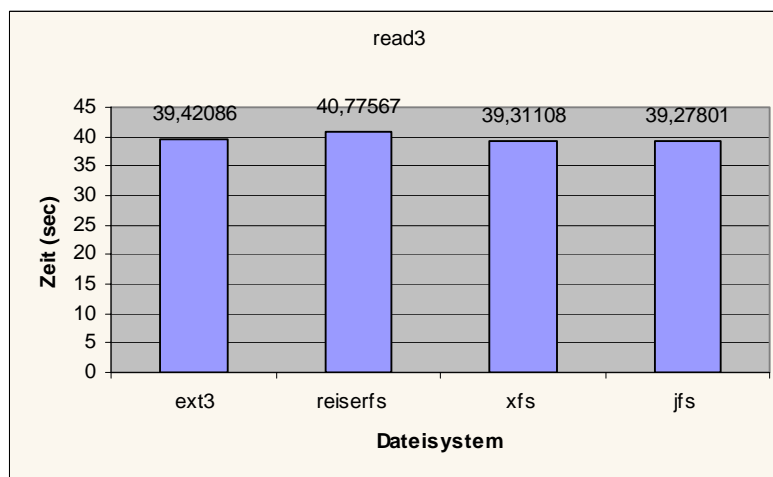
zu kopieren, um die Dateien dem hashkey entsprechend auf der Festplatte abzulegen; dies würde jedoch zum einen den Test aufgrund der Dateisystempuffer verfälschen, zum zweiten entspricht die Testausführung durchaus realitätsnahen Szenarien, mit denen reiserFS auch ohne zusätzliches Kopieren zurechtkommen muss.

3.5 read2



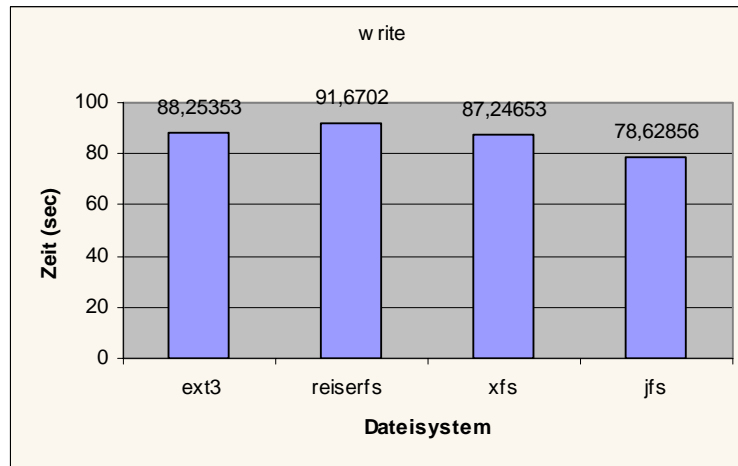
Im Gegensatz zu read1 schneidet hier reiserFS wiederum am besten ab. Die hier relativen geringen Unterschiede in der Ausführungszeit zwischen reiserFS und ext3 sind darin begründet, dass nicht gleichzeitig zwischen den Metadateneinträgen und den tatsächlichen Daten hin- und hergesprungen werden muss, was aufgrund der bei Test1 erläuterten Probleme bei reiserFS sich hier nicht entsprechend stark auswirken kann.

3.6 read3



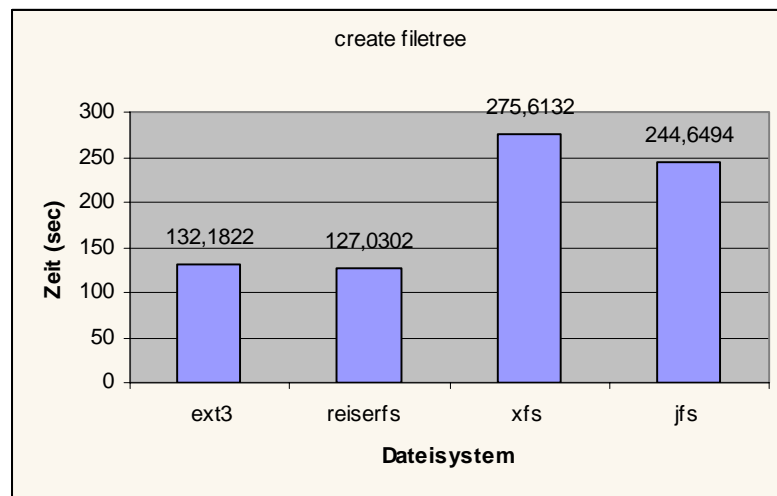
Die Ausführungszeiten bei diesem Test unterscheiden sich kaum und zeigen, dass sämtliche getesteten Dateisysteme beim Lesen großer Datenmengen aus einer Datei eine ähnliche, fast ideale Performanz aufweisen.

3.7 write



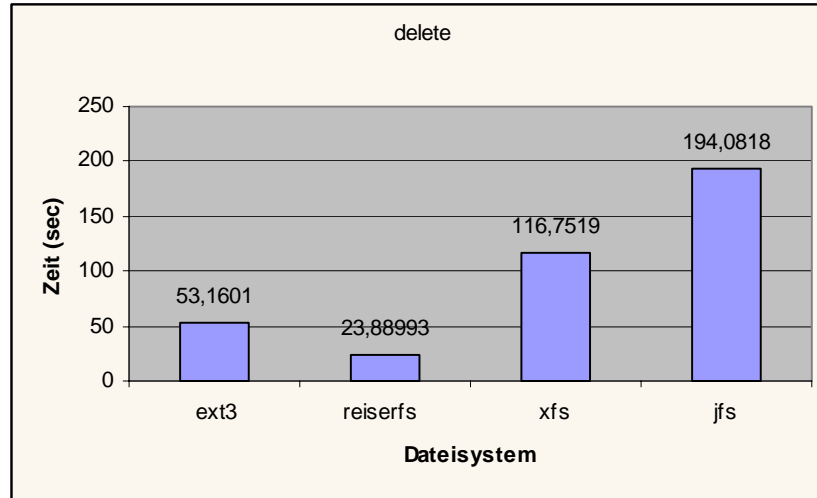
Interessanterweise liegen bei diesem Test die bisher nicht auffallend schnellen Dateisysteme xfs und jfs vorne. Diese beiden Dateisysteme sind nicht direkt für ein möglichst großes Anwendungsfeld konzipiert, sondern insbesondere auf hohe Durchsatzraten bei großen Dateisystemen wie im Umfeld von Datenbanken oder Multimedia-Anwendungen konzipiert. Die Baumstruktur von reiserFS, welche bei großen Datenmengen auch splits notwendig macht, scheint sich hier negativ auszuwirken.

3.8 create filetree



Im Gegensatz zu den beiden eben angesprochenen Tests zeigen sich hier wiederum die Vorteile von reiserFS und ext3 beim Handhaben von Metadaten und vielen, kleinen Dateien bei einer verschachtelten Verzeichnisstruktur.

3.9 delete



Der hohe Vorsprung von reiserFS bei diesem Test scheint seinen Ursprung in der sehr effizienten Ausführung des syscalls `getdents64()` zu haben; unter reiserFS ist dies insbesondere durch die gute Ausnutzung des zur Verfügung stehenden Platzes auch bei kleinen Datenmengen wie Metadaten erklärbar, was einen höheren Durchsatz für Metadaten bedeutet.

4 Schlussfolgerung

Trotz der relativ klaren Ergebnisse zugunsten von reiserFS und xfs muss hier angemerkt werden, dass selbst die angesprochenen Journaling-Dateisysteme nicht alle für das gleiche Einsatzgebiet konzipiert sind. Insbesondere xfs wurde von SGI für IRIX-Systeme entwickelt, um für multimediale Anwendungen hohe Durchsatzraten zu erlauben, insbesondere besitzt xfs die (unter Linux fehlende) Eigenschaft, angeforderte Durchsatzraten zu garantieren, was insbesondere für Videoprocessing-Anwendungen sehr praktisch ist. Zusätzlich wurde unter Linux XLV, ein für XFS entwickelter Volume Manager, nicht implementiert.

JFS kann sein Potential unter Linux noch nicht zeigen, da die gegenwärtige Implementierung, obwohl stabiler als seine Vorgänger, noch keinen Einsatz in Produktionsumgebungen erlaubt; die für einen relativ reibungslosen Betrieb notwendigen Änderungen wurden erst in den letzten Betriebssystemkernen der Serie 2.6 implementiert.

Weiterhin wurden die Dateisysteme ohne Optimierungsmaßnahmen angelegt, wie sie beispielsweise ext3 in Form von Hashtrees oder Orlov-Inode-Allokation erlaubt; in einer Produktionsumgebung ist es durchaus sinnvoll, den Anforderungen entsprechende Optimierungen zu aktivieren.

Außer den in diesem Projekt ausgeführten Tests wären auch weitere Experimente denkbar. Beispiele dafür wäre

- der Vergleich der Aussagekraft unserer Tests mit denen, die fertige Programme durchführen (z.B. „bonnie++“),
- das Einbeziehen des Funktionsumfangs eines Dateisystems in die faktorielle Analyse,
- eine weitere Unterteilung der Tests, z.B. in Tests nur mit großen oder kleinen Dateien.
- Zusätzliches Kopieren des verwendeten Archivs vor dem Start der Tests, um dem Dateisystem zu ermöglichen, die Dateien geordnet anzulegen.

Allerdings erschienen uns die geführten Experimente im Rahmen dieses Praktikums ausreichend aussagekräftig für realitätsnahe Anwendungen, zudem wären umfassende Tests um ein Vielfaches umfangreicher gewesen, daher haben wir auf weitere Ergänzungen verzichtet.

Anhang : Mittelwerte der Ergebnisse

create-filetree ext3 132.1822
create-filetree xfs 275.6132
create-filetree reiserfs 127.0302
create-filetree jfs 244.6494
recls1 ext3 37.5798
recls1 xfs 45.13198
recls1 reiserfs 16.68597
recls1 jfs 61.15373
recls2 ext3 41.49385
recls2 xfs 67.57773
recls2 reiserfs 25.27903
recls2 jfs 80.66488
read1 ext3 93.43308
read1 xfs 198.9019
read1 reiserfs 233.1959
read1 jfs 249.7129
read2 ext3 1281.418
read2 xfs 1294.456
read2 reiserfs 1263.628
read2 jfs 1315.95
read3 ext3 39.42086
read3 xfs 39.31108
read3 reiserfs 40.77567
read3 jfs 39.27801
delete ext3 53.1601
delete xfs 116.7519
delete reiserfs 23.88993
delete jfs 194.0818
write ext3 88.25353
write xfs 87.24653
write reiserfs 91.6702
write jfs 78.62856

sum ext3 1766.942
sum xfs 2124.990
sum reiserfs 1822.154
sum jfs 2264.119