LFS verglichen mit FFS

Michael Abmayer, E535/9425680,
e9425680@studbimb.tuwien.ac.at
Technische Universität Wien

185.188 Seminar (mit Bakkalaureatsarbeit) im WS 2003/04

29. Oktober 2004

Spätestens mit der Vorstellung des log-strukturierten Dateisystems Sprite LFS (Log structured File System) in ROSENBLUM UND OUSTERHOUT [RO92] hat rege Forschungstätigkeit zu log-strukturierten Dateisystemen eingesetzt. Sowohl in ROSENBLUM UND OUSTERHOUT [RO92] als auch in SELTZER ET AL. [SBMS93] und SELTZER ET AL. [SSB+95] wurden mit LFS erzielte Benchmarkwerte mit denen für FFS (Fast File System) verglichen.

In ROSENBLUM UND OUSTERHOUT [RO92] wird gezeigt, daß Sprite LFS „70% der Datenübertragungsrate der Festplatte beim Schreiben ausnutzen kann“. Insbesonders “beim Schreiben kleiner Dateien“ ist Sprite LFS damit „um Größenordnungen“ effizenter als die (Anm. des Verfassers: damals) „üblichen Unix-Dateisysteme“.

Die vorliegende Arbeit gibt einen Überblick über die historische Entwicklung von LFS und FFS, liefert einen Einblick auf die Entwicklung der zugrunde liegenden Festplattenhardware und zitiert einige Performance-Messungen bzw. Benchmarks.

Inhaltsverzeichnis

1 Einleitung

Ziel dieser Seminararbeit ist es, einen Überblick über wissenschaftliche Literatur zu geben, die insbesonders die Dateisysteme LFS (log-structered File System) und FFS (Fast File System) sowie einen Vergleich zwischen diesen zum Inhalt hat. Ausgangspunkt dieser Arbeit ist die Arbeit von SELTZER ET AL. [SSB+95], in der Implementierungen von LFS und FFS im Betriebssystem 4.4BSD-Lite verglichen werden. Weiters soll eine Zusammenfassung einiger Literaturfundstellen und versucht diese in einen größeren Zusammenhang zu stellen.

Im Abschnitt 2 wird ein Überblick über die konzeptuellen Unterschiede von LFS und FFS gegeben. Wesentliche technische Voraussetzungen bei Festplatten werden im Abschnitt 3 dargestellt. Abschnitt 4 zitiert einige Performance-Untersuchungen zu LFS und FFS.

2 Überblick über FFS und LFS

2.1 FFS – Fast File System

FFS wird in MCKUSICK ET AL. [MJLF84] detailliert vorgestellt und beschrieben. Ausgangspunkt der Entwicklung war, daß das „originale UNIX-Dateisystem [...] Blöcke zu 512 Byte" belegte und sowohl bei Lese- als auch Schreibzugriffen auf Dateien einige zeitaufwendige Zugriffe machen mußte. Dies führte zu einer „nutzbaren Bandbreite von rund 20 KB/s pro Plattenarm", obwohl die Festplatten selber eine wesentlich höhere Bandbreite ermöglicht hätten. FFS führte einige Verbesserungen, mit dem Ziel, die Bandbreite zu erhöhen, ein.

Ein weiterer Schwachpunkt des traditionellen Dateisystems war, daß die Transfer-Rate eines bestehenden Systems nach „wenigen Wochen mäßiger Benutzung" auf einen Bruchteil der Transfer-Rate eines „gerade angelegten" Dateisystems sank. Die Ursache dafür lag in der *„free list"*, die „zu Beginn für optimalen Zugriff geordnet war, im Zuge des Anlegens und Löschens von Dateien aber schnell ungeordnet wurde".

Mit FFS wurde die Blockgröße auf 4096 Bytes erhöht. Wird ein Block nicht zur Gänze benötigt, so kann er in „2, 4 oder 8 *Fragmente* geteilt werden", „wobei ein Fragment nicht kleiner als ein Platten-Sektor sein darf, üblicherweise 512 Bytes". Durch die größere Blockgröße wird ein größerer „Durchsatz" (*troughput*) erreicht.

FFS führt sog. *cylinder groups* ein. Eine *cylinder group* umfaßt gemäß SELTZER ET AL. [SBMS93] „typischerweise 16 bis 32 Zylinder" der Festplatte. FFS versucht, „Daten Blöcke einer Datei in der selben *cylinder group* zu speichern, bevorzugt an rotationsmäßig optimalen Orten im gleichen Zylinder". Als konfigurierbarer Parameter wird *„rot_delay"* genannt (bei MCVOY UND KLEIMAN [MK91] *„rotdelay"*). Es handelt sich dabei laut SELTZER ET AL. [SBMS93] um eine „Verzögerung in ms aus Sicht der Festplatte, die zwischen der Kenntnisnahme der Beendigung einer Ein-/Ausgabeanforderung durch die CPU und dem Absetzen einer nachfolgenden Ein-/Ausgabeanforderung seitens der CPU vergeht". Diese Verzögerung soll verhindern, daß beim sequentiellen Lesen von Daten zwischen der Bearbeitung aufeinanderfolgender Ein-/Ausgabeanforderungen die Dauer fast einer ganzen Festplattenumdrehung gewartet werden muß - schließlich hat sich die Platte während der Bearbeitung der Daten durch die CPU bereits weitergedreht. Wären die Blöcke nicht mit einem Versatz positioniert, so würde der gesuchte Block schon zum Teil oder zur Gänze unter dem Schreib-/Lesekopf der Platte vorbeigedreht sein. Weitere Forschungen - unter anderem MCVOY UND KLEIMAN [MK91] - führten zur Performanceverbesserungen durch Techniken wie „Prefetching" (vorausschaudendes Holen von Daten von der Festplatte) und „Clustering" (Datenblöcke in größeren Einheiten transferieren). MCVOY UND KLEIMAN [MK91] nennt auch den Parameter *„maxcontig"*, der die „Anzahl an

Blöcken“ auf der Festplatte angibt, die „in einem Stück“ positioniert werden, bevor eine Lücke, deren Länge der Dauer *rot_delay* entspricht, eingefügt wird.

2.2 LFS – Log structured File System

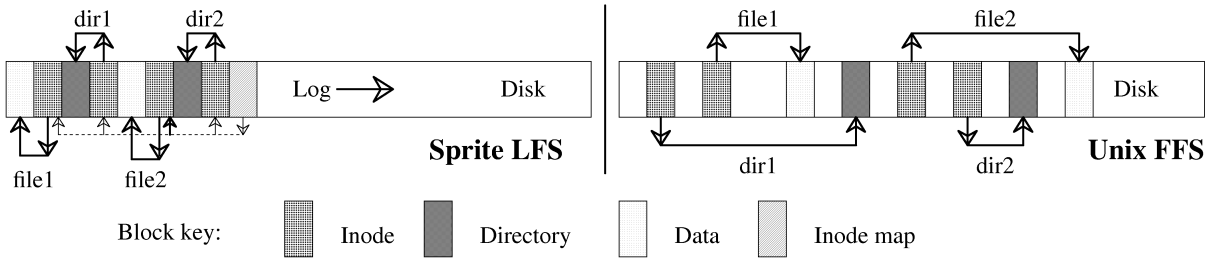

Abbildung 1: **Ein Vergleich zwischen Sprite LFS und Unix FFS aus ROSENBLUM UND OUSTERHOUT [RO92].** *Übersetzung der Bildunterschrift aus* ROSENBLUM UND OUSTERHOUT *[RO92]:* „Das Beispiel zeigt die modifizierten Platten-Blöcke, die beim Anlegen zweier Dateien mit Namen `dir1/file1` und `dir2/file2`, die je einen Block groß sind, geschrieben werden. Beide Systeme müssen die neuen Daten-Blöcke sowie inodes für die Dateien `file1` sowie `file2` zuzüglich der neuen Daten-Blöcke und Inodes für die Verzeichnisse schreiben. Unix FFS benötigt dazu zehn nicht-sequentielle Schreibzugriffe (die inodes für die Dateien werden zweimal geschrieben, um die Wiederherstellung im Falle eines Systemabsturzes zu erleichtern), während Sprite LFS einen großen Schreibzugriff ausführt. Die gleiche Anzahl an Platten-Zugriffen ist zum Lesen der Dateien nötig. Sprite LFS schreibt weiters neue Inode-map-Blöcke, um die neuen Inode-Orte zu verzeichnen.“

Eine Implementierung von LFS – Sprite-LFS – wird in ROSENBLUM UND OUSTERHOUT [RO92] vorgestellt. Als wesentliche Probleme von (Anm. des Verfassers: damals) „üblichen Dateisystemen“ werden dort folgende Sachverhalte angeführt:

- „Informationen werden weit auseinandergestreut auf der Platte verteilt“. So werden z.B im FFS die „Attribute zu einer Datei, ihr Inhalt und der zugehörige Dateiname an verschiedenen Orten“ gespeichert. Es werden daher „zumindest fünf einzelne Plattenzugriffe benötigt, jedem vorhergehend ein *Seek*, um eine Datei neu anzulegen“: „Zwei Zugriffe für die Dateiattribute, einer für den Dateiinhalt, einer für die Verzeichnisdaten und einer für die Verzeichnisattribute. Das führt insbesonders beim Schreiben kleiner Dateien dazu, daß nur rund 5% der Plattenbandbreite tatsächlich für die neuen Daten benutzt werden, der Rest der Zeit wird mit Suchen auf der Platte verbracht.“
- „Die (Anm. des Verfassers: damals) üblichen Dateisysteme [...] tendieren zum synchronen Schreiben“. „FFS schreibt zwar den Dateiinhalt asynchron, die Metadaten werden aber synchron geschrieben“. Eine „Anwendung“, die „kleine Dateien“ schreibt, wird dadurch gebremst und kann z.B „von einem schnelleren Prozessor kaum profitieren.“

LFS verfolgt daher einen grundlegend anderen Ansatz als die traditionellen Dateisysteme, an welchem Ort Daten auf der Platte aufgezeichnet werden. In ROSENBLUM UND OUSTERHOUT [RO92] wird die Implementation Sprite LFS beschrieben. Gemäß ROSENBLUM UND OUSTERHOUT [RO92, Table 1] sind lediglich *Superblocks* und *Checkpoints* an einem festen Platz gespeichert; *Inodes*, *Inode map*, *Indirekt blocks*, *Segment summaries* und *Segment usage tables* sowie *Directory change logs* finden ihren Platz im *Log*. „Die grundlegende Idee bei einem logstrukturierten Dateisystems ist, die Schreibe-Performance durch Puffern einer Abfolge von Dateisystem-Änderungen in einem Datei-Cache und anschließendes sequentielles Schreiben aller Änderungen auf die Platte in einer einzigen Platten-Schreibe-Operantion“. Dieses Schreiben erfolgt kontinuierlich fortschreitend über die gesamte Platte. Damit einhergehend ändert

sich das Layout auf der Festplatte grundlegend. Abbildung 1, übernommen aus ROSENBLUM UND OUSTERHOUT [RO92], illustriert dies anschaulich.

Für den Fall einer Wiederherstellung (*recovery*) nach einer Betriebsunterbrechung (*crash*) führt ROSENBLUM UND OUSTERHOUT [RO92] aus: „Es sollte möglich sein, nach einer Betriebsunterbrechung schnell wiederanzulaufen. [...] Wie viele andere loggende Systeme benutzt Sprite-LFS eine zweizinkige Annäherung an die Wiederherstellung (Anm. des Verfassers: im englischen Original *two-pronged approach to recovery*): *checkpoints*, die konsistente Zustände des Dateisystems defninieren, und *roll-forward*, welches zur Wiederherstellung der seit dem letzten Checkpoint geschriebenen Information benutzt wird."

Eine Herausforderung beim Design eines LFS besteht allerdings „in der Verwaltung freien Speicherplatzes". Die Überlegung von ROSENBLUM UND OUSTERHOUT [RO92] war, daß zu Beginn der freie Platz auf der Platte als ein großes Stück vorliegt. „Erreicht das Log jedoch einmal das Ende der Platte, ist der freie Platz durch Löschen und Ändern von Dateien in viele kleine Stücke fragmentiert. Es bestehen nun zwei Möglichkeiten, wie weiter verfahren werden kann" (siehe auch Abbildung 2):

- *„threading"*: „Die bestehenden Daten bleiben an ihrem Platz, das Log wird an jenen Stellen fortgesetzt, die frei sind. Unglücklicherweise wird der freie Platz durch threading ernstlich fragmentiert, sodaß große zusammenhängende Schreibzugriffe nicht mehr möglich wären und ein log-strukturiertes Dateisystem nicht schneller als ein traditionelles Dateisystem wäre."
- *„copying"*: „Die benutzten Daten werden aus dem Log kopiert, um große freie Blöcke zu erhalten. Es wird in diesem Papier angenommen, daß sie am Kopf des Logs in kompakter Form zurückgeschrieben werden. [...] Der Nachteil am Kopieren ist der Aufwand, besonders für langlebige Daten." Vor allem wenn „alle langlebigen Dateien bei jedem Durchgang des Logs über die Platte kopiert werden müssen".

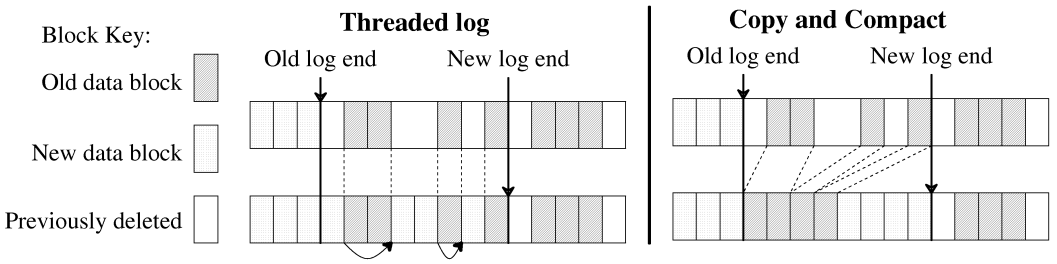

Abbildung 2: **Management des freien Platz bei einem LFS aus ROSENBLUM UND OUSTERHOUT [RO92].** Beim „Threaded log" wird das Log in den bestehenden freien Blöcken fortgesetzt. Die Verkettung der Blöcke des Logs geschieht über Zeiger. „Copy and compact" zeigt die Taktik, durch Kopieren und Zusammenfassen der bestehenden Blöcke den neu entstandenen Platz in größeren Stücken herzustellen.

In ROSENBLUM UND OUSTERHOUT [RO92] wurde daher eine Kombination dieser beiden Verfahren gewählt: „Die Platte wird in große Bereiche gleicher Größe unterteilt, sog. Segmente. Jedes Segment wird immer sequentiell von Anfang bis Ende geschrieben. [...] Das Log ist auf Ebene der Segmente gesehen *threaded*". Im Prozeß des *„segment cleaning* [...] werden mehrere Segmente eingelesen". Danach wird festgestellt, welche darin enthaltenen Daten noch benötigt werden. Diese werden anschließend „auf eine kleinere Anzahl sauberer Segmente geschrieben". Dadurch entsteht freier Platz für neue Daten. Für den Betrieb benötigt LFS daher einen

Cleaner. Dieser kann laut SELTZER ET AL. [SBMS93] sowohl im Kernel, als auch als Userprozeß realisiert werden.

In SELTZER ET AL. [SBMS93] wird einerseits eine Implementierung von LFS auf BSD beschrieben, andererseits ein Vergleich mit FFS geführt. In SELTZER ET AL. [SSB+95] werden ebenso LFS und FFS verglichen, ein Großteil dieses Artikels befaßt sich mit Performance-Gesichtspunkten. MATTHEWS ET AL. [MRC+97] zeigt Möglichkeiten auf, die Performance von LFS zu verbessern, insbesonders durch die „Wahl von Segmentgrößen, die zur Platten- und Arbeitslast-Charakteristik passen“, einer geeigneten „*cleaning policy*“ (Cleaner-Taktik) und der „Reorganisation von Daten“. SOULES ET AL. [SGSG02] stellt CVFS (comprehensive Versioning File System) vor, das verschiedene Versionen einer Datei speichern kann und vergleicht die Performance dieses Systems mit LFS, FFS und ext2 (second extended file system). CVFS benutzt ebenfalls ein „Log-strukturiertes Daten-Layout ahnlich dem von LFS“, auch der „Cleaner ist ähnlich“ aufgebaut.

3 Entwicklung von Hardware und Caches

ANDERSON [AN03] führt zu Festplatten wie folgt aus: „Traditionell besteht das Arbeits-Modell von Programmierern für Platten-Speicher aus einem Satz einheitlicher Zylinder, jeder mit einem Satz einheitlicher Spuren, jede davon hält eine feste Anzahl von 512-Byte-Sektoren, und jeder dieser hat eine eindeutige Adresse. Die Zylinder entstehen aus konzentrischen Kreisen (oder Spuren) auf jeder Seite einer Plattenoberfläche bei einem mehrscheibigen Laufwerk (Anm. des Verfassers: im englischen Original *disk platter in a multiplatter drive*). Jede Spur ist in Sektoren ähnlich Tortenstücken unterteilt. Weil jeder Ort in diesem dreidimensionalen Speicherraum eindeutig durch Zylindernummer, Kopfnummer (Oberflächennummer) und Sektornummer identifiziert werden kann, ergeben diese die Basis des originalen Programmiermodells für Plattenlaufwerke: Zylinder-Kopf-Sektor-Zugriff.“

Zu Zeiten, als Daten zwischen CPU und Festplatte sektorweise ausgetauscht wurden, wurden die Sektoren nicht immer hintereinander numeriert, sondern versetzt. Denn nach der Beendigung eines Datentransfers und dem Beginn eines neuen Datentransfers brauchte die CPU bzw. der Festplattencontroller eine gewisse Zeit zur Verarbeitung. Bei PCs geschah dieses durch Festlegung des Interleave-Faktors mit der Low-Level-Formatierung der Festplatte, wie bei TISCHER [TI92, S. 547] beschrieben. Abbildung 3 zeigt eine Darstellung aus TISCHER [TI92]. FFS benutzt dafür laut SELTZER ET AL. [SBMS93] den *rot_delay*-Parameter, der auch bei bestehenden Dateisystemen problemfrei geändert werden kann und alle nachfolgenden Zugriffe betrifft (siehe auch MCVOY UND KLEIMAN [MK91]). Neben dem Versatz der Sektoren einer Spur zueinander ist gemäß TISCHER [TI92, S. 549] ebenso ein Versatz zwischen den Spuren eines Zylinders (*cylinder skew*) und den Zylindern selbst (*track skew*), in den die Zeit zum Umschalten zwischen den Köpfen und zum Schwenken des Kopfarms eingeht, bei der Low-Level-Formatierung einstellbar. Zu beachten dabei ist, daß sich TISCHER [TI92] auf PC-Systeme bezieht.

Versucht man bei einem Interleave-Faktor von 1:3 (oder einem korrespondierenden *rot_delay*) die Sektoren in aufsteigender Folge zu lesen, so benötigt die Platte drei Umdrehungen, um alle Daten zu liefern. Die mögliche Platten-Transferrate wird somit gleichsam gedrittelt. Bei einem Interleave-Faktor von 1:6 werden sechs Umdrehungen benötigt, die mögliche Transferrate erscheint somit gesechstelt usw. Es ist leicht zu erkennen, daß somit ein Interleave-Faktor von 1:1 zu bevorzugen ist.

In den letzten Jahren wurden Festplatten mit wachsenden Caches ausgestattet. Tabelle 1 stellt exemplarisch drei (willkürlich ausgewählte) Festplattenmodelle aus den Jahren 1991, 1995 und

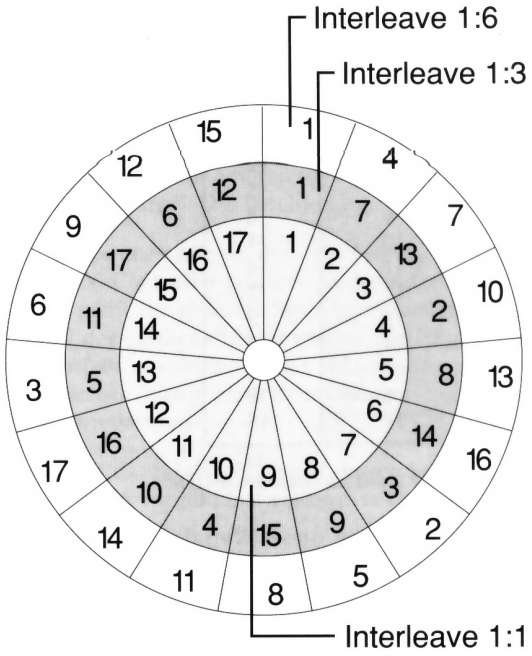

Abbildung 3: Sektoren-Anordnung aus TISCHER [TI92, S. 548] für verschiedene Interleave-Faktoren. Leicht zu erkennen ist, daß z.B. bei einem Interleave-Faktor von 1:3 der jeweils in der Zählung nachfolgende Sektor physikalisch 3 Sektoren weiter liegt.

2000 der Firma Seagate gegenüber, um überblicksmäßig die Entwicklung von Festplatten zu veranschaulichen.

Ebenso wurden in verschiedenen Betriebssystemen Puffer bzw. Caches für die Festplatten eingeführt. Bereits im BETRIEBSSYSTEM DOS (DISK OPERATING SYSTEM) REFERENZ-HANDBUCH [IBM84, S. 4-8f] findet sich unter der Beschreibung des „BUFFERS"-Befehls „Zur Angabe der Anzahl von Plattenpuffern, die DOS im Hauptspeicher zuordnen soll, wenn es gestartet wird" folgendes: „Ein Plattenpuffer ist ein Speicherblock, den DOS zur Speicherung von Daten benutzt, die von einer Platte gelesen oder auf eine Platte geschrieben werden (wobei es sich um eine Festplatte oder um eine Diskette handeln kann), wenn die übertragene Datenmenge nicht ein genaues Vielfaches der Sektorengröße darstellt. [...] Immer wenn DOS aufgefordert wird, einen Datensatz zu lesen oder zu schreiben, bei dem es sich nicht um ein genaues Vielfaches der Sektorengröße handelt, prüft es zuerst, ob der Sektor mit diesem Datensatz schon in einem Puffer steht. Ist dies nicht der Fall, so muß DOS den Sektor [...] lesen. Stehen die Daten jedoch schon in einem Puffer, so überträgt DOS den Datensatz einfach in den Speicherbereich der Anwendung, ohne daß der Sektor von der Platte gelesen werden muß. Dadurch wird Zeit gespart. Diese Zeitersparnis wird sowohl beim Lesen, als auch beim Schreiben von Datensätzen erzielt, da DOS zuerst einen Sektor lesen muß, bevor es einen Datensatz einfügen kann, der durch die Anwendung des Benutzers geschrieben werden soll." Weiters nennt das BETRIEBSSYSTEM DOS (DISK OPERATING SYSTEM) REFERENZ-HANDBUCH [IBM84, S. 4-8f] eine maximale Anzahl von 99 möglichen Plattenpuffern.

STALLINGS [ST98, S. 487] beschreibt den „Cache Manager" als Bestandteil des „I/O Manager" des Betriebssystems Windows NT wie folgt: „**Cache-Manager:** Der Cache-Manager führt das Caching für das gesamte Ein-/Ausgabe-Subsystem durch. Der Cache-Manager stellt im Hauptspeicher einen Caching-Dienst für alle Dateisysteme und Netzwerkkomponenenten zur Verfügung. Er kann dynamisch die Größe des Caches, der einer bestimmten Aktivität gewidmet ist, erhöhen oder erniedrigen, wenn sich die Menge physisch verfügbaren Speichers ändert. Der Cache-Manager beinhaltet zwei Dienste, um die Gesamt-Performance zu verbes-

sern:

- **Lazy write:** Das System erfaßt Updates nur im Cache und nicht auf der Platte. Später, wenn die Beanspruchung des Prozessors gering ist, schreibt der Cache-Manager die Änderungen auf die Platte. Wenn ein spezieller Cache-Block in der Zwischenzeit aktualisiert wurde, ergibt das netto eine Ersparnis.
- **Lazy commit:** Dieses ist ähnlich dem Lazy write für Transaktions-Verarbeitung. Anstatt umgehend eine Transaktion als erfogreich abgeschlossen zu kennzeichnen, cachet das System die übergebene Information und schreibt sie später als ein Hintergrundprozeß in das Dateisystemlog."

ROSENBLUM UND OUSTERHOUT [RO92] sehen allerdings auch folgende Einschränkung: „Schreibpuffern hat natürlich den Nachteil, daß die Menge verlorenen Daten während eines Crash wächst. [...] für Anwendungen, die besseres Crash-Recovery benötigen, kann nichtvolatiles RAM für den Schreibe-Puffer verwendet werden."

Weiters hält TISCHER [TI92, S. 550] zum Thema „Multiple Zone Recording" fest: „Mit moderenen SCSI- und IDE-Platten, die dem BIOS die Kenndaten in bezug auf die Anzahl der Köpfe, Spuren und Sektoren ohnehin nur vorspiegeln, ist jedoch auch eine variable Formatierung der einzelnen Spuren möglich geworden. Denn diese Controllergespanne rechnen die vom BIOS angegebene Kopf-, Zylinder- und Sektornummer ohnehin in eine andere Sektoradresse um, wodurch dann auch eine Erweiterung der äußeren Spuren berücksichtigt werden kann."

Modell	ST 4702N	ST 19171W	Cheetah 73LP
Jahr (Quelle)	1991 (Copyright Specification)	1995 (Copyright Specification)	2000 (Product Manual Rev. A)
Kapazität (MB)	601	9.100	73.400
Drehzahl	3.600	7.200	10.000
Zugriffszeit (ms)	16,5	9,7	5,1
Cache (KB)	32	512 (optional 2048)	4.096
Zylinder	1.546	5.274	29.549
Köpfe	15	20	8
Firmenangabe Transferrate Firmenangabe interne Transferrate Firmenangabe mittlere Transferrate	1,5-2 mbytes/sec	80-124 mbits/sec	399-671 Mbits/sec 52 Mbytes/sec
Firmenangabe Byte/Spur (Mittelwert) Firmenangabe durchschnittliche Anzahl Sektoren/Spur	50	168 (abgerundet)	397.385
berechnete Werte			
Byte/Spur (berechnet mit Annahme 512 Bytes Nutzdaten pro Sektor)	25.600	86.016	
Spuren im Cache	1,28	6,09/24,38	10,55

Tabelle 1: **Überblick über drei Festplattenmodelle der Firma Seagate.** Die Spezifikationen der Festplatten wurden online am 6.12.2003 unter folgenden URLs abgefragt: ST 4702N – `http://www.seagate.com/support/disc/specs/scsi/st4702n.html`; ST 19171W – `http://www.seagate.com/support/disc/specs/scsi/st19171w.html`; Cheetah 73LP – `http://www.seagate.com/cda/products/discsales/enterprise/tech/0,1084,321,00.html`. Das Product Manual zur Cheetah 73LP fand sich mit Stand 20.4.2004 unter `http://www.seagate.com/cda/products/discsales/enterprise/tech/0,1084,321,00.html`

Weiters besteht laut ANDERSON [AN03] die Möglichkeit, die Plattenzugriffe in einer Queue zu sammeln und für einen schnelleren Zugriff neu zu sortieren. Dieses Umsortieren der Zugriffe

kann gemäß ANDERSON [AN03] sowohl vom Betriebssystem als auch von der Platte durchgeführt werden.

Laut STALLINGS [ST98, S. 456] sind „DMA"-Zugriffe (Direct Memory Access), bei denen die Daten zwischen Geräten und Speicher ohne Interaktion mit dem Prozessor transferiert werden (außer zu Beginn und Ende des Transfers), wesentlich „effizienter" als die Methoden *„interrupt-driven* oder *programmed I/O"*.

4 Performance Vergleiche

Die im folgenden zitierten Performance-Vergleiche stammen aus ROSENBLUM UND OUSTERHOUT [RO92], SELTZER ET AL. [SBMS93] und SELTZER ET AL. [SSB+95].

4.1 Vergleiche aus ROSENBLUM UND OUSTERHOUT [RO92]

Im Zuge der Benchmarks in ROSENBLUM UND OUSTERHOUT [RO92] wurden mehrere LFS-Cleaner-Algorithmen mit FFS verglichen, Maßzahl für den Vergleich ist die $write\ cost$ nach Gleichung 1 aus ROSENBLUM UND OUSTERHOUT [RO92]:

$$write\ cost = \frac{total\ bytes\ read\ and\ written}{new\ data\ written} \tag{1}$$

Der Benchmark simulierte die Last des Dateisystems. Dazu wurde eine „feste Anzahl an 4K-Dateien angelegt, mit einer so gewählten Anzahl, daß eine spezielle Gesamt-Platten-Kapazitäts-Ausnutzung erzeugt wurde". Anschließend wurde mit zwei verschiedenen Zugriffsmustern (*„uniform"* und *„hot-and-cold"*) der Betrieb des Dateisystems simuliert. Der eigentliche Zugriff war das Überschreiben einer Datei. Beim *„uniform"*-Muster wurden alle Dateien mit der gleichen Wahrscheinlichkeit überschrieben, beim *„hot-and-cold"*-Muster 90% der Dateien mit einer Wahrscheinlichkeit von 10%, 10% der Dateien mit einer Wahrscheinlichkeit von 90%. Abbildung 4 zeigt die dabei erzielten Resultate.

Ebenso wurde ein Benchmark für verschiedene „Cleaner-Taktiken" beim „Hot-and-Cold-Muster" präsentiert (siehe Abbildung 5). *„Greedy"* wählte die am „wenigsten genutzten" Segmente zur Bearbeitung aus, *„cost-benefit"* wählte nach Gleichung 2 aus ROSENBLUM UND OUSTERHOUT [RO92] mit u als „Nutzungsgrad eines Segments":

$$\begin{aligned} \frac{benefit}{cost} &= \frac{free\ space\ generated\ *\ age\ of\ data}{cost} \\ &= \frac{(1-u) * age}{(1+u)} \end{aligned} \tag{2}$$

Es zeigte sich, daß *cost-benefit* in diesem Test besser abschneidet. Die zugrunde liegende Überlegung war, daß freier Platz in „kalten" Segmenten „wertvoller" wäre. Wenn ein „kaltes" Segment gereinigt würde, wird es gemäß ROSENBLUM UND OUSTERHOUT [RO92] wahrscheinlich für einen längeren Zeitraum nicht mehr gereinigt werden müssen.

Weiters wurde ein Benchmark druchgeführt, bei dem die Datei-Erzeuge-, Lese- und Lösch-Performance von LFS mit dem Dateisystem von SunOS verglichen wurde. Es zeigte sich dabei,

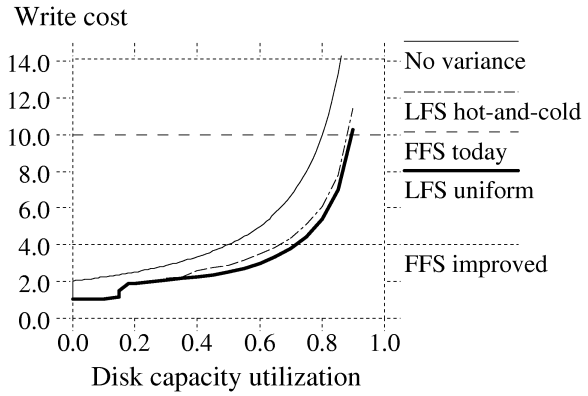

Abbildung 4: **Write Cost nach Plattenbelegung aus ROSENBLUM UND OUSTERHOUT [RO92]:** „FFS today" bezeichnet den technischen Stand von Unix FFS zum Zeitpunkt der Arbeit, „FFS improved" eine Schätzung der Autoren der besten mit FFS erreichbaren Performance. „No variance" bezeichnet ein LFS, bei dem „alle Segmente exakt gleiche Belegung" hätten. „LFS uniform" ist ein LFS, das mit dem *„uniform"*-Muster beschrieben wurde, „LFS hot-and-cold" ist ein LFS, das mit dem *„hot-and-cold"*-Muster beschrieben wurde. Der Cleaner benutzte in dieser Simulation eine „einfache *greedy*-Taktik", d.h. er wählte immer jene Segmente zum Säubern, die am „wenigsten genutzt waren. Beim Schreiben" der neuen Segmente hat er „keine Reorganisation der Daten vorgenommen", sondern sie in der selben Reihenfolge, in der er sie gelesen hat, geschrieben. „Der Simulator lief, bis alle sauberen Segmente verbraucht waren, dann arbeitete der Cleaner, bis wieder ein Schwellwert an sauberen Segmenten erreicht war. Der Simulator lief in jedem Testdurchlauf so lange, bis sich die Werte für die *write cost* stabilisiert hatten und damit die Varianz durch den Kaltstart entfernt war."
Man erkennt, daß mit zunehmendem Plattenbelegungsgrad die *write cost* bei LFS-Systemen zunehmend ansteigt, aber erst bei höheren Plattenauslastungen die *write cost* von FFS erreicht bzw. übertrifft.

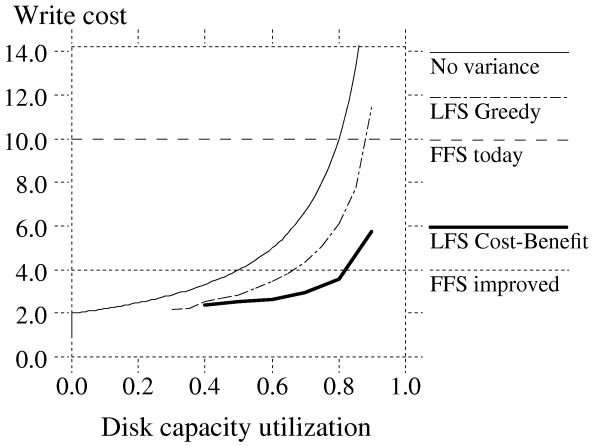

Abbildung 5: **Write Cost nach Plattenbelegung aus ROSENBLUM UND OUSTERHOUT [RO92] bei den Cleaner-Taktiken *„greedy"* und *„cost-benefit"* beim Test mit dem „Hot-and-Cold"-Muster.**

daß LFS beim Erzeugen und Löschen „um eine Größenordnung" schneller war. Außerdem „skaliert diese Performance mit der CPU-Geschwindigkeit".

Abschließend wurde die Datei-Schreibe- und Lese-Performance für „große Dateien" bestimmt. Es zeigte sich, das LFS beim Schreiben vor dem SunOS-Dateisystem liegt, beim Lesen allgemein leicht zurück. Beim sequentiellen Lesen einer Datei, die zuvor mit einem Random-Muster geschrieben wurde, liegt LFS weiter zurück.

Für eine genauere Beschreibung der Benchmarks und Diskussion der Ergebnisse sei auf ROSENBLUM UND OUSTERHOUT [RO92] verwiesen.

4.2 Vergleiche aus SELTZER ET AL. [SBMS93]

[SBMS93] beschreibt eine LFS Implementierung unter dem Betriebssystem BSD. Dabei wurden einige Modifikationen bezüglich des Vorbildes Sprite LFS gemacht. So wurde der „Speicherverbrauch reduziert", der „Cleaner in den User space verschoben", das *„directory operation log* eliminiert" und das „Segment-Layout auf der Platte geändert". Der Cleaner wurde mit der *cost-benefit*-Taktik implementiert.

Untersucht wurde unter anderem die „rohe Dateisystemperformance". Der Benchmark bestand darin, „eine Datei mit einer Größe S anzulegen". Anschließend wurde diese „Datei 50mal geschrieben bzw. gelesen". Im Benchmark werden LFS, FFS und EFS verglichen, bei letzerem handelte es sich um Dateisystem, das „ähnlich dem FFS aus MCVOY UND KLEIMAN [MK91] ist." Dem gegenübergestellt bezeichnet „RAW"-Performance die Geschwindigkeit einer „rohen" Platte.

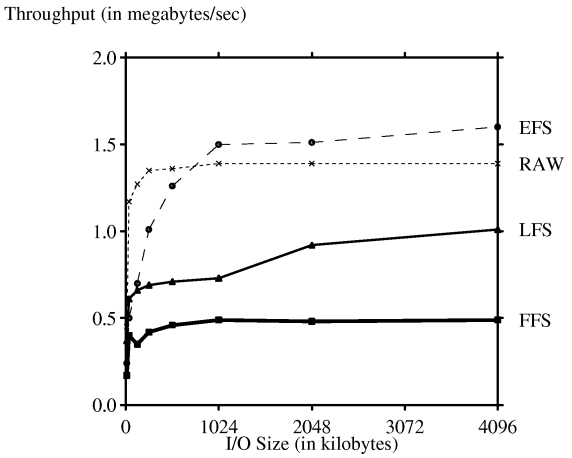

Abbildung 6: **Schreib-Durchsatz nach I/O-Größe aus SELTZER ET AL. [SBMS93]:** Zu erkennen ist, daß EFS ab einer bestimmten Transfergröße sogar besser als RAW abschnitt. Die Autoren erklärten das mit dem Puffern von Daten bei EFS, während bei RAW synchron geschrieben wurde. LFS hatte eine geringere Performance als EFS und RAW. Die dafür präsentierte Erklärung lautete, daß LFS erst dann auf die Platte schrieb, wenn ein Schwellwert an nicht geschriebenen Daten von „rund 800 kB" überschritten war, oder die Anwendung (hier der Benchmark) *„fsync"* aufrief. Dadurch fand der Datentransfer verspätet statt, wie auch Abbildung 7 zeigt.

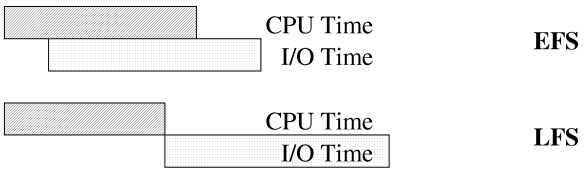

Abbildung 7: **Einfluß des Sammelns von Schreib-Zugriffen bei LFS aus SELTZER ET AL. [SBMS93]:** Die Balken zeigen die „einzelnen Phasen des Benchmarks beim Schreiben eines halben Megabytes". Zu erkennen ist, daß EFS in Summe schneller war.

FFS war sowohl beim Lesen, als auch beim Schreiben vergleichsweise langsam. Die Autoren SELTZER ET AL. [SBMS93] erklären das mit dem eingestellten *rot_delay* von 4 Millisekunden. In Abbildung 9 erkennt man den Zusammenhang von *rot_delay* und damit übersprungenen Blöcken (vgl. auch Abbildung 3 zum Interleave-Faktor).

Weiters nahmen SELTZER ET AL. [SBMS93] den „Andrew-Benchmark" vor, der „Software-Entwicklung" simulieren soll. Dazu wurden folgende Datei- und Verzeichnis-Operationen vor-

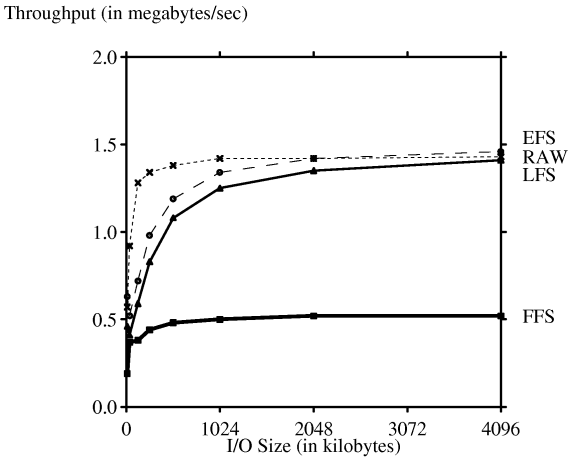

Abbildung 8: **Lese-Durchsatz nach I/O-Größe aus SELTZER ET AL. [SBMS93]:** Beim Lesen lagen EFS, RAW und LFS eng beieinander, nur FFS war vergleichsweise langsam.

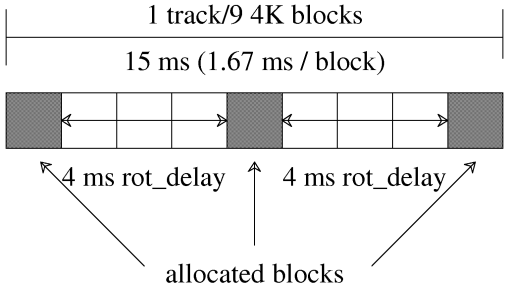

Abbildung 9: **Einfluß von *rot_delay* bei FFS aus SELTZER ET AL. [SBMS93]:** Beim eingestellten *„rot_delay* von 4 ms“ wird nur „jeder vierte Block auf der Platte belegt“. „FFS erreicht dadurch etwa ein Viertel der Platten-Bandbreite“ von 2,2 MB/s.

genommen:

1. „das Anlegen einer Verzeichnis-Hierarchie
2. Kopieren der Daten
3. Rekursives Untersuchen des Status jeder Datei
4. Untersuchen jedes Bytes jeder Datei
5. Kompilieren mehrerer Dateien“

Im „Single-User“-Betrieb erreichte „LFS in Summe einen 9% besseren Wert als EFS und FFS“. Insbesonders bei „Phase 1 Erzeuge Verzeichnisse“ und „Phase 2 Kopiere Daten“ war LFS schneller als EFS und FFS. Getestet wurde auch LFS während der Cleaner gelaufen ist, dieser hatte jedoch „eigentlich keinen Einfluß auf die Performance“.

Im „Multi-User“-Betrieb ergab sich ein anderes Bild, wie Abbildung 10 zeigt.

Abschließend wurde noch die Transaktions-Performance gemessen. Es handelte sich um einen „modifizierte Version des [...] TPC-B-Benchmark“. Dabei stellte sich heraus, daß „LFS eine 15%ige Performance-Verbesserung zu EFS brachte, solange der Cleaner nicht lief“. Sobald der Cleaner lief, brach die Performance von LFS „um 40%“ ein. Als die Segmentgröße von 1MB auf 256KB reduziert wurde, betrug der Verlust immer noch „35%“.

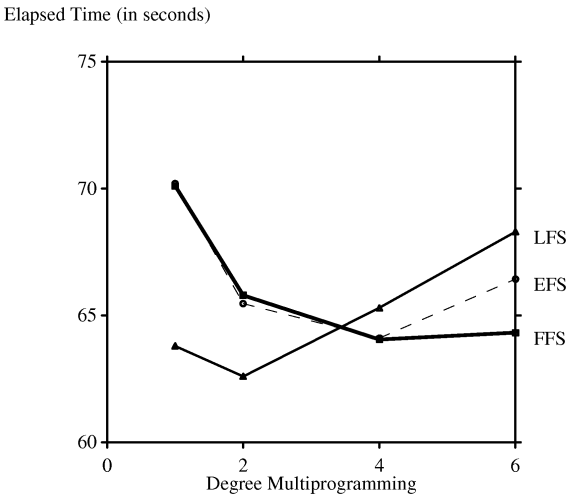

Abbildung 10: **Multi-User Andrew-Performance aus SELTZER ET AL. [SBMS93]:** Bei einem „MultiProgramming Degree“ von 4 überholte FFS LFS.

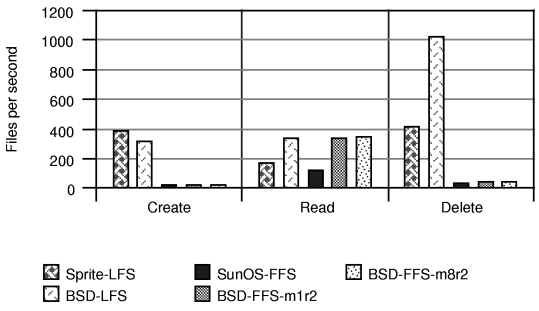

Abbildung 11: **Validierung von BSD-LFS in einem Benchmark für kleine Dateien (1KB) aus SELTZER ET AL. [SSB$^+$95]:** Datei-„Erzeugung“ und -„Löschen“ waren wesentlich schneller als bei den FFS-Versionen. Beim „Lesen“ profierte BSD-LFS von „größeren Spuren und Spur-Puffern“.

4.3 Vergleiche aus SELTZER ET AL. [SSB$^+$95]

In SELTZER ET AL. [SSB$^+$95] wurden sowohl LFS als auch FFS unter dem Betriebssystem BSD erneut verglichen. Dabei wurde BSD-LFS auch mit dem Sprite-LFS und dem SunOS-FFS aus ROSENBLUM UND OUSTERHOUT [RO92] verglichen. Da allerdings Sprite-LFS und SunOS-FFS auf einer anderen Hardware-Plattform getestet wurden, wurden zuerst „Skalierungsfaktoren“ für die „*CPU (SPECint92)*“, die „*Disk Bandwitdh*“ und die „*Avg Access (I/Os per second)*“ ermittelt, um die Performance von Sprite-LFS und SunOS-FFS auf die Hardware, auf der die Tests gemacht wurden, hochzurechnen.

Der erste Benchmark aus SELTZER ET AL. [SSB$^+$95] (Abbildung 11) verglich Datei-„Erzeugung“, -„Schreiben“ und -„Löschen“. Die Autoren SELTZER ET AL. [SSB$^+$95] hielten explizit fest, daß sie keine Erklärung für die Lösch-Performance von BSD-LFS haben. Im Test waren weiters auch BSD-FFS-m1r2 und BSD-FFS-m8r2. m*x* steht hier laut SELTZER ET AL. [SSB$^+$95] für den eingestellten *maxcontig*-Parameter.

Im nächsten Benchmark wurde die Performance für große Dateien verglichen. „Der Test bestand aus den folgenden fünf Schritten durch eine 100 MB-Testdatei:

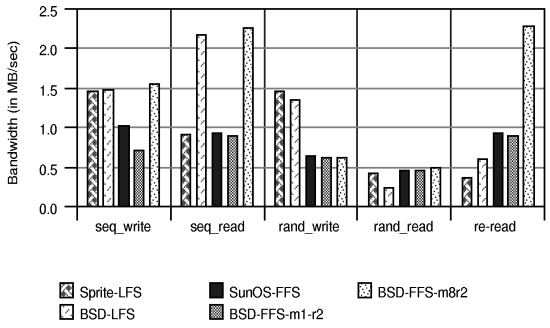

Abbildung 12: **Validierung von BSD-LFS für große Dateien aus SELTZER ET AL. [SSB+95]:** BSD-LFS ist beim „Schreiben“ „äquivalent“ wie Sprite-LFS, beim „Lesen“ jedoch schneller, außer beim Random-Zugriff.

1. Anlegen der Datei durch sequentielles Schreiben in 8 KB-Blöcken.
2. Einlesen der Datei sequentiell in 8 KB-Blöcken.
3. Schreiben von 100 KB in Random-Zugriffen mit einer Block-Größe von 8KB. [1]
4. Lesen von 100 KB in Random-Zugriffen mit einer Block-Größe von 8 KB.
5. Wiedereinlesen der Datei sequentiell in 8 KB Blöcken.“

Abbildung 12 zeigt die Ergebnisse von SELTZER ET AL. [SSB+95].

Mit den beiden vorangegangenen Benchmarks versuchte SELTZER ET AL. [SSB+95] zu zeigen, daß die Implementation von BSD-LFS „valid“ bzw. „faithful“ ist.

In den folgenden Benchmarks wurde von SELTZER ET AL. [SSB+95] die „sequentielle Performance als eine Funktion der Dateigröße verglichen“: „Der Datensatz bestand aus 32 Megabytes Daten, zerlegt in die passende Anzahl von Dateien für die zu messende Dateigröße. Im Fall kleiner Dateien, bei dem die Verzeichnis-Such-Zeit jeden anderen Verarbeitungs-Overhead dominiert, werden die Dateien in Unterverzeichnisse aufgeteilt, die nicht mehr als 100 Dateien enthalten. Im Fall großer Dateien werden entweder 32 MB oder zehn Dateien – was immer mehr Daten erzeugt – benutzt.“

Auch SELTZER ET AL. [SSB+95] führte einen „modifizierten TPC-B“-Transaktions-Benchmark durch. In Abbildung 17 erkennt man, daß LFS ohne laufenden Cleaner erheblich mehr Transaktionen pro Sekunde bewältigte, allerdings mit laufenden Cleaner eine „vergleichbare Performance“ zu FFS hatte.

Abschließend untersuchte SELTZER ET AL. [SSB+95] noch den Einfluß von Fragmentierung auf FFS über einen längeren Beobachtungszeitraum. SELTZER ET AL. [SSB+95] führt dazu aus:

[1] Hier unterscheidet sich SELTZER ET AL. [SSB+95] vom dort ebenfalls zitierten ROSENBLUM UND OUSTERHOUT [RO92]: letzterer führte den dritten und vierten Schritt mit 100 MB statt 100 KB durch. Ich vermutete vorerst einen Druckfehler, und habe die vorliegende Fassung von SELTZER ET AL. [SSB+95] mit der unter `http://www.usenix.org` – genauer der Postscript-Fassung unter `http://www.usenix.org/publications/library/proceedings/neworl/full_papers/seltzer.ps` – per Stand vom 17.10.2004 verglichen. Wie sich zeigte, ist auch in dieser Fassung die Größe 100 KB angegeben, allerdings unterscheidet sich diese Fassung in ein paar anderen Punkten von der von mir zitierten Fassung unter `http://www.eecs.harvard.edu/~margo/usenix.195/usenix.195.ps.gz`. Am augenscheinlichsten ist sicherlich – neben mehreren anderen – der Unterschied beim Skalierungsfaktor für die durchschnittliche Anzahl möglicher Ein-/Ausgabeanforderungen pro Sekunde. Während bei der von mir zitierten Fassung dieser Wert mit 1,7 angegeben wird, beträgt er in der anderen Fassung 1,2. Eine Gegenüberprüfung, ob ROSENBLUM UND OUSTERHOUT [RO92] in verschiedenen Versionen existiert, konnte nicht mehr vorgenommen werden.

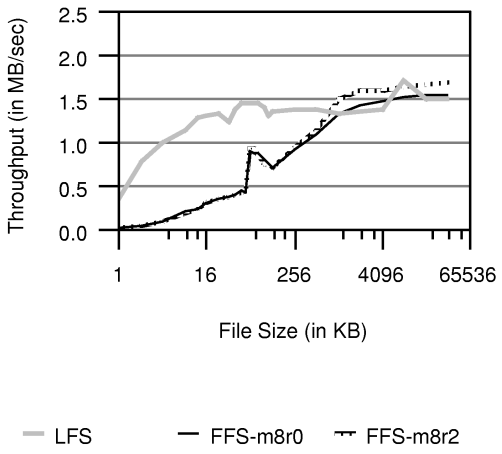

Abbildung 13: **„Erzeugungs"-Performance nach Dateigröße aus SELTZER ET AL. [SSB+95]:** Für kleine Dateigrößen war die „LFS Performance irgendwo zwischen 4 bis 10 mal besser als FFS." Bei größeren Dateien war die Geschwindigkeit „vergleichbar".

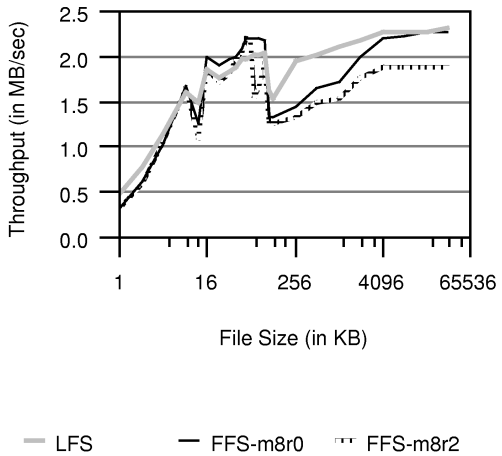

Abbildung 14: **„Lese-Performance" nach Dateigröße aus SELTZER ET AL. [SSB+95]:** *Übersetzung der Bildunterschrift aus* SELTZER ET AL. *[SSB+95]:* Für Dateien, die kleiner als 64 KB sind, ist die Performance für alle Datei-Systeme vergleichbar. Bei 64 KB werden die Dateien aus mehreren Clustern zusammengesetzt und Such-Strafen treten auf. Im Intervall zwischen 64 KB und 2 MB dominiert die Performance von LFS, weil FFS zwischen cylinder groups sucht, um die Daten gleichmäßig zu verteilen."

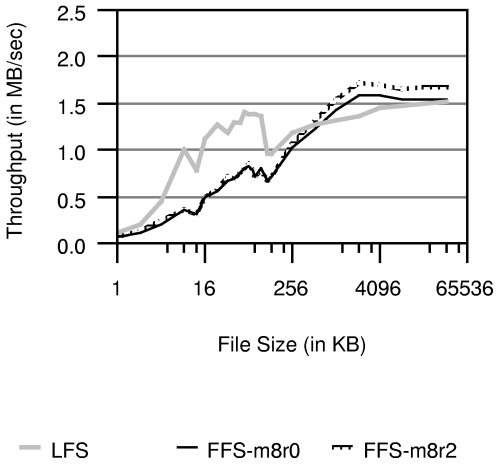

Abbildung 15: **„Überschreibe-Performance" nach Dateigröße aus SELTZER ET AL. [SSB+95]:** Verglichen mit dem „Erzeugungs-Test" zeigte sich, daß FFS und LFS „näher" beieinander lagen, weil LFS „tote Blöcke invalid" setzen mußte. Bei Dateigrößen über 256 KB war FFS schneller.

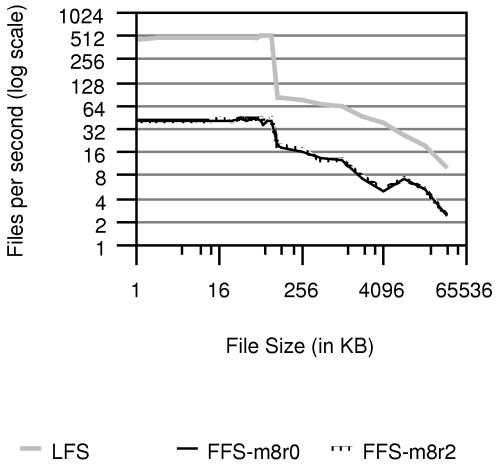

Abbildung 16: **„Lösch-Performance" nach Dateigröße aus SELTZER ET AL. [SSB+95]:** LFS war wegen des „asynchronen Operierens" generell schneller.

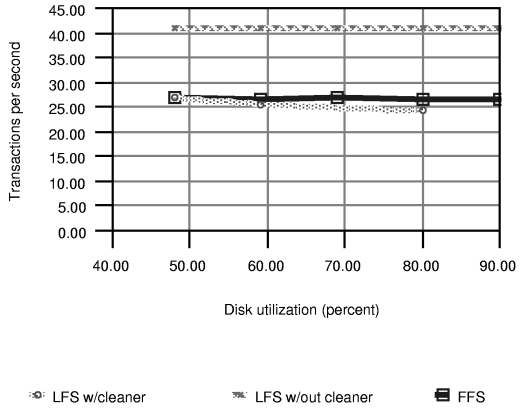

Abbildung 17: **Transactions-Performance nach Dateigröße aus SELTZER ET AL. [SSB+95]:** LFS ist mit laufendem *Cleaner* deutlich langsamer als ohne *Cleaner*.

„Die meisten Dateisysteme entfalten eine kleine Verschlechterung während des Zeitraums, mit einer Perfomance von 85-90% jener eines leeren Dateisystems. Wenn Änderungen auftreten, korrelieren sie oft mit der Belegung. Wir sehen keine Korrelation zwischen Dateisystem-Alter und der Performance-Verschlechterung."

4.4 Vergleiche aus SOULES ET AL. [SGSG02]

[SGSG02] verglich CVFS (comprehensive Versioning File System) in einer asynchronen und einer synchronen Betriebsart mit LFS, FFS und ext2. Da diese Benchmarks aber in einer NFS-Umgebung gemacht wurden, sind sie aus meiner Sicht nur bedingt zum direkten Vergleich der Dateisysteme geeignet.

4.5 Kritik von John K. Ousterhout

John K. Ousterhout, Co-Autor in ROSENBLUM UND OUSTERHOUT [RO92] hat sowohl an SELTZER ET AL. [SBMS93] als auch an SELTZER ET AL. [SSB+95] Kritik geübt.

Margo Seltzer, Co-Autorin von SELTZER ET AL. [SBMS93] und SELTZER ET AL. [SSB+95], hat Beiträge von Ousterhout auf der Website SELTZER [SE04] verlinkt. Am selben Ort finden sich auch Beiträge von Seltzer zu dieser Diskussion.

Kernpunkte der Kritik Ousterhout's an SELTZER ET AL. [SBMS93] sind gemäß SELTZER [SE04]: „Schleche BSD-LFS Implementierung", „Schlechte Benchmarkauswahl" und „Schlechte Analyse". Weiters kritisiert Ousterhout an SELTZER ET AL. [SSB+95] einen „falschen Benchmark-Zugang" zum „Messen des Effekts der Fragmentierung", „falsche Berechnungen" in „Abschnitt 4" und das Fehlen einer „simplen Optimierung".

Seltzer entgegnet dieser Kritik auf der Website SELTZER [SE04]. Die Chronologie der Auseinandersetzung ist leider nicht unmittelbar ersichtlich.

5 Zusammenfassung

Aus den Benchmarkergebnissen von SELTZER ET AL. [SSB+95] ersieht man, daß LFS beim Schreiben kleiner Dateien entschieden schneller als FFS war. Ebenso waren bei SELTZER ET AL. [SSB+95] das Anlegen und Löschen kleiner Dateien schneller als bei FFS. Bei großen Dateien und beim Lesen ist die Performance von LFS und FFS laut SELTZER ET AL. [SSB+95] vergleichbar. Ebenso sind die Unterschiede beim Transaktions-Benchmark aus SELTZER ET AL. [SSB+95] und beim Andrew-Benchmark aus SELTZER ET AL. [SBMS93] gering. Auffallend ist jedoch, daß LFS beim Transaktions-Benchmark aus SELTZER ET AL. [SSB+95] bei laufendem Cleaner wesentlich langsamer ist, als bei nicht laufendem Cleaner. Zumindest auf den ersten Blick erscheinen die Ergebnisse aus dem Transaktions-Benchmark aus SELTZER ET AL. [SBMS93] nicht mit denen aus SELTZER ET AL. [SSB+95] zu korrespondieren.

LFS hat mit der Idee, Daten- und Metadatentransfers zuerst zu sammeln, um sie dann in einem Zug auf die Platte zu schreiben, wie in ROSENBLUM UND OUSTERHOUT [RO92] beschrieben, eine interessante Entwicklungsrichtung aufgezeigt, die in anderen Systemen – wenn auch in modifizierter Form – genutzt werden kann. Das Führen eines Logs (zumindest für bestimmte Datenstrukturen) eröffnet außerdem gemäß ROSENBLUM UND OUSTERHOUT [RO92] die Möglichkeit einer beschleunigten Wiederherstellung nach einen Systemabsturz.

Aus heutiger Sicht erscheinen jedoch einige Gedanken aus den zitierten Arbeiten relativiert, da die fortschreitende Entwicklung von Festplatten genauso wie die Entwicklung von Caching- und Queueing-Technologien auf anderen Schichten – nämlich unterhalb des Dateisystems – für schnellere Transfers sorgen. Insbesonders der *rot_delay*-Parameter und der *maxcontig*-Parameter erscheinen mir aus heutiger Sicht mit dem erreichten Stand der Festplattentechnologie nicht mehr relevant.

Auch ist die Idee der *cylinder groups* aus meiner Sicht mittlerweile eingeschränkt nutzbar, da diese aufgrund der seitens der Festplatte berichteten Geometrie nicht zwingend zusammenhängende, schnell zugreifbare Bereiche beschreiben müssen.

Entscheidend für die Auswahl eines Dateisystems können aber unter anderem auch Faktoren wie Zuverlässigkeit, Versionierbarkeit (das System speichert mehrere Versionen einer Datei) und Sicherheit sein – nicht nur Performance.

Literatur

[AN03] D. Anderson. You Don't Know Jack about Disks. ACM Queue vol. 1, no. 4 - June 2003 Verwendete Version (verifiziert 20.10.2004): `http://www.acmqueue.org/modules.php?name=Content&pa=showpage&pid=46&page=1`

[IBM84] IBM Deutschland Produkt-Vertrieb GmbH. Betriebssystem DOS (Disk Operating System) Referenz-Handbuch. Übersetzung von IBM Personal Computer Software, Disk Operating System, Version 3.00, Reference 6322666. © Copyright International Business Machines Corporation 1983. © Copyright IBM Deutschland GmbH 1984. Herausgegeben von MS NLS Kleine und Mittlere Systeme 0426, Oktober 1984.

[MJLF84] Marshall K. McKusick, William N. Joy, Samuel J. Leffler and Robert S. Fabry. A Fast File System for UNIX. *Computer Systems*, 2(3):181–197, 1984. Verwendete Version (verifiziert 30.9.2004): `http://www.cs.cornell.edu/Courses/cs614/2003SP/papers/KJL84.pdf`.

[MK91] L. W. McVoy and S. R. Kleiman. Extent-like Performance from a UNIX File System. In *Proceedings of the USENIX Winter 1991 Technical Conference*, pages 33–43, Dallas, TX, USA, 21–25 1991. Verwendete Version (verifiziert 30.9.2004): `http://ficus-www.cs.ucla.edu/classes/239_2.spring96/papers/extent_like_filesystem_performance.ps`.

[MRC+97] J. N. Matthews, D. Roselli, A. M. Costello, R. Y. Wang and T. E. Anderson. Improving the Performance of Log-Structured File Systems with Adaptive Methods. In *Proc. Sixteenth Symposium on Operating Systems Principles*, pages 238–251, Saint Malo, France, October 1997. Verwendete Version (verifiziert 30.9.2004): `http://www.cs.princeton.edu/~rywang/berkeley/papers/sosp97.ps`.

[RO92] Mendel Rosenblum and John K. Ousterhout. The Design and Implementation of a Log-Structured File System. *ACM Transactions on Computer Systems*, 10(1):26–52, 1992. Verwendete Version (verifiziert 30.9.2004): `http://bbcr.uwaterloo.ca/~brecht/courses/756/readings/filesys/lfs-SOSP91.ps`.

[SBMS93] Margo I. Seltzer, Keith Bostic, Marshall K. McKusick and Carl Staelin. An Implementation of a Log-Structured File System for UNIX. In *USENIX Winter*, pages 307–326, 1993. Verwendete Version (verifiziert 30.9.2004): `http://www.pha.com.au/papers/usenix.1.93.ps`.

[SE04] Margo I. Seltzer. LFS and FFS Supplementary Information. Verwendete Version (verifiziert 17.10.2004): `http://www.eecs.harvard.edu/~margo/usenix.195/`, verifiziert 30.9.2004.

[SGSG02] C. Soules, G. Goodson, J. Strunk and G. Ganger. Metadata efficiency in a comprehensive versioning file system. Technical report CMU–CS–02–145, Carnegie Mellon University, 2002. Verwendete Version (verifiziert 30.9.2004): `http://www.pdl.cmu.edu/PDL-FTP/Secure/FAST03.pdf`.

[SSB+95] Margo I. Seltzer, Keith A. Smith, Hari Balakrishnan, Jacqueline Chang, Sara McMains, and Venkata N. Padmanabhan. File System Logging versus Clustering: A Performance Comparison. In *USENIX Winter*, pages 249–264, 1995. Verwendete Version (verifiziert 17.10.2004): `http://www.eecs.harvard.edu/~margo/usenix.195/usenix.195.ps.gz`.

[ST98] William Stallings. Operating Systems: Internals and Design Principles. Third Edition. ©1998 Prentice-Hall. Inc., Simon & Schuster / A Viacom Company, Upper Saddle River, New Jersey 07458, USA. ISBN 0–13–917998–4.

[TI92] M. Tischer. *PC Intern 3.0*. DATA-Becker-GmbH, Düsseldorf, 1992.