Hochschule
Augsburg University of
Applied Sciences

Diplomarbeit

Studienrichtung Informatik

Andreas Rid

Performanzsteigerung des Dateisystems Ext4
durch physikalische Umsortierung

Verfasser der Diplomarbeit:
Andreas Rid
Ewigkeitsweg 4
86836 Obermeitingen
andreas@rid-net.de

Fakultät für Informatik
Telefon: +49 821 5586-3450
Fax: +49 821 5586-3499

Erstprüfer: Prof. Dr. Gundolf Kiefer

Zweitprüfer: Prof. Dr. Wolfgang Klüver

Abgabe der Arbeit: 19.05.11

Hochschule Augsburg
University of Applied Sciences
Baumgartnerstraße 16
D 86161 Augsburg

Telefon: +49 821 5586-0
Fax: +49 821 5586-3222
http://www.hs-augsburg.de
poststelle@hs-augsburg.de

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides Statt, dass ich die vorliegende Arbeit selbständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Die aus fremden Quellen (einschließlich elektronischer Quellen) direkt oder indirekt übernommenen Gedanken sind ausnahmslos als solche kenntlich gemacht.

Zusammenfassung

Die im Rahmen der Diplomarbeit entwickelte Software reduziert die Startzeit von Programmen und des Bootvorgangs durch eine sequentielle Anordnung von Dateien in ihrer Aufrufreihenfolge. Durch Überwachung von Dateizugriffen wird eine Liste relevanter Dateien erstellt. Unter der Verwendung der Online-Defragmentierungsschnittstelle des Dateisystems Ext4 werden anschließend die Dateien auf dem Datenträger entsprechend der Dateiliste neu angeordnet. Weiter ist ein Programm entstanden, mit dessen Hilfe parallel zum Startvorgang Dateien vorzeitig von der Festplatte in den Arbeitsspeicher übertragen werden können. Mit den entwickelten Werkzeugen ließ sich die Bootzeit eines frisch installierten Debian Linux von 44,5 auf 13,7 Sekunden reduzieren.

Abstract

The software, that was developed in the scope of this thesis accelerate the boot process as well as application startups by placing files on disk sequentially in their access order. A list of relevant files is created by monitoring file accesses. The list is used to reorder files on disk through the online defragmentation ioctl of the ext4 filesystem. Furthermore files are read-ahead into memory in parallel to program startup. Using the developed tools, the boot time of a freshly installed Debian Linux could be cut from 44.5 to 13.7 seconds.

Inhaltsverzeichnis

1 Einleitung **1**
- 1.1 Motivation . . . 1
- 1.2 Ziel der Arbeit . . . 2
- 1.3 Aufbau der Arbeit . . . 3

2 Grundlagen und Stand der Technik **4**
- 2.1 Das Fragmentierungsproblem . . . 4
  - 2.1.1 Abschätzung der Festplattenzugriffszeit . . . 4
  - 2.1.2 Fragmentierungsarten . . . 4
  - 2.1.3 Streuung von Dateien . . . 5
  - 2.1.4 Fragmentbildung . . . 6
- 2.2 Techniken, um den Start- und Bootvorgang zu beschleunigen . . . 7
  - 2.2.1 *readahead-fedora* . . . 7
  - 2.2.2 *sreadahead* und *ureadahead* . . . 8
  - 2.2.3 *preload* . . . 8
  - 2.2.4 *upstart, parstart, systemd* . . . 9
  - 2.2.5 Bewertung der Ansätze . . . 9
- 2.3 Das Ext4-Dateisystem . . . 10
  - 2.3.1 Das Layout . . . 10
  - 2.3.2 Der I-Node-Allocator . . . 13
  - 2.3.3 Der Block-Allocator . . . 14
  - 2.3.4 Möglichkeit zur Online-Defragmentierung . . . 16
- 2.4 Linux Audit . . . 17
  - 2.4.1 Audit-Filter . . . 17
  - 2.4.2 Audit-Nachrichten . . . 18

3 e4rat-collect – Sammeln von Dateizugriffen **21**
- 3.1 Relevante Systemaufrufe . . . 22
- 3.2 Kommunikation mit Linux Audit . . . 24
  - 3.2.1 Verbindung herstellen . . . 24
  - 3.2.2 Audit-Regeln . . . 24
  - 3.2.3 Empfangen und Parsen von Audit-Nachrichten . . . 25
  - 3.2.4 Kollision mit auditd . . . 27
- 3.3 Die Datenstruktur der Dateiliste . . . 28
- 3.4 Beenden des Sammelvorgangs . . . 30

4 e4rat-realloc – physikalische Block-Umsortierung **32**
- 4.1 Vorbereitung . . . 32
  - 4.1.1 Dateisystem prüfen . . . 32
  - 4.1.2 Dateiattribute prüfen . . . 33
  - 4.1.3 Ermitteln der Dateigröße . . . 33

4.2 Erstellen der Spenderdateien . . . 35
4.2.1 Methode „Pre-Allocation“ . . . 35
4.2.2 Methode „Top Level Directory“ . . . 39
4.2.3 Methode „Locality Group“ . . . 40
4.3 Blockverschiebung . . . 41
4.3.1 EXT4_IOC_MOVE_EXT . . . 42
4.3.2 Derzeit ausgeführte Dateien . . . 43

5 e4rat-preload – Vorladen von Dateien 45
5.1 Problemanalyse . . . 45
5.2 Angewendete Zugriffsstrategie . . . 46

6 Besonderheiten bei der Implementierung 47
6.1 Ausführung als Init-Prozess . . . 47
6.2 Statisch und dynamisch gelinkte Bibliotheken . . . 47
6.3 Konfigurationsdatei . . . 48
6.4 Fehlerbehandlung . . . 49

7 Ergebnisse 51
7.1 Versuchsumgebung . . . 51
7.2 Optimierung des Bootvorgangs . . . 53
7.3 Optimierung von Anwendungen . . . 54

8 Fazit und Ausblick 57
8.1 Fazit . . . 57
8.2 Ausblick . . . 57
8.2.1 Defragmentieren von freien Speicherbereichen . . . 57
8.2.2 Verschieben nicht regulärer Dateien . . . 58
8.2.3 Verschieben der I-Node-Strukturen . . . 58
8.2.4 Stufenweises Lesen . . . 58
8.2.5 Verschieben wachsender Dateien . . . 59

Literaturverzeichnis 60

1 Einleitung

1.1 Motivation

In den letzten Jahren entwickelte sich die Festplatte zunehmend zum Flaschenhals eines PC's. Während die Anforderung an diese - durch immer leistungsfähigere Prozessoren und den immer größer werdenden Softwareumfang - anstieg, blieb die Zugriffszeit annähernd unverändert [1, 14].

Das sieht man daran, dass jeder Rechner – ob alt oder neu – eine ähnlich lange Startzeit benötigt. Die meiste Zeit verstreicht beim Warten auf Daten von der Festplatte. Auch ein frisch installierter Rechner ist davon betroffen. Das Problem ist schlichtweg, dass jede Anfrage an die Festplatte immer mit einer gewissen Verzögerung verbunden ist. Während auf die Übermittlung der Daten gewartet wird, vergeht also wertvolle Zeit.

Die Übertragungsrate der Festplatte wird stets weiterentwickelt. Das Bussystem SATA bietet derzeit eine maximale Übertragungsrate von 300 $\frac{MB}{s}$ an. Kaum weiterentwickeln konnte sich die mechanische Komponente der Festplatte aufgrund physikalischer Grenzen. So lässt sich beispielsweise die Rotationsgeschwindigkeit durch entstehende Vibrationen nicht beliebig erhöhen. Derzeit stellt die Geschwindigkeit von 15000 Umdrehungen pro Minute das Maximum dar. Die Bewegungen des Magnetkopfes sind ebenso begrenzt. Ständiges Beschleunigen und Abbremsen führt zu Materialermüdung.

Insbesondere beim Bootvorgang werden sehr viele und kleine Dateien angefordert. Um sie zu lesen, sind zahlreiche Magnetkopfbewegungen notwendig. In der Abbildung 1 sind CPU- und Festplattenaktivitäten während eines Bootvorgangs dargestellt. Bei dem Bootvorgang handelt es sich um frisch installiertes Debian Linux. Für den Startvorgang werden rund 2600 Dateien von der Festplatte gelesen. Die maximale Übertragungsrate der Festplatte beträgt 61 $\frac{MB}{s}$. Aufgrund der Verzögerungen wird diese jedoch nie erreicht. Die CPU weist keine hohe Auslastung auf.

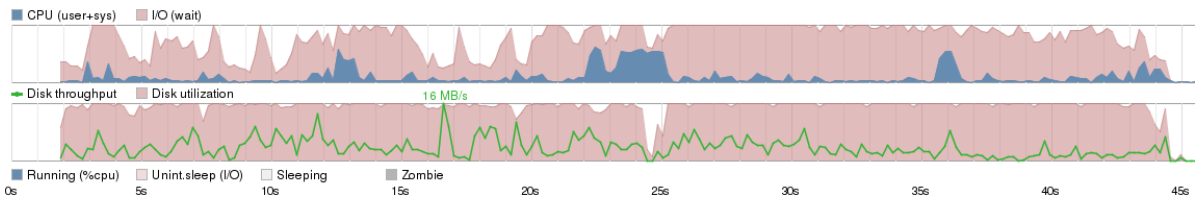


Abbildung 1: Bootchart Debian Squeeze

Ein schneller Startvorgang ist gerade unterwegs bei Laptops erwünscht. Immer häufiger möchte man kurzfristig und schnell Informationen abrufen bzw. sie mit anderen tauschen. Lange Startzeiten werden daher als sehr störend wahrgenommen.

1.2 Ziel der Arbeit

Das Ziel dieser Arbeit ist es, die Startzeit von Programmen sowie die Dauer des Bootvorgangs durch Optimierung der Festplattenbelegung zu reduzieren. Dabei soll zum ersten Mal der Ansatz verfolgt werden, durch geeignete Umsortierung die mechanisch bedingten Verzögerungen zu minimieren. Eine geringere Verzögerung erhöht die mittlere Übertragungsrate und beschleunigt den Lesezugriff.

Logisch zusammengehörende Dateien sollen auf der Festplatte entsprechend ihrer Zugriffsreihenfolge gruppiert und physikalisch der Reihe nach neu angeordnet werden. Die Neuanordnung führt zu einer permanenten Verbesserung der Übertragungsrate, was folglich zu einer Verkürzung der Startzeit führt.

Im Rahmen dieser Diplomarbeit ist das Projekt e4rat (**R**educe disk **A**ccess **T**ime on **E**xt**4** filesystems) entstanden. Mit den entwickelten Anwendungsprogrammen *e4rat-collect*, *e4rat-realloc* und *e4rat-preload* kann jeder beliebige Vorgang bzw. jedes beliebige Programm durch Reduzierung der Festplattenzugriffszeiten beschleunigt werden.

e4rat-collect Bevor der Festplattenzugriff optimiert werden kann, wird mit *e4rat-collect* eine Liste aus relevanten Dateien erstellt. Es überwacht Dateizugriffe von einzelnen Programmen oder des Gesamtsystems. Die Dateiliste stellt die Grundlage zur Umsortierung durch *e4rat-realloc* dar.

e4rat-realloc Das Werkzeug *e4rat-realloc* ordnet Dateien anhand einer Dateiliste sequentiell auf dem Datenträger an. Dieses Werkzeug stellt damit den zentralen Kern der Arbeit dar. Die Umsortierung kann im laufenden Betrieb erfolgen.

e4rat-preload Mit *e4rat-preload* lässt sich der Startvorgang zusätzlich beschleunigen. Es liest parallel zur Ausführung Dateien vorzeitig von der Festplatte in den Arbeitsspeicher. Es wendet eine bessere Zugriffsstrategie an und macht sich die Festplattenruhephasen zu Nutze.

e4rat steht unter der Open-Source Lizenz GPLv3 und kann von der Projektseite auf SourceForge unter http://e4rat.sf.net herunter geladen werden. Neben dem Quellcode werden darüber hinaus Debian-Pakete für die Architekturen *i386* und *amd64* angeboten. Ferner kann e4rat über die Paketverwaltung von Arch Linux aus dem Arch Linux User Repository (AUR) installiert werden.

1.3 Aufbau der Arbeit

Im Kapitel 2 befinden sich die Grundlagen für die Arbeit. Es werden bisherige Projekte vorgestellt und der Aufbau des Dateisystems Ext4 sowie die Funktionsweise von Linux Audit erläutert. Jedem der entwickelten Werkzeuge wird ein eigenes Kapitel 3, 4 und 5 gewidmet. Besonderheiten bei der Implementierung, die alle Werkzeuge betreffen, finden sich im Kapitel 6. Die Ergebnisse können dem Kapitel 7 entnommen werden. Im Kapitel 8 befinden sich das Fazit und weitere Maßnahmen, um Startvorgänge noch weiter zu optimieren.

2 Grundlagen und Stand der Technik

Dieses Kapitel erklärt Begriffe und Grundlagen, die für diese Arbeit entscheidend sind. Zu Beginn wird das Problem von Fragmentierung genauer beschrieben. Anschließend werden bisherige Projekte vorgestellt, die jeweils in eigenen Ansätzen Startvorgänge optimieren. Darüber hinaus werden Grundlagen über das Ext4 Dateisystems vermittelt. Zum Schluss folgt ein Überblick über die Linux-Monitoring-Schnittstelle *Linux Audit*.

2.1 Das Fragmentierungsproblem

2.1.1 Abschätzung der Festplattenzugriffszeit

Die Zugriffszeit misst die Zeit von der Auslösung einer Anfrage bis zur erfolgreichen Übermittlung eines Blockinhalts vom Datenträger in den Hauptspeicher. Die Festplatte, mit der der Bootchart aus Abbildung 1 erstellt wurde, hat eine mittlere Zugriffszeit von 13,1 ms[1]. Sie setzt sich zusammen aus der Spurwechselzeit t_s, der Latenzzeit, auch bekannt als Rotationsverzögerung t_r, und der Kommando-Latenz t_k.

$$Zugriffszeit = t_s + t_r + t_k \tag{1}$$

Die benötigte Zeit für einen Spurwechsel ist abhängig von der Wegstrecke, die der Magnetkopf zurücklegt. Für die Latenzzeit wird der Mittelwert, nämlich die Zeit für eine halbe Umdrehung, verwendet. Standardmäßig arbeiten Festplatten mit einer Drehzahl von 7200 $\frac{Umdrehungen}{min}$, was nahezu die Hälfte der Gesamtverzögerung ausmacht.

$$t_r = \frac{1}{2 * 7200\frac{1}{min}} = \frac{60 * 1000}{2 * 7200} ms = 4,1\overline{6} ms \tag{2}$$

Die Kommando-Latenz entspricht der Verzögerung, in der der Festplattencontroller den Befehl interpretiert und koordiniert. Diese ist sehr klein und wird für die Berechnung der Zugriffszeit häufig vernachlässigt [31]. Gut zu erkennen ist, dass die meiste Zeit für eine Neupositionierung des Magnetkopfes aufgewendet wird.

2.1.2 Fragmentierungsarten

Unter Fragmentierung versteht man die Aufteilung von logisch zusammengehörenden Datenblöcken in zwei oder mehrere Blockbereiche. Man unterscheidet bei einer Fragmentierung drei unterschiedliche Arten:

[1] Der Wert der Zugriffszeit errechnet sich aus den Herstellerangaben [29]

Die Fragmentierung einzelner Dateien
Der Dateiinhalt ist auf dem Datenträger in mindestens zwei Bereiche unterteilt. Um die Datei zu lesen sind daher mehrere Kopfbewegungen notwendig. Die höhere Zugriffszeit verlangsamt den Lesevorgang. Moderne Dateisysteme versuchen durch die Verwendung von Heuristiken Fragmentierung zu vermeiden. Da das Dateisystem jedoch keine Kenntnis darüber besitzt, wieweit eine Datei in Zukunft wachsen wird, können Heuristiken das Problem der Fragmentierung häufig eindämmen jedoch nicht verhindern.

Die Verstreuung von logisch zusammengehörenden Dateien
Programme bestehen meist aus mehreren Dateien. Beim Start werden diese der Reihe nach von der Festplatte angefordert. Sie gehören somit logisch zusammen. Dennoch sind die Dateien häufig auf der Festplatte verteilt. Es entstehen beim Lesen zwischen den Dateien vermeidbare Wartezeiten. Durch Aneinanderreihung der Dateien könnten diese jedoch vermieden werden.

Die Fragmentierung ungenutzten Speicherplatzes
Der Datenträger besitzt viele aber nur sehr kleine freier aufeinanderfolgende Blockbereiche. Dies wirkt sich zwar nicht direkt auf die Performanz der Festplatte aus, erschwert jedoch das Anlegen neuer Dateien. Wird kein ausreichend großer Bereich gefunden, wird die Datei trotz geringer Festplattenauslastung in mehrere Fragmente zerrissen.

2.1.3 Streuung von Dateien

Unter UNIX-Betriebssystemen herrscht eine strikte Verzeichnisstruktur [23]. Je nach Art und Typ findet eine jede Datei einen vordefinierten Platz in der Verzeichnishierarchie. So werden beispielsweise Anwendungsdaten von Programmen, Bibliotheken und Konfigurationsdateien getrennt. Die Aufteilung verhilft den Überblick über die zahlreichen Dateien zu bewahren. Ebenso können Programme leichter Dateien, beispielsweise Bibliotheken, mit anderen Programmen teilen.

Abbildung 2 zeigt einen Ausschnitt der Verzeichnisstruktur, in der sich das Wurzelverzeichnis "/" aufteilt.

Ausführbare Dateien werden in der Regel in den Verzeichnissen /bin, /sbin, /usr/bin oder /usr/sbin abgelegt; Bibliotheken hingegen meistens in /lib oder /usr/lib. Abhängig von der Anzahl installierter Programme können diese Verzeichnisse zum Teil mehrere tausend Dateien enthalten. Das Dateisystem versucht diese physikalisch nahe beieinander abzuspeichern. Dennoch erstrecken sich insbesondere große Verzeichnisse über mehrere Spuren. Beim Lesen von Dateien aus dem gleichen Verzeichnis ist die Reihenfolge entscheidend. Bei jedem Spurwechsel kommt die volle Rotationsverzögerung zum Tragen.

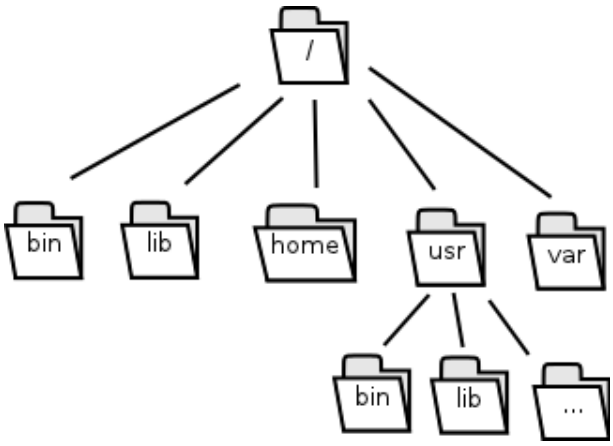


Abbildung 2: Unix Verzeichnisstruktur

Sind Dateien eines Programms auf mehrere Verzeichnisse aufgeteilt, vergrößert sich entsprechend der Abstand zu den Dateien. Um sie zu lesen legt der Magnetkopf weite Strecken zurück. Lange Spurwechselzeiten sind die Folge.

2.1.4 Fragmentbildung

Zusätzlich dazu, dass die Verzeichnisstruktur Dateien voneinander entfernt, kann auch der Dateiinhalt auf der Festplatte fragmentiert vorliegen.

Jede Datei behält ihre Blockbelegung auf der Festplatte bis zu ihrem Lebensende. Demzufolge führt es zu einer permanenten Verschlechterung der Performanz, sobald Dateien fragmentiert abgespeichert werden. Insbesondere sehr langsam wachsende Dateien laufen Gefahr zu fragmentieren. Befinden sich direkt hinter der Datei keine freien Blöcke, ist das Dateisystem gezwungen, einen entfernten Block zu allozieren. Die Datei fragmentiert.

Wenn kein ausreichend großer Blockbereich auf der Festplatte ungenutzt vorliegt, muss die Datei bereits zum Zeitpunkt der Erstellung fragmentiert abgelegt werden. Das Dateisystem hat keine andere Möglichkeit, als die Datei in zwei oder mehr Teilen abzulegen. Eine hohe Festplattenauslastung führt auf Dauer zu einem stark fragmentierten Dateisystem. Sind Dateien einmal fragmentiert, können diese nur manuell mit Hilfe eines Defragmentierwerkzeugs wieder zu einem fortlaufenden Stück vereint werden.

2.2 Techniken, um den Start- und Bootvorgang zu beschleunigen

2.2.1 *readahead-fedora*

readahead von Fedora optimiert den Bootvorgang durch eine bessere Festplattenzugriffsstrategie, mit der weite Magnetkopfbewegungen minimiert werden [22].

In der Vorbereitungsphase wird der Bootvorgang analysiert. Mit Hilfe von Linux Audit, siehe Kapitel 2.4, überwacht das Werkzeug *readahead-collector* Dateizugriffe und erstellt daraus eine Liste relevanter Dateien. Diese bildet die Grundlage für die Optimierung des Bootvorgangs. Um alle Dateien zu erfassen, die im Bootvorgang benötigt werden, kann der Sammelvorgang auch als Init-Prozess ausgeführt werden.

Im optimierten Bootvorgang, mit dem Werkzeug *readahead*, werden die Dateien in der Liste nach der physikalischen Position auf dem Datenträger umsortiert[2]. Anschließend wird die Datei vorzeitig in den Arbeitsspeicher (Page-Cache) mit dem Befehl readahead(2) übertragen. Die Strecke, die der Magnetkopf überwinden muss, wird durch die Umsortierung verkürzt. Das entspricht einer Verringerung der Zugriffszeit aufgrund kleinerer Spurwechselzeiten.

Wurden alle Dateien erfolgreich von der Festplatte in den Cache übertragen, sollten keine großen Verzögerungen auf Grund der Festplatte mehr entstehen. Der Geschwindigkeitsvorteil resultiert demnach aus der besseren Zugriffsstrategie, die durch das Umsortieren zustande gekommen ist.

Mit *readahead-collector* können die Dateien auf zwei Listen, nämlich *early* und *later* aufgeteilt werden. Die Aufteilung stellt einen Kompromiss dar zwischen Verringerung der Zugriffszeit aufgrund der Umsortierung der Dateiliste und der Problematik auf Grund der Aufrufreihenfolge. Das Risiko, dass während des vorzeitigen Einlesens der Dateien der Bootvorgang unterbrochen wird, ist somit gemindert. In der Dateiliste *later* werden alle Dateien gespeichert, die von der Festplatte gelesen werden, nachdem auf die Datei */etc/init.d/readahead_later* zugegriffen wurde.

Bei der Erstellung der Dateiliste findet keine Analyse über die Art des Zugriffs statt. In der Liste können sich daher temporäre Dateien bzw. solche, auf die ausschließlich schreibend zugegriffen wird, befinden. Jede Datei wird als Ganzes betrachtet.

[2]Die Umsortierung der Liste erfolgt nur im sog. *full-mode*. Liegt die Dateiliste bereits sortiert vor, entfällt die Sortierung. *readahead-fedora* bezeichnet diesen Fall als *fast-mode*.

2.2.2 *sreadahead* und *ureadahead*

Ähnlich wie bei *readahead-fedora*, werden in den Projekten *sreadahead* und *ureadahead* Dateien vorzeitig in den Arbeitsspeicher übertragen [26, 28]. Dies führt wiederum zu einer hohen Cache-Trefferrate. *sreadahead* steht für „super-readahead“ und ist speziell auf Solid State Disks ausgerichtet. Die Projektabspaltung *ureadahead* (über-readahead) verspricht neben Solid-State-Disks auch eine Optimierung des Bootvorgangs bei rotierenden Festplatten.

Der Bootvorgang wird im ersten Schritt mit Hilfe eines gepatchten Kernels analysiert. Der Patch ermöglicht nicht nur die Ermittlung der Dateien, welche im Zugriff stehen, sondern auch das Herausfinden der Bereiche, die tatsächlich gelesen werden. Über das virtuelle Dateisystem *debugfs* werden die Informationen vom Kernel entgegen genommen und in eine sog. „Pack“-Datei gespeichert. Im Falle einer rotierenden Festplatte sortiert ureadahead die gelisteten Dateien, bevor die Pack-Datei geschrieben wird, nach den jeweiligen I-Node-Nummern. Die Blockbereiche innerhalb einer Datei hingegen werden nach der physikalischen Position sortiert.

Ab dem zweiten Bootvorgang kann die Pack-Datei genutzt werden, um den Bootvorgang zu optimieren. *ureadahead* liest im Falle einer rotierenden Festplatte zu Beginn alle I-Node-Tabllen ein, in denen sich mindestens eine I-Node-Struktur der gelisteten Dateien befindet. Anschließend liest *ureadahead* die Blockbereiche der Dateien der Reihe mit dem Befehl readahead(2) ein. Für SSDs hingegen werden keine I-Node-Tabellen vorsorglich eingelesen. Durch eine Vielzahl von Threads führt es den Einlesevorgang teilweise parallel aus.

Die Pack-Datei von *sreadahead* und *ureadahead* enthält keine Dateien aus dem frühen Stadium des Bootvorgangs. Um die Programme nutzen zu können, muss ein Kernel-Patch eingespielt werden, was den Bau eines neuen eigenen Kernels bedarf. Für unerfahrene Benutzer ist das eine nicht triviale Angelegenheit. Bei der Distribution Ubuntu ist *ureadahead* Bestandteil des Basissystems. Das Einspielen des Patches ist dort nicht mehr notwendig.

2.2.3 *preload*

preload ist ein Hintergrunddienst, entwickelt von Behdad Esfahbod, der die Startzeit von Programmen verkürzt, indem er Dateien von der Festplatte vorlädt [7]. Mit Hilfe einer Markow-Kette implementiert Esfahbod einen lernenden Algorithmus, der das Verhalten eines Anwenders am Rechner studiert. In einem festen Intervall von 20 Sekunden liest *preload* Prozessinformationen und derzeit geöffnete Dateien aus dem Proc-Dateisystem ein.

Anhand dieser Daten wird eine Dateiliste für jeden Prozess generiert. Jedes ausgeführte

Programm bewirkt eine Zustandsänderung. Je nach Zustand liest *preload* die Dateien der Programme ein, die wahrscheinlich als nächstes gestartet werden.

Die Intervalle von 20 Sekunden bilden Momentaufnahmen. Das System wird nur lückenhaft überwacht. Die erstellten Dateilisten sind daher unvollständig und enthalten nur die ausführbare Datei sowie die verlinkten Bibliotheken. Der Einlesevorgang ist nicht optimiert, da die Dateien in der Liste nicht nach ihrer physikalischen Position sortiert werden. *preload* wirkt sich nur auf den Start von Anwendungen positiv aus, der Bootvorgang hingegen wird nicht beschleunigt.

2.2.4 *upstart, parstart, systemd*

Die Projekte *upstart*, *parstart* und *systemd* sind Hintergrundprogramme, die als Init-Prozess ausgeführt werden. Sie stellen eine Alternative zu SysVinit dar. Mit dem Ziel den Startvorgang möglichst zu parallelisieren, sind sie eine weitere Möglichkeit den Bootvorgang zu beschleunigen. Anstelle jedoch den Zugriff auf die Festplatte zu optimieren bewirkt die Parallelisierung eine bessere Auslastung der CPU. Während ein Dienst auf die Hardware wartet, kann inzwischen ein anderer Prozess fortgesetzt werden. Auch lassen sich durch Parallelisierung Multicore-Prozessoren besser nutzen.

2.2.5 Bewertung der Ansätze

Jedes der oben vorgestellten Programme nutzt einen ganz eigenen Ansatz Startzeiten zu reduzieren. Dabei ist jedes für sich entweder auf die Optimierung von Anwendungsprogrammen oder auf die des Bootvorgangs ausgerichtet.

Das zu lösende Problem ist stets dasselbe. Die Dauer eines Startvorgangs ist stark von den Verzögerungen, bedingt durch die Festplatte, abhängig. Zum Teil kann das Problem durch vorzeitiges Einlesen gelöst werden. Die Übertragungsrate der Festplatte lässt sich steigern, wenn der Zugriff koordiniert abläuft. Verzögerungen, verursacht durch lange Spurwechselzeiten, können somit verringert, aber nicht eliminiert werden. Darüber hinaus stellt die bessere Zugriffsstrategie kein Mittel zur Reduzierung der Rotationsverzögerung, die bei jedem Spurwechsel entsteht, dar. Außerdem kann der Fortschritt des Bootvorgangs während des vorzeitigen Einlesens ins Stocken geraten. Durch die Sortierung entscheidet die physikalische Position der Datei auf dem Datenträger über die Position in der Liste. So kann es sein, dass eine Datei, die relativ zu Beginn des Bootvorgangs benötigt wird, sich im hinteren Teil der Liste befindet. Wird der Einleseprozess mit einer höheren Priorität ausgeführt, muss der Bootvorgang warten.

Der Ansatz der Parallelisierung von Startskripten durch eine SysVinit-Alternative lässt sich gut mit dem Vorzeitigen Einlesen kombinieren. Er darf daher als Mittel den Bootvorgang zu optimieren nicht fehlen. Sind alle Dateien von der Festplatte gelesen, lässt

sich Startzeit einsparen, indem die CPU optimal ausgelastet wird.

Eine bessere Zugriffsstrategie auf die Festplatte kann den Bootvorgang beschleunigen, löst aber nicht das Problem, dass Dateien auf der Festplatte verstreut liegen. Mit einer physikalischen Umsortierung von Dateien auf dem Datenträger könnte sowohl Spurwechselzeit als auch die wechselbedingte Rotationsverzögerung eingespart werden. Zu dem, dass sich somit Kopfbewegungen verhindern lassen, kann der Lesevorgang parallel zum Startvorgang ausgeführt werden, ohne diesen zu behindern.

2.3 Das Ext4-Dateisystem

Ext4 steht für *Fourth Extended Filesystem* [8, 20, 24]. Zum Erfolgskonzept des Dateisystems zählt unter anderem seine Abwärtskompatibilität. Sie geht erst verloren, wenn inkompatible Erweiterungen (engl. features), siehe Abschnitt 2.3.1, aktiviert werden. Diese Arbeit basiert auf einigen dieser Erneuerungen. Eine Abwärtskompatibilität zum Vorgänger Ext3 besteht daher nicht mehr.

2.3.1 Das Layout

Die kleinste logische Einheit im *Extended Filesystem* ist ein Block. Die Blockgröße wird beim Anlegen des Dateisystems bestimmt. Standardmäßig wird eine Größe von 4096 Bytes (4 KB) verwendet.

Der Superblock

Der erste Block wird auch als Superblock bezeichnet. In ihm gespeichert sind alle relevanten Informationen über den Typ des Dateisystems. Dazu zählen unter anderem die Identifikationsnummer des Dateisystems, die Blockgrößen, sowie die Anzahl der Blöcke, aus denen das Dateisystem besteht. Der Superblock besitzt eine feste Größe von 1024 Bytes. Um ihn vor Verlust und Zerstörung zu schützen, sind auf dem Datenträger zahlreiche Kopien gleichmäßig verteilt angelegt. Im Notfall kann er mit Hilfe der Kopien wieder hergestellt werden.

Die Blockgruppen

Alle Blöcke in dem Dateisystem sind in sog. Blockgruppen unterteilt. Zu Beginn jeder Gruppe werden Metainformationen hinterlegt. Dazu zählt unter anderem eine Block-Bitmap, eine I-Node-Bitmap, sowie eine I-Node-Tabelle.

Jede Bitmap belegt einen Block. Innerhalb der Block-Bitmap steht jedes Bit für den Belegungszustand eines Blocks in der Gruppe. Ist das Bit auf eins gesetzt, ist der Block

belegt, andernfalls frei. Die Anzahl der Bits in der Bitmap definiert, wie viele Blöcke zu einer Blockgruppe zusammen gefasst sind. Das bedeutet, dass eine Blockgruppe bei einer Blockgröße von 4096 Bytes aus 32768 Blöcken besteht.

$$4096\frac{Bytes}{Block} * 8\frac{Bits}{Byte} = 32768\frac{Bits}{Block} \tag{3}$$

Der I-Node

Ein I-Node (Index-Knoten) repräsentiert ein Dateisystemobjekt. Jeder I-Node besitzt eine eindeutige I-Node Nummer, kurz *ino*. Zu den Dateisystemobjekten zählen unter anderem reguläre Dateien, Soft-Links und Verzeichnisse. Jedes dieser Objekte wird im Dateisystem als einfacher I-Node verwaltet. Nur der innerhalb der Datenstruktur gespeicherte Typ verrät, um welche Art von Objekt es sich handelt.

Unter Ext4 besitzt die I-Node-Datenstruktur eine Größe von 256 Bytes. Sie enthält unter anderem Informationen über die Dateigröße, die Zugriffsrechte, sowie eine Liste aus Zeigern, die auf Blöcke verweisen, in denen der Dateiinhalt hinterlegt ist. Gespeichert sind die I-Nodes in der I-Node-Tabelle. Anhand der I-Node-Nummer *ino* lässt sich die Blockgruppe, sowie die Position innerhalb der I-Node-Tabelle einfach berechnen.

$$Blockgruppe = \frac{ino}{inodes\_per\_group}$$

$$Position = ino \ \ \% \ \ indoes\_per\_group$$

Die Anzahl der I-Nodes pro Gruppe wird beim Anlegen des Dateisystems festgelegt und kann nachträglich nicht geändert werden. Die Zahl sollte wohl überlegt sein, denn sie bestimmt, wieviele Objekte maximal in dem Dateisystem angelegt werden können. Ist die Zahl zu groß gewählt, wird unnötig Speicherplatz für eine zu große I-Node-Tabelle vergeudet. Andernfalls, beispielsweise bei vielen kleinen Dateien, kann die Festplattenkapazität nicht ausgenutzt werden. Als Standardeinstellung umfasst eine I-Node-Tabelle 16384 Einträge. Die I-Node-Bitmap ist demnach nur halb ausgelastet. Daraus ergibt sich eine Tabellengröße von 512 Blöcken, die von Beginn an reserviert sind.

$$inode\_table\_size = inodes\_per\_group * \frac{256}{blocksize} = 512$$

An welcher physikalischen Position sich sowohl die Bitmaps als auch die I-Node-Tabelle befinden, kann der Group-Descriptor-Tabelle entnommen werden. Gespeichert ist die Tabelle im selben Block wie auch der Superblock. Die Group-Descriptor-Tabelle beginnt somit direkt hinter den bereits belegten 1024 Bytes und verwendet den bislang noch ungenutzten Speicher.

Features

Features sind Erweiterungen bzw. Änderungen am Dateisystemlayout. Der Dateisystemtreiber, der sich im Betriebssystemkern befindet, interpretiert die Daten auf dem Datenträger. Problematisch wird es, wenn der Kerneltreiber eines der aktivierten Features nicht kennt. Im *Extended Filesystem* werden daher die Features in drei Gruppen gegliedert.

- Kompatible Features erlauben Lese- und Schreibzugriff auf den Datenträger, ohne diesen zu zerstören, auch wenn der Dateisystemtreiber die Features nicht kennen sollte. Auch mit älteren Kernelversionen kann das Dateisystem ohne Einschränkungen genutzt werden.

- Sofern ein Feature nur lesend kompatibel ist und der Dateisystemtreiber dieses nicht kennt, kann auf das Dateisystem nur lesend und nicht mehr schreibend zugegriffen werden. Ein Schreibzugriff könnte Schäden am Dateisystem verursachen. Das Dateisystem kann dennoch lesend ins System eingehängt werden.

- Unter inkompatible Features fallen solche, die – sofern sie für den Dateisystemtreiber unbekannt sind – das Auslesen der Informationen auf dem Datenträger verhindern. Auf den Datenträger kann mit älteren Kernelversionen nicht mehr zugegriffen werden.

Die Liste der eingeschalteten Features befindet sich im Superblock des Dateisystems. Die folgenden Features *Extents* und *flexible Blockgruppen* gehören zu den inkompatiblen Features.

Extents
: Die wohl wichtigste Erneuerung bei Ext4 ist die Einführung von *Extents*. Ein Extent ist eine Datenstruktur, die zur einfachen Blockadressierung von Bereichen verwendet wird. Das daraus resultierende Extent-Mapping ersetzt damit die bisher verwendete indirekte Blockadressierung, wodurch es zu den inkompatiblen Features gehört. Während in der indirekten Blockadressierung jeder Block einzeln verwaltet wird, erlauben *Extents* eine Vielzahl von aufeinanderfolgenden Blöcken zu betrachten. Damit minimiert sich der Verwaltungsoverhead stark. Ebenso lässt sich durch *Extents* die maximale Dateigröße erhöhen.

  Die Betrachtung von Blockbereichen minimiert die Komplexität der Blockverwaltung. Das Feature *Extents* ist damit die Grundlage vieler weiterer Neuentwicklungen im Dateisystemtreiber, die insbesondere darauf aus sind, die Fragmentierung von Dateien zu verhindern.

Flexible Blockgruppen
: Unter flexiblen Blockgruppen versteht man das Vereinen von mehreren Blockgruppen zu einer großen virtuellen Gruppe. Jede Blockgruppe wird mit diesem Feature einer virtuellen Gruppe zugeordnet. Die Anzahl der Blockgruppen in so einer vir-

tuellen Gruppe wird in einer exponentiellen Zahl der Basis zwei angegeben.

Flexibel in dieser virtuellen Gruppe ist die Platzierung der Metainformationen. Während bisher die Metainformationen immer zu Beginn einer jeden Blockgruppe gespeichert wurden, werden sie in die erste Blockgruppe der virtuellen Gruppe verschoben. Die stark anwachsende Festplattenkapazität hat zur Folge, dass die Anzahl an Blockgruppen drastisch ansteigt. Durch die Einteilung in Blockgruppen wird der Speicherplatz auf dem Dateisystem durch die Metainformationen in regelmäßigen Abständen unterbrochen. Der größte Bereich von aufeinanderfolgenden freien Blöcken ist daher der Rest einer jeden Blockgruppe. Mit einer Blockgröße von 4 Kilobyte ist jede Gruppe 128 Megabyte groß. Sehr große Dateien müssen daher auf mehrere Gruppen verteilt, sprich in Fragmenten, abgespeichert werden.

Das Zusammenfassen von Metainformationen erhöht die maximale Bereichsgröße freier Blöcke. Als Standardeinstellung wird der Wert 4 verwendet. Demnach werden 16 Blockgruppen jeweils zu einer flexiblen Gruppe zusammengefasst. Die Größe einer Flexiblen Gruppe ist demnach

$$2^4 * 128MB = 2048MB$$

Weiter lässt sich durch das Feature der flexiblen Blockgruppen die Performanz und die Skalierbarkeit des Dateisystems steigern [3].

2.3.2 Der I-Node-Allocator

Um ein Objekt im Dateisystem anzulegen muss ein I-Node alloziert werden. Diese Aufgabe übernimmt der I-Node-Allocator [17]. Er sucht nach einem ungenutzten Eintrag in den I-Node-Tabellen und markiert diesen anschließend in der I-Node-Bitmap als belegt.

Wie schon erwähnt, existiert für jede Blockgruppe eine eigene I-Node-Tabelle. Dahinter versteckt sich die Idee, Metainformationen der Datei, nämlich den I-Node, sowie ihren Dateiinhalt möglichst nahe beieinander zu speichern. Der Dateiinhalt soll also möglichst in der selben Blockgruppe platziert werden, in der sich auch der I-Node befindet. In welcher Blockgruppe der I-Node angelegt wird, ist von weitreichender Bedeutung. Bereits hier wird entschieden, welche Dateien in unmittelbarer Nähe zueinander gespeichert werden.

Für das Anlegen von I-Nodes unterscheidet der I-Node-Allocator zwischen Dateien und Verzeichnissen.

Anlegen von Dateien

Das Dateisystem besitzt keine Kenntnis darüber, welche Dateien logisch zusammengehören. Unter der Annahme, dass Dateien, die sich im selben Verzeichnis befinden,

gemeinsam angefordert werden, alloziert der I-Node-Allocator die I-Nodes in derselben Blockgruppe.

Bevor eine Datei angelegt wird, ermittelt der I-Node-Allocator die I-Node-Nummer des Zielverzeichnisses. Befindet sich in der Blockgruppe kein freier I-Node, wird in den umliegenden Gruppen danach gesucht.

Ein weiteres Kriterium für die Wahl der Blockgruppe ist, dass sie ausreichend über freie Blöcke zur Speicherung des Dateiinhalts verfügt. Ist das nicht der Fall, wird ebenso nach einer neuen Gruppe gesucht.

Anlegen eines Verzeichnisses

Das Anlegen von Verzeichnissen ist in Bezug auf die Blockbelegung im Dateisystem von großer Bedeutung. Für das Anlegen des I-Nodes eines Verzeichnisses gelten daher strengere Kriterien. Schließlich werden in den Verzeichnissen später möglicherweise zahlreiche Dateien angelegt. Es ist daher sinnvoll, zusätzliche I-Nodes und freie Blöcke für zukünftige Dateien einzuplanen.

Selbst Dateien, die sich über eine tiefere Verzeichnisstruktur erstrecken, stehen ebenso in einer, wenn auch etwas schwächeren Beziehung zueinander. Das bedeutet für Verzeichnisse, die im selben Unterverzeichnis liegen ebenso, dass sie nicht zu weit voneinander entfernt abgelegt werden sollten. Es gilt daher, Anfangsverzeichnisse im Dateisystem mit großem Abstand zueinander anzulegen.

Die Streuung der Anfangsverzeichnisse ist durch den sog. Orlov-Algorithmus realisiert [4]. Er vergleicht alle Blockgruppen miteinander und sucht aus diesen die am besten geeignete heraus. Geprüft wird die Anzahl freier I-Nodes und die Anzahl freier Blöcke in der Gruppe. Ist diese vollständig frei, kann die Suche abgebrochen werden. Welche Gruppe als erstes betrachtet wird, entscheidet eine Zufallszahl. Damit erreicht das Dateisystem ohne viel Aufwand eine statistisch gesehen gute Streuung von Verzeichnissen.

Das Erstellen eines Unterverzeichnisses ähnelt dem Anlegen einer Datei. Sobald die Gruppe des Zielverzeichnisses einen bestimmten Füllgrad überschreitet, sucht der Algorithmus umliegend nach einer alternativen Gruppe. Damit bleiben auch Unterverzeichnisse in relativer Nähe.

2.3.3 Der Block-Allocator

Das Herzstück eines jeden Dateisystems ist der Block-Allocator [3, 18]. Er gehört zur Verwaltungssoftware des Dateisystems und befindet sich im Betriebssystemkern. Seine Aufgabe ist es, I-Nodes bei Bedarf Blöcke zuzuweisen. Der Block-Allocator bestimmt demnach die Blockbelegung im Dateisystem. Um die Fragmentierung von Dateien möglichst zu verhindern, verfügt der Block-Allocator über bestimmte Techniken.

Wird eine Datei neu angelegt, sucht der Block-Allocator nach einem möglichst großen freien Blockbereich. Existiert die Datei bereits, werden, wenn möglich, physikalisch hinter dem Dateiende die noch freien Blöcke verwendet. Ist der Block direkt hinter dem Dateiende belegt, muss nach einem neuen Bereich gesucht werden. Die Datei fragmentiert.

Zu den wichtigsten Erneuerungen im Ext4-Dateisystem zählt der sog. Multi-Block-Allocator. Anders als die bisherige Implementierung erlaubt der Multi-Block-Allocator das Allozieren mehrerer Blöcke gebündelt in einer einzelnen Anfrage. Die zusätzliche Information über die gesuchte Blocklänge ermöglicht dem Block-Allocator eine bessere und gezieltere Entscheidungsfindung für einen Blockbereich. Der größte Teil der Fragmentbildungen kann jedoch verhindert werden, da innerhalb einer Anfrage direkt nach einem ausreichend großen Bereich gesucht werden kann. Problematisch bleiben weiterhin langsam wachsende Dateien.

Pre-Allocation

Eine weitere Technik unter Ext4, die mögliche Fragmentierung verhindern soll, sind sog. *Pre-Allocation-Spaces*. Bei sehr langsam wachsenden Dateien ist das Kombinieren der Blockanfragen nicht möglich. Um dennoch den Dateien auf der Festplatte genügend Raum zum Wachsen zu lassen, wird jede Blockanfrage bis zu einem bestimmten Limit, auf eine Zahl einer Zweier-Potenz aufgerundet. Der entstandene Überschuss wird in den I-Node eigenen Pre-Allocation-Space eingetragen. Bei einem I-Node-Pre-Allocation-Space handelt es sich um einen zugesicherten Blockvorrat für einen bestimmten I-Node aus noch freien ungenutzten Blöcken. Dieser Vorrat ist temporär und wird nicht auf die Festplatte geschrieben. Sobald das Dateisystem ausgehängt wird, gehen alle Vorallozierungen verloren.

Für Dateisysteme sind besonders temporäre Dateien ein Problem. Diese Dateien sind oft klein und werden nicht selten nach kurzer Zeit wieder gelöscht. Die durch das Löschen wieder frei gewordenen sehr kleinen Bereiche sind nur schwer wiederverwendbar, da sie nur wiederum sehr kleinen Dateien sinnvoll zuzuordnen sind. Der Block-Allocator des Dateisystems verwaltet daher Allozierungen kleiner Dateien gesondert. Solange die Datei eine Größe von 16 Blöcken nicht überschreitet, gilt sie unter Ext4 als kleine Datei.

Für kleine Dateien gibt es einen weiteren Pre-Allocation-Space, die sog. Locality-Group. Jede Blockanfrage wird aus dem Blockvorrat der Locality-Group erfüllt, solange die Datei nach der Anfrage das Limit kleiner Dateien nicht überschreitet. Das Dateisystem speichert somit alle kleinen Dateien physikalisch beieinanderliegend.

Um auf Synchronisation zwischen CPUs verzichten zu können, existiert für jede CPU eine eigene Locality-Group.

Delayed Allocation

Mit Hilfe der Delayed Allocation werden Blockanfragen verzögert. Das Schreiben in Dateien wird mit dem Systemaufruf write(2) durchgeführt. Reichen die zugewiesenen Blöcke einer Datei nicht mehr aus, wird die Anfrage eines weiteren Blocks zeitlich nach hinten verschoben. Erst wenn die Datei geschlossen wird und fest steht, dass alle Daten geschrieben sind, wird eine große Anfrage durchgeführt. Nur durch die Delayed-Allocation kann der Multi-Block-Allocator seine Wirkung voll erzielen.

Buddy Cache

Um geeignete Blockbereiche schnell ausfindig zu machen, bedient sich der Multi-Block-Allocator des sog. Buddy Caches. Dieser enthält Informationen über die Anzahl freier Blöcke innerhalb der Gruppe, sowie eine Auflistung freier aufeinanderfolgender Blockbereiche. Diese sind in vordefinierte Größen einer Zweier-Potenz von 0 bis 13 gegliedert. Anders als bei den Block-Bitmaps berücksichtigt der Buddy Cache alle Vorallozierungen. Wo sich der Bereich innerhalb der Blockgruppe befindet, ist dem Buddy Cache nicht zu entnehmen.

2.3.4 Möglichkeit zur Online-Defragmentierung

Eine weitere Erneuerung im Dateisystem Ext4 ist der Befehl EXT4_IOC_MOVE_EXT. Mit ihm lassen sich Blockbelegungen zwischen zwei Dateien während des Betriebs, sprich online, vertauschen. Das Dateisystem selbst übernimmt die Synchronisation parallel ausgeführter Dateizugriffe. Die Gefahr eines Datenverlustes besteht daher nicht.

Während des Vorgangs ist jeder weitere Zugriff auf die I-Node-Strukturen blockiert. Um Blöcke zwischen Dateien zu vertauschen, hängt der Befehl Seiten im Page-Cache des Betriebssystems um. Durch das Markieren der Seite als *dirty* wird veranlasst die Änderung auf die Festplatte zu übernehmen. Wie bei jeder Änderung übernimmt der Kernel-Thread *jbd2* die Aufgabe die Daten sicher auf die Festplatte zu schreiben.

Die Möglichkeit, Blockbelegungen zwischen Dateien zu tauschen, macht sich das Werkzeug *e4defrag* zu Nutze. Das Programm ist Teil der Sammlung der *e2fsprogs* und stellt die Implementierung der unter Ext4 bekannten Online-Defragmentierung dar. Mit dem Programm lassen sich Dateien, die in mehreren Fragmenten (Teilstücken) auf der Festplatte verteilt liegen, wieder zu einer fortlaufenden Sequenz verbinden. Jede Datei wird bei *e4defrag* einzeln behandelt. Um eine Datei zu defragmentieren wird eine gleichgroße Spenderdatei erstellt. Weist die Spenderdatei eine weniger starke Fragmentierung auf als die der Originaldatei, so führt *e4defrag* den Befehl EXT4_IOC_MOVE_EXT aus, um alle Blöcke zwischen den Dateien zu vertauschen. Anschließend wird die Spenderdatei wieder gelöscht.

Das Anlegen der Spenderdatei übernimmt der Block-Allocator. Er alleine bestimmt wo und wie stark fragmentiert die Datei angelegt wird. *e4defrag* kann keinen Einfluss darauf nehmen. Die Schnittstelle sieht dies nicht vor. Der Grund liegt in der Annahme, dass eine geeignete Platzierung der Datei am besten vom Block-Allocator des Dateisystems bewerkstelligt werden kann.

2.4 Linux Audit

Der Linux-Kernel bietet seit der Version 2.6.11 eine Schnittstelle zum Überwachen und Protokollieren von Systemaufrufen an [27]. Das Analysieren von Systemaufrufen erlaubt einen tiefen Einblick in das Systemgeschehen. Linux Audit wird daher unter anderem von Host Intrusion Detection Systeme (HIDS) genutzt [12]. Über Systemaufrufe wird beispielsweise Speicher alloziert und auf Dateien zugegriffen. Bei der Entwicklung von Linux Audit wurde besonders auf einen geringen Ressourcenverbrauch geachtet. Der durch die Überwachung entstehende Mehraufwand ist somit so gering wie möglich.

Linux Audit eignet sich besonders gut um Dateizugriffe zu ermitteln. Im Kapitel 3 wird Linux Audit genutzt, um eine Liste von Dateien zu erstellen. Bei den Dateien handelt es sich um jene, die zum Startvorgang eines Programms oder des Betriebssystems benötigt werden. Auch die Art des Zugriffs – lesend oder schreibend – lässt sich mit Linux Audit ermitteln.

Der Kommunikationskanal von Linux Audit basiert auf einem Netlink-Socket. Netlink [16] ist eine Methode der asynchronen Inter-Prozess-Kommunikation (IPC) zwischen Kernel und Prozessen. Die zu übertragenden Daten werden in einer Warteschlange gepuffert. An dem Socket lauscht in der Regel der Audit Dämon. Seine Aufgabe ist es, so schnell wie möglich die empfangenen Daten auf die Festplatte zu speichern. Alle gelesenen Daten werden aus dem Puffer gelöscht, um Platz für neue Ereignisse zu schaffen. Über einen Event-Multiplexer können die empfangenen Daten direkt an andere Programme weitergeleitet werden. Hierfür stellt das Programm ein Plugin zur Verfügung, welches vom Event-Multiplexer geladen wird. Zur Laufzeit können Programme über den Code im Plugin Audit Ereignisse empfangen.

2.4.1 Audit-Filter

Ein Systemaufruf durchläuft vor und nach der Ausführung verschiedene Audit Filter. Was in diesen gefiltert wird, bestimmen Audit Regeln, die zu den Filtern hinzugefügt werden. Greift eine Regel, wird das Ereignis protokolliert. Abbildung 3 zeigt einen Überblick über die verschiedenen Filter.

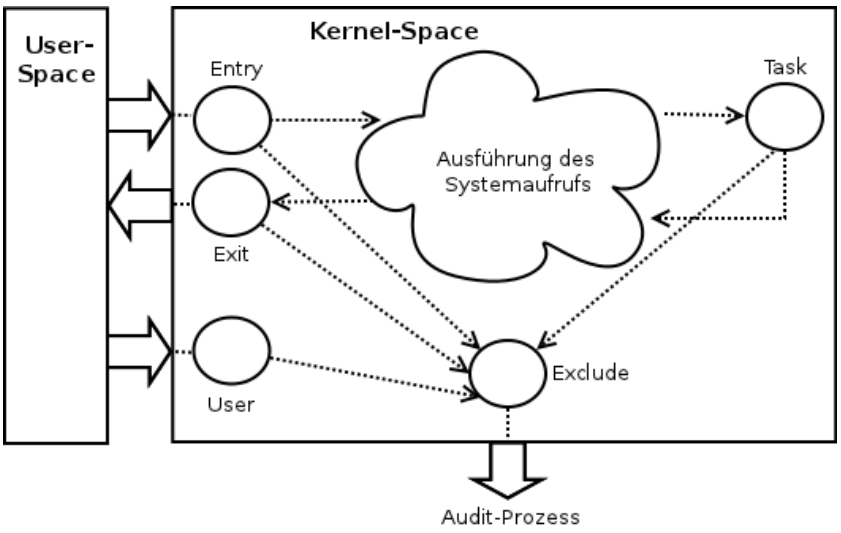


Abbildung 3: Linux Audit Filter im Überblick

Entry Der Filter Entry wird vor Ausführung des Systemaufrufs durchlaufen. Der Filter gilt als veraltet und wird in den nächsten Kernelversionen verschwinden. Für diese Diplomarbeit wurde daher auf die Verwendung dieses Filters verzichtet.

Exit Der Filter Exit wird nach der Ausführung des Systemaufrufs durchlaufen. Wird ein Ereignis generiert, enthält dieses zusätzlich den Rückgabewert und die Information, ob der Befehl erfolgreich ausgeführt werden konnte.

Task Regeln im Filter Task dürfen sich nur auf die Prozesserstellung beziehen. Der Filter wird nur nach der Ausführung von fork(2), vfork(2) oder clone(2) eingesehen. Für die Erstellung der Regel dürfen daher nur Schlüsselwörter verwendet werden, die zur Erstellung eines Prozesses gesetzt sind.

User Linux Audit kann auch Ereignisse aus dem User-Space entgegen nehmen. Somit werden neben Systemaufrufen auch Ereignisse von Anwendungsprogrammen, wie z.B. login(1), protokollierbar. Ein fehlgeschlagener Anmeldeversuch durchläuft den Filter *User*.

Exclude Jedes Ereignis, bei dem eine Filter-Regel zutraf, durchläuft zusätzlich den Filter *Exclude*. Mit ihm ergibt sich die Möglichkeit Ereignisse nachträglich wieder auszuschließen, bevor sie tatsächlich über den Socket protokolliert werden.

2.4.2 Audit-Nachrichten

Sobald die Regeln gesetzt sind, können am Socket aufgetretene Ereignisse empfangen werden. Jedes Ereignis ist aufgeteilt in mehrere Nachrichten, die jeweils die selbe Audit Identifikationsnummer besitzen. Die jeweiligen Nachrichten enthalten unterschiedliche

Informationen. Für diese Arbeit interessant sind die Nachrichten vom Typ SYSCALL, CWD, PATH, EOE, AUDIT_GET und AUDIT_CONFIG_CHANGE .

SYSCALL enthält Aufrufparameter, den Rückgabewert sowie Prozessinformationen über den jeweiligen Systemaufruf. Häufig sind Übergabeparameter Zeiger auf eine Speicheradresse in Speicherumgebung des jeweiligen Prozesses. Was an der Adresse hinterlegt ist, kann aufgrund der virtuellen Speicherumgebung von anderen Prozessen nicht eingesehen werden. Dazu gehört unter anderem jeder übergebene Dateipfad. Aus diesem Grund bietet Linux Audit neben *SYSCALL* zusätzliche Nachrichtentypen wie *PATH* und *CWD* an.

PATH enthält Informationen über die jeweilige Datei. Dazu zählt unter anderem der Pfad, die I-Node-Nummer, sowie die Geräte-Nummer dev_t des Geräts, auf dem sich das Dateisystem befindet.

CWD enthält das aktuelle Verzeichnis des jeweiligen Prozesses. Mit Hilfe des aktuellen Verzeichnisses kann ein relativer Pfad aus der Nachricht *PATH* zu einem absoluten Pfad überführt werden.

EOE signalisiert das Ende der Nachrichtenkette (**E**nd **O**f multi-record-**E**vent)

AUDIT_CONFIG_CHANGE benachrichtigt über Änderung der Linux Audit Konfiguration.

AUDIT_GET enthält Verbindungsinformationen. Im Gegensatz zu den oben genannten Nachrichten-Typen wird diese Nachricht manuell angefordert. Sie kann daher auch von anderen Prozessen aus empfangen werden, die derzeit nicht als aktiver Audit Prozess eingetragen sind.

Ein Ereignis könnte beispielsweise eine Nachrichtenkette wie in Abbildung 4 auslösen:

Für diese Arbeit relevant sind folgende Felder:

arch ist eine codierte Zahl, die die verwendete Architektur des Programms beschreibt.

syscall ist die Nummer des Systemaufrufs. In der *x86*-Architektur steht die 5 für open(2).

success teilt mit, ob der Befehl ausgeführt wurde oder nicht.

a0,a1,a2,a3 enthalten die Übergabeparameter.

pid ist die ID des Prozesses, der das Ereignis auslöst.

```
type=SYSCALL msg=audit(1298544344.486:302): arch=40000003 syscall=5
   success=yes exit=5 a0=808383c a1=203c1 a2=1a4 a3=808383c items=2
   ppid=2401 pid=2406 auid=0 uid=0 gid=0 euid=0 suid=0 fsuid=0
   egid=0 sgid=0 fsgid=0 tty=tty2 ses=1 comm="gdm-binary"
   exe="/usr/sbin/gdm-binary" key=(null)
type=CWD msg=audit(1298544344.486:302):  cwd="/root"
type=PATH msg=audit(1298544344.486:302): item=0 name="/var/run/"
   inode=918537 dev=08:02 mode=040755 ouid=0 ogid=0 rdev=00:00
type=PATH msg=audit(1298544344.486:302): item=1
   name="/var/run/gdm.pid" inode=917630 dev=08:02 mode=0100644
   ouid=0 ogid=0 rdev=00:00
type=EOE msg=audit(1298544344.486:302):
```

Abbildung 4: Empfangene Audit Nachrichten ausgelöst durch einen open(2)-Systemaufruf. Der Nachrichtentyp wurde den empfangenen Zeichenketten zur einfacheren Veranschaulichung hinzugefügt. Die Syntax entspricht der Log-Dateien des Audit Dämons.

comm ist die Bezeichnung des Prozesses (maximal 16 Zeichen).

exe ist der Pfad zur derzeit ausgeführten Datei.

cwd gibt das aktuelle Verzeichnis an.

name ist der Dateipfad, auf den der Systemaufruf zugreift.

inode ist die I-Node-Nummer des Dateisystemobjektes *name*

dev ist die Geräte-Nummer des Dateisystem auf der sich *name* befindet. Der Doppelpunkt trennt die MAJOR:MINOR Nummern.

3 e4rat-collect – Sammeln von Dateizugriffen

Dieses Kapitel beschreibt, wie durch das Werkzeug *e4rat-collect* Dateizugriffe über einen gewissen Zeitraum überwacht werden und daraus eine Liste von relevanten Dateien erstellt wird.

Mit Hilfe von Linux Audit lassen sich aus dem User-Space Systemaufrufe wie z.B. open(2) überwachen, siehe Kapitel 2.4. Die Art des Zugriffs wird ermittelt und anhand der empfangen Informationen eine Dateiliste generiert. Die Audit-Nachrichten werden gesammelt und einem Ereignis zugeordnet. Nur wenn die Datei über den gesamten Überwachungszeitraum rein im Lesezugriff steht, wird die Datei in die Liste aufgenommen. Mehrfache Einträge gleicher Dateien werden verhindert.

Nicht jeder Dateizugriff hat eine Festplattenaktivität zur Folge. Einmal von der Festplatte gelesen, werden Dateien im Page-Cache des Betriebssystems zwischengespeichert. Jeder weitere Zugriff erfolgt dann direkt aus dem Cache des Betriebssystems. Es treten somit keine Verzögerungen durch die Festplatte auf. Wenn möglich, sollten gecachte Dateien daher nicht in die Liste aufgenommen werden. Ihr Zugriff ist nicht weiter optimierbar. Die Liste ist die Grundlage für die Umsortierung durch das Werkzeug *e4rat-realloc*. Zugriffszeiten lassen sich nur reduzieren, wenn Dateien physikalisch dem Programm zugeordnet werden, das auf die Datei das erste Mal zugreift. Werden Dateien in mehrere Listen zugeordnet, werden diese im Fall einer Umsortierung der vorherigen physikalischen Zuordnung wieder entrissen. Übrig bleibt eine Lücke, die die Lesegeschwindigkeit wieder verschlechtert.

Welche Dateien sich bereits im Page-Cache befinden, kann nicht ermittelt werden. Seit Linux Kernelversion 2.4 sind der Buffer- und der Page-Cache vereint[19]. Ein Einblick von außen ist aus Gründen der Sicherheit nicht vorgesehen. Um Dateien nicht wieder in die Liste aufzunehmen, können mit einem Übergabeparameter bereits bestehende Listen angegeben werden. All diese Dateien werden dann ignoriert. Ebenso nicht von der Festplatte übertragen werden alle bereits geöffneten Dateien. Sie werden durch den Kommandozeilenbefehl lsof(1) ermittelt und gleichermaßen ignoriert.

Weiter lässt sich mit *e4rat-collect* die Überwachung auf ein oder mehrere Programme bzw. Skripte beschränken. Neben der Überwachung von Dateizugriffen wird weiter die Prozesserstellung überwacht. Jeder Systemaufruf ist einem Prozess zugeordnet. Spaltet sich ein Prozess in Mutter- und Kindprozess, wird die Prozess-ID des Kindprozesses in die Liste der zu überwachenden Prozesse aufgenommen.

Die Linux Audit Schnittstelle ist sehr mächtig. Die Kommunikation mit der Schnittstelle wurde durch Hinzuziehen der Bibliothek libaudit realisiert.

3.1 Relevante Systemaufrufe

Mit dem Systemaufruf read(2) wird von einer Datei gelesen, während mit write(2) in eine Datei geschrieben wird. Es ist naheliegend genau diese zwei Befehl zu überwachen, um zu ermitteln welche Dateien im Zugriff stehen. Es wäre damit sogar möglich herauszufinden, welche Teile einer Datei gelesen werden und welche nicht. Allerdings treten diese Befehle sehr häufig auf, wodurch es zu einer Flut an Informationen käme. Die Verarbeitung wäre mit einem hohen CPU-Aufwand verbunden.

Vor jedem read(2) oder write(2) wird jede Datei zuerst geöffnet. Die Befehle greifen über einen sog. Dateideskriptor auf die Datei zu. Der Deskriptor wird entweder für einen Lese- und/oder einen Schreibzugriff erstellt. Um Ressourcen zu sparen, werden nicht read(2) und write(2) überwacht, sondern lediglich das Öffnen der Datei. Mit welchen Rechten der Deskriptor angelegt wird, gibt darüber Aufschluss in welche Art, nämlich lesend oder schreibend, auf die Datei zugegriffen wird. Dass nach dem Öffnen ein read(2)- oder write(2)-Befehl folgt, kann zwar nicht mehr festgestellt, aber angenommen werden.

Im Folgenden werden die von *e4rat-collect* überwachten Systemaufrufe vorgestellt, um eine vollständige Dateiliste zu erhalten.

int execve(const char *file, char **const argv, char **const envp);
: Führt eine Datei aus. Der Parameter *file* zeigt auf den auszuführenden Dateipfad. Bei execve(2) handelt es sich um einen reinen Lesezugriff.

int open(const char *path, int flags);
: Öffnet einen Dateideskriptor zum Dateipfad *path*. Durch den Parameter *flags* wird die Art des Zugriffs festgelegt. Folgende Bits sind für die Erstellung der Dateiliste relevant:

0x001	O_WRONLY	Datei wird für einen reinen Schreibzugriff geöffnet.
0x002	O_RDWR	Datei wird für Lese- und Schreibzugriff geöffnet.
0x200	O_CREAT	Legt eine neue Datei an, falls diese nicht bereits existiert. Datei wird zum Schreibzugriff geöffnet.

int openat(int dirfd, const char *path, int flags);
: Öffnet eine Datei relativ zu einem bereits geöffneten Dateideskriptor eines Verzeichnisses. Der Parameter *path* zeigt auf einen relativen Pfad. Der Parameter *flags* unterscheidet sich nicht vom open(2) Befehl.

int creat(const char *path, mode_t mode);
: Erstellt eine leere Datei mit dem Pfad *path*. Wenn die Datei bereits existiert, wird ihr Inhalt gelöscht. Bei Erfolg gibt sie einen Dateideskriptor zurück mit dem schreibend zugegriffen werden kann. Es handelt sich hierbei um einen rein schreibenden Zugriff.

int truncate(const char *path, off_t length);
: Verkleinert eine Datei auf eine Länge von *length* in Bytes. Der Dateizugriff wird als schreibend eingestuft.

int truncate64(const char *path, off_t length);
: Ist eine Erweiterung von truncate(2). Mit dem Systemaufruf können sehr große Dateien behandelt werden. Dieser existiert nur in 32-Bit Systemen.

int mknod(const char *path, mode_t mode, dev_t dev);
: Erstellt eine Spezialdatei ähnlich wie creat(2). Der Dateityp wird im Parameter *mode* mit angegeben. Da der Befehl auch die Erstellung regulärer Dateien erlaubt, wird er mit überwacht.

Für das Überwachen ein oder mehrerer ausgewählter Programme müssen Prozesse Programmen zugeordnet werden. Dies erfolgt anfangs über die Prozessbezeichnung. Zu jedem Programmstart ist die Prozessbezeichnung gleich dem Dateinamen. Während der Ausführung kann jeder Prozess seine Bezeichnung ändern. Besteht das Programm aus mehreren Prozessen, reicht die Bezeichnung nicht mehr aus. Die Prozesse können, auch wenn sie demselben Programm angehören, unterschiedliche Bezeichnungen annehmen. Um dennoch Prozesse einem Programm zuzuordnen, wird zusätzlich die Erstellung eines jeden Prozesses überwacht.

pid_t fork();
: Erzeugt einen Kindprozess. Der Rückgabewert im Mutterprozess ist die Prozess-ID des Kindprozesses, der im Kindprozess hingegen ist eine 0.

pid_t vfork();
: Gabelt ähnlich wie unter fork(2) einen Prozess. Im Unterschied zu fork(2) besitzen Eltern- und Kindprozess die gleiche Speicherumgebung. Der Befehl führt zu undefiniertem Verhalten, wenn nicht direkt im Anschluss exit(2) oder execve(2) ausgeführt wird.

pid_t clone(int (*fn)(void *), void *child_stack, int flags, void *arg, ...); Der
: Befehl clone(2) ist eine Weiterentwicklung von fork(2) unter Linux. Der Befehl lässt spezifischere Angaben über den Speicheraufbau des Kindprozesses zu.

Als Darstellung der Systemaufrufe mit seinen Übergabeparametern wurde die Notation für eine Funktion in der Programmiersprache C verwendet. Bei einem Systemaufruf handelt es sich jedoch nicht um eine Funktion. Während Funktionen Parameter auf dem Stack in umgekehrter Reihenfolge ablegen und anschließend durch einen koordinierten Sprung den Programmcode der Funktion fortsetzen, sind Systemaufrufe Software-Interrupts. Ein Software-Interrupt wird unter *x86* mit dem CPU-Befehl int 0x80 ausgelöst. Von da ab übernimmt die Ausführung der Betriebssystemkern. Die Übergabeparameter werden nicht auf dem Stack sondern in Registern übergeben. Das erste Register enthält die Nummer des aufzurufenden Systemaufrufs. Die Nummer ist eine Positionsan-

gabe in der Interrupt-Vektor-Tabelle zum Programmcode der Interrupt-Service-Routine im Kern des Betriebssystems.

Unter Linux unterscheiden sich die Nummern der Systemaufrufe zwischen 32-Bit und 64-Bit Architekturen. Insbesondere die Architekturen *x86_64* und *PPC64* erhalten hierbei eine Sonderrolle. Beide sind 64-Bit Architekturen, die trotz der Umstellung auf den größeren Adressraum kompatibel zu den 32-Bit CPU-Befehlen bleiben [15]. Ermöglicht wird das durch einen 32-Bit-Kompatibilitätsmodus, der auch das Linux Betriebssystem unterstützt. Ein 64-Bit Betriebssystemkern kann somit auch 32-Bit Anwendungen ausführen. Jede Anwendung, ob 32- oder 64-Bit, verwendet je nach ihrer Architektur eine unterschiedliche Nummerierung der Systemaufrufe. Bei der Analyse eines Systemaufrufs gilt daher, auf die Architektur des Anwendungsprogramms zu achten.

3.2 Kommunikation mit Linux Audit

3.2.1 Verbindung herstellen

Bevor mit Linux Audit kommuniziert werden kann, wird eine Verbindung, wie in Abbildung 5, hergestellt. Steht die Netlink Socketverbindung durch den Befehl audit_open(3), wird die Verbindung weiter konfiguriert. Um Nachrichten zu empfangen, muss die empfangsberechtigte Prozess-ID gesetzt werden. Nur der Prozess mit der Prozess ID kann Audit Nachrichten über den Socket erhalten. In Zeile 11 wird Linux Audit im Kern des Betriebssystems eingeschaltet.

Die Auswertung der Nachrichten ist immer mit einem gewissen CPU-Aufwand verbunden. Um den Startvorgang möglichst wenig zu beeinträchtigen, kann das Empfangen und Auswerten der Nachrichten durch Erhöhung der Anzahl maximal gepufferter Nachrichten von 32 auf 256 bei Bedarf nach hinten verschoben werden.

3.2.2 Audit-Regeln

Die zu überwachenden Systemaufrufe werden als eine gemeinsame Audit-Regel formuliert und in den Audit-Filter *EXIT* eingetragen. Die Regeln müssen dabei in Abhängigkeit der vorliegenden Prozessor-Architektur formuliert sein. Die jeweilige Architektur ist eine codierte hexadezimale Zahl, die die Maschine bzw. die CPU durch gesetzte Bits beschreibt[3].

Die Architekturbezeichnung lässt sich zur Laufzeit durch den Befehl uname(3) ermitteln. Eine gesonderte Rolle spielen die 64-Bit-Architekturen *x86_64* und *ppc64*. Beide

[3]Maschinenzahl enthält Bits unter anderem für die Unterscheidung von 32/64 Bit, Little/Big Endian, ...

```
1     audit_fd = audit_open();
          if (-1 == audit_fd)
3             throw std::logic_error("Cannot open audit socket");
  ....
5
      if(0 > audit_set_pid(audit_fd, getpid(), WAIT_YES))
7         error("Cannot set pid to audit");

9     //set 1 to enable auditing
      //set 2 to enable auditing and lock the configuration
11    if(0 > audit_set_enabled(audit_fd, 1))
          error("Cannot enable audit");
13
      if(0 > audit_set_backlog_limit(audit_fd, 256))
15        audit_request_status(audit_fd);
```

Abbildung 5: Verbindung mit Linux Audit herstellen.

sind abwärtskompatibel und erlauben weiterhin die Ausführung 32-Bit-Befehle. Bei beiden Architekturen werden daher, wie in Abbildung 6, zwei Regelsätze erstellt. Auch wenn *e4rat-collect* als 32-Bit-Anwendung compiliert ist, kann es sowohl 32-Bit- als auch 64-Bit-Systemaufrufe überwachen. Mit der Funktion audit_name_to_machine(3) wird die Architekturbezeichnung aus der Struktur *utsname* von uname(3) in eine gültige Maschinenzahl überführt. Die Nummer der Systemaufrufe für die Regelsätze der jeweiligen Architektur wird mit audit_name_to_syscall(3) abgefragt.

Die übertragene Datenmenge von Linux Audit kann verringert werden, indem nur erfolgreich ausgeführte Systemaufrufe übermittelt werden. Fehlgeschlagene Zugriffe sind für die Erstellung der Dateiliste von keinerlei Interesse und führen nur zu einem höheren CPU-Aufwand. Jede Regel wird weiter mit der Bedingung *success=1* verknüpft.

3.2.3 Empfangen und Parsen von Audit-Nachrichten

e4rat-collect empfängt Daten über eine Socketverbindung. Durch passives Warten mit der Funktion select(3) wird in einer Endlosschleife geprüft, ob Daten am Socket anliegen. Nach Ablauf des Timeouts oder beim Empfang eines Unix-Signals, wird das Warten vorzeitig unterbrochen. Nur wenn eine Nachricht erfolgreich gelesen werden konnte, wird die Schleife verlassen.

Die empfangenen Daten werden mit der Funktion audit_get_reply(3) in die Struktur *audit_reply* (Abbildung 7) eingelesen.

Die Variable *type* enthält den Nachrichtentyp. Der Nachrichteninhalt, siehe Abbildung

```
1   struct utsname uts;
    if(-1 == uname(&uts))
3       throw std::logic_error(std::string("Cannot receive machine
            hardware name: ") + strerror(errno));

5   if(0 == strcmp(uts.machine, "x86_64"))
    {
7       activateRules(MACH_86_64);
        activateRules(MACH_X86);
9   }
    else if(0 == strcmp(uts.machine, "ppc64"))
11  {
        activateRules(MACH_PPC64);
13      activateRules(MACH_PPC);
    }
15  else
    {
17      int machine = audit_name_to_machine(uts.machine);
        if(-1 == machine)
19          throw std::logic_error(std::string("Unknown machine
                hardware name ")+ uts.machine);
        activateRules(machine);
21  }
```

Abbildung 6: Regelsätze in Abhängigkeit der Architektur

4, befindet sich in *msg.data*. Die Zeichenkette ist nicht mit '\0' terminiert. Die Anzahl gültiger Zeichen in der Zeichenkette gibt die Variable *len* an.

Aus der vergleichsweise großen Datenstruktur sind nur ein paar Informationen für die Generierung der Dateiliste relevant. Um Ressourcen zu sparen wird daher jede empfangene Nachricht sofort eingelesen. Die Werte aus relevanten Feldern werden in eine Struktur eingetragen, die jeweils ein Ereignis beschreibt.

Die Nachrichten über einen bestimmten Systemaufruf werden immer in der gleichen Reihenfolge übermittelt. Während der Übertragung von Nachrichten eines Ereignisses können sich jedoch Nachrichten anderer Ereignisse dazwischen schieben. Durch die Nachrichten-ID lassen sich die empfangenen Nachrichten dem jeweiligen Ereignis zuordnen. Die Datenstrukturen der derzeit empfangenen Ereignisse werden in einer Liste verwaltet. Ist die Nachricht vom Typ Syscall, wird ein neuer Eintrag in der Liste erstellt. Erst wenn das Ereignis vollständig empfangen wurde, wird es weiter verarbeitet und aus der Liste entfernt.

Die Bibliothek *libauparse* stellt Funktionen zur Verfügung, die das Lesen der Zeichenkette vereinfachen. Bevor die Funktionen verwendet werden können, muss der Zeichenkette

```
struct audit_reply {
    int                         type;
    int                         len;
    struct nlmsghdr             *nlh;
    struct audit_message {
          struct nlmsghdr       nlh;
          char                  data[MAX_AUDIT_MESSAGE_LENGTH];
    } msg;

    union {
        struct audit_status     *status;
        struct audit_rule_data  *ruledata;
        struct audit_login      *login;
        const  char             *message;
        struct nlmsgerr         *error;
        struct audit_sig_info   *signal_info;
        struct daemon_conf      *conf;
    };
};
```

Abbildung 7: Datenstruktur *audit_reply* zur Übertragung von Audit-Nachrichten

„type=MSG_TYP msg=“ vorangestellt werden.

Die Dateipfade aus *name* und *cwd* sind umhüllt von Hochkomma. Enthält der Dateipfad jedoch Leerzeichen, wird jedes Zeichen aus dem Pfad in seinen hexadezimalen Ascii-Wert codiert.

3.2.4 Kollision mit auditd

Am Socket kann immer nur die empfangsberechtigte Prozess-ID Audit-Nachrichten empfangen, siehe Kapitel 3.2.1. Verbinden sich weitere Programme, wie beispielsweise der Audit-Dämon *auditd* mit dem Audit-Socket, überschreiben diese den eingetragenen Empfänger-Prozess. Die Verbindung wird somit unterbrochen.

Bevor die Änderung der neuen Empfänger-Prozess-ID übernommen wird, generiert Linux-Audit die Nachricht vom Typ AUDIT_CONFIG_CHANGE, siehe Abbildung 8. Die neue Prozess-ID ist im Feld *audit_pid*, die vorherige unter *old* zu entnehmen.

```
type=AUDIT_CONFIG_CHANGE msg=audit(1303836969.573:3112):
   audit_pid=5080 old=4933 auid=4294967295 ses=4294967295 res=1
```

Abbildung 8: Audit Dämon übernimmt die Verbindung der Netlink-Socket-Verbindung

Linux-Audit weckt für diese Nachricht ein letztes mal den vorherigen Prozess. Dieser kann die Nachricht jedoch nur empfangen, wenn sich keine weiteren ungelesenen Nachrichten im Puffer befinden, da immer nur die älteste Nachricht gelesen wird. In diesem Fall empfängt *e4rat-collect* diese Nachricht nicht und kann auf die Änderung nicht reagieren.

Werden über einen Zeitraum von 10 Sekunden keine Daten auf dem Audit-Socket übertragen besteht Grund zur Annahme, dass ein anderer Prozess die Verbindung übernommen hat. Nach Ablauf der Zeit wird der Status über den Netlink-Socket mit dem Befehl audit_request_status(3) eingefordert. Der Kernel sendet daraufhin eine Nachricht vom Typ AUDIT_GET. In der Struktur *audit_reply->status.pid*, siehe Abbildung 9, steht die derzeit aktive Prozess-ID. Entspricht dieser Wert in *pid* nicht dem der eigenen, wurde die Verbindung übernommen.

```
1  struct audit_status {
       __u32   mask;           /* Bit mask for valid entries       */
3      __u32   enabled;        /* 1 = enabled, 0 = disabled        */
       __u32   failure;        /* Failure-to-log action            */
5      __u32   pid;            /* pid of auditd process            */
       __u32   rate_limit;     /* messages rate limit (per second) */
7      __u32   backlog_limit;  /* waiting messages limit           */
       __u32   lost;           /* messages lost                    */
9      __u32   backlog;        /* messages waiting in queue        */
   };
```

Abbildung 9: Audit Socket Status Informationen

Ist die Verbindung übernommen worden, tritt *e4rat-collect* bewusst in den Hintergrund, um systemrelevanteren Programmen wie dem Audit Dämon den Vortritt zu lassen. *e4rat-collect* beendet sich mit einer Fehlermeldung.

3.3 Die Datenstruktur der Dateiliste

Die Bedingungen an die Dateiliste sind einerseits, dass jede Datei nur einmal vorkommen darf und andererseits, dass die Aufrufreihenfolge beibehalten werden muss.

Die Zugriffsart entscheidet darüber, ob die Datei in die Liste aufgenommen wird. Da die aufgelisteten Dateien später physikalisch aneinandergereiht werden, dürfen nur solche in die Liste, auf die lediglich lesend zugegriffen wird. Alle anderen sind für eine Umsortierung nicht geeignet, da diese – sofern sie wachsen – nach der Umsortierung fragmentieren. Es wird eine Möglichkeit benötigt, Dateien dauerhaft auszuschließen.

Häufig werden Dateien erst lesend und später schreibend geöffnet. Es muss also möglich sein, eine Datei nachträglich wieder aus der Liste zu entfernen. Die Generierung

der Liste ist ein Zusatzaufwand. Die Bemühungen sollte dahin gehen, den zusätzlichen Ressourcenverbrauch (CPU und Arbeitsspeicher) so gering wie möglich zu halten. Die Suche nach bereits vorhandenen Einträgen sollte möglichst effizient verlaufen.

Ein Dateipfad ist nicht eindeutig. Durch sog. Hard-Links können mehrere Dateipfade auf ein und dieselbe Datei verweisen. Hinter jedem dieser Pfade ist dieselbe I-Node Nummer hinterlegt. Nur die I-Node Nummer identifiziert innerhalb eines Dateisystems eindeutig Dateien und andere Objekte. Um eine Datei auf einem Computer eindeutig zu identifizieren, wird ein eindeutiger Schlüssel aus der Gerätedateinummer (dev_t), auf dem sich das Dateisystem befindet und aus der Nummer des I-Nodes (ino_t) generiert.

Für jede Datei, ausgewertet durch die Linux Audit Nachrichten, wird ein Datei-Objekt mit folgenden Attributen angelegt.

```
struct FilePtrPrivate {
     fs::path   path;
     dev_t      dev;
     ino_t      ino;
     bool       valid;
};
```

Die Variable *valid* wird anfangs immer mit „wahr“ initialisiert. Erst wenn auf die Datei schreibend zugegriffen wird, ändert sich ihr Wert und beschreibt das Objekt für die Liste als ungültig.

Alle Datei-Objekte werden in eine einfach verkettete Liste *std::list* eingetragen. Damit bewahrt die Dateiliste ihre Aufrufreihenfolge. Um eine Datei im Nachhinein als ungültig zu markieren, muss das Datei-Objekt zuvor ausfindig gemacht werden. Die Suche in der Liste erfolgt linear, da die Objekte unsortiert vorliegen. Um ein Objekt zu finden, wird im Schnitt die halbe Liste durchlaufen. Im schlimmsten Fall existiert das gesuchte Objekt nicht. Hierfür wird die ganze Liste durchsucht, wobei n Vergleichsoperationen notwendig sind. Für eine schnelle Suche ist die verkettete Liste ungeeignet.

Effizienter ist es, eine Baumstruktur zu verwenden. Der Datentyp std::map aus der C++ STL ist ein Rot-Schwarz-Baum mit einer Suchzeit von $\mathcal{O}(n \log n)$. Als eindeutiger Schlüssel wird die Kombination aus {dev;ino} verwendet. Wird kein Eintrag in der Baumstruktur gefunden, so existiert auch kein Objekt in der verketteten Liste. Die Suchzeit wird mit dem Hinzuziehen der Baumstruktur stark reduziert.

Existiert ein Eintrag, so muss kein neues Objekt hinzugefügt werden. Handelt es sich jedoch um einen Schreibzugriff, wird das Objekt, falls dies noch nicht geschehen ist, als ungültig markiert. Auf der Suche nach dem Objekt muss, wie vorher, die verkettete Liste durchlaufen werden. Abhilfe schaffen hier Smart-Pointer. In der Liste, als auch in der Baumstruktur, werden nur Zeiger auf das Objekt referenziert. Damit kann über beide

Strukturen auf das Objekt zugegriffen werden. Die Struktur erlaubt beides, eine schnelle Suche sowie die Erhaltung der Aufrufreihenfolge.

Die vorgestellte Struktur verbirgt sich hinter der Klasse *FilePtr*, bei der es sich um eine Ableitung von *boost::shared_ptr* handelt, ein Smart-Pointer, der auf ein *FilePtrPrivate*-Objekt referenziert. Der Referenzzähler gibt an, wie viele *FilePtr*-Objekte auf dasselbe *FilePtrPrivate*-Instanz mit dem Schlüssel {dev;ino} zeigen. Der Konstruktor der Klasse *FilePtr* sucht in der Baumstruktur nach einer bereits existierenden Referenz. Sofern ein Eintrag mit dem gleichen Schlüssel existiert, übernimmt *FilePtr* die Referenz und erhöht den Referenzzähler. Wird kein Eintrag gefunden, wird eine neue *FilePtrPrivate*-Instanz angelegt und in den Baum eingetragen.

Bei den Zeigern, die in die Baumstruktur eingetragen sind, handelt es sich um sog. Weak-Pointer. Sie haben nur eine schwache Bindung und verändern den Referenzzähler nicht. Im Destruktor der Klasse *FilePtr* wird der Referenzzähler wieder um eins dekrementiert. Steht der Referenzzähler danach auf null, wird der Eintrag in der Baumstruktur entfernt und das referenzierte *FilePtrPrivate*-Objekt gelöscht. Referenziert nur ein FilePtr auf das Datei-Objekt, so steht der Referenzzähler auf 1. Mit der Methode unique() kann geprüft werden ob mehr als nur ein *FilePtr* auf die Datei zeigt. Ist der Rückgabewert von unique() „wahr“, wird das neu erstellte Datei-Objekt an die Liste angehängt.

3.4 Beenden des Sammelvorgangs

Je nachdem, welchen Vorgang der Anwender optimieren möchte, muss zu einem unterschiedlichen Zeitpunkt der Sammelvorgang beendet werden.

Der Sammelvorgang von *e4rat-collect* kann auf folgende Arten beendet werden. Jede Variante ist mit Hilfe von Unix-Signalen realisiert.

- Manuell durch den Benutzer

  Das Kommandozeilen-Programm kill(1) dient dem Beenden von Programmen. Hierfür sendet es an den jeweiligen Prozess das Signal *SIGTERM*. Programme, die in einem Terminal ausgeführt werden, können darüber hinaus durch die Tastenkombination STRG-C beendet werden. Dies bewirkt das Senden des Signals SIGINT an den jeweiligen Prozess.

  Für *e4rat-collect* wurden die Standardroutinen, die üblicherweise einen raschen Programmabbruch bewirken, ersetzt. Anstatt das Programm zu beenden, wird der Sammelvorgang abgebrochen und anschließend die Dateiliste ausgegeben. Anstelle des Befehls kill(1) gibt es zusätzlich die Möglichkeit den Sammler zu beenden, indem *e4rat-collect* mit dem Pa-

rameter *--kill* aufgerufen wird. Die Prozess ID wird aus einer Pid-Datei gelesen. Anschließend wird an den jeweiligen Prozess das Signal SIGINT geschickt.

- Automatisch nach Ablauf eines Timeouts

  Eine weitere Möglichkeit den Sammelvorgang zu stoppen besteht durch die Verwendung eines Timeouts. Wenn *e4rat-collect* als Init-Prozess gestartet wurde, beendet es sich automatisch nach 120 Sekunden. Länger sollte ein Bootvorgang nicht andauern.

  Implementiert ist der Timeout als Wecksignal mit Hilfe der Funktion alarm(3). Nach Ablauf der Zeit sendet das Betriebssystem zum Sammelprozess das Signal SIGALRM. Ebenso wie *SIGTERM* und *SIGINT* wird das Programm beim Empfang von SIGALRM durch einen Signal-Handler kontrolliert beendet.

- Der zu überwachende Prozess beendet sich

  Wird ein Programm mit Hilfe des Parameters --execute von *e4rat-collect* ausgeführt, beendet sich der Sammelvorgang, sobald sich das angegebene Programm beendet. Das Programm wird als Mutterprozess ausgeführt, *e4rat-collect* lebt als Kindprozess weiter. Der Mutterprozess bleibt weiterhin über das Signal SIGCHLD mit seinen Kindprozessen in Kontakt. Sobald sich etwas ändert, wird das Signal an alle seine Kindprozesse geschickt. Darunter fällt z.B. die Beendigung des Mutterprozesses. Dieses Ereignis kann von allen anderen Ereignissen gefiltert werden, wenn ein alternatives Todessignal eingestellt wird. Mit folgendem Befehl wird das neue Todessignal gesetzt.

  ```
  prctl(PR_SET_PDEATHSIG, SIGINT);
  ```

  Die Entscheidung für das Signal SIGINT liegt nahe, da auf dieses ohnehin schon reagiert wird.

Damit sich das Programm kontrolliert beenden lässt, sind an zahlreichen Stellen im Programmcode sog. Interruption-Points verteilt. Nur an diesen vordefinierten Punkten darf das Programm beendet werden. Durchläuft das Programm einen solchen Punkt, wird geprüft ob ein Signal empfangen wurde. Ist dies nicht der Fall, wird das Programm fortgesetzt.

4 e4rat-realloc – physikalische Block-Umsortierung

Ist die Dateiliste der relevanten Dateien mit *e4rat-collect* erstellt, können mit *e4rat-realloc* die Blöcke der Dateien physikalisch auf dem Datenträger verschoben werden. Die Umsortierung hängt dabei stark von der Größe und Anzahl freier Blockbereiche ab. Befindet sich auf dem Datenträger ein ausreichend großer freier Blockbereich, werden alle Dateien entsprechend der Liste sequentiell angeordnet.

Die Blockverschiebung wird durch den Befehl EXT4_IOC_MOVE_EXT, siehe Kapitel 2.3.4, durchgeführt. Hierfür erstellt *e4rat-realloc* für jede zu verschiebende Datei aus der Liste eine gleich große Spenderdatei. Die Spenderdatei dient als Blockquelle. Die sequentielle Verschiebung ist nur dann möglich, wenn die Spenderdateien auf dem Datenträger der Reihe nach angeordnet werden können. In Kapitel 4.2 werden drei unterschiedliche Methoden vorgestellt, die eine sequentielle Anordnung der Spenderdateien ermöglichen. Weist die Anordnung der Spenderdateien weniger Fragmente auf als die der Originaldateien, wird die Blockverschiebung durchgeführt. Anschließend werden die Spenderdateien wieder gelöscht.

4.1 Vorbereitung

Die Dateiliste kann Dateien von mehreren Dateisystemen enthalten. Der erste Schritt ist daher, die Dateien nach der Gerätenummer aufzugliedern. Jedes Gerät gilt es einzeln zu behandeln.

Bevor die Blöcke der Dateien verschoben werden, wird auf mögliche Fehlerquellen im Voraus geprüft. Durch die Prüfung können Dateien im Voraus aussortiert und eine geeignete Fehlermeldung ausgegeben werden. Sollte eine Datei nicht verschoben werden können, bleiben die Spenderdatei und die allozierten Blöcke ungenutzt. Nachdem Löschen hinterlässt sie eine Lücke in der sequentiellen Neuanordnung. Dies kann sich negativ auf die Leseperformanz der Festplatte auswirken.

Der Befehl EXT4_IOC_MOVE_EXT existiert nur bei Ext4-Dateisystemen. Dateien, die nicht auf einem Ext4-Dateisystemen liegen, werden daher ignoriert. Die Überprüfung des Dateisystems wird in Kapitel 4.1.1 beschrieben. Weiter werden die Attribute einer jeden Datei aus der Liste, siehe Kapitel 4.1.2, überprüft.

4.1.1 Dateisystem prüfen

Die Identifikationsnummern der Dateisysteme im Superblock sind bei den Dateisystemen Ext2/3/4 identisch. Wie das Dateisystem im System eingehängt ist und welchen Treiber

es verwendet, steht in der Datei /proc/self/mounts [4].

Der Befehl EXT4_IOC_MOVE_EXT setzt voraus, dass das Feature *extents*, siehe Kapitel 2.3.1, aktiviert ist. Es wurde bereits in Ext3 eingeführt und ist deshalb kein eindeutiges Merkmal für Ext4. Unterstützt das Dateisystem *extents*, so ist das Bit EXT3_FEATURE_INCOMPAT_EXTENTS im Superblock in der Variable *s_feature_incompat* gesetzt.

4.1.2 Dateiattribute prüfen

Durch die Überprüfung der Dateiattribute können Dateien von vornherein aussortiert werden, bei denen der Befehl EXT4_IOC_MOVE_EXT fehlschlägt.

Die folgenden Dateiattribute werden geprüft:

- Es handelt sich um eine reguläre Datei.
- Die Datei belegt Speicherplatz. Ist das nicht der Fall, kann sie ignoriert werden.
- Der Anwender verfügt über die notwendigen Schreibrechte auf die Datei.
- Die Datei ist veränderbar. Hierfür darf das immutable Flag *FS_IMMUTABLE_FL* für die Datei nicht gesetzt sein.
- Die Blockindizierung wird mit *extents* verwaltet. Die Datei muss über das Flag *EXT4_EXTENTS_FL* verfügen.

Die Flags werden mit dem Befehl FS_IOC_GETFLAGS abgefragt. Ist das Flag *EXT4_EXTENTS_FL* nicht gesetzt, wird es nachträglich mit *FS_IOC_SETFLAGS* hinzugefügt, siehe Abbildung 10.

4.1.3 Ermitteln der Dateigröße

Um Spenderdateien anzulegen wird die Größe der Originaldatei ermittelt. Dieser Schritt wird bereits in der Vorbereitungsphase ausgeführt, da der Dateideskriptor ohnehin schon geöffnet ist.

Der Defragmentierbefehl EXT4_IOC_MOVE_EXT setzt voraus, dass die Spenderdatei dieselbe logische Größe wie die Originaldatei aufweist. Dabei interessant ist nicht die Größe in Bytes, sondern die Anzahl der Blöcke, die die Datei im Dateisystem belegt. Die Anzahl der Bytes lassen jedoch nicht auf die Anzahl belegter Blöcke schließen. Einer

[4] Ist das Proc-Dateisystem nicht im System eingehängt, wird alternativ die Datei /etc/mtab verwendet.

```
        if(0 > ioctl(fd, FS_IOC_GETFLAGS, &flags))
        {
            info("Cannot receive inode flags: %s: %s", path,
               strerror(errno));
            invalid_file_type++;
            goto cont;
        }

        if(!(flags & EXT4_EXTENTS_FL))
        {
            flags |= EXT4_EXTENTS_FL;
            if(0> ioctl(fd, FS_IOC_SETFLAGS, &flags))
            {
                info("Cannot convert file %s to be extent based:
                   %s", path, strerror(errno));
                not_extent_based++;
                goto cont;
            }
        }

        if(flags & FS_IMMUTABLE_FL)
        {
            info("%s is immutable.", path);
            not_writable++;
            goto cont;
        }
```

Abbildung 10: Prüfen der I-Node Flags der Originaldatei

Datei können mehr Blöcke zugewiesen werden als sie im Moment benötigt. Um die Datei vollständig zu verschieben, gilt es daher die Anzahl der Blöcke für die Blockverschiebung zu ermitteln.

Problematisch sind sog. dünnbesetzte Dateien (engl. sparse-files). Sie weisen logische Lücken in ihrer Blockallozierung auf. Ungenutzte Bereiche in der Datei werden auf dem Datenträger nicht alloziert, was den Speicherverbrauch der Datei reduziert. Die unallozierten Bereiche bleiben physikalisch zwischen den allozierten frei. Werden die Bereiche zu einem späteren Zeitpunkt gefüllt, können die bislang freigelassenen Blöcke genutzt werden, sofern sie nicht anderweitig von einer anderen Datei belegt sind.

Wie viele Blöcke für die Spenderdatei eingeplant werden müssen, ist abhängig von der Erstellungsart der Spenderdatei. In der Methode *Pre-Allocation*, siehe Kapitel 4.2.1, können belegte Bereiche zusammengeschoben werden. Die Größe der Spenderdatei ist in diesem Fall die Anzahl physikalisch belegter Blöcke. Bei den Methoden *Top-Level-Directory*, sowie *Locality-Group* können ungenutzte Bereiche hingegen nicht verbunden

werden. Die Größe der Spenderdateien entspricht demnach der logischen Größe der Originaldatei.

Die belegten Blockbereiche einer Datei können mit dem Befehl GET_FIEMAP abgefragt werden. Jeder Bereich besitzt eine Länge, den physikalischen Offset auf dem Datenträger und den logischen Offset innerhalb der Datei. Durch das Zusammenzählen der Offsets erhält man die tatsächlich belegten Blöcke (physikalische Größe) der Datei. Um die logische Größe zu ermitteln wird das Dateiende gesucht. Dieser Blockbereich ist mit dem Flag *EOF* gekennzeichnet. Die logische Größe ist demnach der logische Offset in der Datei addiert mit der Länge des Bereichs.

4.2 Erstellen der Spenderdateien

Das Erstellen der Spenderdateien ist der heikle Schritt bei der Blockverschiebung. Hier wird das Fundament für den Erfolg der Umsortierung gelegt. Nur wenn die Spenderdateien sequentiell in der gegeben Reihenfolge auf dem Datenträger angeordnet werden können, lässt sich die Festplattenzugriffszeit reduzieren.

Der Entwurf der Online-Defragmentierung sieht keine Möglichkeit vor auf die Erstellung der Spenderdatei Einfluss zu nehmen. Allein der Block-Allocator des Dateisystems ist für die Platzierung der Datei verantwortlich. Welche Blöcke dieser verwenden soll, ist von außen nicht auf direktem Wege bestimmbar. Durch die Anwendung bestimmter Techniken kann jedoch das Verhalten des Block-Allocators beeinflusst werden.

Im folgenden werden drei unterschiedliche Methoden vorgestellt, die möglichst eine sequentielle Anordnung hervorrufen sollen.

Während dieser Vorgänge wird die Priorität des Prozesses erhöht, um Fremdeinwirkungen durch andere Programme möglichst zu verhindern.

4.2.1 Methode „Pre-Allocation“

Die Methode *Pre-Allocation* beeinflusst das Verhalten des Block-Allocators über das Hinzufügen von Blöcken in den I-Node-Pre-Allocation-Space, siehe Kaptitel 2.3.3. Möglich macht dies ein Kernel Patch von Kazuya Mio, der die Befehle EXT4_IOC_CONTROL_PA und EXT4_IOC_GET_PA einführt [21]. Mit EXT4_IOC_CONTROL_PA lassen sich Blöcke vorallozieren und wieder freigeben. Die Liste an vorallozierten Blöcken lässt sich daraufhin mit dem Befehl EXT4_IOC_CONTROL_PA abfragen.

Werden Blöcke mit fallocate(2) alloziert, verwertet der Block-Allocator erst Blöcke aus dem Blockvorrat des I-Node-Pre-Allocation-Space. Durch gezielte Vorallozierungen kann

somit die Platzierung einer Datei auf dem Datenträger bestimmt werden. Jeder beliebige noch freie Block kann den Spenderdateien zugewiesen werden. Einer sequentiellen Anordnung von Dateien steht nichts mehr im Wege.

Um Blöcke den Spenderdateien zuzuordnen, muss zuvor ein ausreichend großer freier Blockbereich gefunden werden.

Datenstrukturen

Den Befehlen werden jeweils die folgenden Datenstrukturen übergeben.

EXT4_IOC_CONTROL_PA

Der Befehl EXT4_IOC_CONTROL_PA steuert den I-Node-Pre-Allocation-Space. Die Anweisung wird durch die zu übergebene Struktur ext4_prealloc_info, siehe Abbildung 11, formuliert.

```
struct ext4_prealloc_info {
    __u64 pi_pstart; /* physikalischer Offset auf dem Datenträger */
    __u32 pi_lstart; /* logischer offset in der Datei             */
    __u32 pi_len;    /* Länge des Blockbereichs                   */
    __u32 pi_free;   /* Ausgabe: Länge gefundener Bereich         */
    __u16 pi_flags;  /* Auszuführende Anweisung                   */
};
```

Abbildung 11: Datenstruktur zur Vorallozierung von Blöcken

Von besonderer Bedeutung ist die Variable *pi_flags*. In ihr kann eine der folgenden drei Anweisungen zur Steuerung der Vorallozierungen abgesetzt werden:

EXT4_MB_MANDATORY
: voralloziert Blöcke. Sind die Blöcke belegt, wird mit einem Fehler abgebrochen. Die Struktur enthält den nächsten freiliegenden Blockbereich.

EXT4_MB_ADVISORY
: voralloziert Blöcke. Sind diese belegt, wird automatisch nach der bestmöglichen Alternative gesucht.

EXT4_MB_DISCARD_PA
: löscht alle Vorallozierungen.

Die größtmögliche Zahl, die in *pi_len* angegeben werden darf, ist auf

die Anzahl der Blöcke pro Blockgruppe - 10 begrenzt. Diese Grenze ist willkürlich gesetzt und beachtet nicht das Feature der Flexiblen Blockgruppen, siehe Features im Kapitel 2.3.1.

EXT4_IOC_GET_PA

Mit dem Befehl EXT4_IOC_GET_PA können vorallozierte Blöcke abgefragt werden. Dem Systemaufruf wird die Struktur aus Abbildung 12 übergeben.

```
1 struct ext4_prealloc_list {
      /* Arraygröße von pl_space */
3     __u32 pl_count;
      /* Anzahl gülter Einträge in pl_space */
5     __u32 pl_mapped;
      /* Die Anzahl Bereiche über die der I-Node verfügt */
7     __u32 pl_entries;
      struct ext4_prealloc_info pl_space[0];
```

Abbildung 12: Datenstruktur um vorallozierte Blöcke abzufragen

Die Variable *pl_count* gibt die Größe des Arrays *pl_space* an. In ihm finden sich die vorallozierten Extents wieder. In wie weit das Array verwendet ist, ist in der Variable *pl_mapped* abzulesen.

Suche nach freien Blöcken

In der Methode *Pre-Allocation* wird die Aufgabe des Block-Allocators, nämlich nach Blöcken zu suchen, übernommen. In jeder Blockgruppe befindet sich eine Block-Bitmap, die den Belegungszustand jedes einzelnen Blocks angibt. Im Ext4 Dateisystem werden Daten durch den Kernel-Thread *jbd2* auf den Datenträger geschrieben. In regelmäßigen Abständen erwacht der Thread und überträgt alle Änderungen auf die Festplatte. Schreibvorgänge lassen sich somit bündeln und werden daher verzögert auf den Datenträger geschrieben. Auch nicht berücksichtigt in der Block-Bitmap werden Vorallozierungen. Vorallozierte Blöcke sind für andere I-Nodes nicht verwendbar. Ein vermeintlich freier Bereich in der Bitmap kann bereits reserviert worden sein. Das Fehlen der Vorallozierung, sowie das späte Schreiben auf die Festplatte sorgen dafür, dass die Blockbitmaps für die Bereichssuche ungeeignet sind.

Suche nach Blockbereichen kleiner als eine Blockgruppe

Das Ausfindigmachen von Blockbereichen, die kleiner als eine Blockgruppe sind, erfolgt durch die Suche des Block-Allocators selbst. Hierfür wird eine neue Datei mit der gesuchten Größe angelegt. Wo der Block-Allocator die Datei angelegt hat, wird als Suchergebnis festgehalten. Anschließend wird die Datei wieder gelöscht.

Die Suche wird effizienter, wenn keine Festplattenaktivität durch die Allozierung mehr entsteht. Deshalb werden Blöcke nur voralloziert. Dies geschieht durch den Befehl EXT4_IOC_CONTROL_PA mit der Anweisung EXT4_MB_ADVISORY. Die vorallozierten Blöcke werden anschließend mit EXT4_IOC_GET_PA abgefragt. Der größte belegte Bereich wird gemerkt und der Vorgang solange wiederholt, bis ausreichend viel Speicher gefunden ist. Anschließend wird die Datei gelöscht um den vorallozierten Speicher wieder frei zugeben. Der Algorithmus findet somit immer die größten noch freien Blockbereiche.

Suche über die Grenze von Blockgruppen hinaus

Eine Suche über die Grenze von Blockgruppen hinaus wird mit Hilfe des Buddy-Caches realisiert. Ist das Feature der flexiblen Blockgruppen aktiviert, sind die die Bitmaps sowie die I-Node-Tabelle mehrerer Blockgruppen in der ersten der virtuellen Gruppe zusammengefasst, was dazu führt, dass sich ein freier Bereich über alle Mitglieder der virtuellen Gruppe erstrecken kann. Die bisherige Implementierung des Block-Allocators ist derzeit nicht in der Lage mehrere Gruppen gleichzeitig zu betrachten.

Aus dem Proc-Dateisystem unter dem Pfad /proc/fs/ext4/<device>/mb_groups kann der Inhalt des Buddy-Caches ausgelesen werden.

```
#group: free  frags first [ 2^0   2^1   2^2   2^3   2^4   2^5   2^6   2^7   2^8   2^9   2^10  2^11  2^12  2^13  ]
#0    : 24543 1     8225  [ 1     1     1     1     1     0     1     1     1     1     1     1     1     2     ]
#1    : 32768 1     0     [ 0     0     0     0     0     0     0     0     0     0     0     0     0     4     ]
...
```

Der Algorithmus zählt leere Blockgruppen zusammen. Betrachtet werden Gruppen mit einer Fragmentzahl *frags = 1*. Ist diese Bedingung nicht erfüllt, kann kein Block der darauffolgenden Gruppe dem bisher gefundenen Bereich hinzugefügt werden. Unklar ist, wie die freien Bereiche in der Gruppe verteilt liegen.

Gilt *frags = 1*, beginnt der Bereich an der Position *first*. Die Länge dieses Bereichs ist gleich dem Wert von *free*. Erstreckt sich dieser Bereich bis ans Ende der Blockgruppe, wird auch die darauffolgende Gruppe näher untersucht. Dies gilt sofern die Position addiert mit der Länge des Bereichs der Gesamtblockzahl der Gruppe entspricht.

$$first + free = total\_blocks\_per\_group \tag{4}$$

Nur wenn die darauffolgende Gruppe völlig ungenutzt ist, können die freien Blöcke dem Bereich hinzugezählt werden. Diese Bedingung setzt impliziert voraus, dass für die Gruppe automatisch gilt *first = 0*. Der erste Block der folgenden Gruppe knüpft somit an den Bereich an.

$$free = total\_blocks\_per\_group \tag{5}$$

Ohne viel Rechenaufwand erlaubt der Buddy Cache schnell große freie Blockbereiche im Dateisystem ausfindig zu machen. Eine genaue Analyse der einzelnen Blockgruppen

ist nicht möglich. Dennoch sind die gefundenen Bereiche größer als die, die der Block-Allocator finden kann.

Blockzuweisung

Die gefundenen freien Blockbereiche werden den Spenderdateien mit Hilfe des Befehls EXT4_IOC_CONTROL_PA zugeteilt. Die Spenderdateien lassen sich somit auf der Festplatte in den gefundenen freien Bereich der Reihe nach anordnen.

Im Fall der dünnbesetzen Dateien können die allozierten Bereiche der Datei zusammengeschoben werden, indem bei den Spenderdateien die logischen Lücken der Datei ebenso nicht voralloziert werden.

Zusammenfassung

Mit der Methode Pre-Allocation können Dateien optimal angeordnet werden. Über Vorallozierungen lässt sich der Block-Allocator des Dateisystems bei der Erstellung der Spenderdateien steuern. Mit dieser Methode können somit die besten Ergebnisse erzielt werden.

Der Nachteil ist, dass die notwendigen Befehle dem Kernel manuell über einen Patch hinzugefügt werden müssen. Vorallozierungen wurden bereits im September 2010 in [9] angekündigt und sollen fester Bestandteil des Kernels werden. Bis dahin gilt der Patch als instabil.

4.2.2 Methode „Top Level Directory“

Eine andere Möglichkeit Spenderdateien der Reihe nach auf dem Datenträger anzuordnen stellt die Methode *Top Level Directory* dar. Im Gegensatz zur Methode *Pre-Allocation* muss kein Kernel Patch eingespielt werden.

Die Methode macht sich zu Nutze, dass das Dateisystem Dateien im selben Wurzelverzeichnis auf dem Datenträger nahe beieinander ablegt. Auf dieses Verhalten, ausgelöst durch den Orlov-Algorithmus, wurde bereits im Kapitel 2.3.3 näher eingegangen.

Der Algorithmus legt zu Beginn ein neues Anfangsverzeichnis auf dem Datenträger an. In diesem Verzeichnis werden anschließend alle Spenderdateien entsprechend ihrer Aufrufreihenfolge angelegt.

Wo die Dateien schlussendlich physikalisch liegen ist ungewiss. Erste Tests zeigen hinsichtlich der Anordnung gute Ergebnisse, besonders dann, wenn der Orlov-Algorithmus für das neue Anfangsverzeichnis eine noch vollständig ungenutzte Blockgruppe finden konnte. In der Regel entstehen durch den Block-Allocator zwischen den Dateien kleine Lücken. Dieser rundet jede Blockanfrage auf die nächst höhere Zweier-Potenz auf. Der

Blocküberschuss wird nicht der Datei, sondern dem I-Node-Pre-Allocation-Space zugewiesen. Durch die Methode *Top Level Directory* kommt es kaum zu einer Vertauschung der Reihenfolge von den Spenderdateien.

Um zu verhindern, dass kleine Dateien in der Locality-Group, also außerhalb der Blockgruppe des Anfangsverzeichnisses, angelegt werden, wird zuvor das Limit für kleine Dateien auf 0 gesetzt. Dieser Wert lässt sich individuell für jedes Ext4 Dateisystem unter dem Pfad

$$/sys/fs/ext4/ < device > /mb\_stream\_req$$

einstellen. Jede Blockanfrage, die einschließlich der bisherigen Dateigröße, gleich oder größer als der angegebene Wert in *mb_stream_req* ist, wird außerhalb der Locality-Group alloziert. Sind alle Dateien angelegt, wird der Originalwert wieder hergestellt.

In der Zeit, in der die Spenderdateien angelegt werden, ist durch das neue Limit für kleine Dateien das gesamte Verhalten des Dateisystems abgeändert. Das Dateisystem verhält sich somit wie sein Vorgänger Ext3.

Die Abstände zwischen den Dateien können sich negativ auf die Lesegeschwindigkeit auswirken. Die Übertragungsrate der Festplatte wird somit nicht optimal ausgenutzt.

4.2.3 Methode „Locality Group“

Die Methode *Locality Group* verfolgt im Vergleich zu *Top Level Directory* genau den umgekehrten Ansatz. Anstatt zu verhindern, dass kleine Dateien in der Locality Group angelegt werden, wird das anlegen aller Spenderdateien in der Locality-Group provoziert. Die Größe des Grenzwertes für kleine Dateien wird so angehoben, dass selbst die größte anzulegende Spenderdatei kleiner ist.

Eine Blockanfrage, die mit Blöcken aus der Locality-Group erfüllt wird, wird nicht auf die nächst höhere Zweier-Potenz aufgerundet. Zwischen den Dateien entstehen keine Lücken durch freie Blöcke. Die Anordnung ist somit optimal.

Sind die Blöcke aus dem Blockvorrat der Locality-Group aufgebraucht, legt der Block-Allocator eine neue Locality-Group an, deren Größe über den Pfad

$$/sys/fs/ext4/ < device > /mb\_group\_prealloc$$

bestimmt werden kann. Als Wert setzt der Algorithmus die Blockanzahl aller Spenderdateien, jedoch nicht mehr als Blöcke pro Blockgruppe existieren. Grund ist, dass der Block-Allocator nicht in der Lage ist nach größeren Bereichen zu suchen. Der Wert in *mb_group_prealloc* wird nur zum Anlegen einer neuen Locality-Group berücksichtigt.

Jede CPU besitzt eine eigene Locality-Group. Um zu verhindern, dass der Prozess während des Anlegens der Spenderdateien die CPU wechselt und dadurch Blöcke aus einer

anderen Gruppe erhält, wird die Ausführung des Programms durch Setzen der Prozessoraffinität, siehe Abbildung 13, auf die CPU der Nummer 0 eingeschränkt.

```
        cpu_set_t sched_mask;
2
        CPU_ZERO(&sched_mask);
4       CPU_SET(0, &sched_mask);
        sched_setaffinity(gettid(), sizeof(cpu_set_t), &sched_mask))
```

Abbildung 13: Setzen der Prozessoraffinität

Die Methode *Locality-Group* legt Dateien ohne Abstand zueinander auf dem Datenträger an. Das zu erzielende Resultat ist somit besser als das der *Top Level Directory*. Während der Erstellung der Spenderdateien wird das Verhalten des Dateisystems massiv geändert.

Ein Schreibvorgang ausgeführt von einem Prozess, der auf der gleichen CPU arbeitet, kann eine Blockanfrage auslösen, die aus derselben Locality Group erfüllt wird. Zwischen den Spenderdateien werden Blöcke für andere fremde Dateien reserviert. Das kann sich negativ auf die Lesegeschwindigkeit auswirken. Problematischer ist es jedoch, wenn fremde Dateien, ausgelöst durch die Blockanfrage, fragmentieren.

Solange eine Datei noch Blöcke im eigenen I-Node-Pre-Allocation-Space reserviert, werden diese als Erstes aufgebraucht. Dennoch gilt es, wenn möglich, andere Schreibzugriffe während des Anlegens der Spenderdateien zu verhindern. Ist dies nicht möglich, sollte die Methode *Locality Group* nicht angewendet werden.

4.3 Blockverschiebung

Ist für jede Originaldatei eine exakt gleich große Spenderdatei erstellt, kommt der eigentliche Schritt – das Vertauschen der Blockbelegungen. Bevor Blöcke zwischen dem Paar – Orignal- und Spenderdatei – getauscht werden, ist die Anzahl der Fragmente zu zählen. Als Fragmente werden sowohl Fragmentierungen innerhalb einer Datei, als auch große Abstände zwischen Dateien gezählt. Weisen die Spenderdateien zusammen weniger Fragmente als die der Originaldateien auf, so ist eine Vertauschung sinnvoll. Die Blockbelegung jeder Datei kann mit dem Befehl GET_FIEMAP abgefragt werden. Jeder Blockbereich wird in eine Liste eingetragen und nach der physikalischen Position sortiert. Hängt ein Bereich nicht am vorherigen, wird die Anzahl der Fragmente um eins erhöht.

4.3.1 EXT4_IOC_MOVE_EXT

Mit dem Befehl EXT4_IOC_MOVE_EXT lassen sich zur Laufzeit Blöcke aus Original- und Spenderdatei tauschen. Dem Befehl wird die Struktur *move_extent* aus Abbildung 14 übergeben.

```
1 struct move_extent {
      __s32 reserved;          // ungenutzt
3     __u32 donor_fd;          // Descriptor der Spenderdateidonor
      __u64 orig_start;        // Logischer Offset in der Orignaldatei
5     __u64 donor_start;       // Logischer Offset in der Spenderdatei
      __u64 len;               // Die zu verschiebende Länge in Bytes
7     __u64 moved_len;         // Anzahl Bytes erfolgreich verschoben
  };
9
  int ioctl(__u32 orig_fd, int request = EXT4_IOC_MOVE_EXT, struct
     move_extent*);
```

Abbildung 14: Datenstruktur move_extent zur online Defragmentierung

Die Funktion gibt bei Erfolg 0, ansonsten -1 zurück. Damit der Befehl ausgeführt werden kann, müssen bestimmte Bedingungen erfüllt sein. Sowohl der Dateideskriptor der Spenderdatei (*donor_fd*), als auch der der Originaldatei (*orig_fd*) müssen mit Schreibrechten geöffnet sein. Beide Dateien müssen über das Extent-Flag verfügen. Die logischen Offsets innerhalb der Dateien *orig_start* und *donor_start* müssen identisch sein. Auch wenn die Struktur das Angeben von verschiedenen Offsets zulässt, wird es bislang nicht unterstützt. Sind die beiden Zahlen ungleich, wird mit einem Fehler abgebrochen. Die zu verschiebende Blocklänge *len* ist in Bytes angegeben und sollte immer ein Vielfaches der Blockgröße sein. Die Variable *moved_len* dient als Rückgabewert und gibt an wie viele Blöcke bislang verschoben wurden. Die Angabe ist ebenfalls in Bytes. Entspricht *moved_len* nicht dem Wert hinterlegt in *len*, muss der Befehl wiederholt aufgerufen werden. Zuvor sind Länge und Offsets anzupassen.

Der Befehl selbst greift nicht auf die Festplatte zu. Das Umhängen verläuft allein im Arbeitsspeicher. Nur die Seiten im Page Cache, in denen sich der Blockinhalt befindet, werden umgehängt. Der Kernelthread *jbd2* schreibt die Änderungen auf die Festplatte. Durch die Blockvertauschung entsteht daher keine Gefahr eines Datenverlustes. Sind die Blöcke beider Dateien getauscht, werden alle Vorallozierungen gelöscht.

Mit dem Befehl *posix_fadvice* kann dem Betriebssystem mitgeteilt werden, dass beide Dateien nicht mehr weiter im Arbeitsspeicher gehalten werden müssen.

Zu guter Letzt werden die Spenderdateien wieder gelöscht.

4.3.2 Derzeit ausgeführte Dateien

Wie bereits erwähnt setzt der Befehl EXT4_IOC_MOVE_EXT voraus, die Originaldatei schreibend zu öffnen. Bei Dateien, die sich derzeit in Ausführung befinden ist dies jedoch nicht möglich. Trotzt ausreichender Zugriffsrechte schlägt der Befehl open(2) fehl und die Variable *errno* wird auf *ETXTBSY* gesetzt. Bevor diese Dateien also verschoben werden können, muss das entsprechende Programm zuvor beendet werden.

Für die Optimierung des Bootvorgangs empfiehlt es sich daher, vor der Umsortierung möglichst viele Dienste zu beenden. Am einfachsten wechselt man hierfür in den Runlevel 1.

```
init 1
```

Dass die Originaldatei schreibend geöffnet werden muss war jedoch nicht immer so.

Die Prüfung der Zugriffsrechte wurde in der Kernelversion 2.6.32.1 eingeführt. Bis dahin war es Angreifern möglich anhand des Befehls EXT4_IOC_MOVE_EXT, als nicht privilegierter Nutzer, Dateien zu überschreiben [6]. Die Sicherheitslücke wurde geschlossen, indem vorausgesetzt wurde, dass der Dateideskriptor der Originaldatei lesend und schreibend geöffnet sein muss, der der Spenderdatei nur schreibend. Genau diese Änderung führt dazu, dass der Befehl bei derzeit ausgeführten Dateien nicht mehr aufgerufen werden kann.

Ein alternativer Lösungsansatz die Sicherheitslücke zu schließen ist, die Rechteprüfung anstatt auf dem Dateideskriptor besser direkt in der I-Node-Struktur durchzuführen. Die notwendigen Änderungen sind in Abbildung 15 als Kernel-Patch dargestellt. Es ist nicht mehr vorausgesetzt den Dateideskriptor der Originaldatei schreibend zu öffnen. Somit können Blöcke von derzeit ausgeführten Dateien verschoben werden.

Bis zur Fertigstellung dieser Ausarbeitung kam bislang noch kein Feedback bezüglich des Kernel-Patches aus der Open-Source-Community.

```
diff --git a/fs/ext4/ioctl.c b/fs/ext4/ioctl.c
index 808c554..4afb59c 100644
--- a/fs/ext4/ioctl.c
+++ b/fs/ext4/ioctl.c
@@ -230,8 +230,7 @@ setversion_out:
                struct file *donor_filp;
                int err;

-               if (!(filp->f_mode & FMODE_READ) ||
-                   !(filp->f_mode & FMODE_WRITE))
+               if (inode_permission(inode, MAY_READ | MAY_WRITE)
                        return -EBADF;

                if (copy_from_user(&me,
```

Abbildung 15: Vorschlag der abgeänderten Rechteprüfung

5 e4rat-preload – Vorladen von Dateien

Mit dem Programm *e4rat-preload* werden Dateien schnellst möglich von der Festplatte in den Arbeitsspeicher transferiert. Die Umsortierung bewirkt bereits eine Performanzsteigerung. Mit dieser ist jedoch das Optimierungspotential noch nicht vollständig ausgeschöpft. Dieses Kapitel beschreibt, wie der Lesevorgang auf der Festplatte noch effizienter wird. Das vorzeitige Einlesen durch *e4rat-preload* wirkt sich positiv auf die Cache Trefferrate aus und soll weitere Kopfbewegungen verhindern. Beides führt zu einem schnelleren Startvorgang der jeweiligen Programme.

5.1 Problemanalyse

Eine Datei besteht aus den Metainformationen im I-Node und aus dem Dateiinhalt. Beim Einlesen der Datei werden beide Teile von der Festplatte angefordert. Die Umsortierung mit *e4rat-realloc* ordnet die Inhalte der Dateien auf dem Datenträger möglichst sequentiell an. Die Informationen im I-Node hingegen bleiben weiterhin an der ursprünglichen Position in der I-Node-Tabelle der jeweiligen Blockgruppe. Die Umsortierung führt möglicherweise dazu, dass sich die Entfernung zwischen I-Node und dem dazugehörigen Dateiinhalt vergrößert. Durch die größere Entfernung legt der Magnetkopf eine längere Wegstrecke zurück, wodurch die Spurwechselzeit steigt.

Vorteilhaft ist, dass die Festplatte meist mehr Blöcke von der Spur liest, als angefordert sind. Diese werden in den internen Cache der Festplatte übertragen. Befinden sich weitere Dateien durch die Neuanordnung auf derselben Spur, werden diese mitgelesen. Obgleich sich die Entfernung vom I-Node zum Dateiinhalt vergrößert, reduziert sich die Anzahl der Kopfbewegungen.

Ein weiterer Grund für die Performanzsteigerung stellt das vorzeitige Einlesen von I-Nodes durch das Dateisystem dar. Die I-Node Strukturen weisen eine Größe von 256 Bytes auf. Wird ein I-Node von der Festplatte gelesen, liest das Ext4 Dateisystem umliegende Einträge mit in den Arbeitsspeicher ein. Standardmäßig sind es 32 Blöcke, in denen Platz für 512 I-Node-Strukturen ist[5].

$$32 Blöcke * \frac{4096 \frac{Bytes}{Block}}{256 \frac{Bytes}{I-Node}} = 512$$

Liegen weitere zu lesende I-Nodes in diesem Fenster, werden dadurch weitere Kopfbewegungen verhindert.

[5] Die Anzahl der Blöcke kann unter dem Pfad /sys/fs/ext4/<device>/inode_readahead_blks eingestellt werden.

5.2 Angewendete Zugriffsstrategie

Um Kopfbewegungen möglichst zu verhindern arbeitet *e4rat-preload* in den folgenden drei Schritten:

1. Das vorzeitige Einlesen der I-Nodes. Dieser Schritt verursacht viele Kopfbewegungen, da die I-Nodes auf der Festplatte weiterhin weit verteilt liegen. Um die Strecke der Kopfbewegungen zu verringern, wird die Liste vor dem Einlesen nach den I-Node-Nummern sortiert. Gelesen werden die I-Nodes mit dem Befehl stat(2)

2. Anschließend wird das zu optimierende Programm als Mutterprozess ausgeführt. *e4rat-preload* lebt weiter als Kindprozess.

3. Parallel zur Ausführung wird mit dem Lesen der Dateiinhalte begonnen. Der Vorgang des Einlesens und der der parallelen Programmausführung beeinflussen sich nicht gegenseitig. Durch die Umsortierung der Dateiinhalte durch *e4rat-realloc* sind die Dateien der Reihe nach entsprechend ihrer Aufrufreihenfolge angeordnet. Weitere Kopfbewegungen sind daher nicht mehr zu erwarten. Dateiinhalte werden mit dem Befehl readahead(2) angefordert.

Durch die parallele Ausführung des Programms zum Lesevorgang werden darüber hinaus die Festplattenruhephasen für die Übertragung genutzt. *e4rat-preload* erlangt im Laufe des Startvorgangs einen immer größeren Vorsprung, der vollständig die Verzögerungen durch die Festplatte eliminiert.

Um den Bootvorgang zu beschleunigen wird *e4rat-preload* als Init-Prozess in die Kernelparameter eingetragen.

init = /sbin/e4rat-preload

Wenn nicht durch die Konfigurationsdatei abgeändert, liest *e4rat-preload* die Dateiliste unter dem Pfad /var/lib/e4rat/startup.log ein und führt nach dem Einlesen der I-Nodes /sbin/init aus.

Für Anwendungsprogramme wird die Dateiliste als Parameter übergeben und das auszuführende Programm mit --execute <command> angegeben.

6 Besonderheiten bei der Implementierung

Dieses Kapitel beschreibt Besonderheiten bei der Implementierung, die jeweils zum Teil für alle drei Werkzeuge relevant sind.

6.1 Ausführung als Init-Prozess

e4rat-collect und *e4rat-preload* lassen sich als Init-Prozess vor allen anderen Programmen ausführen. Der gesamte Bootprozess kann somit erfasst und beschleunigt werden. Beide Programme setzen sich vor die Ausführung des eigentlichen Init-Systems.

Sie ersetzen jedoch das Init-System nicht, sondern führen es sobald wie möglich im Mutterprozess mit der Prozess-ID 1 aus. Für gewöhnlich wird das Init-System über den Pfad */sbin/init* ausgeführt. Über die Konfigurationsdatei, siehe Kapitel 6.3, kann ein alternativer Dateipfad *init* angegeben werden.

Unter der Unix Verzeichnisstruktur [23] können Verzeichnisse wie /usr oder /var auf andere Datenträger ausgelagert werden. Sie werden im Laufe des Bootvorgangs ins System eingehängt. Für die Ausführung als Init-Prozess muss daher auf die Verzeichnisse verzichtet werden. Ebenso ist daher in der Konfigurationsdatei der Pfad zur Dateiliste mit *startup_log_file* einstellbar.

Dateien, die im frühen Stadium des Bootvorgangs benötigt werden, sprich bevor es zum Einhängen weiterer Dateisysteme gekommen ist, befinden sich in den Verzeichnissen /sbin, /bin, /etc und /lib. Um die Werkzeuge früh ausführen zu können, dürfen diese nicht abhängig von Dateien aus anderen Verzeichnissen sein. Bestimmte Bibliotheken werden daher statisch, siehe Kapitel 6.2, gelinkt. Installiert werden die Binaries in das Verzeichnis /sbin.

6.2 Statisch und dynamisch gelinkte Bibliotheken

Der von *e4rat-collect*, *e4rat-realloc* und *e4rat-preload* gemeinsam genutzte Code ist in der Bibliothek *libe4rat-core* zusammengefasst. Die Bibliothek ist, wenn möglich, dynamisch und wird zur Laufzeit zu jeder der drei Binaries gelinkt.

Tabelle 1 zeigt weitere Bibliotheken, von denen e4rat abhängig ist.

Einige dieser Bibliotheken, siehe Tablle 1, liegen im Verzeichnis /usr/lib. Auf Systemen kann sich das /usr-Verzeichnis auf einer anderen Partition befinden, was dann wie bereits in Kapitel 6.1 erwähnt, die Ausführung als Init-Prozess verhindert. Zur gemeinsam

[6] Bibliothek wird automatisch zu Programmen und Bibliotheken gelinkt.

Bibliothek	Verzeichnis	Paket
libext2fs	/lib	Ext2 Filesystem Library
libaudit libauparse	/usr/lib	Linux Audit
libboost-system libboost-filesystem libboost-regex	/usr/lib	Boost
libgcc [6] libstdc++ [6]	/usr/lib	GNU C++ Compiler

Tabelle 1: Abhängigkeiten

genutzten Bibliothek *libe4rat-core* werden daher die im Verzeichnis /usr/lib liegenden Bibliotheken statisch gelinkt.

Bei einer statischen Bibliothek kopiert der Linker die benötigten Code-Bereiche in die Binary. Die Größe der Binary nimmt dementsprechend zu. Während der Programm-Code bei dynamischen Bibliotheken in der Regel positionsunabhängig (PIC) übersetzt ist, wird er bei statischen Bibliotheken positionsabhängig (non-PIC) generiert [30]. Dynamische Bibliotheken werden zur Laufzeit durch den Programmlader *ld.so* geladen. Während dieser bei der *x86*-Architektur das Laden von PIC als auch von non-PIC-Code unterstützt, fehlt die Unterstützung von non-PIC bei *x86_64* [13]. Es kann daher unter *x86_64* nur der Code in eine shared library aufgenommen werden, der mit dem Compiler-Flag *-fPIC* übersetzt wurde. Die Option *-fPIC* gehört jedoch nicht zu den Standardeinstellungen bei statischen Bibliotheken. So sind daher in der Regel die statischen Standardbibliotheken *libgcc* und *libstdc++* non-PIC.

Im Fall einer 64-Bit Architektur wird die Bibliothek *libe4rat-core* daher statisch und nicht mehr dynamisch gebaut. Nur so lässt sich die Abhängigkeit zum /usr Verzeichnis durch statisches Linken der Standardbibliotheken lösen.

6.3 Konfigurationsdatei

Die Konfigurationsdatei dient dem Zweck, das fest einprogrammierte Verhalten der Werkzeuge zu beeinflussen. Mit ihr können gewisse Standardeinstellungen dauerhaft geändert werden.

Ebenso möglich ist es, das Programmverhalten durch Übergabeparameter zu steuern. Das ist jedoch nur bedingt möglich, wenn das Werkzeug als Init-Prozess gestartet wird. Das Programm wird nicht von Hand in einem Kommandozeileninterpreter, sondern direkt vom Betriebssystem aus aufgerufen. Alle dem Kernel unbekannten Kernelparameter werden dem Init-Prozess als Aufrufparameter übergeben. Um Kollisionen bei Übergabe-

parametern zu vermeiden ist eine Konfigurationsdatei notwendig.

Optionen in der Konfigurationsdatei können global für alle und speziell in einer Sektion für eines der drei Werkzeuge gesetzt werden. Der Name der Sektion leitet sich aus dem Suffix des Werkzeugs ab. In folgender Reihenfolge wird gesucht, bis ein entsprechender Wert hinterlegt ist:

1. Optionen in der eigenen Sektion
2. Globale Optionen
3. Standardeinstellungen

Die Konfigurationsdatei wird mittels *INFO-Parser* aus der Boost-Bibliothek gelesen.

6.4 Fehlerbehandlung

Programme, die als Init-Prozess ausgeführt werden sind schwer zu debuggen. Aus diesem Grund ist ein umfangreiches Logging notwendig, um Verhalten und Programmfehler nachvollziehen zu können. Jede Nachricht ist einer Priorität zugeordnet. Wie ausführlich Nachrichten am Bildschirm angezeigt oder in die Log-Datei geschrieben werden sollen, lässt sich über Aufrufparameter und der Konfigurationsdatei einstellen.

Weiter können in der Konfigurationsdatei folgende drei Ziele, an die die Nachricht gesendet werden soll, eingestellt werden.

Kernel-Ring-Puffer [5, 25]
: Der Linux Kernel schreibt seine Log-Nachrichten in den sog. Kernel-Ring-Puffer. Der Dienst klogd sorgt dafür, dass die Nachrichten auf die Festplatte geschrieben werden. Über die Gerätedatei */dev/kmsg* können auch Anwendungsprogramme Nachrichten an den Puffer senden. Unabhängig vom Dateisystem oder laufenden Diensten können Programme durch den Ring-Puffer, auch im frühen Stadium des Bootvorangs, Ereignisse protokollieren. Die Größe des Puffers ist begrenzt. Werden zu viele Nachrichten generiert, läuft der Ring-Puffer über und ältere Nachrichten werden überschrieben.

Syslog Dämon [5, 11]
: Der Syslog Dämon ist ein Protokollierungsdienst. Über den Pfad /dev/log können Programme Nachrichten einfach und global protokollieren. Der Dienst wird im Laufe des Bootvorgangs gestartet. Log-Nachrichten können verloren gehen, wenn sich das Programm beendet, bevor der Dienst gestartet wird.

Log-Datei

Weiter gibt es die Möglichkeit Log-Nachrichten in eine Datei zu schreiben. Auch hier besteht die Gefahr, dass Nachrichten verloren gehen, nämlich dann, wenn sich das Programm beendet, bevor das Dateisystem schreibend im System eingehängt wurde.

Kann eine Nachricht nicht abgesetzt werden, weil der Syslog Dämon nicht läuft oder auf das Dateisystem nicht geschrieben werden kann, werden diese im Speicher in einer Warteschlange gehalten. Nachrichten gehen verloren, wenn das Programm sich beendet bevor das Ziel zur Verfügung steht.

7 Ergebnisse

Dieses Kapitel zeigt das Vebesserungspotential auf, welches durch die Umsortierung und den anschließenden Einsatz von *e4rat-preload*, bezüglich Boot- und Startvorgang möglich ist. Alle Untersuchungen wurden auf dem gleichen Rechner durchgeführt. Dieser wird im Abschnitt 7.1 näher beschrieben. Im Abschnitt 7.2 wird der Bootvorgang optimiert. Hier findet sich auch ein direkter Vergleich mit den Projekten *ureadahead* und *readahead-fedora*. Weiter werden die Ergebnisse zur Optimierung der Startvorgänge der Anwendungsprogramme *Gimp* und *OpenOffice* im Abschnitt 7.3 vorgestellt.

Für die graphische Darstellung des Bootvorgangs wurde das Programm Bootchart verwendet [2].

7.1 Versuchsumgebung

Als Versuchsumgebung wurde ein Rechner der Hochschule Augsburg verwendet. Um aufschlussreiche Ergebnisse zu erhalten wurde auf diesem Rechner ein Debian Linux frisch installiert. Ausgeschlossen wird somit eine Beschönigung der Messungen durch zusätzlich fragmentierte Dateien oder eine vorangegangene ungünstige Nutzung des Dateisystems.

Hardware

Ausgestattet mit einem Intel(R) Core(TM)2 Duo E8200 CPU mit 2.66GHz, 2 GB Arbeitsspeicher und einer Wester Digital WD2500YS-01S Festplatte stellt der Rechner einen heutzutage gängigen Desktop-PC dar.

Laut Hersteller ist die Festplatte in [29] mit folgenden Werten ausgeschrieben:

Allgmein	Kapazität	250 GB
	Drehzahl	7200 $\frac{1}{min}$
Zugriffszeiten	Spurwechselzeit	8,9 ms (Durchschnitt) 2,0 ms (nächste Spur)
	Rotationsverzögerung	4.2 ms
Übertragungsrate	SATA-Controller	300 $\frac{MB}{s}$
	Festplatte	61 $\frac{MB}{s}$

Installation

Der gesamte Festplattenspeicher ist einer Partition zugeordnet. Darauf befindet sich eine Debian (6.0) Squeeze Standardinstallation mit unter anderem einer graphischen Oberfläche und der Gnome-Desktop-Umgebung. Für die Erstellung der Graphen wurde das Werkzeug Bootchart aus Debian Testing installiert. Auf eine SWAP-Partition wurde verzichtet. Nach der Installation war das Dateisystem mit 6,7 GB belegt. Dies entspricht einer gesamten Festplattenauslastung von 3%.

Für die automatische Anmeldung ist die Konfigurationsdatei /etc/gdm/custom.conf des Anmeldedienstes GDM (Gnome Display Manager) wie folgt abgeändert:

```
[daemon]
AutomaticLoginEnable=true
AutomaticLogin=root
[security]
AllowRoot=true
```

Ein Eintrag im *Gnome Autostart* lässt Firefox automatisch starten.

Um den gesamten Bootvorgang aufzuzeichnen, muss verhindert werden, dass *bootchartd* sich automatisch beendet. Mit folgendem Eintrag in der Konfigurationsdatei wird dieses Verhalten deaktiviert. /etc/bootchart.conf:

```
AUTO_STOP_LOGGER="no"
```

Für die Messung der Bootvorgänge wurden folgende Kernelparameter im Grub hinzugefügt:

```
init=/sbin/bootchartd bootchart_init=/sbin/init
```

Mit dem Parameter *bootchart_ init* kann ein alternativer Programmpfad angegeben werden, der als Init-Prozess ausgeführt wird. Im Bootvorgang mit *e4rat-preload* wurde der Wert durch */sbin/e4rat-preload* ersetzt.

7.2 Optimierung des Bootvorgangs

Dieses Kapitel vergleicht die Dauer des Bootvorgangs im Ausgangszustand, nach der Umsortierung und bei anschließendem Einsatz von *e4rat-preload*. Die Ergebnisse werden anschließend mit den Verfahren von *ureadahead* und *readahead-fedora* verglichen.

Die Bootvorgänge sind jeweils in Abbildung 16 dargestellt.

Die Ergebnisse sind in der Tabelle 2 zusammengefasst. Die Dateiliste /var/lib/e4rat/startup.log umfasst 2548 Einträge.

	Bootzeit	Verbesserungsfaktor
Ausgangszustand	44,5 s	–
ureadahead	25,5 s	1,7
readahead-fedora	24,4 s	1,8
Nach der Sortierung	23,4 s	1,9
Mit *e4rat-preload*	13,7 s	3,2

Tabelle 2: Messungen zur Reduzierung des Bootvorgangs

Der Bootvorgang des frisch installierten Debian Linux benötigt im Ausgangszustand 44,5 Sekunden. Die CPU ist kaum ausgelastet. Die Festplatte weist trotz hoher Auslastung nur eine geringe Übertragungsrate auf.

Allein durch das Umsortieren der Bootdateien konnte die Dauer des Bootvorgangs um fast die Hälfte reduziert werden. Die durchschnittliche Übertragungsrate ist angestiegen. Die Festplattenzugriffszeit ist gesunken.

Zieht man zusätzlich *e4rat-preload* hinzu, konnte der Bootvorgang um weitere 18,7 Sekunden auf insgesamt 13,7 Sekunden reduziert werden. Im Vergleich zum Ausgangszustand bedeutet es, dass der Bootvorgang 3,2 mal schneller ausgeführt wird.

In der Abbildung 16e sind die zwei Arbeitsschritte, das Lesen der I-Nodes und das Lesen der Dateiinhalte, gut zu erkennen. Die I-Nodes liegen auf dem Datenträger immer noch weit verteilt. Um sie zu lesen sind weiterhin viele Kopfbewegungen notwendig. Die Datenmenge ist gering, dennoch werden hierfür ungefähr 3 Sekunden aufgewendet. Die kurze Spitze von 64 $\frac{MB}{s}$ ist höher als die maximale Übertragungsrate der Festplatte mit 61 $\frac{MB}{s}$. Erreicht wird sie durch die gute Ausnutzung des internen Festplatten-Caches und die höhere Übertragungsrate des SATA-Busses.

Nachdem sich *e4rat-preload* beendet hat, finden kaum mehr Festplattenzugriffe statt. Wartezeiten aufgrund der Festplatte bleiben fast vollständig aus. Nachdem die graphische Oberfläche gestartet ist, sind beide Kerne der CPU optimal ausgelastet.

Im Vergleich dazu fällt die Optimierung ohne Umsortierung durch *ureadahead* und

readahead-fedora weniger stark aus. Beide Werkzeuge können die Startzeit fast halbieren. Die Bootzeiten unterscheiden sich kaum. Die Übertragungsrate der Festplatte ist erhöht. Während die Daten von der Festplatte gelesen werden, findet fast keine CPU-Aktivität statt. Das liegt daran, dass die Dateien nach der physikalischen Position und nicht in der Aufrufreihenfolge gelesen werden. Der Bootvorgang wird somit zunehmend unterbrochen. Der Bootvorgang dauert in beiden Fällen etwas länger als nach der Umsortierung ohne den Einsatz von *e4rat-preload*.

7.3 Optimierung von Anwendungen

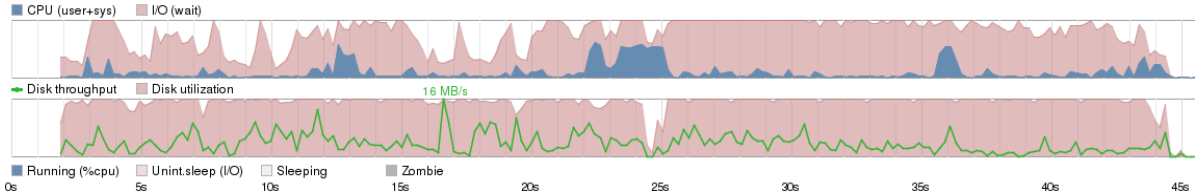
Weiter wurde untersucht, wie sich die Umsortierung auf den Startvorgang von Anwendungsprogrammen auswirkt. Hierfür wurden das Bildverarbeitungsprogramm *Gimp* und das Textverarbeitungsprogramm *OpenOffice* optimiert. Die genauen Zahlen sind der Tabelle 3 zu entnehmen.

		Startzeit	Verbesserungsfaktor
Gimp	Ausgangszustand	5,2 s	–
	Nach der Sortierung	3,5 s	1,5
	Mit *e4rat-preload*	3.0 s	1,7
OpenOffice	Ausgangszustand	5,1 s	–
	Nach der Sortierung	3.1 s	1,6
	Mit *e4rat-preload*	2,5 s	2,0

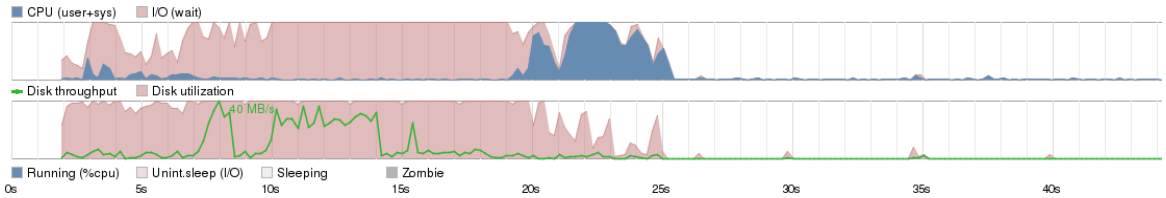
Tabelle 3: Messungen zur Reduzierung der Startzeiten von Gimp und OpenOffice

Die Dateiliste von *Gimp* umfasst 466 Dateien. Die Umsortierung reduzierte die Startzeit um ca. 1,7 Sekunden. Vergleicht man die Startvorgänge in Abbildung 17 erkennt man, dass sich die Effizienz der Festplatte steigern ließ. Das vorzeitige Einlesen von Dateien durch *e4rat-preload* verbessert die Startzeit nur geringfügig.

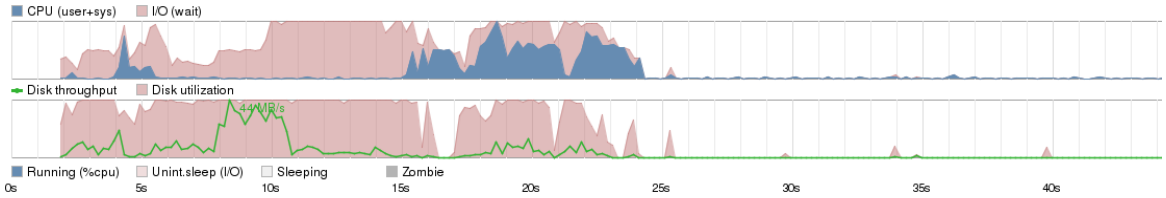
Ein ähnliches Bild zeigt sich bei der Optimierung des Textverarbeitungsprogramms *OpenOffice*. Die Dateiliste umfasst 137 Einträge. Durch Umsortierung und durch den Einsatz von *e4rat-preload* ließ sich die Startzeit halbieren. In Abbildung 18 ist eine deutliche Steigerung der Übertragungsrate zu erkennen.



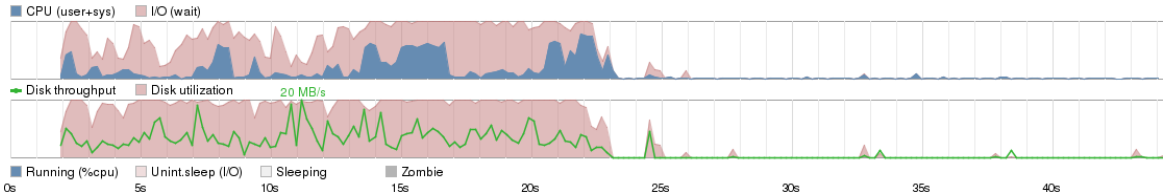
(a) Ausgangszustand



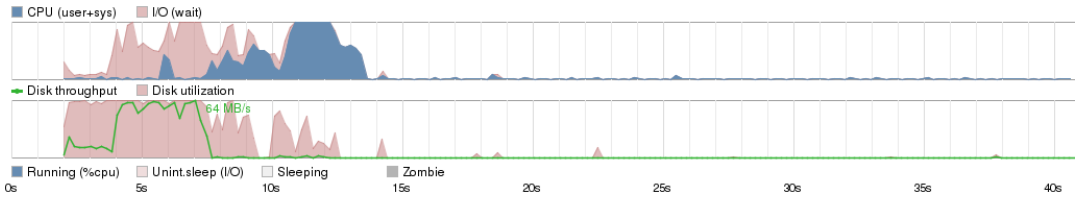
(b) *ureadahead*



(c) *readahead-fedora*

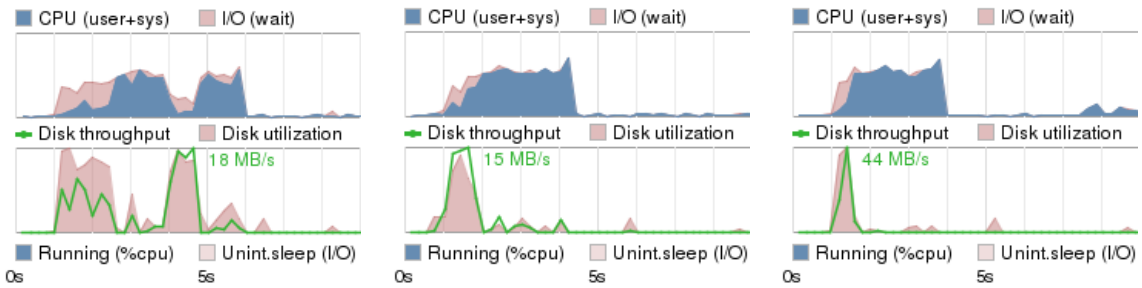


(d) Nach der Umsortierung



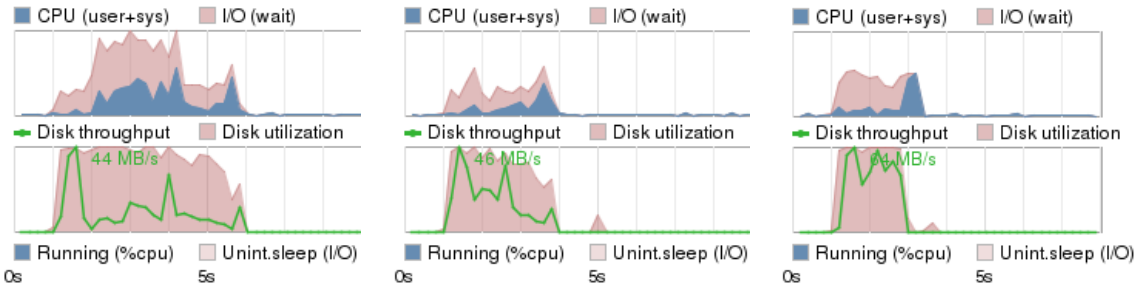
(e) Mit *e4rat-preload*

Abbildung 16: Bootvorgänge im Vergleich



(a) Ausgangszustand (b) Nach der Umsortierung (c) Mit *e4rat-preload*

Abbildung 17: Optimierung des Startvorgangs von *Gimp*



(a) Ausgangszustand (b) Nach der Umsortierung (c) Mit *e4rat-preload*

Abbildung 18: Optimierung des Startvorgangs von *OpenOffice*

8 Fazit und Ausblick

8.1 Fazit

Durch eine absolut sequentielle Anordnung der Dateiinhalte auf der Festplatte konnte die Startzeit von Programmen halbiert und der Bootvorgang gedrittelt werden. Die Übertragungsrate stieg bis auf ihr Maximum an. Mechanisch bedingte Verzögerungen durch Bewegungen des Magnetkopfes ließen sich größtenteils verhindern. Auch ohne Vorladeprogramm *e4rat-preload* reduziert sich die Startzeit spürbar.

Die Optimierung basiert auf der Erstellung einer Dateiliste mit dem Werkzeug *e4rat-collect*. Die Dateiliste wird vom Anwender einmal erstellt und kann fortan zur Reduzierung der Startzeit verwendet werden. Dateizugriffe lassen sich zur Erstellung der Dateiliste von jedem beliebigen Programm oder Vorgang überwachen. Auch die Erstellung einer eigenen Liste von Hand ist somit möglich. Die Werkzeuge lassen sich somit individuell einsetzen. Wichtig ist nur darauf zu achten, dass keine Datei in mehreren Listen vorkommt.

Während *ureadahead* und *readahead-fedora* den Startvorgang beim Einlesen stören, da sie Dateien entsprechend der physikalischen Position und nicht nach der Aufrufreihenfolge einlesen, erfolgt der Lesevorgang mit *e4rat-preload* störungsfrei parallel zum Startvorgang.

Trotz Umsortierung wird weiterhin viel Zeit für das Lesen von der Festplatte aufgewendet. In Abbildung 16e benötigt *e4rat-preload* rund 7 Sekunden von insgesamt 15, um alle Daten von der Festplatte zu lesen. Weiter optimierbar sind die 3 Sekunden, in denen die I-Nodes gelesen werden.

8.2 Ausblick

Im folgenden werden Ideen vorgestellt, mit denen das bisher erreichte Verbesserungspotenzial möglicherweise noch weiter ausgebaut werden kann.

8.2.1 Defragmentieren von freien Speicherbereichen

Der Erfolg der Umsortierung hängt stark von der Größe freier Blockbereiche ab. Ist selbst der größte Bereich zu klein, können nicht alle Dateien der Reihe nach angeordnet werden. Das mögliche Potential der Umsortierung ist nicht vollständig ausgeschöpft.

Durch das Werkzeug *e4defrag* soll das Defragmentieren von freiem Speicher in Zukunft

möglich sein. Der Ansatz besteht bislang nur in der Theorie und ist noch nicht implementiert.

8.2.2 Verschieben nicht regulärer Dateien

In dieser Arbeit wurden lediglich Blöcke von regulären Dateien auf der Festplatte verschoben. Andere Dateisystemobjekte wie Verzeichnisse oder symbolische Verknüpfungen (engl. soft-link) werden nicht berücksichtigt. Dabei wäre das Defragmentieren von Verzeichnissen sinnvoll. Sie wachsen in der Regel langsam, was im Laufe der Zeit zu einer starken Fragmentierung führt.

Eine symbolische Verknüpfung verweist auf ein anderes Dateisystemobjekt. Der Dateipfad wird als Dateiinhalt in einem separatem Block gespeichert [7].

Bislang akzeptiert der Befehl EXT4_IOC_MOVE_EXT nur Dateideskriptoren von regulären Dateien. Die Startzeit könnte möglicherweise noch weiter reduziert werden, wenn sich auch Verzeichnisse oder Soft-Links physikalisch umsortieren ließen.

8.2.3 Verschieben der I-Node-Strukturen

Die I-Node-Strukturen werden in der Umsortierung nicht erfasst. Sie bleiben weiterhin an ihrem ursprünglichen Ort auf dem Datenträger. Umso stärker die I-Nodes verteilt liegen, desto langsamer wird die erste Phase von *e4rat-preload*, was den Startvorgang verlängert.

Jeder I-Node besitzt eine eindeutige Nummer. Anhand der Nummer wird die Blockgruppe und die Postion in der I-Node-Tabelle berechnet. Durch Verschieben eines I-Nodes ändert sich auch die I-Node-Nummer. Wird sie verschoben, müssen alle Objekte, die auf den I-Node referenzieren, aktualisiert werden.

Verweisen mehrere Verzeichnispfade auf diesen I-Node, müssen die anderen Pfade mit großen Aufwand gesucht werden. Hierfür muss die komplette Baumstruktur durchlaufen werden.

8.2.4 Stufenweises Lesen

Während *e4rat-preload* die I-Nodes liest, ist der Bootvorgang unterbrochen. Dabei verstreicht jedoch wertvolle Zeit. Dieser Vorgang könnte verkürzt werden, wenn

[7] Werden zur Speicherung des Dateipfades weniger als 64 Bytes benötigt, wird der Dateipfad nicht in einem separaten Block, sondern direkt in der I-Node-Struktur gespeichert. Diese Art des Soft-Links wird als Fast-Link bezeichnet.

e4rat-preload die Dateien stufenweise einlesen würde.

e4rat-preload erreicht schnell einen Vorsprung vor dem parallel ausgeführten Startvorgang. Dieser Vorsprung könnte genutzt werden, um das Lesen der I-Nodes im hinteren Teil der Liste zeitlich nach hinten zu verschieben. Die Dauer der Unterbrechung, bevor das Programm oder der Bootvorgang fortgesetzt wird, ließe sich somit verkürzen.

8.2.5 Verschieben wachsender Dateien

Bei der Erstellung der Dateiliste wurden Dateien, auf die schreibend zugegriffen wird, bisher aus der Liste ausgeschlossen. Würden diese Dateien wachsen, hätten sie nach der Umsortierung keinen Platz dafür. Die Dateien würden schnell fragmentieren und das Einlesen verlangsamen.

Sie könnten dann in die Liste aufgenommen werden, wenn bei der Umsortierung ausreichend Platz einplant würde. Dies kann einfach realisiert werden, wenn die Spenderdateien größer als die Originaldateien angelegt würden.

Literaturverzeichnis

[1] Brian Beeler: „Hard Disk Drive Reference Guide“, 16. März 2010
http://www.storagereview.com/hard_disk_drive_reference_guide
Stand: April 2011

[2] Bootchart, Version 0.10
http://www.bootchart.org
Stand: April 2011

[3] M. Cao, A. Dilger, A. Kumar, J. Santos: „Ext4 block and inode allocator improvements“, 2008
http://ols.fedoraproject.org/OLS/Reprints-2008/kumar-reprint.pdf
Stand: April 2011

[4] Jonathan Corbet: „The Orlov block allocator“, 5. November 2002
http://lwn.net/Articles/14633/ Stand: April 2011
Stand: April 2011

[5] Alan Cox: „LINUX ALLOCATED DEVICES (2.6+ version)“, 6. April 2009
http://www.kernel.org/doc/Documentation/devices.txt
Stand: April 2011

[6] CVE-2009-4131 kernel: ext4: Fix insufficient checks in EXT4_IOC_MOVE_EXT
https://bugzilla.redhat.com/show_bug.cgi?id=544471
Stand: April 2011

[7] Behdad Esfahbod: „Preload – An Adaptive Prefetching Daemon“, 2006

[8] Dr. Oliver Diedrich: „Das Linux-Dateisystem Ext4“, c't 10/09, S. 180

[9] Akira Fujita: “Outline of Ext4 File System & Ext4 Online Defragmentation Foresight“, 28 September 2010
http://events.linuxfoundation.org/slides/2010/linuxcon_japan/linuxcon_jp2010_fujita.pdf
Stand: April 2011

[10] GCC online documentation
http://gcc.gnu.org/onlinedocs/
Stand: April 2011

[11] „The GNU C Library“, Version 2.13, 10. Februar 2011
http://www.gnu.org/software/libc/manual/
Stand: April 2011

[12] Steve Grubb: „Audit and IDS“, 06. Juni 2008
http://people.redhat.com/sgrubb/audit/audit-ids.pdf
Stand: April 2011

[13] Jan Hubička, Andreas Jaeger, Michael Matz, Mark Mitchell: „System V Application Binary Interface AMD64 Architecture Processor Supplement “, 3. September 2010
http://www.x86-64.org/documentation/abi-0.99.pdf
Stand: April 2011

[14] Dr. Volker Jaenisch, Martin Klapproth, Patrick Westphal: „I/O-Scheduler und RAID-Performance“
www.linuxtechnicalreview.de/content/download/420/3357/file/I-O-Scheduler-und-RAID-Performance.pdf
Stand: April 2011

[15] Olof Johansson: „A look at 32 vs 64bit userspace“
http://lixom.net/~olof/lca.pdf
Stand: April 2011

[16] Eva-Katharina Kunst, Jürgen Quade: „Kernel- und Treiberprogrammierung mit dem Kernel 2.6 - Folge 40: Kern Technik“, Linux-Magazin 2008/07

[17] Linux Kernel: I-Node-Allocator des Ext4 Dateisystems:
linux-2.6.36/fs/ext4/ialloc.c

[18] Linux Kernel: Multi-Block-Allocator des Ext4 Dateisystems:
linux-2.6.36/fs/ext4/mballoc.c

[19] Robert Love: „Linux Kernel Development“, 2004, S. 211-251

[20] Wesley McGrew: „Ext2 and Ext3 Filesystems: Intro to the data structures, and points of interest for forensic examiner“
http://mcgrewsecurity.com/training/extx.pdf
Stand: April 2011

[21] Kazuya Mio: „[RFC][PATCH V3 0/4] ext4: inode preferred block allocation“
http://kt1.osuosl.org/mailarchive/linux-fsdevel/2010/12/1/6887724
Stand: April 2011

[22] readahead-fedora, Version 1.5.6
https://fedorahosted.org/readahead
Stand: April 2011

[23] Rusty Russell, Daniel Quinlan, Christopher Yeoh: „Filesystem Hierarchy Standard“, V2.3, 29 Januar 2004
http://www.pathname.com/fhs/
Stand: April 2011

[24] Peter Schmid: „Das Filesystem ext2“, 17. April 2009
http://pubwww.hsz-t.ch/~pschmid/bs/script/ext2-p.pdf
Stand: April 2011

[25] Michael A. Schwarz: „The Kernel Logging Dæmon“, Linux Journal, 1. August 2000
http://www.linuxjournal.com/article/4058
Stand: April 2011

[26] super-readahead, Version 1.0
http://code.google.com/p/sreadahead
Stand: April 2011

[27] SUSE Linux Enterprise: „The Linux Audit Framework“, 10SP1, 08. März 2008
http://www.novell.com/documentation/sled10
Stand: April 2011

[28] über-readahead, Version 0.100.0
https://launchpad.net/ureadahead
Stand: April 2011

[29] WD RE Enterprise Hard Drives, August 2007
www.wdc.com/en/library/eide/2879-001119.pdf
Stand: April 2011

[30] Ian Wienand: „Position Independent Code and x86-64 libraries“, 26 November 2008
http://www.technovelty.org/code/c/amd64-pic.html
Stand: April 2011

[31] Wikipedia „Festplattenlaufwerk“
http://de.wikipedia.org/w/index.php?title=Festplattenlaufwerk&oldid=88477773
Stand: Mai 2011