TECHNISCHE
UNIVERSITÄT
WIEN

VIENNA
UNIVERSITY OF
TECHNOLOGY

DIPLOMARBEIT

# Schnelles Starten grosser Programme

ausgeführt am

Institut für Computersprachen
Abteilung für Programmiersprachen und Übersetzerbau
der Technischen Universität Wien

unter Anleitung von
Ao.Univ.Prof. Dipl.-Ing. Dr.techn. M. Anton Ertl

durch

Stefan Strauß-Haslinglehner

Mitterndorf 97
A-4643 Pettenbach

Datum

Unterschrift

**Kurzfassung**

Diese Arbeit zeigt eine Möglichkeit zur Reduktion der Zeit für den Kaltstart von Anwendungen durch Preloading der für den Start benötigten Blockgerätedaten. Dabei wird die Zugriffsfolge des Demand Pagings so umgeordnet, dass die Blockgeräte I/O Performance beim Preloading optimiert wird.

Gezeigt wird das Design und die Implementierung eines Linux Kernelmoduls, bestehend aus einer Recording Einheit, einer Datenbank und einer Preloading Einheit. Das Recording zeichnet die Zugriffsfolge eines Anwendungsstarts auf die Blockgeräte des Systems auf. Die so gewonnenen Daten werden persistent in einer Datenbank gespeichert. Zu Beginn zukünftiger Starts der Anwendung führt das Preloading die optimierte Zugriffsfolgen konzentriert aus.

Die Auswirkung des Preloadings wurde auf zwei unterschiedlichen Testplattformen mit verschiedenen Anwendungen untersucht. Messungen zeigten dabei eine Reduktion der Startzeit um den Faktor 1.5 bis 1.8.

**Abstract**

This thesis shows a way to reduce application startup time. This is done by preloading the data needed during application startup in an order optimized for block device I/O performance. The thesis describes the design and implementation of a Linux kernel module, consisting of a recording unit, a database and a preloading unit. The recording unit is responsible for recording access patterns on block-devices at application startup. The recorded access patterns are optimized and stored persistently in the database. The preloading unit is responsible for preloading the corresponding access patterns on future application startups.

The effect of preloading was analyzed using several applications on two test systems. Measurements showed a speedup by a factor of 1.5 to 1.8.

# Inhaltsverzeichnis

# Kapitel 1

# Einleitung

Das freie Betriebssystem GNU/Linux konnte in den letzten Jahren seine Marktposition im Serverbereich ausbauen und ist in manchen Bereichen, wie dem der Webserver, sogar führend [NC04]. Obwohl inzwischen ausgereifte Windowmanager wie Gnome oder KDE verfügbar sind, ist verglichen mit dem Serverbereich die Verbreitung von GNU/Linux am Desktop noch gering [MG04]. Ein wichtiger Schritt zur Steigerung der Akzeptanz ist die Verfügbarkeit von Desktop-Schlüsselanwendungen wie Browser, Personal Information Management (PIM), Textverarbeitung oder Tabellenkalkulation.

Wie auch auf anderen Plattformen steigen Funktionsumfang, Programmgrösse und Speicheranforderung dieser Anwendungen stetig. Verbunden mit einer im Vergleich zu Serversystemen relativ kurzen Uptime der Desktopsystemen, werden diese Anwendungen oft unter der Voraussetzung gestartet, dass sich keine oder nur wenige für den Start benötigten Daten im Speicher befinden. Diese Inhalte müssen beim Programmstart von den langsamen Blockgeräten des Systems in den Speicher gelesen werden, was entsprechend lange Startzeiten bedingt.

## 1.1 Subjektive Performance

Entscheidend, ob ein User ein System als schnell empfindet, sind die Antwortzeiten bei der Ausführung einer Aktion. Diese können folgendermassen klassifiziert werden [JN93, S.135]:

- Beträgt die Antwortzeit weniger als 0.1 Sekunden, so hat der User den Eindruck von unmittelbaren Feedback des Systems, der Arbeitsfluss ist nicht gestört.

- Bei Antwortzeiten von weniger als einer Sekunden bemerkt der User die Unterbrechung des Arbeitsflusses und verliert das Gefühl unmittelbar an den Daten zu arbeiten.

- Länger als 10 Sekunden ist der User nicht bereit zu warten. Er möchte sich mit anderen Aufgaben während der Wartezeit beschäftigen.

Die erste Aktion des Users bei der Verwendung eines Programms ist der Anwendungsstart. Die Performance des Anwendungsstarts ist ein erster und somit wichtiger subjektiver Eindruck der Anwendungsperformance.

Ziel dieser Arbeit ist es daher, den Start grosser Anwendungen zu beschleunigen. Das könnte ein Schritt zur Akzeptanzsteigerung von GNU/Linux am Desktop sein.

## 1.2 Ausgangssituation

Zur Definition der Ausgangssituation soll einigen Messergebnissen aus Kapitel 3 vorgegriffen werden. Sie beschreiben das Zeitverhalten der Anwendung OpenOffice 1.0.1 am Testsystem Laptop [1].

Die Messung der Zeit für den Start von OpenOffice nach Reboot (Tab.3.4, Kaltstart) ergibt eine Zeit von 14.2s. Startet man die Anwendung nochmals, so sinkt die Startzeit deutlich (Tab.3.4, Warmstart) auf 3.0s. Der Unterschied zwischen den beiden Startzeiten könnte einen Ansatzpunkt zur Verbesserung der Startperformance bieten. In den nächsten Abschnitten soll daher näher auf die Ursache dieser Zeitdifferenz eingegangen werden.

### 1.2.1 Die Speicherhierachie

Eine Erklärung liefert das Kommando `time`, welches u.a. die Anzahl der Kontextswitches wegen I/O Aktivität auflistet. Es besagt, dass OpenOffice beim Start nach Reboot 959 mal die CPU entzogen wird, weil die Anwendung auf das Ende eines Datentransfers von der Festplatte in den Speicher warten muss (Tab.3.7, Kontext-Switches I/O). Ursache ist die in allen Computersystemen implementierte Speicherhierachie.

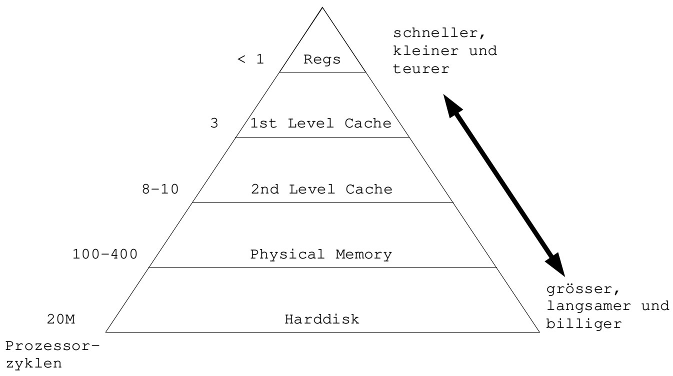


Abbildung 1.1: Tradeoff in der Speicherhierachie

Bei den Speichermedien eines Computersystems existiert ein Tradeoff zwischen Speicherkapazität und Zugriffsgeschwindigkeit, bzw. zwischen Kosten per Bit und

[1] siehe Kapitel 3.1 und 3.2

Zugriffsgeschwindigkeit [PH98, S. 542]. Speichermedien sind daher in eine Hierarchie eingeteilt (Abb.1.1). An der linken Kante der Grafik ist eine Abschätzung der Zugriffszeiten in Prozessorzyklen auf die jeweilige Ebene der Hierachie angegeben. Das Speichermedium mit der schnellsten Zugriffsgeschwindigkeit befindet sich in der Hierarchie am weitesten oben, d.h. dem Prozessor am nächsten, während sich das Speichermedium mit der langsamsten Zugriffsgeschwindigkeit am unteren Ende der Speicherhierachie befindet.

Aufgrund der aufgezeigten Konvergenz ist das Speichermedium mit der schnellsten Zugriffsgeschwindigkeit auch am teuersten und somit in kleinerer Kapazität vorhanden, als das Speichermedium mit der langsamsten Zugriffsgeschwindigkeit, üblicherweise das Blockgerät.

### Das Prinzip der Lokalität

Durch die Implementierung einer Speicherhierachie wird die Illusion eines grossen Speicherbereichs mit schneller Zugriffszeit geschaffen. Diese Illusion funktioniert auf Grund des Prinzips der Lokalität, wobei zwei Arten von Lokalität unterschieden werden [PH98, S. 541 ff.]:

- *zeitliche Lokalität*: Wurde auf ein Element im Speicher zugegriffen, so ist die Wahrscheinlichkeit gross, dass erneut auf dieses Element zugegriffen wird.

- *räumliche Lokalität*: Wurde auf ein Element im Speicher zugegriffen, so ist die Wahrscheinlichkeit gross, dass auch auf im Speicher benachbarte Elemente zugegriffen wird.

Versagt das Prinzip der Lokalität, so spricht man von einem *Miss* in einer Ebene der Speicherhierachie. Die Entwicklung am Hardwaresektor zeigt, dass ein Miss in einer Ebenen der Speicherhierachie auf Grund der ständig steigenden Taktfrequenz immer teurer im Sinne von Prozessorzeit wird. Auf allen Ebenen der Speicherhierachie werden daher zur Reduktion der Zugriffe auf die nächste Ebene der Speicherhierachie Caches sowohl in Hardware als auch in Software implementiert ([PH98, S.545])([STA98, S.479]).

Je weiter man sich in der Speicherhierachie nach unten bewegt, desto grösser werden die Kosten eines Miss in der entsprechenden Ebene der Hierarchie. Die höchsten Kosten verursacht demnach ein Miss in der vorletzten Ebene der Speicherhierachie, dem Arbeitsspeicher. Ein Miss bedingt hier einen Zugriff auf das langsamste Medium der Speicherhierachie, dem Blockgerät.

Die Eingangs erwähnten 959 I/O Kontextswitches beim Start von OpenOffice nach Reboot bedeuten also 959 Zugriffe auf das langsamste Medium der Speicherhierachie. Der Anwendungsstart zeigt I/O-lastigen Charakter. Wird OpenOffice ein weiters mal gestartet, so ist hingegen kein Zugriff auf die Festplatte mehr notwendig. Der Anwendungsstart ist dann CPU-lastig, da sich alle für den Start der Anwendung benötigten Daten bereits im Speicher befinden.

### 1.2.2 Asequentieller Zugriff

Im letzten Abschnitt wurde als Ursache der Abweichung der Startzeiten zwischen I/O- und CPU-lastigen Start von OpenOffice die Zugriffe auf die Festplatte festgestellt. Diese Zugriffe sind nicht zu vermeiden, da OpenOffice die Blockgerätedaten für den Anwendungsstart braucht. Dem Kommando `time` nach wird der I/O ausschliesslich durch Memory Mappings verursacht. Datei I/O findet während des Starts von OpenOffice nicht statt.

Für eine Steigerung der Startperformance muss man also die Effizienz der Blockgerätezugriffe während des Anwendungsstarts steigern.

OpenOffice liest am Laptop beim Start nach Reboot 11386 Seiten an Daten vom Blockgerät ein (Tab.3.9, Umfang ZF). Bei einer Seitengrösse von 4K sind dies 44.5M Daten. Das Einlesen dieser Daten dürfte bei der mit `hdparm -t` ermittelten Transferrate von 20 MB/s 2.2s dauern. Addiert man diese 2.2s zur Zeit für den Start von OpenOffice wenn alle Daten im Speicher sind (3.0s), so ist man noch immer weit von der Startzeit nach Reboot (14.3s) entfernt. Der Grund dafür wird in den nächsten Abschnitten untersucht.

#### Der virtuelle Adressraum

Das Konzept des virtuellen Adressraums basiert ebenfalls auf dem Prinzip der Lokalität. Durch den virtuellen Adressraum wird die Speicherlimitierung eines Prozesses auf den verfügbaren physikalischen Speicher umgangen, in dem jedem Prozess ein virtueller Adressraum, in der Grösse abhängig vom adressierbaren Bereich eines Prozessors (bei 32-Bit-Systemen $2^{32} = 4GB$), zur Verfügung gestellt wird. Die Abbildung vom virtuellen auf den physikalischen Adressraum erfolgt mit Hilfe der Page Tables eines Prozesses.

#### Memory Mapping

Memory Mapping ermöglicht das Einblenden von Ausschnitten beliebiger Dateien in den virtuellen Adressraum eines Prozesses, wie z.B. den Code- oder Datenbereich einer ausführbaren Datei oder von Shared Libraries. R/W-Operationen auf einem Memory Mapping werden vom Kernel transparent für den Prozess in R/W-Operationen am Backing Store, meistens eine Datei, umgewandelt.

#### Demand Paging

Demand Paging wird durch das Konzept des virtuellen Adressraums ermöglicht. Ziel des Demand Pagings ist die Verschiebung der Allokierung physikalischen Speichers auf den letztmöglichen Zeitpunkt: den des ersten Zugriffs auf die entsprechende Adresse im virtuellen Speicher.

Der Prozessor unterbricht die Ausführung des Prozesses und generiert einen Page Fault. Führt die Page ein Memory Mapping durch, so sucht der Page Fault Handler die entsprechenden Daten im Page Cache. Sind die Daten im Page Cache (minor Page Fault), so kann der Prozess fortfahren. Im Falle eines Miss (major Page Fault) wird dem Prozess die CPU entzogen und er muss warten, bis die Daten vom Backing Store eingelesen wurden.

### Demand Paging beim Anwendungsstart

Wie u.a. die Zahl der I/O-Kontextswitches beim Start von OpenOffice zeigt, kommt es beim Anwendungsstart zu einer Häufung von major Page Faults, falls die von der Anwendung benötigten Daten nicht im Speicher sind. Bei so einer Häufung von major Page Faults zeigen sich einige Nachteile des Demand Paging Mechanismus:

- Moderne Blockgeräte können Requests bearbeiten, die viele Seiten umfassen. Da ein Prozess auf die Klärung eines major Page Faults in einer Wait Queue warten muss, generiert dieser zu diesem Zeitpunkt auch keine weiteren Zugriffe in diesen Datenbereich des Blockgeräts. Der Page Fault Mechanismus nutzt damit weder die maximale Kapazität der Requests, noch zeitliche und räumliche Lokalität der Zugriffe auf das Blockgerät. Theoretisch werden immer nur Daten im Umfang einer Speicherseite vom Blockgerät gelesen.

- Demand Paging versucht, mit Hilfe des Readahead Mechanismus spekulativ Blockgerätedaten zu lesen. Bei einer Folge sequentieller Zugriffe wird der spekulativ gelesene Datenanteil vergrössert, bei asequentiellen Zugriffen verkleinert. Um eine Verschmutzung des Page Caches mit nicht benötigten Blockgerätedaten zu vermeiden, wird bei einer Serie sequentieller Zugriffe der spekulative Datenanteil nur langsam erhöht. Aus dem selben Grund wird bei asequentiellen Zugriffen der spekulative Datenanteil schnell eingeschränkt. Beim Anwendungsstart werden so integrale Datenbereiche am Blockgerät weiterhin durch viele, schlecht ausgelastete Requests vom Blockgerät gelesen.

- Sprünge im Programmfluss wie Verzweigungen, Schleifen oder Funktionsaufrufe bewirken eine asequentielle Zugriffsfolge auf das Blockgerät. Für die Zugriffe sind entsprechend umfangreiche Aktionen der Komponenten des Blockgeräts notwendig. Dieser Overhead, wie Aktuatorenbewegungen oder Rotation ist mechanischer Natur und dauert aus Sicht der CPU entsprechend lange.

- Die Daten einer Datei sind nicht notwendigerweise physikalisch sequentiell am Blockgerät abgelegt. Bei grossen Dateien und hoher Auslastung des Blockgeräts kann es sein, dass ein Dateisystem die Datenblöcke einer Datei fragmentiert am Blockgerät ablegen muss.

Die angeführten Punkte reduzieren die Blockgeräte I/O Performance und somit die Performance beim Anwendungsstart, was auch die zu Beginn des Kapitels gezeigten Messergebnisse zeigen.

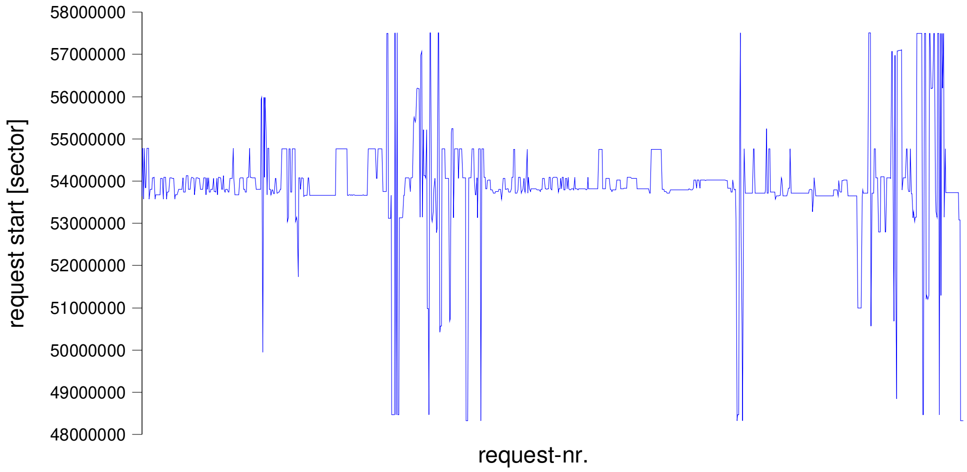


Abbildung 1.2: Start OpenOffice 1.0.1 - Sektorzugriffsfolge Demand Paging

Das Kommando `hdparm -t` misst die Blockgeräte I/O Performance bei sequentiellen Zugriff. Das Einlesen der für den Start von OpenOffice benötigten Daten von der Festplatte dauert jedoch $14.3 - 3.0 = 11.3s$ anstatt der errechneten 2.2s. Daraus errechnet sich eine Transferrate von 4 MB/s anstatt der mit `hdparm -t` gemessenen 20 MB/s. Ursache kann nur der asequentielle Charakter der Zugriffsfolge des Demand Pagings sein.

Das ist auch in Abbildung 1.2 visualisiert. Das Diagramm zeigt auf der Ordinate den von der Zugriffsfolge betroffenen Sektorbereich der Festplatte. Die Abszisse stellt die Reihenfolge der Requests dar.

## 1.3 Das Konzept

Demand Paging basiert auf der Annahme, dass der Programmablauf und somit die Zugriffsfolge auf ein Blockgerät nicht vorhersagbar ist. In der Regel trifft dies auch zu. So ist z.B. die Interaktion eines Users mit einer Anwendung mit grafischer Benutzeroberfläche nicht vorhersagbar. Eine Ausnahme stellt jedoch der Anwendungsstart dar. Das Konzept dieser Arbeit zur Beschleunigung von Anwendungsstarts basiert auf folgender Hypothese:

> Die Zugriffe einer startenden Anwendung auf die Blockgeräte des Systems weisen grösstenteils statischen Charakter auf, d.h. sind vorhersagbar.

Grund dafür ist die eingeschränkte Möglichkeit der Einflussnahme auf die startende Anwendung durch den User, z.B. meistens nur möglich durch Kommandozeilenparameter oder Konfigurationsdateien.

Um die aufgestellte Hypothese untersuchen zu können, soll im Zuge dieser Arbeit eine Anwendung implementiert werden, welche den statischen Charakter der Blockgerätezugriffsfolge beim Anwendungsstart zur Optimierung der Blockgeräte I/O-Performance nutzt.

Dazu soll die Folge der Blockgerätezugriffe einer startenden Anwendung aufgezeichnet und so umgeordnet werden, dass eine möglichst sequentielle Zugriffsfolge entsteht. Vor späteren Anwendungsstarts soll diese optimierte Zugriffsfolge dann reproduziert werden.

Die Requests der Zugriffsfolge werden out-of-order konzentriert zu Beginn des Anwendungsstarts durchgeführt. Dabei soll Demand Paging und Read Ahead umgangen werden. Durch die optimierte Sequentialität der Zugriffe sollen die Aktuatorenbewegungen der Blockgeräte minimiert und die Auslastung der Requests maximiert werden.

# Kapitel 2

# Design

Dieses Kapitel beschreibt die Konzeption und das Design eines Preload-Services (PLS), welches die in Kapitel 1.3 erläuterte Hypothese zur Reduktion der Zeit von Anwendungsstarts untersuchen soll.

Das PLS geht davon aus, dass bei einer Folge von Kaltstarts einer Anwendung auf die gleichen Blockgerätedaten zugegriffen wird. Mit anderen Worten: die Zugriffsfolge einer startenden Anwendung auf die Blockgeräte des Systems weist statischen Charakter auf und ist somit vorhersagbar.

Anders geht der Demand Paging Mechanismus des Kernels von einer begrenzten Vorhersagbarkeit einer Zugriffsfolge, basierend auf dem Prinzip der Lokalität, aus. Dies ist im Regelbetrieb sinnvoll, ignoriert aber den statischen Charakter der Zugriffsfolgen von Anwendungsstarts.

Das PLS versucht daher, den statischen Charakter der Blockgerätezugriffe während des Anwendungsstarts zu nutzen, um die Zugriffe vom Demand Paging Mechanismus zu entkoppeln. Die Zugriffe können dann gesammelt und hinsichtlich Blockgeräte I/O Performance optimiert zu Beginn eines Anwendungsstarts durchgeführt werden.

## 2.1 Allgemein

### 2.1.1 Spezifikation

Um die eingangs beschriebene Aufgabe durchführen zu können, muss das PLS folgende Aufgaben erfüllen:

- Aufzeichnen von Blockgerätezugriffe
- Umsortierung aufgezeichneter Zugriffe
- Persistenz der aufgezeichneten Daten
- Reproduktion der umsortierten Folge der Zugriffe
- Konfiguration und Administration des PLS

- Ausgabe verschiedener Informationen und Statistiken

Das PLS soll weitgehend autonom arbeiten. Modifikationen an den Kernelquellen sollen auf ein Minimum beschränkt werden, Plattform- und Softwareabhängigkeiten sollen vermieden werden. Wenn möglich werden daher archtekturunabhängige Schnittstellen des Kernels, wie z.B. das Virtual File System, verwendet.

### 2.1.2 Begriffsdefinitionen

#### PLS Komponenten

Das PLS besteht aus einer *Recording* Einheit zur Aufzeichnung der Blockgerätezugriffe verschiedener Anwendungsstarts. Die aufgezeichneten Zugriffe werden in einer *Datenbank* nach Anwendung gruppiert und sortiert verwaltet. Eine Import- und Exportfunktion ermöglicht Datenpersistenz bei einem Neustart des Systems. Die *Preloading* Einheit führt die aufgezeichneten Blockgerätezugriffe gesammelt zu Beginn eines neuerlichen Anwendungsstarts aus.

#### Zugriffsfolgen

Die konzentrierte Durchführung aller der für einen Anwendungsstart notwendigen Blockgerätezugriffe ist nur dann möglich, wenn die Zugriffe zu Beginn des Starts bekannt sind, d.h. bei früheren Starts dieser Anwendung aufgezeichnet wurden. Alle einem Anwendungsstart zuordenbare Blockgerätezugriffe werden in einer *Zugriffsfolge* organisiert.

Ein Zugriff umfasst in der Regel mehrere Speicherseiten. Für eine Neusortierung der Blockgerätezugriffe hinsichtlich maximaler Sequentialität werden diese in Einheiten vom Umfang einer Speicherseite, im Kernel definiert durch `PAGE_SIZE`, zerlegt. Ein so entstandenes Element einer Zugriffsfolge wird in weiterer Folge als *Page* bezeichnet.

### 2.1.3 Linux und das PLS

#### Modularisierung

Das PLS kann entweder in den Kernelquellen, als Kernelmodul, oder als Anwendung ausserhalb des Kernels realisiert werden.

Eine wichtige Regel des Kernelhackings besagt, dass so viel Funktionalität wie möglich aus dem Kernel in den User Space auszulagern ist. Da die Funktionalität des PLS teilweise eng mit dem Kernel verwoben ist, konnte dies nur begrenzt umgesetzt werden. So ist es z.B. naheliegend, Preloading und Recording im Kernel Space zu implementiert, da beide Komponenten eng mit dem bio-Layer verwoben sind.

Um die notwendigen Modifikationen am Kernel trotzdem zu minimieren, wurde die Anwendung als Modul implementiert. Die Kernelquellen müssen somit nur um zwei Einsprungpunkte in das PLS erweitert werden. Der erste führt das Preloading und die Initialisierung des Recordings einmalig zu Beginn eines Anwendungsstarts durch, der zweite dient zur Aufzeichnung eines Blockgerätezugriffs. Ist das PLS Modul geladen, wird jeder lesende Blockgerätezugriff aufgezeichnet.

#### Kernelversion

PLS wurde für die Speicherverwaltung von Linux 2.6.0 konzipiert. Zu Beginn der Diplomarbeit war die Versionsnummer des aktuellen Kernels 2.4.20. Auf Grund umfassender Modifikationen der Speicherverwaltung im Entwicklerkernel 2.5 entschloss ich mich, das PLS für den Kernel 2.5.x zu konzipieren und später auf 2.6.0 zu portieren. Die wichtigste Neuerung für das PLS war die Einführung des bio Layers [RL04, S.218] und damit der Möglichkeit, Multipage Requests zu realisieren.

### 2.1.4 Anwendung

Das PLS Modul kann beim Systemstart geladen werden. Auf einen Datenträger exportierte Zugriffsfolgen werden in die Datenbank importiert.

Ab den Zeitpunkt, an dem das PLS Modul geladen wurde, erfolgen Preloading und Recording kontinuierlich für alle Anwendungsstarts. Interaktion mit den User ist dazu nicht notwendig.

Bei Shutdown des Systems werden die aufgezeichneten Zugriffsfolgen auf einen nichtflüchtigen Datenträger exportiert und somit persistent gehalten. Das Laden und Entladen des PLS Moduls, sowie der Import und Export der Datenbank kann durch entsprechende Einträge in die Init-Skripte des Systems automatisiert werden.

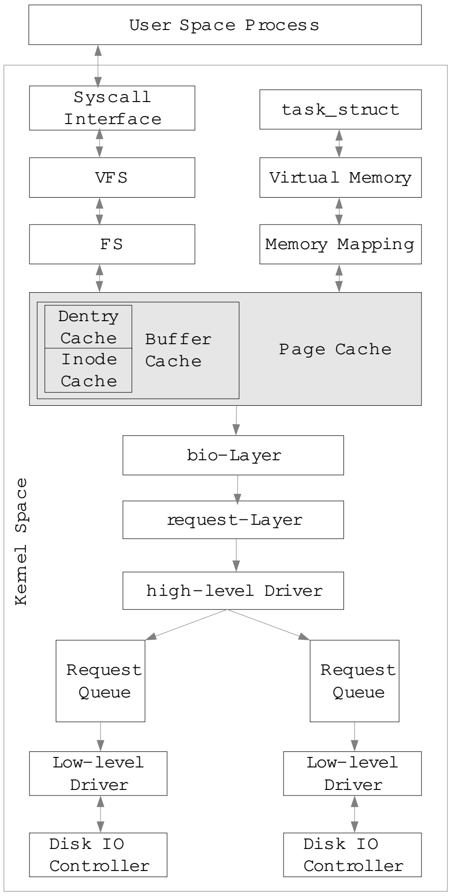


Abbildung 2.1: Blockgeräte-I/O System von Linux

## 2.2 Recording

Die Aufgabe des Recordings ist die Datengewinnung für das Preloading. Dazu soll die Zugriffsfolge einer startenden Anwendung auf die Blockgeräte des Systems aufgezeichnet werden. Bei späteren Anwendungsstarts soll die durch das Recording aufgezeichnete Zugriffsfolge beim Preloading hinsichtlich Blockgeräte I/O Performance optimiert reproduziert werden.

### 2.2.1 Kernelintegration

Die Kernfrage beim Design des PLS war, an welcher Stelle des Blockgeräte I/O Systems (Abb. 2.1) das Recording im Kernel integriert werden soll. Diese Entscheidung beeinflusst, wie vollständig die Zugriffsfolge einer startenden Anwendung aufgezeichnet werden kann und wo das Preloading zu implementieren ist.

Auf den verschiedenen Ebenen des Blockgeräte I/O Systems werden Blockgerätezugriffe auf den gleichen Datenbereich unterschiedlich dargestellt:

- *virtuelle Speicherverwaltung*: Memory Area und Seitenoffset
- *Page Cache*: Device, Inode und Seitenoffset
- *bio Layer*: Device, Startsektor und Sektorgrösse

Art und Typ dieser Parameter bestimmen, welche Methoden für das Preloading verwendet werden können. Findet das Recording z.B. in der virtuellen Speicherverwaltung statt, so kann das Preloading nicht im bio Layer implementiert werden. Der Grund dafür ist, dass beim Recording in der virtuellen Speicherverwaltung keine Informationen über die physikalische Anordnung der Daten am Blockgerät, d.h. den Startsektor, gewonnen werden können.

Zur Entscheidungsfindung wurden verschiedene Lösungen implementiert, die das Recording auf Ebene der virtuellen Speicherverwaltung, des Page Caches und des bio Layers implementieren. Die verschiedenen Lösungen werden in den folgenden Abschnitten diskutiert.

Um die Implementierungen vergleichen zu können, wurde die Qualität der gewonnenen Zugriffsfolgen beim Preloading hinsichtlich Requestauslastung und Sequentialität verglichen. Die Ergebnisse wurden am Beispiel des Anwendungsstarts von GNU Emacs in den Abbildungen 2.2, 2.3 und 2.5 visualisiert. Die Messmethode wird detailliert in Kapitel 3.5 beschrieben.

**Recording in der virtuellen Speicherverwaltung**

Der erste Ansatz dieser Arbeit war, das Recording dort durchzuführen, wo beim Start einer Anwendung eine Häufung der Blockgerätezugriffe verursacht wird: in der virtuellen Speicherverwaltung eines Prozesses.

Der virtuelle Speicher eines Prozesses wird in Linux mit Hilfe von Memory Areas organisiert. Allokiert ein Prozess Speicher, so erhält der Prozess nicht tatsächlich physikalischen Speicher zugeteilt, sondern nur das Recht, einen Ausschnitt seines virtuellen Adressraums zu nutzen. Dieser Ausschnitt wird im Kernel durch eine Memory Area repräsentiert. Jeder Prozess verwaltet eine Liste der von ihm genutzten Memory Areas.

Eine Memory Area kann ein Mapping einer Datei in den Speicher durchführen. Greift ein Prozess auf so ein Memory Mapping zu, so wird dieser Zugriff vom Kernel transparent in einen Zugriff auf die entsprechende Datei am Blockgerät übersetzt. Beim Start eines Prozesses werden Memory Mappings, z.B. eines Code- oder Datensegments, nach den Vorgaben der ausführbaren Datei durchgeführt. Weitere Memory Mappings können zur Laufzeit mit Hilfe des Systemcall `mmap()` definiert werden.

Das Konstrukt des virtuellen Speichers ermöglicht es, die tatsächliche Allokierung von physikalischen Speicher auf den letzt möglichen Zeitpunkt, den des ersten Zugriffs, zu verschieben. Diese Vorgehensweise wird Demand Paging genannt. Greift ein Prozess zum erstenmal auf die Seite einer Memory Area zu, so wird ein Page Fault generiert. Der Page Fault Handler versucht diesen Page Fault zu bearbeiten. Führt die betroffene Memory Area ein Memory Mapping durch so spricht man von Demand Loading. Sind die Daten des Memory Mappings nicht im Speicher, so müssen die Daten vom Backing Store, meistens ein Blockgerät, gelesen werden.

Das Recording auf Ebene der virtuellen Speicherverwaltung erfolgt im Page Fault Handler des Kernels. Im Zuge des Recordings wird jeder Page Fault einer startenden Anwendung aufgezeichnet. Die Page einer Zugriffsfolge wird durch Memory Area und Seitenoffset in dieser Area beschrieben. Entsprechend werden die Daten nach Memory Areas gruppiert und nach Offset sortiert.

Für das Preloading muss die `readpage` Methode einer Memory Area verwenden werden. Beim Anlegen einer Memory Area erfolgt das Preloading für alle, dieser Memory Area zuordenbare Pages der Zugriffsfolge. Das Preloading der restlichen Pages der Zugriffsfolge erfolgt, wenn die entsprechende Memory Area angelegt wird.

Aus diesem Ansatz resultieren verschiedene Nachteile:

- Das Recording erfasst nicht den gesamten Blockgeräte I/O einer startenden Anwendung. Es wird nur der durch Demand Paging verursachte Blockgeräte I/O aufgezeichnet. Der durch Dateioperationen, dem Buffer-, Dentry- oder Inodecache verursachter I/O wird hingegen nicht erfasst.

- Beim Recording in der virtuellen Speicherverwaltung kann keine Information über die tatsächliche Anordnung der Request-Daten auf den Blockgeräten gewonnen werden. Die Sortierung der Zugriffsfolge basiert auf logischer Adressierung und nimmt somit keine Rücksicht auf die physikalische Anordnung der Blockgerätedaten. Sie vertraut auf das Bestreben der Dateisysteme, die Blöcke einer Datei möglichst sequentiell am Blockgerät abzulegen. Bei grossen Dateien und hohen Füllgrad des Blockgeräts kann es aber notwendig sein, Dateien fragmentiert am Blockgerät abzulegen.

- Das Preloading der Zugriffsfolge einer startenden Anwendung erfolgt stückweise dann, wenn eine Memory Area angelegt wird. Somit werden nicht alle Pages einer Zugriffsfolge gleichzeitig geladen.

- Kriterium für die Reihenfolge des Preloading der Fragmente der Zugriffsfolge ist die Folge, in der die Memory Areas angelegt werden. Diese Reihenfolge hat aber nichts mit der Anordnung der Dateien der Memory Mappings am Blockgerät zu tun.

- Die Verwendung der `readpage` Methode einer Memory Area für das Preloading erzeugt Overhead, da für jede Page einer Zugriffsfolge ein Aufruf der Methode notwendig ist. Zum Zeitpunkt der Implementierung wurden noch keine Multipage Requests durch Memory Areas unterstützt.

Die angeführten Nachteile bewirken, dass die Zugriffsfolge des Preloadings auf die Blockgeräte des Systems keinen optimal sequentiellen Charakter aufweist,

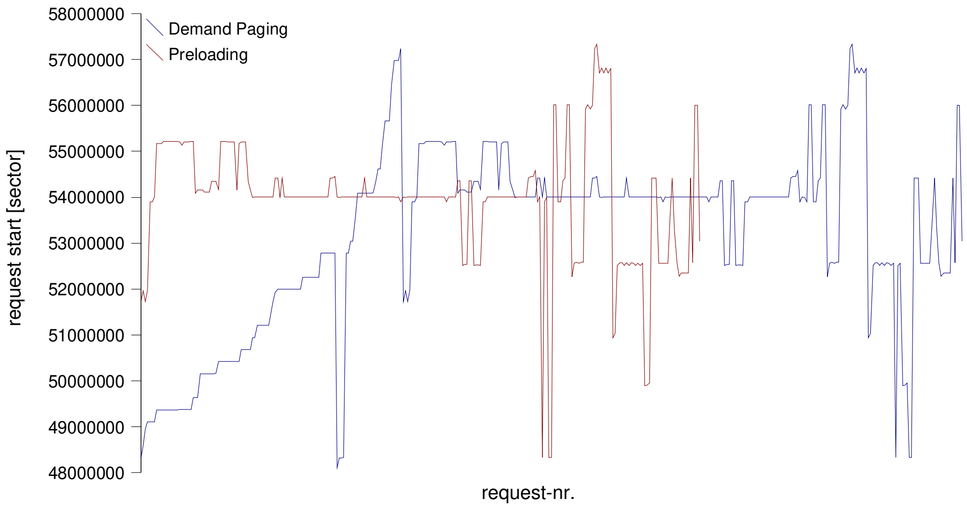


Abbildung 2.2: Zugriffsfolge GNU Emacs - Preloading in der virtuellen Speicherverwaltung

wie in Abbildung 2.2 ersichtlich ist. Dennoch ist die so sortierte Zugriffsfolge (rot) deutlich kürzer als die des Demand Pagings (blau). Für das Laden der gleichen Datenmenge wurden daher weniger Requests benötigt, was auf eine bessere Auslastung der Requests schliessen lässt. Dies zeigt wiederum, dass die sortierte Zugriffsfolge sequentieller ist, als die des Demand Pagings.

### Recording im Page Caches

Ziel beim Design der nächsten Version war es, möglichst den gesamten Blockgeräte I/O einer startenden Anwendung zu erfassen. Weiters sollte das Preloading der Zugriffsfolge nicht mehr fragmentiert, sondern konzentriert zu Beginn eines Anwendungsstarts erfolgen.

Dazu musste das Recording um einige Ebenen im Blockgeräte I/O System des Kernels nach unten verlagert werden. Die erste Ebene unterhalb der virtuellen Adressverwaltung, welche vom gesamte Blockgeräte I/O passiert wird, ist die Ebene der Caches.

Der wichtigste Systemcache des Kernels ist der Page Cache. Alle anderen Caches wie Buffer-, Inode- oder Dentry Cache verwalten ihre Daten ebenfalls im Page Cache. Die Daten dieser Caches sind Untermengen der Daten des Page Caches. Somit werden alle von einem Blockgerät gelesenen Daten im Page Cache verwaltet.

Soll der gesamte Blockgeräte I/O beim Recording an zentraler Stelle erfasst werden, so ist dies unterhalb der virtuellen Adressverwaltung zum ersten Mal im Page Cache möglich. Recording und Preloading wurden daher aus der Ebene der virtuellen Speicherverwaltung in die der Caches, genauer in den Page Cache verlagert.

Die Page einer Zugriffsfolge wird im Page Cache eindeutig durch Device, Inode und Seitenoffset in dieser Inode beschrieben. Für das Preloading wird die aufgezeichnete Zugriffsfolge nach Device, Inodenummer und Seitenoffset sortiert.

Im Entwicklerkernel 2.5.x wurde eine Schnittstelle für Multipage Requests implementiert. Dadurch ist es jetzt möglich, beim Preloading ganze Seitencluster mit einem Request zu lesen, was den Overhead des Preloadings reduziert.

Beim Preloading muss nun für jede Page der Zugriffsfolge ein Page Frame allokiert, mit dem Seitenoffset initialisiert und in den Page Cache eingefügt werden. Anschliessend werden, alle eine Inode betreffenden Pages mit einen einzelnen Multipage Request eingelesen. Die für das Preloading verwendete Funktion `mpage_readpages` verlangt dazu als Argument das Inode Objekt und ein Array mit den allokierten Page Frames.

Aufgabe der Funktion ist es, aus den Multipage Requests einen oder mehrere bio-Requests zu konstruieren. Dazu prüft die Funktion mit Hilfe der Blockmapperfunktion des Dateisystems, ob die Daten des Multipage Requests sequentiell am Blockgerät angeordnet sind. Wird eine asequentielle Datenanordnung entdeckt, oder übersteigt die Kapazität des Multipage Requests die Maximalkapazität eines bio-Requests, wird der Multipage Request auf mehrere bio-Requests aufgeteilt.

Die Analyse der Zugriffsfolge des Preloadings und ein Profiling zeigten verschiedene Probleme dieses Ansatzes:

- Wie beim Recording in der virtuellen Speicherverwaltung können auch hier keine Informationen über die physikalische Anordnung der Requestdaten am Blockgerät gewonnen werden. Die gewählte Sortierung muss nicht unbedingt eine Zugriffsfolge mit maximaler Sequentialität beim Preloading bedeuten. Aufgrund möglicher Dateisystemfragmentierung können logisch sequentielle Daten physikalisch asequentiell am Blockgerät angeordnet sein.

- Inodes werden für jedes Device eindeutig durch ihre Inodenummer identifiziert. Aus der fortlaufenden Inodenummer zweier Dateien folgt nicht, dass deren Daten auch sequentiell am Blockgerät abgelegt worden sind.

- Multipage Requests im Page Cache unterliegen der Einschränkung, dass die Daten eines Requests von einer Datei stammen müssen. Die Daten zweier unmittelbar hintereinander am Blockgerät abgelegten Dateien können nicht mit einem Request eingelesen werden.

- Der Preloading Prozess wird immer wieder durch synchrone Requests des Dateisystems, genauer der Blockmapperfunktion des Dateisystems, nach Indirektionsblöcken gestoppt. Da die Request Queue beim Preloading einen hohen Füllgrad aufweist und die eintreffenden Requests, falls kein Request Clustering durchgeführt werden kann, nach dem FIFO Prinzip bearbeitet werden, kann die Bearbeitung der synchronen Requests sehr lange dauern. Dem Preloading Prozess wird der Prozessor solange entzogen, bis der synchrone Request bearbeitet wurde.

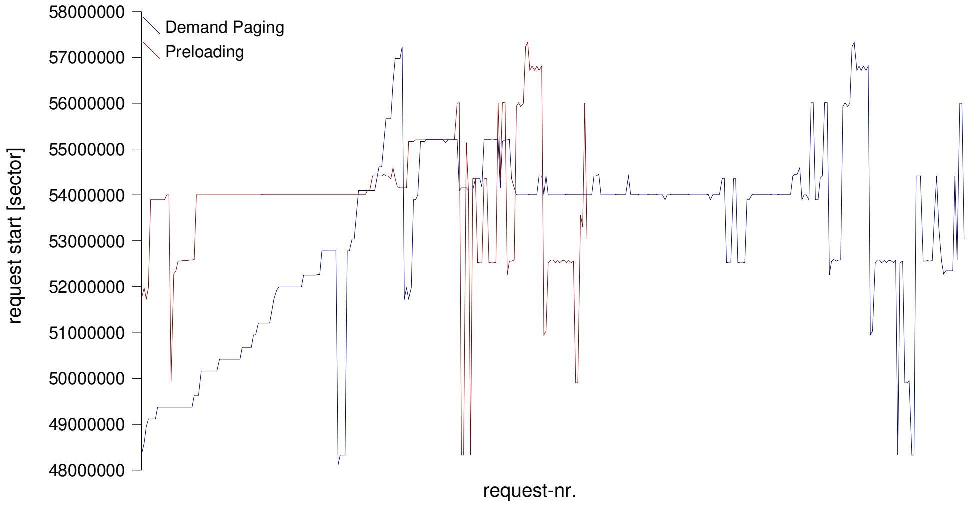


Abbildung 2.3: Zugriffsfolge GNU Emacs - Preloading im Page Cache

- Die ungeplanten synchronen Requests der Blockmapperfunktion stören den sequentiellen Charakter der Zugriffsfolge und mindern somit die Blockgeräte I/O Performance.

Da das Recording oberhalb der Blockmapperfunktion des Dateisystems arbeitet, kann keine Information über die tatsächliche Anordnung der Daten einer Page einer Zugriffsfolge am Blockgerät gewonnen werden. Daher ist die Sortierung nach Inode und Seitenoffset nicht optimal, wie auch die Analyse der Zugriffsfolge des Preloadings für GNU Emacs (Abb. 2.3) zeigt.

Dennoch ist eine Verbesserung gegenüber der Version, die das Recording in der virtuellen Speicherverwaltung durchführt (Abb.2.2), erkennbar. Vergleicht man die Zugriffsfolgen beim Preloading (rot), so ist zu sehen, dass durch Recording und Preloading im Page Cache eine kürzere Zugriffsfolge erzeugt wird. Daraus kann man auf eine gesteigerte Requestauslastung durch verbesserte Sequentialität schliessen. Die verbesserte Sequentialität ist auch optisch, durch Vergleich der beiden Zugriffsfolgen des Preloadings zu sehen.

### Recording im bio Layer

Durch Recording im bio Layer soll die I/O Performance beim Preloading gegenüber der letzten Version verbessert werden. Das grösste Potential dazu liegt einerseits in einer verbesserten Sequentialität der Zugriffsfolge beim Preloading und andererseits in der Vermeidung der Blockierung des Preloading Prozesses durch synchrone Requests.

Um eine Zugriffsfolge mit sequentielleren Charakter als in den ersten beiden Versionen zu erhalten, muss das Recording dorthin verlagert werden, wo Informationen über die physikalische Anordnung der Requestdaten am Blockgerät verfügbar sind. Diese Information kann in weiterer Folge zu einer Sortierung der

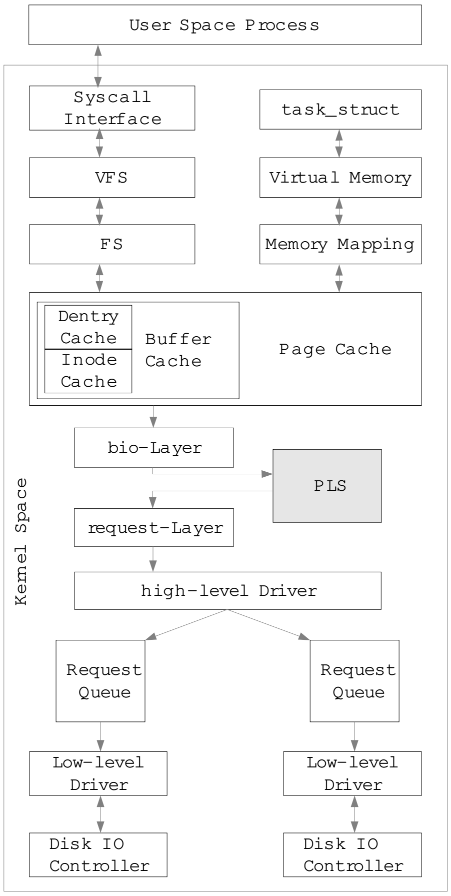


Abbildung 2.4: Recording im Blockgeräte-I/O System von Linux

Zugriffsfolge unter Berücksichtigung der physikalischen Anordnung der Daten am Blockgerät verwendet werden. Eine solche Sortierung müsste theoretisch optimal sequentielle Zugriffsfolgen beim Preloading zur Folge haben.

Die physikalische Anordnung der Daten am Blockgerät wird durch die Blockmapperfunktion des Dateisystems ermittelt. Das Blockmapping bildet das Tripel Device/Inode/Seitenoffset auf den Tupel Device/Sektoroffset ab. Das Ergebnis des Blockmappings wird zur Konstruktion von bio-Requests aus den Multipage Requests des Page Caches verwendet. Verlagert man das Recording in den bio Layer (Abb. 2.4), kann man das Ergebnis der Blockmapperfunktion zur optimalen Sortierung einer Zugriffsfolge verwenden.

Synchrone Requests werden ausschliesslich von Page-, Inode- und Buffer Cache, z.B. durch Indirektion, generiert. Will man diese vermeiden, so muss man das Preloading unterhalb der Caches durchführen. Für das Preloading bietet sich daher ebenfalls der bio Layer an.

Ein bio-Request beschreibt den Transfer eines sequentiellen Datenbereichs am Blockgerät in einen nicht notwendigerweise sequentiellen Speicherbereich. Ein

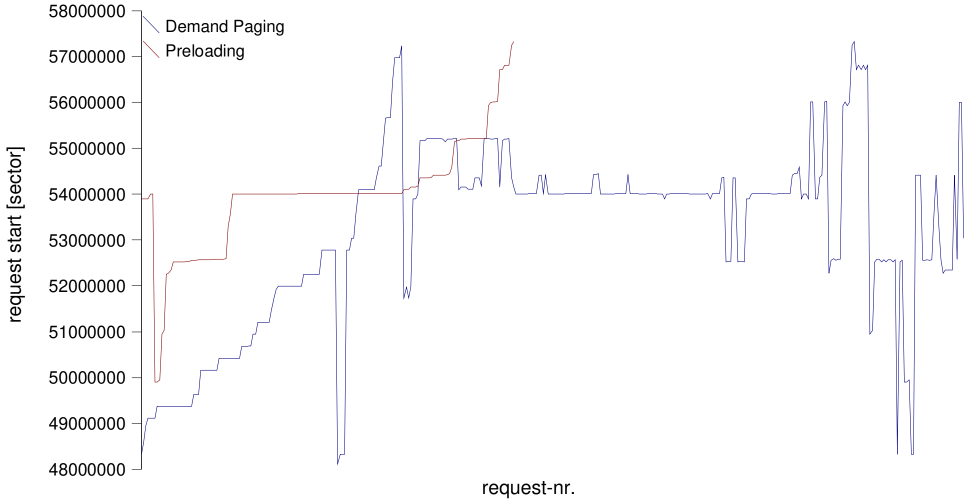


Abbildung 2.5: Zugriffsfolge GNU Emacs - Preloading im bio Layer

Request ist nicht mehr nur auf die Daten einer Datei eingeschränkt, wie beim Preloading im virtuellen Adressbereich, bzw. im Page Cache. Im bio Layer existiert keine Einschränkung für die Konstruktion von Requests, ausser dass die Daten eines Requests sequentiell am Blockgerät angeordnet sein müssen.

Durch diese Faktoren wird die Requestauslastung weiter verbessert, wie Abbildung 2.5 zeigt. Die Zugriffsfolge des Preloadings (rot) zeigt einen annähernd optimal sequentiellen Verlauf. Mit dem gewählten Ansatz ist daher eine weitere Verbesserung der Blockgeräte I/O Performance kaum mehr möglich. Für eine weitere Performancesteigerung ist ein Blockreordering der Daten einer Zugriffsfolge notwendig.

Die Vorteile der Verlagerung von Recording und Preloading in den bio Layer werden mit einer geringeren Flexibilität hinsichtlich der Anordnung der Datenblöcke am Blockgerät erkauft. Das Ergebnis der Blockmapperfunktion wird zur Sortierung verwendet und muss somit in den Zugriffsfolgen gespeichert werden. Verändert sich die Anordnung der Blockgerätedaten, z.B. durch Defragmentierung, so müssen die aufgezeichneten Zugriffsfolgen verworfen werden.

Da das PLS eine experimentelle Anwendung ist und die maximal mögliche Minimierung der Startzeit von Anwendungen gezeigt werden soll, wird dieser Nachteil in Kauf genommen. Recording und Preloading werden wie beschrieben im bio Layer des Kernel I/O Systems durchgeführt.

### 2.2.2 Recording Methoden

Im PLS sind zwei unterschiedliche Recording-Methoden verfügbar:

- *PID gebundenes Recording*: für jeden Anwendungsstart wird eine eigene Zugriffsfolge angelegt. Die Zuordnung eines Requests zu einer Zugriffsfolge erfolgt mit Hilfe der PID des Anwendungsprozesses.

- *Statisches Recording*: mehrere Anwendungsstarts werden in einer Zugriffsfolge verwaltet.

Im Normalfall wird PID gebundenes Recording durchgeführt. Die Blockgerätezugriffe werden nach Prozess getrennt gesammelt. Für jeden Anwendungsstart wird eine eigene Zugriffsfolge gespeichert.

Beim statischen Recording werden alle Blockgerätezugriffe ab Aktivierung des statischen Recordings einer einzigen Zugriffsfolge zugeordnet. Diese Variante ist beim Recording einer immer wiederkehrenden Folge von Anwendungsstarts sinnvoll. Beispiel hierfür sind der Systemstart oder der Start des X-Servers incl. eines Window-Managers. Das Preloading mehrerer Zugriffsfolgen aufeinanderfolgender Anwendungsstarts kann dadurch in Zukunft gemeinsam beim Start der ersten Anwendung durchgeführt werden. Durch den grösseren Umfang zusammengesetzter Zugriffsfolgen ergeben sich weitere Optimierungsmöglichkeiten hinsichtlich Reqeustauslastung, bzw. Sequentialität.

Die Methode des Recordings wird manuell durch ein Flag gesteuert, welches mit Hilfe der `ioctl` Schnittstelle des PLS-Moduls kontrolliert werden kann. Ist das Flag gesetzt, wird statisches Recording, sonst PID gebundenes Recording durchgeführt.

### 2.2.3 Der Recording Prozess

Aus Sicht einer startenden Anwendung ist das Recording ein zustandsbehafteter Prozess. Das Recording wird zu Beginn des Anwendungsstarts initialisiert und ist während des gesamten Anwendungsstarts aktiv. Der Vorgang des Recordings kann in drei Abschnitte unterteilt werden: die Initialisierung des Recordings, das Recording der Blockgerätezugriffe und das Beenden des Recordings.

#### Recording Sessions

Zu einem Zeitpunkt können mehrere Anwendungsstarts aktiv sein, wobei für jeden dieser Anwendungsstarts ein Recording möglich sein soll. Das Recording eines Anwendungsstarts wird daher durch eine *Recording Session* verwaltet. Eine Recording Session wird durch die Prozess ID (PID) des entsprechenden Prozesses der startenden Anwendung eindeutig identifiziert.

Alle zu einen Zeitpunkt aktiven Recording Sessions werden in einer Liste verwaltet. Zur performanten Suche werden die Recording Sessions zusätzlich durch eine Hashtable indiziert.

#### Initialisierung einer Recording Session

Ist das PLS Modul geladen und das Recording aktiviert, so wird im Falle von PID bezogenen Recording zu Beginn eines jeden Anwendungsstarts eine Recording Session initialisiert.

Die Funktion `do_execve` des Kernels behandelt alle Systemaufrufe zur Prozesserzeugung, wie `exec` oder `execve`. Es liegt daher nahe, die Initialisierung einer Recording Session am Anfang dieser Funktion durchzuführen. Den Einsprungpunkt in das PLS definiert ein globaler Zeiger, welcher beim Laden des PLS Moduls auf die Funktion zur Initialisierung einer Recording Session gesetzt wird. Beim Entladen des PLS-Moduls wird dieser Zeiger wieder auf Null gesetzt.

Zu Beginn eines jeden Anwendungsstarts wird nun in der Funktion `do_execve` geprüft, ob der Einsprungpunkt in das PLS gesetzt, d.h. ob das PLS Modul geladen ist. Wenn ja, wird die durch den Zeiger bezeichnete Methode zur Initialisierung einer Recording Session aufgerufen.

In dieser Methode wird ein Recording Session Objekt allokiert, initialisiert und in die Liste, bzw. Hashtable zur Verwaltung der Recording Session Objekte eingefügt. Im Zuge der Initialisierung wird die entsprechende Zugriffsfolge in der Datenbank gesucht, und der neuen Recording Session zugewiesen.

Erfolgt für diese Anwendung zum ersten Mal ein Recording, so wird eine neue Zugriffsfolge in der Datenbank angelegt, andernfalls wird eine vorhandene Zugriffsfolge ergänzt.

### Recording von Blockgeräte I/O

Das Recording des Blockgeräte I/Os wird zwischen bio- und Request Layer durchgeführt (Abb. 2.4). Die Funktion `submit_bio` des Kernels bildet die zentrale Schnittstelle zwischen diesen beiden Layern. Die Aufgabe dieser Funktion ist es, einen bio Request an den Request Layer zu übergeben. Ist das PLS-Modul geladen und das Recording aktiviert, so werden in der Funktion `submit_bio` alle bio-Requests über die Recording Funktion des PLS-Moduls umgeleitet.

Als Einsprungpunkt in das PLS wird ein globaler Zeiger definiert, welcher, wenn das PLS Modul geladen ist, auf die Recording Funktion des Moduls zeigt. Der Zeiger wird beim Laden des PLS Moduls gesetzt. Jeder Datentransfer vom oder zum Blockgerät passiert den bio Layer und somit die Funktion `submit_bio`. In `submit_bio` wird geprüft, ob der Zeiger auf die Recording Funktion gesetzt ist, d.h. ob das PLS Modul geladen ist. Ist dies der Fall, so wird für jeden lesenden bio Request die Recording Funktion des Moduls aufgerufen. Einziges Argument ist ein Zeiger auf den bio Request.

Der aufzuzeichnende bio Request beschreibt den Transfer eines sequentiellen Datenbereichs am Blockgerät in einen nicht notwendigerweise sequentiellen Speicherbereich. Der sequentielle Datenbereich am Blockgerät wird durch Startsektor und Sektoranzahl, der Speicherbereich durch einen Vektor von Page Frames, genannt bio Vektor, beschrieben. Je nach Blockgrösse des Blockgeräts sind in diesen Page Frames die Daten eines oder mehrerer Datenblöcke des Blockgeräts gespeichert. Die Zuordnung eines Datenbereichs am Blockgerät zu einer Page Frame erfolgt durch das Dateisystem, genauer durch die Blockmapperfunktion.

Die Recording Funktion des PLS Moduls zerlegt nun einen übergebenen bio Request in die Elemente des bio Vektors und errechnet für jedes Element den

Startsektor am Blockgerät. Die so gewonnene Zuordnung einer Sektoradresse zum Tripel Device/Inode/Seitenoffset wird in der Page einer Zugriffsfolge gespeichert. Wurde der bio Request bearbeitet, so wird im Recording Session Objekt noch die aktuelle Kernelzeit gespeichert.

Die Zerlegung der bio Requests ermöglicht eine Umordnung der Pages einer Zugriffsfolge auf Basis der Sektoradresse. Die resultierende Zugriffsfolge für das Preloading weist in der Regel einen erheblich sequentielleren Charakter auf, als die Zugriffsfolge des Demand Paging (Abb. 2.5).

### Ende einer Recording Session

Per Definition soll das Recording nur die Blockgerätezugriffe während der Startphase einer Anwendung aufzeichnen. Weiters soll das Recording automatisiert, d.h. ohne Interaktion mit dem User erfolgen. Das PLS muss daher erkennen, wann ein Anwendungsstart beendet ist.

Eine Möglichkeit wäre, dass die Anwendung dem PLS das Ende der Startphase signalisiert. Dazu müssten aber alle Anwendungen, für die ein Recording durchgeführt werden soll, modifiziert werden, was nicht praktikabel ist.

Um das Ende eines Anwendungsstarts zu erkennen, nutzt das PLS daher die für eine startende Anwendungen charakteristische Häufung Page Faults. Daraus folgt auch eine Häufung der Requests an die Blockgeräte des Systems. Eine Recording Session wird genau dann beendet, wenn die Zeitspanne zwischen zwei der Recording Session zuordenbare bio Requests einen definierten Grenzwert überschreitet. Dieser Grenzwert kann über das `ioctl` Interface des PLS zur Laufzeit modifiziert werden. Beim Laden des PLS-Moduls ist der Grenzwert auf zwei Sekunden voreingestellt.

Ob eine Recording Session zu beenden ist, wird periodisch durch einen Timer überprüft, der beim Laden des PLS-Moduls gestartet wird. Die Interrupt Funktion des Timers iteriert über die Liste der aktiven Recording Sessions. Dabei wird für jede Session geprüft, ob die maximal erlaubte Zeitspanne zwischen zwei bio Requests überschritten wurde. Wurde eine Zeitüberschreitung festgestellt, so wird die Recording Session aus der Liste und der Hashtable zur Verwaltung der Recording Sessions entfernt und freigegeben. Das Recording ist somit für diese Anwendung beendet.

## 2.3 Datenbank

Die Aufgabe der Datenbank ist die Verwaltung der beim Recording gewonnenen Zugriffsfolgen. Für späteres Preloading müssen diese Zugriffsfolgen über Systemstarts hinweg verfügbar gemacht werden.

### 2.3.1 Art der Datenverwaltung

Für die Verwaltung und Speicherung der aufgezeichneten Zugriffsfolgen sind verschiedene Ansätze möglich. Die Daten der Zugriffsfolgen können entweder in eigen entwickelten Lösungen, oder durch bereits bestehende Anwendungen, wie z.B. einer SQL Datenbank, verwaltet werden. Bei einer Eigenentwicklung kann die Verwaltung der Daten im Speicher, oder auf einem Blockgerät erfolgen. Bei der Verwaltung im Speicher ist zwischen User- und Kernelspace zu unterscheiden. Weiters sind Mischformen möglich.

Eine Eigenentwicklung kann im User-, oder Kernelspace implementiert werden. Verwaltet man die Daten im Userspace, so minimiert man den Speicherverbrauch des PLS im Kernelspace. Da jedoch Recording und Preloading im Kernelspace implementiert sind, steigt der Overhead durch Kommunikation und den Transfer der benötigten Zugriffsfolgen in den Kernelspace. Bei der Verwaltung der Zugriffsfolgen im Kernelspace fällt dieser Overhead weg.

Verwaltet man alle Daten im Speicher, sind die Speicheranforderungen entsprechend hoch. Besonders fällt dies bei der Datenhaltung im Kernelspace ins Gewicht, da das ganze Design des Kernels auf minimale Speicheranforderungen ausgelegt ist. Die Speicheranforderung der Datenbank könnte dadurch vermindert werden, dass nur die gerade von Recording und Preloading benötigten Daten im Speicher gehalten werden. Daraus resultiert aber wieder Overhead, da beim Preloading und Recording benötigte Zugriffsfolgen erst vom langsamen Blockgerät gelesen werden müssen.

Bei der Verwendung einer Fremdlösung könnte eine der zahlreich verfügbaren SQL-konformen Datenbanken verwendet werden. Diese bieten optimierte Datenstrukturen für Such- und Einfügeoperationen und es wird viel Funktionalität aus dem Kernel- in den Userspace verlagert. Weiters muss keine Import- und Exportfunktion implementiert werden.

Da die in dieser Arbeit vorgestellte Version des PLS experimentellen Charakter besitzt und der maximal mögliche Performancegewinn beim Anwendungsstart gezeigt werden soll, wurde eine einfache Datenbank im Kernelspace implementiert. Alle Zugriffsfolgen der Datenbank werden im Speicher verwaltet und nur beim Export auf einen nichtflüchtigen Speicher des Systems geschrieben.

Hinsichtlich Speicheranforderung ist dies sicher die ungünstigste Variante, da alle Daten im Kernelspace verwaltet werden. Dafür entfällt der Overhead durch Kommunikation und Datentransfer mit dem, bzw. vom Userspace. Soll das PLS später über den Status einer Konzeptstudie weiter entwickelt werden, muss die Datenbank dennoch in den Userspace ausgelagert werden. Eine so speicherintensive

Komponente wie eine Datenbank würde von der Community nicht im Kernelspace toleriert werden.

### 2.3.2 Datentypen

Um die Datenbank modellieren zu können, muss zuerst untersucht werden, welche Datentypen verwaltet und gespeichert werden sollen.

Eine Page einer Zugriffsfolge beschreibt den Transfer eines kontinuierlichen Datenbereichs am Blockgerät in einen kontinuierlichen Speicherbereich. Aus Sicht des Kernels wird eine Page Frame mit einem Memory Mapping beschrieben. Die Daten dieses Page Frames sind ein integraler Teil einer im Kernel durch eine Inode repräsentierten Datei. Eine Datei ist auf der Partition eines Blockgeräts gespeichert, im Kernel repräsentiert durch ein Device.

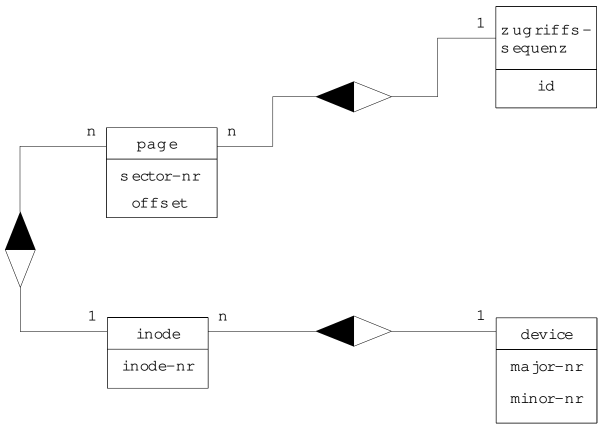


Abbildung 2.6: Beziehungen zwischen den Objekttypen der Datenbank

Abbildung 2.6 versucht diese Beziehungen in UML Notation zu modellieren. Die Page einer Zugriffsfolge wird einerseits eindeutig durch den Startsektor am Blockgerät, andererseits ebenfalls eindeutig durch Inode, Device und Seitenoffset identifiziert. Die Zuordnung dieser beiden eindeutigen Schlüssel ist das Ergebnis der Blockmapperfunktion des Dateisystems und muss, da das Preloading unterhalb dieser Blockmapperfunktion agiert, in der Datenbank abgebildet werden. Die Datenbank muss also Zugriffsfolgen, deren Pages, sowie Inode- und Device Objekte verwalten können.

#### Identifizierung einer Zugriffsfolge

Um eine Zugriffsfolge bei späteren Anwendungsstarts reproduzieren zu können, muss diese eindeutig einer Anwendung zugeordnet werden können. Jedes Zugriffsfolgenobjekt in der Datenbank benötigt dazu eine eindeutige ID. Ein Problem stellen dabei die vielen Faktoren beim Anwendungsstart dar, welche den Ausführungspfad und somit die Zugriffsfolge beeinflussen können:

- Verschiedene Kommandozeilenoptionen ergeben verschiedene Ausführungspfade im Code und somit verschiedene Zugriffsfolgen beim Start.
- Ebenso können die Konfigurationsdateien der Anwendung den Ausführungspfad beim Start beeinflussen.
- Die Anwendung benutzt Environment Variablen. So kann z.B. das Kommando `tree` je nachdem in welchem Verzeichnis es aufgerufen wird, definiert durch die Environment Variable `PWD`, verschiedene Zugriffsfolgen zur Folge haben.
- Die Anwendung kann durch verschiedene User gestartet werden. Ob z.B. root oder ein normaler User eine Anwendung startet, kann Einfluss auf den Ausführungspfad und somit die Zugriffsfolge der Anwendung haben.
- uva ...

Um eine Zugriffsfolge eindeutig zu identifizieren, müssten all diese Faktoren in die ID einfließen, was nicht praktikabel ist. Um die Anzahl der Zugriffsfolgen und die Grösse der Datenbank, sowie die Zeit für Import und Export dennoch einzuschränken, ist eine gewisse Unschärfe bei der Identifizierung einer Zugriffsfolge sogar erwünscht.

Das Preloading und die Initialisierung einer Recording Session werden zu Beginn des Anwendungsstarts in der Funktion `do_execve` durchgeführt. Der Funktion werden als Argument der Pfadname der ausführbaren Datei, die Argumente und die Umgebungsvariablen übergeben. Dies sind die einzigen Informationen über eine startende Anwendung, die dem PLS zur Verfügung stehen. Die eindeutige ID einer Zugriffsfolge muss also aus diesen Informationen gewonnen werden.

Zur Generierung der ID einer Zugriffsfolge werden daher der Pfadname der ausführbaren Datei und die Kommandozeilen-Parameter der startenden Anwendung verwendet. Aus diesen Komponenten wird mittels der Hashfunktion `jhash` [1] des Kernels eine numerische ID berechnet und im Zugriffsfolge Objekt gespeichert.

### Sortierung einer Zugriffsfolge

Wie am Beispiel von GNU Emacs in Abbildung 2.5 zu sehen ist, zeigen die beim Recording aufgezeichneten Zugriffsfolgen (blau) stark asequentiellen Charakter. Um eine Zugriffsfolge beim Preloading möglichst sequentiell zu gestalten, muss also die Reihenfolge der Blockgerätezugriffe geändert werden.

Beim Preloading werden alle Pages einer Zugriffsfolge zu Beginn des Anwendungsstarts eingelesen. Man könnte nun die Sortierung der vom Preloading generierten Requests dem Disk Scheduler des Kernels überlassen. Der Disk Scheduler versucht ebenfalls, durch Sortierung und Clustering der Requests in der Requestqueue möglichst sequentielle Zugriffsfolgen zu erzeugen. Der Horizont des Disk

[1] siehe `include/linux/jhash.h`

Schedulers hinsichtlich Optimierung ist dabei aber auf die Requests in der Requestqueue beschränkt. Da aber durch die Beschränkung der Anzahl der Requestobjekte oft nicht alle Requests zum Einlesen einer Zugriffsfolge auf einmal in die Requestqueue gestellt werden können, kann das congestion Verhalten einer Requestqueue zu nicht optimal-sequentiellen Zugriffsfolgen führen. Entsprechend sinkt die I/O Performance beim Preloading.

Das PLS ordnet daher die beim Recording eines Anwendungsstarts gewonnenen Zugriffsfolge beim Einfügen in die Datenbank neu. Dazu werden die Pages einer Zugriffsfolge nach ihrem Startsektor aufsteigend sortiert. Da der Startsektor der erste Sektor des Datenbereichs einer Page am Blockgerät ist, definiert dieser die Lage der Daten verschiedener Pages zueinander. Die Sortierung nach Startsektoren ergibt also theoretisch eine optimal sequentielle Zugriffsfolge.

### 2.3.3 Datenstrukturen

Da als Datenbank zur Verwaltung aufgezeichneter Zugriffsfolgen eine Eigenentwicklung im Kernelspace gewählt wurde, stellte sich die Frage nach den für die Anforderungen des PLS optimalen Datenstrukturen. Ein Anwendungsstart unter Verwendung des PLS setzt sich aus Preloading, Anwendungsstart und Recording zusammen. Für eine optimale Gesamtperformance müssen daher bei der Auswahl der Datenstrukturen für die Datenbank die Bedürfnisse von Preloading und Recording berücksichtigt werden.

#### Zugriffsszenarien auf die Datenbank

Hinsichtlich Datenbankzugriff können bei einen Anwendungsstart unter Verwendung des PLS zwei Szenarien für Preloading und Recording unterschieden werden.

Wurde noch nie ein Recording durchgeführt, so ist für einen Anwendungsstart keine Zugriffsfolge in der Datenbank gespeichert. Die Datenbankoperationen des Preloadings beschränkt sich auf die Suche in der Menge der gespeicherten Zugriffsfolgen. Die Suche bleibt erfolglos, der Preloading Prozess wird beendet und eine Recording Session wird initialisiert. Dabei wird eine neue Zugriffsfolge angelegt und in die Datenbank eingefügt. Im Zuge des Recordings wird jeder lesende Blockgerätezugriff in Pages zerlegt. Für jede Page muss geprüft werden, ob diese bereits in der Zugriffsfolge vorhanden ist. Falls nicht, so wird die Page nach Startsektor sortiert in die Zugriffsfolge eingefügt.

Wurde bereits ein Recording für diesen Anwendungsstart durchgeführt, so findet der Preloading Prozess die entsprechende Zugriffsfolge in der Datenbank. Zur Rekonstruktion der Zugriffsfolge iteriert dann der Preloading Prozess über alle Pages der Zugriffsfolge. Nach dem Preloading wird wieder eine Recording Session für diesen Anwendungsstart initialisiert. Zum Zeitpunkt der Initialisierung haben demnach bereits alle bio Requests des Preloadings den Einsprungpunkt ins Recording passiert. Vorhandene Zugriffsfolgen werden durch nochmaliges Recording bestenfalls mit noch nicht erfassten Blockgeräterequests ergänzt. Wurde eine Zugriffsfolge schon beim ersten Recording vollständig erfasst, so sind nach dem Preloading für den Anwendungsstart keine weiteren Blockgeräterequests notwendig. Entsprechend geht die Anzahl der Datenbankoperationen bei mehrmaligen Recording gegen Null.

Für das Preloading ist eine performante Suche in der Menge der Zugriffsfolgen wichtig. In weiterer Folge muss der Preloading Prozess die Möglichkeit haben, auf die Folge der sortierten Pages der gesuchten Zugriffsfolge zuzugreifen.

Das Recording benötigt zur Kontrolle, ob die Page einer Zugriffsfolge bereits erfasst wurde, eine effektive Suche in der Menge der Pages einer Zugriffsfolge. Weiters müssen neue Pages effektiv in die sortierte Menge der Pages einer Zugriffsfolge eingefügt werden können.

### Verwendete Datenstrukturen

Die Auswahl der Datenstrukturen der Datenbank erfolgt auf Basis der im letzten Abschnitt beschriebenen Zugriffsszenarien.

Die wichtigste Datenbankoperation für Preloading und Recording ist die Suche. Alle Objekte der Datenbank werden daher zur performanten Suche in verschiedenen Hashtables verwaltet. Eine Hashtable wird durch ein statisches Array aus Zeigern auf Objekte gebildet. Die Grösse für das Array einer Hashtables zur Verwaltung der Pages einer Zugriffsfolge ist von der Anzahl der Pages der Zugriffsfolge abhängig. Sie wird beim Reimport der Datenbank so gewählt, dass die Hashtable eine Fill Rate von 50% aufweist. Die Grösse aller anderen Hashtable-Arrays ist durch Konstanten definiert. Die Kollisionsbehandlung erfolgt mit Hilfe von Listen. Dazu wird die Standard Listenimplementierung des Kernels, eine doppelt verkettete Liste [2], verwendet.

Beim Preloading und Export muss weiters die Möglichkeit des sequentiellen Zugriffs auf die Objekte der Datenbank gegeben sein. Alle Objekte der Datenbank werden daher zusätzlich in verschiedenen Listen verwaltet. Wie bei den Hashtables wird die Listen-API des Kernels verwendet.

### Besonderheiten bei der Verwaltung der Pages einer Zugriffsfolge

Im Gegensatz zu allen anderen Objekten der Datenbank müssen die Pages einer Zugriffsfolge sortiert nach Startsektor verwaltet werden. Dazu wird ebenfalls eine Liste verwendet, obwohl diese Datenstruktur zur sortierten Verwaltung einer in der Auftrittswahrscheinlichkeit gleichverteilten Folge von Requests nicht optimal ist. Einige Besonderheiten der Zugriffsfolgen sprechen dennoch für die Verwendung einer Liste zur Organisation der Pages.

So profitiert die Folge der Pages einer Zugriffsfolge von zeitlicher und örtlicher Lokalität der Blockgerätezugriffe. Eine beim Recording aufgezeichnete Zugriffsfolge weisst meist stückweise sequentiellen Charakter auf und ist somit nicht gleichverteilt.

[2] siehe Linux 2.6.0: `include/linux/list.h`

| Anwendung | Pages ZF | Hits |
|---|---|---|
| GNU Emacs 21.2.1 | 1468 | 1243 (85%) |
| Mozilla 1.0.2 | 4097 | 3625 (88%) |
| OpenOffice 1.0.1 | 11506 | 10685 (93%) |

Tabelle 2.1: 1-Elemente Cache Hits

Das kann man mit einem 1-Elemente Cache ausnützen, welcher die Einfügeposition der letzten Page einer Zugriffsfolge speichert. Bei der nächsten Einfügeoperation ist zu prüfen, ob die einzufügende Page im Sinne der Sektorsortierung zu der im Cache benachbart ist. Wenn ja, so ist der Aufwand für die Einfügeoperation $O(1)$. Andernfalls muss die Liste nach der korrekten Einfügeoperation durchsucht werden.

Tabelle 2.1 zeigt die Auswirkung dieses Prinzips beim Einfügen in die sortierte Elementliste der Zugriffsfolgen beim erstmaligen Recording des Anwendungsstarts der drei Testszenarien. Die Hit-Rate beträgt im Durchschnitt 88%, d.h. der Aufwand für 88% der Einfügeoperationen ist $O(1)$.

Bei erneutem Recording wird eine vorhandene Zugriffsfolge nur noch ergänzt. Die Anzahl der aufgezeichneten Requests und somit die Anzahl der Einfügeoperationen in die Datenbank gehen gegen Null.

Durch die Verwendung eines 1-Elemente Caches und der stagnierenden Anzahl der Einfügeoperationen bei mehrmaligen Recording ist die Verwendung einer Liste zur sortierten Verwaltung der Pages einer Zugriffsfolge vertretbar.

### Zusammenfassendes Beispiel

Abbildung 2.7 zeigt die Datenstrukturen der Datenbank. Zeiger auf Strukturen sind hellgrau dargestellt. Die Zeiger der doppelt verketteten Listen wurden zu strichlierten Linien zusammengefasst. Elemente der Hashtable wurden zur besseren Übersicht dunkelgrau dargestellt.

Im Beispiel sind vier Zugriffsfolgen vom Start verschiedener Anwendungen in der Datenbank gespeichert. Die Zugriffsfolge der ersten Anwendung besteht aus drei, die der zweiten Anwendung aus zwei Pages. Für die dritte Zugriffsfolge wurden noch keine Pages aufgezeichnet. Daher wurde für diese Zugriffsfolge noch keine Hashtable zur Verwaltung der Pages allokiert. Für die vierte Zugriffsfolge wurde bereits eine Hashtable allokiert, was nur dann erfolgt, wenn mindestens eine Page einer Zugriffsfolge aufgezeichnet wurde.

Da die vierte Zugriffsfolge aber keine Page enthält, kann diese nur mit einer anderen Zugriffsfolge zusammengelegt worden sein. Alle Elemente der vierten Zugriffsfolge wurden daher in eine andere Zugriffsfolge eingefügt. Das Zusammenlegen der Zugriffsfolgen mehrerer Anwendungsstarts ist dann sinnvoll, wenn ein Anwendungsstart andere Anwendungsstarts verursacht. Zum Beispiel kann auf den Start des X-Servers der Start eines Windowmanagers folgen. Die zusammengelegte Zugriffsfolge bietet durch ihren grösseren Umfang mehr Optimierungspotential. Zugriffsfolgen können manuell, mit Hilfe des `ioctl`-Interfaces zusammengelegt

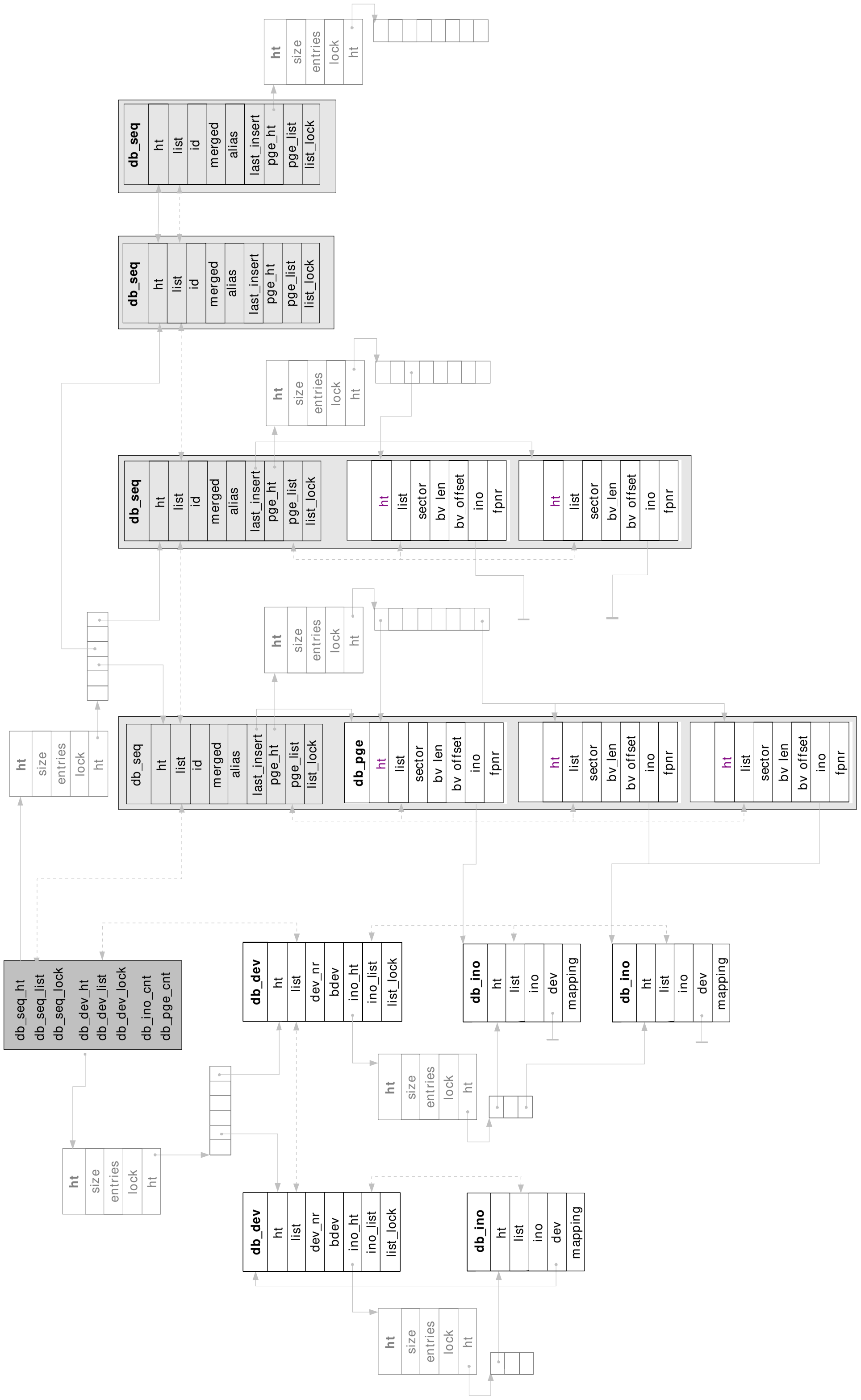


Abbildung 2.7: Datenstrukturen der Datenbank

werden.

Die Page einer Zugriffsfolge beschreibt den Transfer von Daten aus drei verschiedenen Dateien in den Speicher, erkennbar durch die drei Inode Elemente in der Datenbank. Diese Dateien sind auf zwei Devices verteilt. Die Zugriffsfolgen drei und vier, sowie die Pages zwei und drei der ersten Zugriffsfolge verursachen je eine Kollision in der entsprechenden Hashtable zur Verwaltung der Pages der Zugriffsfolgen. Die Kollisionsbehandlung erfolgt mit Listen.

### 2.3.4 Abschätzung der Speicheranforderungen

Da die Datenbank im Kernelspace realisiert wird, soll die zu erwartende Speicheranforderung der Datenbank näher untersucht werden.

Die Zugriffsfolge eines Anwendungsstarts kann sehr umfangreich werden. So besteht die Zugriffsfolge des Anwendungsstarts von OpenOffice 1.0.1 aus 11500 Pages. Die Zugriffe während des Anwendungsstarts betrafen 118 Dateien, welche alle auf einem Device gespeichert sind. Eine Zugriffsfolge wird einen Anwendungsstart durch den Pfad der ausführbaren Datei und den Kommandozeilenparametern zugeordnet. Dadurch können für den Anwendungsstart von OpenOffice durchaus mehrere Zugriffsfolgen in der Datenbank, je eine für jeden Parametersatz, existieren.

#### Grundstruktur eines Datenbankobjekts

Ein Objekttyp der Datenbank wird durch eine Datenstruktur definiert. Diese Datenstrukturen weisen eine einheitliche Grundstruktur auf. Die zur Organisation der Datenbankobjekte verwendeten Felder sind in allen Objekten gleich.

Die Objekte der Datenbank werden in Hashtables verwaltet. Zur Verkettung in der Kollisionsliste eines Arrays wird die Listen-API des Kernels verwendet. Die Verkettung in der Liste erfolgt durch ein Listenobjekt (vgl. Abb.2.7, Feld `ht`), bestehend aus zwei Zeigern zur Vorwärts- und Rückwärtsverkettung.

Weiters sind alle Objekte der Datenbank in verschiedenen Listen organisiert. Ein weiteres Listenobjekt (vgl. Abb.2.7, Feld `list`) ist für die Verkettung in diesen Listen zuständig.

Ein Objekttyp der Datenbank besteht also mindestens aus zwei Listenobjekten. Dazu besteht die Datenstruktur, je nach Objekttyp unterschiedlich, aus zusätzlichen Feldern zur Verwaltung weiterer Daten.

#### Speicheranforderungen am Beispiel von OpenOffice

Am Beispiel der Eingangs beschriebene Zugriffsfolge des Anwendungsstarts von OpenOffice 1.0.1 soll der Speicherbedarf der Datenbank errechnet werden. Es wird angenommen, dass die Datenbank ansonsten leer ist.

In Tabelle 2.2 ist der Speicherbedarf angeführt. In Summe werden ca. $0.33M$ Speicher für die eine Zugriffsfolge benötigt. Bei einem Reimport der Zugriffsfolge

| Datentyp | Datenbank-Objekt | Anzahl | Speicherbedarf DB-Objekt [B] | Summe [B] |
|---|---|---|---|---|
| Device | Device Objekt | 1 | 40 | 40 |
| | Hashtable | 1 | 16 | 16 |
| | Hashtable-Array | 13 | 8 | 104 |
| Inode | Inode Objekt | 118 | 28 | 3304 |
| | Hashtable | 1 | 16 | 16 |
| | Hashtable-Array | 1009 | 8 | 8072 |
| Zugriffs-sequenz | ZF Objekt | 1 | 544 | 544 |
| | Hashtable | 1 | 16 | 16 |
| | Hashtable-Array | 1009 | 8 | 8072 |
| ZF Page | Page Objekt | 11500 | 28 | 322000 |
| | Hashtable | 1 | 16 | 16 |
| | Hashtable-Array | 1009 | 8 | 8072 |
| | | | | 350272 |

Tabelle 2.2: Abschätzung Speicherbedarf Zugriffsfolge OpenOffice 1.0.1

wird die Arraygrösse der Hashtable zur Indizierung der Pages nicht mit 1009 Einträgen festgelegt, sondern es wird eine Fill Rate der Hashtable von 50% angestrebt. Dazu wird die Arraygrösse doppelt so gross wie die Anzahl der zu verwaltenden Pages gewählt. Der Gesamtspeicherbedarf steigt dann auf $0.5M$.

Durch das kontinuierliche Recording wird für jeden Anwendungsstart mindestens eine Zugriffsfolge in der Datenbank gespeichert. Variiert der Parametersatz, so können durchaus auch mehrere Zugriffsfolge für ein Anwendung auftreten. Daraus folgt eine hohe Belastung des Kernelspeichers, was unterstreicht, dass bei einer Weiterentwicklung des PLS die Datenbank aus dem Kernelspace auszulagern ist.

## 2.4 Preloading

Die Aufgabe des Preloadings ist das Laden der für den Start einer Anwendung benötigten Daten von den Blockgeräten des Systems. Die Folge der Blockgerätezugriffe beim Preloading soll optimiert hinsichtlich I/O Performance erfolgen. Das Preloading ist der zentrale Prozess des PLS. Die in den Abschnitten Recording, bzw. Datenbank beschriebenen Komponenten dienen zur Datengewinnung und Datenhaltung für das Preloading.

### 2.4.1 Kernelintegration

Wie beim Recording und der Datenbank stellt sich die Frage, an welcher Stelle das Preloading in das Kernel I/O System (Abb.2.1) integriert werden soll. Anders als beim Recording ist der Ort der Integration durch Anforderungen an das Preloading und dem Format der aufgezeichneten Zugriffsfolgen relativ eingegrenzt.

Aus der in Kapitel 2.2.1 beschriebenen Testimplementierung resultiert die Anforderung an das Preloading, synchrone Blockrequests zu vermeiden, da diese den Preloading Prozess blockieren. Dazu muss das Preloading unterhalb des Blockmappings im Page Cache durchgeführt werden. Die erste Ebene unterhalb des Blockmappings der Dateisysteme ist der bio-Layer, wo auch das Recording erfolgt. Es liegt daher nahe, das Preloading ebenfalls im bio-Layer durchzuführen. Dafür spricht auch der modulare Aufbau von bio-Requests, wodurch diese einfach zerlegt, umgeordnet und beim Preloading wieder rekonstruiert werden können.

Das Preloading konstruiert zum Laden der Blockgerätedaten aus den Vorgaben der gespeicherten Zugriffsfolge bio Requests. Diese Requests werden anschliessend in den Request Layer eingespeist. Das Preloading operiert somit, genau wie das Recording, zwischen bio- und Request Layer (Abb.2.4).

Aufgrund dieses Ansatzes muss das Preloading im Kernelspace implementiert werden. Die Nähe zur Datenbank spricht dafür, das Preloading ebenfalls als Funktion im PLS Modul zu implementieren, wodurch weiters die notwendigen Modifikationen an den Kernelquellen minimiert werden.

### 2.4.2 Initialisierung des Preloadings

Da das Preloading als Funktion des PLS Moduls konzipiert ist, muss im Blockgeräte I/O System des Kernels ein Einsprungpunkt in das Preloading definiert werden.

Zu Beginn des Anwendungsstarts sollen alle dafür benötigten Blockgerätedaten im Speicher präsent sein. Ziel ist es daher, das Preloading zum frühest möglichen Zeitpunkt im Anwendungsstart durchzuführen.

Die ausführbare Datei einer Anwendung soll dazu nicht modifiziert werden. Bei der Menge der Programme einer Linux Installation wäre dies nicht praktikabel. Die Initialisierung des Preloadings im Userspace, z.B. durch einen Syscall, ist daher nicht möglich. Ein Anwendungsstart muss somit vom PLS an zentraler Stelle im Kernelspace erkannt werden.

Linux definiert als einzige Schnittstelle für Anwendungsstarts den Syscall `execve`. Der Handler im Kernelspace für diesen Syscall ist die Funktion `do_execve`. Die Initialisierung des Preloadings im Kernelspace wird am Anfang dieser Funktion durchgeführt. Dadurch erfolgt das Preloading zum frühest möglichen Zeitpunkt während des Anwendungsstarts. Falls die geladene Zugriffsfolge vollständig war, sollten nach dem Preloading alle für den Anwendungsstart benötigten Daten im Speicher vorhanden sein. Bei der Fortsetzung des Anwendungsstarts sollte dann kein Blockgeräte I/O mehr notwendig sein.

Wie beim Recording definiert ein globaler Zeiger den Einsprungpunkt in das Preloading. Beim Laden des Moduls wird dieser Zeiger auf die Preloading-Funktion des PLS Moduls gesetzt. Wird das Modul entladen, so wird der Zeiger wieder Null gesetzt. Beim Anwendungsstart wird zu Beginn der Funktion `do_execve` geprüft, ob dieser Zeiger gesetzt ist. Ist dies der Fall, so wird die durch den Zeiger bezeichnete Preloading-Funktion aufgerufen.

### 2.4.3 Der Preloading Prozess

Nach der Initialisierung rekonstruiert das Preloading die dem Anwendungsstart zugeordnete Zugriffsfolge. Die Daten der Zugriffsfolge werden dabei vollständig geladen.

Dazu wird zuerst geprüft, ob eine Zugriffsfolge für die Anwendung in der Datenbank existiert. Zur Suche nach der Zugriffsfolge muss, wie in Kapitel 2.3.2 beschrieben, eine eindeutige ID für die startende Anwendung generiert werden.

Wurde eine Zugriffsfolge gefunden, so konstruiert das Preloading aus den Vorgaben der Zugriffsfolge die bio Requests zum Einlesen der Daten. Dazu iteriert der Preloading Algorithmus über die Pages der Zugriffsfolge. Für jede Page der Zugriffsfolge wird eine Page Frame allokiert und in den Page Cache eingefügt. Der Schlüssel dazu ist das in der Page der Zugriffsfolge gespeicherte Tripel Device/Inode/Seitenoffset.

Der Preloading Algorithmus versucht, so viele Pages der Zugriffsfolge wie möglich in einen bio Request zusammenzufassen. Die Bedingung dazu ist, dass die Daten der Pages sequentiell am Blockgerät angeordnet sind. Die Voraussetzung dazu wird durch die Sektorsortierung der Pages einer Zugriffsfolge geschaffen.

Die maximale Kapazität eines bio Requests ist auch durch das Blockgerät beschränkt. So kann die Festplatte von Testsystem 1 Requests mit maximal 256 Sektoren bearbeiten. Bei einer Sektorgrösse von 512B und einer Seitengrösse von 4K sind dies 32 Pages. Die Maximalkapazität der Requests des Blockgeräts von Testsystem 2 ist hingegen mit 2048 Sektoren beschränkt, was 256 Pages entspricht.

Ist die Maximalkapazität eines bio Requests erreicht, oder sind die Daten zweier aufeinanderfolgender Pages der Zugriffsfolge nicht sequentiell am Blockgerät angeordnet, so muss ein neuer bio Request allokiert werden. Der aktuelle bio Request wird zur weiteren Bearbeitung an den Request Layer übergeben. Die Sequentialität der Blockgerätedaten wird mit Hilfe des in der Page einer Zugriffsfolge gespeicherten Startsektors geprüft.

### 2.4.4 Limitierungen der Preloading-Performance

Das Preloading einer Zugriffsfolge stellt Anforderungen an das I/O System von Linux, die im Normalbetrieb kaum vorkommen. Einige Komponenten des I/O System sind nicht auf diese Anforderungen abgestimmt, was eine Limitierung der Preloading Performance zur Folge hat.

Charakteristisch für das Preloading einer Zugriffsfolge ist, dass innerhalb eines kurzen Zeitraums ungewöhnlich viele Requests generiert werden. Diese Requests sind nach der Anordnung der Daten am Blockgerät sortiert. Weiters versucht der Preloading Algorithmus die Auslastung, bzw. die Grösse der Requests zu maximieren.

#### Request Queue Congestion

Das Congestion Verhalten der Request Queue eines Blockgeräts verlängert die Dauer des Preloadings. Dadurch verzögert sich auch der Anwendungsstart, da dieser unmittelbar nach Ende des Preloadings fortgesetzt wird.

Alle vom Preloading generierten Blockgeräte Requests haben asynchronen Charakter. Die Entkoppelung von Preloading und Blockgeräte I/O erfolgt durch die Request Queue. Das Preloading ist dann beendet, wenn alle Requests zum Laden einer Zugriffsfolge generiert wurden. Das heisst nicht, dass zu diesen Zeitpunkt bereits alle Requests des Preloadings bearbeitet wurden. In der Regel dauert das Einlesen der Daten vom langsamen Blockgerät um ein vielfaches länger als die Generierung der Blockgeräte Requests durch das Preloading. Daraus folgt, dass der Füllgrad der Request Queue durch die vielen in kurzer Zeit generierten Requests entsprechend hoch ist.

Jeder Request einer Request Queue wird durch ein Request Objekt repräsentiert. Um die Speicheranforderungen des Kernels zu minimieren, ist die Anzahl der Request Elemente pro Request Queue beschränkt. Die Obergrenze ist von der Speicherausstattung des Systems abhängig und beträgt üblicherweise 128 Request Objekte. Weiters sind einige dieser Request Objekte für den Kernel reserviert.

Sind alle Request Objekte belegt, dann wird die Request Queue in den Zustand congested versetzt. Stellt ein Prozess einen Request an ein Blockgerät, dessen Request Queue sich im Zustand congested befindet, so muss der Prozess warten, bis wieder genügend Request Objekte frei sind.

Da beim Preloading viele Requests in kurzer Zeit generiert werden, wird die Request Queue während des Preloadings durchaus mehrmals in den Zustand Congested versetzt. Dies ist mit einer Blockierung des Preloading Prozesses gleichzusetzen. Jeder Zustandswechsel der Queue während des Preloadings bedeutet eine Synchronisierung von Blockgeräte I/O und Preloading.

Abbildung 2.8 zeigt ein Beispiel für das Congestion Verhalten der Request Queue eines Blockgeräts während des Preloadings einer Zugriffsfolge. Auf der Ordinate ist der Füllstand der Queue in Requests angegeben. Die Abszisse zeigt den zeitlichen Verlauf des Preloadings. Zu Beginn des Preloadings wird angenommen,

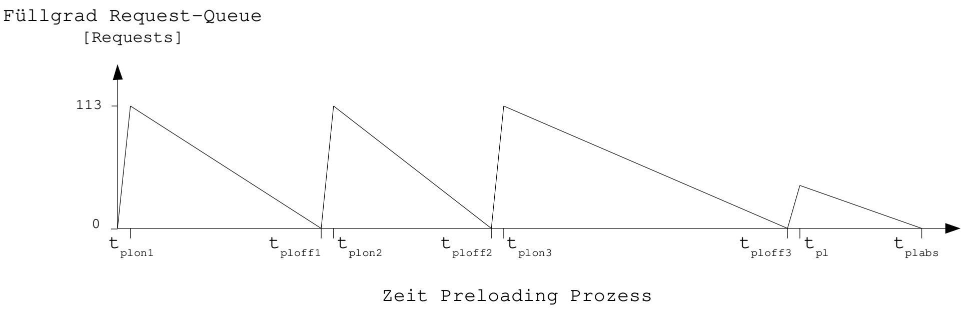


Abbildung 2.8: Congestion Verhalten der Request Queue beim Preloading

dass die Request Queue leer ist.

Der Preloading Prozess generiert innerhalb kurzer Zeit eine grosse Anzahl asynchroner Requests, bis die Queue mangels Request Objekte in den Zustand Congested versetzt wird ($t_{plcon}$). Der Preloading Prozess muss warten, bis die Requests in der Queue bearbeitet wurden und wieder genügend Request Objekte frei sind. Dann wird der Zustand Congested der Queue aufgehoben ($t_{plcoff}$).

Die abfallenden Flanken des Graphen müssen dabei nicht linear sein. Das ist nur dann der Fall, wenn die Bearbeitung aller Requests in der Queue genau gleich lange dauert. Weiters kann auch der Zeitraum, bis der Zustand Congested aufgehoben wird, unterschiedlich gross sein.

Das Congestion Verhaltens der Request Queue zeigt, wie wichtig es ist, die Auslastung der Requests beim Preloading zu maximieren. Für das Lesen der Daten einer Zugriffsfolge vom Blockgerät werden so weniger Requests benötigt. Eventuell kann so die Anzahl der Zustandswechsel der Request Queue und somit die Ausführungszeit des Preloadings reduziert werden.

## 2.5 Synchronisierung der PLS Prozesse

Das PLS kann in die Prozesse Preloading, Anwendungsstart und Recording unterteilt werden. Für eine optimale Performance beim Anwendungsstart ist müssen diese optimal aufeinander abgestimmt sein.

### 2.5.1 Preloading und Recording

Ist das PLS Modul geladen, erfolgen Recording und Preloading automatisch für jeden Anwendungsstart, ohne dass es einer Interaktion mit einem User bedarf. Ziel der Synchronisierung von Preloading und Recording ist die Vermeidung des Recordings von Requests, welche durch das Preloading generierten wurden. So soll Overhead durch zusätzliche Suchoperationen in der Datenbank vermieden werden. Dennoch müssen alle noch nicht erfassten Requests des Anwendungsstarts aufgezeichnet werden.

Dazu wird das Preloading und die Initialisierung der Recording Session zu Beginn des Anwendungsstarts durchgeführt. Zuerst erfolgt das Preloading und danach die Initialisierung einer Recording Session. Erst danach wird der Anwendungsstart fortgesetzt.

Da das Preloading aus Sicht der startenden Anwendung ein atomarer Vorgang ist, wird mit dem Recording des Anwendungsstarts erst dann begonnen, wenn das Preloading alle Requests zum Laden der Zugriffsfolge generiert hat. Da Recording und Preloading im bio Layer arbeiten, haben bei der Initialisierung der Recording Session die Requests des Preloadings den bio Layer und somit den Einsprungpunkt ins Recording bereits passiert.

Weil die Recording Session aber noch vor der Fortsetzung des Anwendungsstart initialisiert wird, werden alle, bei früheren Recordings noch nicht erfassten Requests des Anwendungsstarts trotzdem erfasst.

### 2.5.2 Preloading und Anwendungsstart

Eine weitere Steigerung der Performance von Anwendungsstarts ist durch teilweise Parallelisierung von Preloading und Anwendungsstart möglich.

Das Preloading einer Zugriffsfolge ist ein Prozess, bei dem sich CPU-lastige mit I/O-lastigen Phasen abwechseln. In der CPU-lastigen Phase werden die Requests aus den Vorgaben der gespeicherten Zugriffsfolge generiert. Darauf folgt die I/O-lastige Phase des Einlesens der Blockgerätedaten. Durch das Congestion-Verhalten der Request Queue (Abb.2.8) können diese Phasen mehrmals hintereinander wechseln.

In den I/O-lastigen Phasen könnte die bei wenig belasteten System brachliegende CPU-Kapazität für eine teilweise Fortsetzung des Anwendungsstart genutzt werden. Die Voraussetzung dazu ist, dass die dazu notwendigen Daten durch das Preloading bereits in den Speicher gelesen wurden.

Die beim Recording aufgezeichnete Zugriffsfolge entspricht den Anforderungen der startenden Anwendung. Die Daten werden genau in der für den Anwendungsstart notwendigen Reihenfolge vom Blockgerät gelesen.

Die beim Preloading durchgeführte Zugriffsfolge nimmt hingegen keine Rücksicht auf die Anforderungen des Anwendungsstarts. Die Pages der Zugriffsfolge wurden hinsichtlich best möglicher I/O Performance umgeordnet. Es kann dadurch vorkommen, dass zu Beginn des Anwendungsstarts benötigte Daten aufgrund der Sortierung erst gegen Ende des Preloadings in den Speicher gelesen werden. Der Anwendungsstart wird solange blockiert, bis der Transfer der benötigten Daten abgeschlossen ist. Eine teilweise Parallelausführung von Preloading und Anwendungsstart ist dann nur sehr eingeschränkt möglich.

Des weiteren wird die Möglichkeit der Parallelisierung von Preloading und Anwendungsstart durch das Congestion Verhalten der Request Queue eingeschränkt. Mit dem Anwendungsstart kann erst dann fortgefahren werden, wenn alle Requests des Preloadings generiert wurden. Bei umfangreichen Zugriffsfolgen besteht die Möglichkeit, dass durch das Congestion Verhalten der Request Queue der Preloading Prozess und somit der Anwendungsstart verzögert wird.

Beim in Abbildung 2.8 gezeigten Szenario wechselt die Request Queue viermal in den Zustand Congested. Eine Parallelausführung von Preloading und Anwendungsstart ist erst ab den Zeitpunkt $t_{pl}$ möglich, d.h. wenn die letzten Requests durch das Preloading generiert wurden. Man könnte $t_{pl}$ durch Erhöhung der Anzahl der Request Objekte pro Request Queue vorverlegen. Nachteil währe ein erhöhter Speicherverbrauch durch den Kernel.

## 2.6 Schnittstelle PLS Modul - User Space

Für die in den Userspace ausgelagerte Funktionalität des PLS muss die Möglichkeit zur Kommunikation mit dem PLS Modul im Kernelspace geschaffen werden. Linux stellt dazu verschiedene Möglichkeiten bereit:

- Systemcalls
- `proc`-Dateisystem
- Gerätetreiber

Syscalls haben statischen Charakter. Sie definieren eine dauerhafte Schnittstelle zu einem Service des Kernels. Das PLS hingegen ist nur dann verfügbar, wenn das PLS Modul geladen ist. Eleganter währe daher eine Schnittstelle, die nur dann verfügbar ist, wenn auch das Preload Service einsatzbereit ist. Da weiters neue Syscalls von der Community kaum mehr akzeptiert werden, wurde von deren Verwendung abgesehen.

Einträge des `proc`-Dateisystems oder Gerätetreiber haben hingegen nicht-statischen Charakter. Die Schnittstelle zum Service steht nur dann zur Verfügung, wenn der entsprechende Treiber oder das Modul geladen wurde. Je nach Anwendung haben beide Konzepte Vor-, bzw. Nachteile.

### 2.6.1 Import der Datenbank

Die beim Recording gewonnenen Zugriffsfolgen müssen für das Preloading bei späteren Anwendungsstarts über Neustarts des Systems hinweg verfügbar gemacht werden. Dazu muss der Inhalt der Datenbank im Speicher auf einen nicht-flüchtigen Datenträger exportiert und nach Neustart des Systems wieder importiert werden.

Der Import- und Export-Funktionalität wurden aus den Kernelspace in den Userspace ausgelagert. Die Motivation dazu war die Vermeidung der umfangreichen Dateioperationen des Imports und Exports im Kernel.

#### Performance

Da das PLS u.a. auch zur Beschleunigung des Systemstarts eingesetzt wird, ist eine gute Performance beim Import der Datenbank wichtig.

Die Zugriffsfolgen der Datenbank werden vollständig im Speicher verwaltet. Daher muss der Inhalt der Datenbank beim Import von einem nichtflüchtigen Speicher, üblicherweise einen Blockgerät, gelesen werden. Wie im Kapitel 2.3.4 gezeigt wurde, kann die zu importierende Datenmenge durchaus umfangreich sein. So umfasst die Zugriffsfolge des Anwendungsstarts von OpenOffice ca 12000 Pages.

Zur Entscheidungsfindung wurden Messungen mit verschiedenen Implementierungen der Importfunktion durchgeführt. Tabelle 2.3 zeigt die Import-Performance der verschiedenen Implementierungen beim Import von 1000 Datensätzen. Die

| Messung | proc-FS DS single | proc-FS DS blocked | device char DS | device bin DS |
|---|---|---|---|---|
| 1 | 5.049 | 0.853 | 0.428 | 0.080 |
| 2 | 4.927 | 0.870 | 0.432 | 0.079 |
| 3 | 5.037 | 0.869 | 0.430 | 0.082 |
| arithm. Mittel | 5.004 | 0.860 | 0.430 | 0.080 |

Tabelle 2.3: Performance beim Import von 1000 Datensätzen

Messung wurde für jede Implementierung dreimal durchgeführt. Abschliessend wurde der Mittelwert berechnet.

Die ersten beiden Versionen benutzen als Schnittstelle das `proc`-Dateisystem des Kernels. Da das `proc`-Dateisystem String-basiert arbeitet, müssen die Daten beim Export vom und beim Import ins Binärformat konvertiert werden. Weiters ist die Maximalkapazität einer Schreiboperation in einen Eintrag des `proc`-Dateisystems auf die Seitengrösse des Systems beschränkt. Die erste Version importierte jede Page einer Zugriffsfolge mit je einer Schreiboperation.

Aufgrund der inakzeptablen Zeiten der ersten Implementierung wurden die Datensätze bis zur Maximalkapazität einer Schreiboperation am `proc`-Dateisystem zusammengefasst. Auf den Testsystemen können so 4K Daten pro Aufruf importiert werden. Gegenüber der ersten Version zeigt sich so eine deutliche Steigerung der Importperformance.

Für die Versionen drei und vier wurde ein virtueller Gerätetreiber implementiert. Für den Import wird das `write` Kommando verwendet. Wie in den ersten beiden Versionen werden die Daten für den Import und Export in eine Stringrepräsentation konvertiert. Die Werte für Version drei zeigen eine Performancesteigerung gegenüber den ersten beiden Versionen, welche auf dem `proc`-Dateisystem basieren. Dennoch würde der Import der 12000 Datensätze der OpenOffice Zugriffsfolge noch immer 5s dauern.

In Version vier wurden daher die Datensätze binär exportiert und importiert. Dadurch entfiel die aufwendige Stringkonvertierung. Der Import der OpenOffice Zugriffsfolge benötigt so weniger als eine Sekunde.

### Lösung

Der Im- und Export der Daten wird mit Hilfe eines virtuellen Gerätetreibers durchgeführt. Beim Laden des PLS Moduls wird der Treiber beim Kernel registriert. Darauf kann im `dev` Dateisystem eine Gerätedatei für das PLS angelegt werden, welche von einer Anwendung im Userspace geöffnet werden kann.

Jeder Linux Treiber stellt einen standardisierten Satz von Funktionen zur Verfügung, welche von Anwendungen im Userspace durch entsprechende Wrapperfunktionen, z.B. der libc, aufgerufen werden können. Dabei müssen nicht alle Funktionen vom Treiber implementiert werden. Der virtuelle Gerätetreiber des PLS implementiert nur die `write` Methode für den Import. Der zu importierende Datensatz wird als Argument übergeben.

Wie die Messungen in Tabelle 2.3 zeigen, bedeutet die Verwendung eines Gerätetreibers die beste Performance beim Import der Datenbank. Weiters werden User nicht durch zusätzliche Einträge im `proc`-Dateisystem von Linux irritiert.

### 2.6.2 Administration und Export der Datenbank

Das PLS muss eine Möglichkeit zur Administration aus dem Userspace zur Verfügung stellen. Sie sollte möglichst transparent und unauffällig für nicht berechtigte User, sowie einfach erweiterbar sein.

#### ioctl Interface

Für den Import der Datenbank wird ein virtueller Gerätetreiber verwendet. Es liegt daher nahe, diesen auch für die Administration des PLS zu verwenden.

Für die Administration eines Treibers aus dem Userspace stellt Linux die `ioctl`-Schnittstelle zur Verfügung. Will ein Treiber diese Schnittstelle nutzen, so muss er nur die `ioctl` Funktion des standard Funktionssatztes implementieren. Nach dem Öffnen der dem virtuellen Gerätetreiber des PLS zugewiesenen Gerätedatei kann eine Anwendung im Userspace durch Aufruf der entsprechende Wrapperfunktion einer Library das PLS administrieren. Die Wrapperfunktion des libc heisst z.B. `ioctl` und verlangt als Argument eine geöffnete Gerätedatei, eine Kennzahl und ev. ein Argument. Die Kennzahl beschreibt die auszuführende Aktion.

Der Vorteil dieser Methode ist, dass keine zusätzliche Technik implementiert werden muss, da der für den Import implementierte Gerätetreiber verwendet werden kann. Weiters ist die Möglichkeit zur Konfiguration ist nur dann gegeben, wenn das PLS Modul geladen ist und der Treiber beim Kernel registriert wurde.

#### Funktionalität

Das PLS bietet umfangreiche Möglichkeiten zur Administration. Jede Aktion benötigt einen Aufruf einer `ioctl` Wrapperfunktion mit einer Kennzahl, welche die auszuführende Aktion identifiziert. Nachfolgend eine Auflistung der implementierten Kennzahlen und eine Beschreibung ihrer Funktionalität:

- `PLS_IOCS_CFG_PRE`
  (De)aktiviert Preloading. Der `ioctl` Aufruf ist als Wechselschalter konzipiert und benötigt daher keine Argumente.
- `PLS_IOCG_CFG_PRE`
  Liefert Null, falls das Preloading deaktiviert wurde, sonst eins. Der Aufruf verlangt keine Parameter.
- `PLS_IOCS_CFG_REC`
  (De)aktiviert Recording. Der `ioctl` Aufruf ist als Wechselschalter konzipiert und benötigt daher keine Argumente.

- `PLS_IOCG_CFG_REC`
  Liefert Null, falls das Recording deaktiviert wurde, sonst eins. Der Aufruf verlangt keine Parameter.
- `PLS_IOCS_CFG_PER`
  Setzt den Wert der Variable `rec_period`, welche die Zeitspanne für den Zeitablauf einer Recording Session definiert. [3] Der Wert wird als ein Zeiger auf Integer übergeben. Erlaubt sind Werte zwischen 1 und 10 Sekunden.
- `PLS_IOCG_CFG_PER`
  Gibt den Wert der Variable `rec_period` zurück. Der Aufruf verlangt keine Parameter.
- `PLS_IOCS_CFG_TIM`
  Setzt den Wert der Variable `rec_irq_interval` zur Steuerung des Ablaufes des Recording Timers [4]. Verlangt als Parameter einen Zeiger auf einen Integerwert, welcher im Intervall zwischen 1 und 10 Sekunden liegen muss.
- `PLS_IOCG_CFG_TIM`
  Gibt den Wert der Variable `rec_irq_interval` zurück. Der Aufruf verlangt keine Parameter.
- `PLS_IOC_STAT_DB`
  Schreibt eine Übersicht über die in der Datenbank enthaltenen Zugriffsfolgen nach stdout. Als Parameter wird ein Zeiger auf einen Integerwert verlangt. Erlaubt sind die Werte Null oder eins. Ist der Wert eins, so werden auch leere Zugriffsfolgen aufgelistet.
- `PLS_IOC_STAT_HT`
  Schreibt eine Statistik der im Modul verwendeten Hashtables bezüglich Füllgrad, Kollisionen, etc. nach stdout. Der Aufruf verlangt keine Parameter.
- `PLS_IOC_EXPORT_DB`
  Exportiert die Datenbank. Als Parameter wird ein Zeiger auf einen Integerwert verlangt. Dieser Wert repräsentiert den Dateideskriptor jener geöffneten Datei, in die die Datenbank exportiert werden soll.
- `PLS_IOC_PRELOAD`
  Führt einen manuellen Preload einer Zugriffsfolge in der Datenbank durch. Als Parameter wird ein Zeiger auf einen Wert vom Typ unsigned long verlangt. Dieser Wert ist die ID jener Zugriffsfolge, für die der Preload durchgeführt werden soll.
- `PLS_IOC_MERGE`
  Fügt eine Zugriffsfolge in eine andere Zugriffsfolge ein. Der Aufruf benötigt als Parameter einen Zeiger auf ein Array der Länge zwei vom Typ unsigned long, in dem die IDs der einzufügenden Zugriffsfolgen gespeichert sind. Die zweite Zugriffsfolge wird in die erste Zugriffsfolge des Arrays eingefügt.
- `PLS_IOC_DELETE`
  Löscht eine Zugriffsfolge in der Datenbank. Als Parameter wird ein Zeiger

[3] siehe Kapitel 2.2.3
[4] siehe Kapitel 2.2.3

auf einen Wert vom Typ unsigned long benötigt, welcher die ID der zu löschenden Zugriffsfolge repräsentiert.

# Kapitel 3

# Messungen

## 3.1 Testsysteme

Um einen Eindruck von den Auswirkungen des PLS auf die Performance von Anwendungsstarts zu erhalten, wurden die Messungen auf zwei unterschiedlichen Testsystemen durchgeführt, welche ein unterschiedliches Leistungsspektrum hinsichtlich CPU und Blockgeräte I/O Performance aufweisen:

- *Testsystem Laptop*: Wie bei den meisten Laptops ist für dieses Testsystem die im Vergleich zu Testsystem Desktop geringere CPU und Blockgeräte I/O Performance charakteristisch. Grund dafür ist die Verwendung einer langsamdrehenden Festplatte und eines niedrig getakteten Prozessors zur Minimierung des Energieverbrauchs, bzw. zur Maximierung der Akkulaufzeit.

- *Testsystem Desktop*: Ein Desktopsystem der aktuellen Leistungsklasse mit einer schnelldrehenden IDE Festplatte und einem im Vergleich zum Laptop schnellen CPU. Dieses Testsystem wurde gewählt, um die Auswirkungen der im Vergleich zum Laptop besseren CPU und Blockgeräteperformance beobachten zu können.

Tabelle 3.1 gibt einen Überblick über die Hardwareausstattung der Testsysteme. Für das in dieser Arbeit vorgestellte PLS ist vor allem die unterschiedliche Leistungsfähigkeit der Blockgeräte interessant. Die Softwareausstattung der Testsysteme wurde weitgehend identisch gehalten und besteht aus GNU/Linux in Form einer Standardinstallation der Distribution Red Hat Linux 7.2-2.96-112.7.1 (gcc version 2.96 20000731) mit Kernel 2.6.0. Auf beiden Testsystemen wurde das `ext3` Dateisystem verwendet.

## 3.2 Testszenarien und Messmethoden

Um reproduzierbare Messergebnisse zu erhalten, werden in diesem Kapitel Testszenarien definiert und die verwendeten Messwerkzeuge beschrieben.

| | Testsystem | Laptop | Desktop |
|---|---|---|---|
| CPU | Type | Intel Pentium III | Intel Pentium IV |
| | Speed | 933MHz | 2.53 GHz |
| | L1 Cache | 32K, 16K Data | 12K, 8K Data |
| | L2 Cache | 256K integrated | 512K integrated |
| | FS-Bus Speed | 133MHz | 533MHz |
| Disk | Type | IBM Travelstar | Maxtor |
| | Kapacity | 30 GB | 120 GB |
| | Bytes per Sector | 512 | 512 |
| | Rotational Speed | 4200 rpm | 7200 rpm |
| | Buffer Size | 2 MB | 8 MB |
| | Interface Type | UDMA 100 | DMA/ATA-133 |
| | Average Seek | 13 ms | 8.9 ms |
| | Maximum Seek | 24 ms | 18 ms |
| | Average Latency | 7.1 ms | 4.17 ms |

Tabelle 3.1: Hardwareausstattung der Testsysteme

### 3.2.1 Testszenarien

Am Beispiel dreier Anwendungen sollen die Auswirkungen des PLS untersucht werden. Die drei Anwendungen unterscheiden sich hinsichtlich ihres Ressourcenbedarfs beim Anwendungsstart. Um die Startzeit ohne Modifikation am Sourcecode messen zu können, können die gewählten Anwendungen per Kommandozeilenparameter oder Skriptingschnittstelle unmittelbar nach dem Anwendungsstart wieder beendet werden. Auf Basis dieser Anwendungen werden drei Testszenarien definiert.

#### GNU Emacs

Um die Auswirkung des PLS auf eine eher kleine Anwendung beobachten zu können, wurde als erstes Testszenario der Start von GNU Emacs 21.2.1 gewählt. Gemessen wurde die Ausführungszeit des Kommandos

```
/usr/bin/emacs -nw -f save-buffers-kill-emacs
```

gemessen. Die Option `-nw` bewirkt, dass Emacs ohne X-Fenster gestartet wird, während die Option `-f` nach den Start die Lisp Funktion `save-buffers-kill-emacs` ausführt, welche Emacs unmittelbar nach den Start wieder beendet.

#### Mozilla

Als in Bezug auf Ressourcenbedarf mittlere Anwendung wurde Mozilla 1.0.2 gewählt. Zur Messung der Startzeit wird Mozilla mit der minimalen html-Seite

```
<html>
    <body onload=window.close()></body>
</html>
```

aufgerufen. Das Java Script Kommando `window.close()` beendet den Browser gleich nach dem Laden der Seite, d.h. nach Ende des Anwendungsstarts.

#### OpenOffice

Eine der ressourcenintensivsten Desktopanwendungen unter Linux ist OpenOffice. Als drittes Testszenario wurde daher der Anwendungsstart von OpenOffice 1.0.1 gewählt. Gemessen wird die Ausführungszeit des Kommandos

```
soffice macro:///standard.module1.macro1
```

Mit Hilfe des Parameters `macro` wird OpenOffice angewiesen, nach dem Start ein Makro auszuführen. Unmittelbar nach den Start prüft OpenOffice, ob das angegebene Makro existiert. Falls nicht, beendet sich OpenOffice wieder.

### 3.2.2 Messverfahren und Werkzeuge

Alle Messungen wurden jeweils nach Reboot, Login und, wenn notwendig, Start des X-Servers (Mozilla, OpenOffice) durchgeführt. Bei den Messungen der Startzeit der Testszenarien unter Verwendung des PLS waren Preloading und Recording aktiviert. Jedes Messergebnis bildet den Mittelwert einer aus 10 Messungen bestehenden Messreihe.

Alle Zeiten wurden mit GNU time 1.7 gemessen. Als `cat` Kommando wurde cat 2.0.21 verwendet. Alle Testszenarien wurden per Skript ausgeführt. Die Testskripts wurden unmittelbar nach Systemstart und Login bei subjektiv unbelasteten System ausgeführt.

## 3.3 Zeitverhalten der Testszenarien

In diesem Kapitel soll das Zeitverhalten der Testszenarien mit und ohne Verwendung des PLS untersucht werden. Dazu soll zunächst die theoretisch erreichbare Performancesteigerung ermittelt werden und anschliessend mit den gemessenen Werten verglichen werden.

### 3.3.1 Theoretisch mögliche Performancesteigerung

Um eine Abschätzung der maximal möglichen Performancesteigerung bei der Ausführung der Testszenarien errechnen zu können, muss die Blockgeräte I/O Performance und die während des Anwendungsstarts vom Blockgerät gelesene Datenmenge ermittelt werden.

#### Ermittlung der Blockgeräte I/O Performance

Die Blockgeräte I/O Performance der Testsysteme wurden mit zwei unterschiedlichen Verfahren ermittelt. Zum einen wurde die I/O Performance mit Hilfe des Kommandos `hdparm -t`[1] gemessen. Das Kommando ermittelt die Performance beim Lesen in den Buffercache ohne Dateisystem Overhead.

[1] http://freshmeat.net/projects/hdparm

| | hdparm -t [MB/s] | cat > /dev/null [MB/s] |
|---|---|---|
| Laptop | 20.00 | 9.60 |
| Desktop | 55.27 | 29.64 |

Tabelle 3.2: Blockgeräte I/O Performance der Testsysteme

Zum anderen wurde zur Messung der I/O Performance das Kommandos `cat` verwendet. Zur Messung wurde nach Reboot eine 10M grosse Datei mit `cat` nach `/dev/null` geschrieben. Im Gegensatz zu den Messungen mit `hdparm -t` ist bei den Ergebnissen der Messungen mit `cat` der Overhead von VFS und FS in den Messwerten enthalten.

Die Ergebnisse beider Verfahren sind in der Tabelle 3.2 angeführt. Die Werte zeigen die aufgrund der Herstellerangaben (Tab.3.1) vermuteten, deutlich grössere Leistungsfähigkeit des Blockgeräts im Testsystem Desktop.

Vergleicht man die Verfahren, so ist die Auswirkung des Overheads von VFS und FS bei den Messungen mit `cat` ersichtlich. Wie das Kommando `time` zeigt, findet bei der Ausführung aller Testszenarien kein Dateisystem read und write statt. Im Gegensatz zum Preloading und `hdparm` ist in den mit `cat` gemessenen Werten Overhead für Dateisystemindirektion enthalten. Eine grössere Relevanz besitzen daher die mit `hdparm -t` ermittelten Werte. Wie Messungen später zeigen, wird z.B. die mit `cat` ermittelte I/O Performance beim Preloading der Zugriffsfolge von Mozilla am Testsystem Desktop (Tab.3.9, I/O Perf) deutlich überschritten.

Bei beiden Messverfahren ist zu beachten, dass die Performance sequentieller Lesevorgänge vom Blockgerät gemessen wurde. Die auf Basis dieser Werte errechnete maximale Performancesteigerung ist somit der best-case.

### Ermittlung der beim Demand Paging eingelesenen Datenmengen

Das Kommando GNU `time` zeigt neben Zeitinformationen auch minor und major Page Faults eines Prozesses an. Interessant für die Abschätzung der zu lesenden Datenmenge sind dabei die major Page Faults, also jene Page Faults, zu deren Behebung Blockgeräte-I/O notwendig ist.

Zur Berechnung der eingelesenen Datenmenge bei der Ausführung eines Testszenarios wurden die Anzahl major Page Faults beim Kaltstart ermittelt und mit der Seitengrösse multipliziert (Tab.3.3, Datenmenge). Die Seitengrösse beträgt auf beiden Testsystemen 4K.

Die Berechnung berücksichtigt nicht, dass durch das Readahead des Demand Pagings mit unter nicht nur eine Seite pro Page Fault, sondern spekulativ ein Seitencluster gelesen wird. Tatsächlich werden während des Anwendungsstarts immer mehr Daten als berechnet eingelesen, wie der Umfang der aufgezeichneten Zugriffsfolgen (Tab.3.9, Umfang ZF) zeigt.

Auffallend ist die Differenz der major Page Faults zwischen den Testsystemen beim Szenario OpenOffice. Ursache kann eigentlich nur der unterschiedliche X-Server, bzw. Window-Manager der beiden Testsysteme sein. Vermutlich werden

| | | major PFs | Datenmenge [MB] | seq. read [s] | Start ohne Disk-I/O [s] | theor. Maximum [s] | Steigerung gegenüber Kaltstart |
|---|---|---|---|---|---|---|---|
| Laptop | emacs | 1420 | 5.54 | 0.28 | 0.20 | 0.48 | 5.0 |
| | mozilla | 3466 | 13.54 | 0.68 | 2.41 | 3.17 | 2.3 |
| | ooffice | 8349 | 32.61 | 1.63 | 2.97 | 4.73 | 3.0 |
| Desktop | emacs | 1418 | 5.54 | 0.10 | 0.09 | 0.19 | 5.7 |
| | mozilla | 4223 | 16.50 | 0.30 | 1.37 | 1.67 | 2.3 |
| | ooffice | 10417 | 40.96 | 0.74 | 2.26 | 3.00 | 2.4 |

Tabelle 3.3: Theoretisch erreichbare Performancesteigerung der Ausführungszeiten

am Laptop vor der Messung durch den Start von X-Server und Window-Manager auch von OpenOffice benötigte Shared Libraries geladen, die am Desktop nicht geladen werden.

**Abschätzung der theoretisch möglichen Performancesteigerung**

Durch Division der gelesenen Datenmenge durch die gemessene Blockgeräte I/O Performance erhält man die Zeit für das sequentielle Lesen der errechneten Datenmenge (Tab.3.3, seq. read). Zur Berechnung wurden dabei die mit `hdparm -t` ermittelte I/O Performance verwendet, da beim Start der Testszenarien lt. GNU `time` kein Dateisystem-I/O erfolgt.

Führt man ein Testszenario zweimal hintereinander aus, so befinden sich bei der zweiten Ausführung alle dafür benötigten Daten bereits im Speicher. Misst man die Zeit dieser zweiten Ausführung (Tab.3.3, Start ohne Disk I/O) und addiert man diese mit der Zeit für das sequentielle Lesen der eingelesenen Daten, so erhält man eine Abschätzung der Ausführungszeit der Testszenarien (Tab.3.3, theor. Maximum). Der Faktor der Performancesteigerung gegenüber normalen Anwendungsstart ist in Tabelle 3.3, in der letzten Spalte angegeben.

Diese Abschätzung ist der best-case, da einerseits für die Berechnung die Blockgeräte I/O Performance bei sequentiellem Lesen verwendet wurde und andererseits die errechnete gelesene Datenmenge wegen Readahead nicht ganz der tatsächlich eingelesenen Datenmenge entspricht. Eine mögliche Parallelausführung von Preloading und Anwendungsstart wurde bei der Abschätzung ebenfalls nicht berücksichtigt.

### 3.3.2 Messmethoden und ermittelte Messwerte

Die in Tabelle 3.4 verwendeten Spaltenbezeichnungen haben folgende Bedeutung:

- **Kaltstart**: Zeit für die Ausführung eines Testszenarios ohne Verwendung des PLS nach Reboot und Login.

- **Warmstart**: Zeit für die zweite Ausführung der Hintereinanderausführung der Testszenarios. Dieser Wert gibt die Ausführungszeit an, wenn sich alle für das Szenario benötigten Daten im Speicher befinden.

- **(pld) Warmstart**: Zeit für die Ausführung eines Szenarios nach Preloading der entsprechenden Zugriffsfolge, aber ohne Zeit für das Preloading. Dieser Wert wurde nach manuellem Preloading der Zugriffsfolge des Szenarios gemessen und soll, verglichen mit der Zeit für den Warmstart den Overhead des Recordings zeigen.

- **pld**: Zeit von Beginn des Preloadings bis Ende des Einlesen der Blockgerätedaten. Um diesen Wert zu ermitteln wurde nach dem Generieren der Requests durch das Preloading ein einzelner Block synchron vom Blockgerät gelesen. Der synchrone Blockrequest wurde so gewählt, dass er am Ende der vom Preloading generierten Requests eingefügt wurde und kein Request Clustering möglich ist. Gemessen wurde die Zeit von Beginn des Preloadings bis zum Ende des synchronen Requests. Für den Blockrequest wurde ein Korrekturwert von der gemessenen Zeit abgezogen.

- **exec time pld**: die Ausführungszeit der Preloadingfunktion. Diese Zeit entspricht der Zeit für das Generieren der bio Requests. Subtrahiert man den Wert mit der Zeit für das Einlesen der Daten (pld), so erhält man die Zeitspanne, in der theoretisch eine Parallelausführung von Preloading und Anwendungsstart möglich ist. Weiters zeigt sich in den Zeiten das congestion Verhalten der Request Queue (Kap.3.3.5).

- **pls**: die Ausführungszeit eines Testszenarios unter Verwendung des PLS mit aktiviertem Preloading und Recording.

- **Steigerung Kaltstart**: Performancesteigerung bei der Ausführung der Testszenarien unter Verwendung des PLS mit aktivierten Recording und Preloading gegenüber Kaltstart.

Alle Werte wurden mit dem Kommando `time` (nicht GNU Time) in der `bash` Shell nach Reboot und wenn nötig (bei Mozilla und OpenOffice), Start von X-Server und Window-Manager ermittelt.

### 3.3.3 Tatsächliche Performancesteigerung

Die Ergebnisse der Messungen zum Zeitverhalten der Testszenarien bei Verwendung des PLS sind in Tabelle 3.4 zusammengefasst. Die Performance bei der Ausführung der Testszenarien konnte am Testsystem Laptop im Schnitt um den Faktor 1.7 und am Testsystem Desktop im Schnitt um den Faktor 1.5 gesteigert werden.

Am Laptop konnte also eine grössere Performanceverbesserung als am Desktop erzielt werden. Vermutlich kommt die Optimierung der Zugriffsfolgen hinsichtlich Sequentialität am Laptop mit der wesentlich langsameren Festplatte besser

| | | Kalt-start [s] | Warm-start [s] | (pld) Warmstart [s] | pld [s] | exec time pld [s] | Parallel-ausführung [s] | PLS [s] | Steigerung PLS geg. Kaltstart |
|---|---|---|---|---|---|---|---|---|---|
| laptop | emacs | 2.438 | 0.201 | 0.205 | 1.427 | 0.287 | +0.158 | 1.474 | 1.7 |
| | mozilla | 7.373 | 2.406 | 2.642 | 2.006 | 0.787 | -0.311 | 4.959 | 1.5 |
| | ooffice | 14.226 | 2.967 | 3.017 | 5.679 | 4.137 | +0.711 | 7.985 | 1.8 |
| desktop | emacs | 1.086 | 0.087 | 0.117 | 0.389 | 0.002 | -0.201 | 0.707 | 1.5 |
| | mozilla | 3.897 | 1.368 | 1.600 | 0.383 | 0.006 | -0.650 | 2.633 | 1.5 |
| | ooffice | 7.289 | 2.264 | 2.545 | 2.246 | 0.396 | +0.137 | 4.654 | 1.6 |

Tabelle 3.4: Tatsächliche Performancesteigerung der Ausführungszeiten

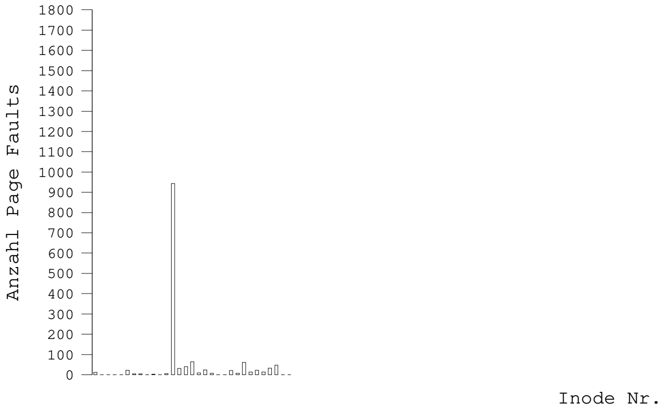


Abbildung 3.1: Fragmentierung der Page Faults beim Start von GNU Emacs

zum tragen. Diese Überlegenheit der Festplatte des Testsystems Desktop zeigt sich auch in den gemessenen Zeiten für den I/O-lastigen Kaltstart: am Desktop starten die Testszenarien ca. um den Faktor 2 schneller. Beim CPU-lastigen Warmstart ist der Vorsprung des Desktops vor allem bei OpenOffice längst nicht so gross.

Auch ist die leichte Tendenz zu erkennen, dass bei Testszenarien mit umfangreicheren Zugriffsfolge, wie OpenOffice, höhere Steigerungsfaktoren erreicht werden konnten. So konnte auf beiden Testsystemen durch den Einsatz des PLS bei OpenOffice die grösste Steigerung erzielt werden, was mit den besseren Optimierungspotential einer umfangreicheren Zugriffsfolge erklärt werden kann.

### Abweichung vom theoretischen erreichbaren Maximum

Wie erwartet, konnte die Performance der Testszenarien durch Einsatz des PLS nicht die in Tabelle 3.3 errechneten theoretischen Maxima erreichen, da selbst optimierte Zugriffsfolgen asequentiellen Charakter aufweisen.

Grund für diese Asequentialitäten ist die in den Abbildungen 3.1, 3.2 und 3.3 für das Testsystem Laptop gezeigte Fragmentierung der Blockgerätezugriffe beim Start der Testszenarien auf Shared Libraries und die ausführbare Dateie. Die Grafiken wurden durch Analyse der beim Recording aufgezeichneten Zugriffsfolgen

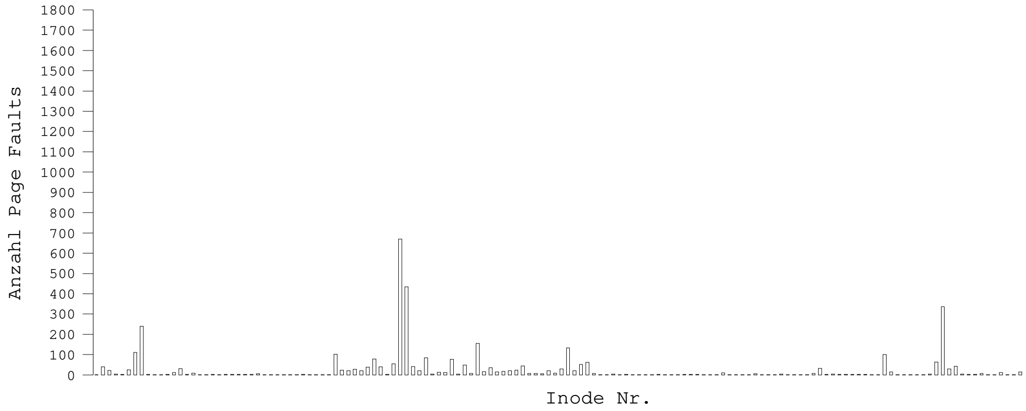


Abbildung 3.2: Fragmentierung der Page Faults beim Start von Mozilla

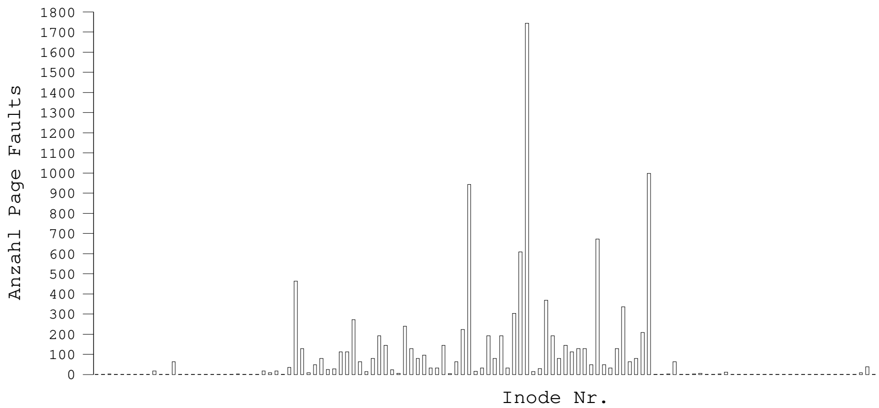


Abbildung 3.3: Fragmentierung der Page Faults beim Start von OpenOffice

erstellt und ergaben eine Verteilung der Page Faults auf 33, 210 und 122 Dateien bei den Testszenarien Emacs, Mozilla und OpenOffice. Die geringste Fragmentierung zeigen die Blockgerätezugriffe von GNU Emacs (Abb.3.1). Über 90% der Zugriffe erfolgen auf die ausführbare Datei.

In welchem Ausmass die Asequentialitäten der Zugriffsfolge des Demand Pagings durch Sektorsortierung beseitigt werden können, hängt von der physikalischen Anordnung der Daten am Blockgerät ab. So betrug die Auslastung des Dateisystems am Testsystem Laptop zum Zeitpunkt der Messungen 93% bei 5% Dateisystemreserve, wobei Mozilla und OpenOffice extra für diese Arbeit, also in ein relativ ausgelastetes Dateisystem, installiert wurden. Es ist daher wahrscheinlich, dass die Dateien dieser Anwendungen teilweise fragmentiert am Blockgerät angeordnet werden mussten.

Dennoch kann durch die Verwendung des PLS die Startzeit der Testanwendungen am betagten Laptop beinahe auf das Niveau des modernen Desktopsystems gesenkt werden.

### 3.3.4 Recording Overhead

Subtrahiert man in Tabelle 3.4 die Werte für den Warmstart der Testszenarien nach Preload mit denen des Warmstarts, so erhält man den Overhead des Recordings.

Voraussetzung für die Berechnung des Recordingoverheads ist, dass beim Warmstart nach Preloading kein Blockgeräte-I/O mehr erfolgt. Die entsprechende Zugriffsfolge muss daher vollständig sein.

Der Wert beschreibt die Zeit für die Initialisierung einer Recording Session sowie in weiterer Folge den Lookup für jedes Vektorelement der bio Requests des Anwendungsstarts in der Hashtable der entsprechenden Zugriffsfolge. Beim Recording von OpenOffice werden so am Laptop ca. 11300 Lookups durchgeführt.

Allgemein ist der Overhead für das Recording auf allen Testsystemen messbar. Grund dafür dürfte die Hashtable zur Indizierung der Pages einer Zugriffsfolge sein. Der Hashwert einer Page wird auf Basis des Startsektors modulo der Arraygrösse der Hashtable berechnet. Obwohl die Arraygrösse der Hashtable beim Reimport der Zugriffsfolge auf eine Fill Rate von 50% dimensioniert wird, kommt es durch die verwendete Hashfunktion dennoch zu vielen Kollisionen.

| Testszenario | Grösse HT-Array | Einträge | Kollisionen | max. Länge Kollisionsliste |
|---|---|---|---|---|
| emacs | 2936 | 1468 | 1101 | 9 |
| mozilla | 7782 | 3891 | 830 | 4 |
| ooffice | 22764 | 11382 | 4393 | 11 |

Tabelle 3.5: Testsystem Laptop - Hashtablestatistik nach Import

Tabelle 3.5 zeigt am Testsytem Laptop die Hashtablestatistik nach Import der Zugriffsfolgen der Testszenarien. Die längste Kollisionsliste der Hashtable von OpenOffice enthält z.B. 11 Elemente. Ist die Zugriffsfolge vollständig und wurden keine überflüssigen Zugriffe aufgezeichnet, so muss beim Anwendungsstart jede

| | | exec time pld [s] | pld [s] | Requests | Request-Queue congested | Requests in der letzen Etappe |
|---|---|---|---|---|---|---|
| laptop | emacs | 0.287 | 1.427 | 164 | 1 | 51 ( 31%) |
| | mozilla | 0.787 | 2.006 | 224 | 1 | 111 ( 50%) |
| | ooffice | 4.137 | 5.679 | 502 | 4 | 50 ( 10%) |
| desktop | emacs | 0.002 | 0.389 | 74 | 0 | 74 (100%) |
| | mozilla | 0.186 | 0.383 | 145 | 1 | 32 ( 22%) |
| | ooffice | 0.396 | 2.246 | 214 | 1 | 101 ( 47%) |

Tabelle 3.6: Congestion-Verhalten der Request Queue beim Preloading

Page der entsprechenden Zugriffsfolge genau einmal gesucht werden. Eine effizientere Datenstruktur, wie z.B. ein B-Tree, könnte den Overhead des Recordings einschränken.

### 3.3.5 Auswirkung des Congestion Verhaltens

In Tabelle 3.6 sind Ergebnisse von Messungen zum Congestion Verhalten der Request Queue während des Preloadings aufgelistet.

Die Spalte drei (exec time pld) zeigt nochmals die Zeit für das Generieren der asynchronen Requests des Preloadings. Spalte vier (pld) zeigt die Zeit für das Einlesen der Requestdaten vom Blockgerät. Die Differenz dieser beiden Zeiten ergibt den Zeitraum, in dem Daten-I/O und Anwendungsstart parallel ausgeführt werden könnten.

Die grösste dieses Zeitraums hängt wesentlich vom Congestion Verhalten der Request Queue ab. In der Tabelle 3.6 ist die Anzahl der Requests und die Anzahl der Wechsel der Request Queue in den Zustand Congested beim Preloading angeführt. Die Messungen zeigen, dass die Request Queue z.B. beim Preloading der Zugriffsfolge von OpenOffice durchaus mehrmals in den Zustand Congested versetzt wird. Wie oft, hängt von der Anzahl der zum Preload der entsprechenden Zugriffsfolge benötigten Requests ab.

Testsystem Desktop profitiert von der höheren Requestkapazität der Festplatte. Ein Request darf am Desktop 2048 Sektoren, oder 256 Page Frames bei einen Seitengrösse von 4K umfassen. Das Blockgerät vom Laptop kann hingegen nur 256 Sektoren, oder 32 Page Frames pro Request bei gleicher Seitengrösse bearbeiten. Entsprechend mehr Requests werden am Testsystem Laptop zum Einlesen einer Zugriffsfolge benötigt, womit die Request Queue womöglich öfter in den Zustand Congested wechseln muss.

Das Preloading der Zugriffsfolge von OpenOffice veranlasst die Queue am Testsystem Laptop vier mal zum Wechsel in den Zustand congested, wonach im Zuge der letzten Etappe des Preloadings noch 50 Requests zu generieren sind. Damit werden 452 oder 90% der Requests synchron eingelesen, was die Annäherung der Zeiten von Spalte drei (exec time pld) und vier (pld) in Tabelle 3.6 erklärt.

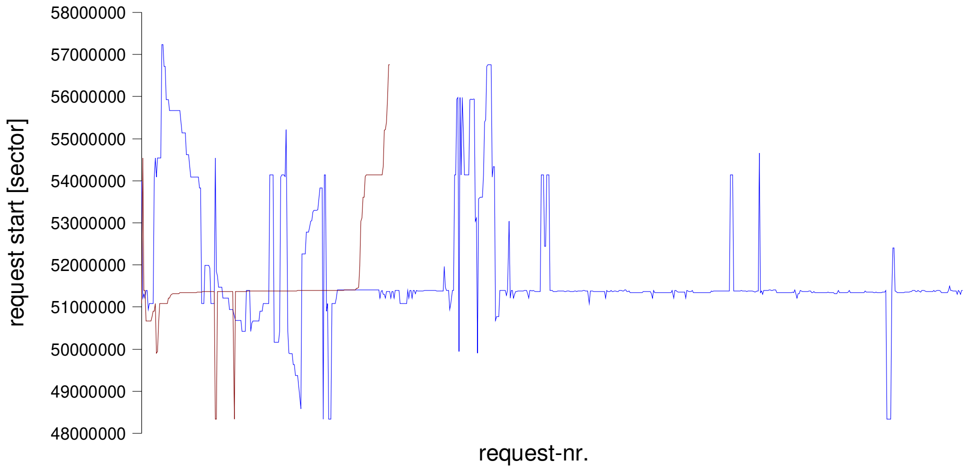


Abbildung 3.4: Laptop: Zugriffsfolgen beim Start von Mozilla 1.0.2

### 3.3.6 Parallelisierung von Preloading und Anwendungsstart

Wie in Kapitel 2.5.2 besprochen, kann es unter bestimmten Voraussetzungen zur Parallelausführung von Preloading und Anwendungsstart kommen. Neben den Congestion Verhalten der Request Queue spielt vor allem die Umsortierung der Zugriffsfolge des Demand Pagings durch das PLS eine Rolle.

Addiert man die Zeiten für den Warmstart nach Preload (Tab.3.4, (pld) Warmstart) mit der Zeit für das Einlesen der Requestdaten (Tab.3.4, pld) und vergleicht diesen Wert mit der gemessenen Zeit für Preload und Start (Tab.3.4, PLS) so erkennt man den Zeitraum der Parallelausführung von Preloading und Anwendungsstart (Tab.3.4, Parallelausführung). Hat Parallelausführung stattgefunden, so ist dieser Wert positiv. Ein negativer Wert bedeutet, dass die sequentielle Ausführung von Preloading und Anwendungsstart schneller ist, als der Versuch der Parallelausführung. Anscheinend blockieren sich Preloading und Anwendungsstart. Die Ursache dafür konnte ich allerdings nicht feststellen.

Am ausgeprägtesten ist die Parallelausführung für GNU Emacs am Testsystem Laptop. Hier wird der Anwendungsstart, sicher auch begünstigt durch die kurze Ausführungszeit des Warmstarts, fast vollständig parallel mit dem Preloading durchgeführt. Für OpenOffice ist der Parallelisierungseffekt ebenfalls auf beiden Testsystemen messbar. Bei Mozilla ist keine Parallelausführung festzustellen.

Auf beiden Testsystemen fällt bei Mozilla die im Verhältnis zur Zeit für das Laden der Zugriffsfolgedaten lange User Time auf (Tabelle 3.8, user time). Beachtet man die Visualisierung der Zugriffsfolge in Abbildung 3.4 und 3.5, so kann man anhand des Graphen der Zugriffsfolge des Demand Pagings (blau) erkennen, dass beim Anwendungsstart sehr früh auf obere Regionen des Sektorbereichs zugegriffen wird. Aufgrund der aufsteigenden Sektorsortierung (roter Graph) werden diese Bereiche aber erst gegen Ende des Preloadings geladen, womit auch der Anwendungsstart entsprechend lange blockiert wird.

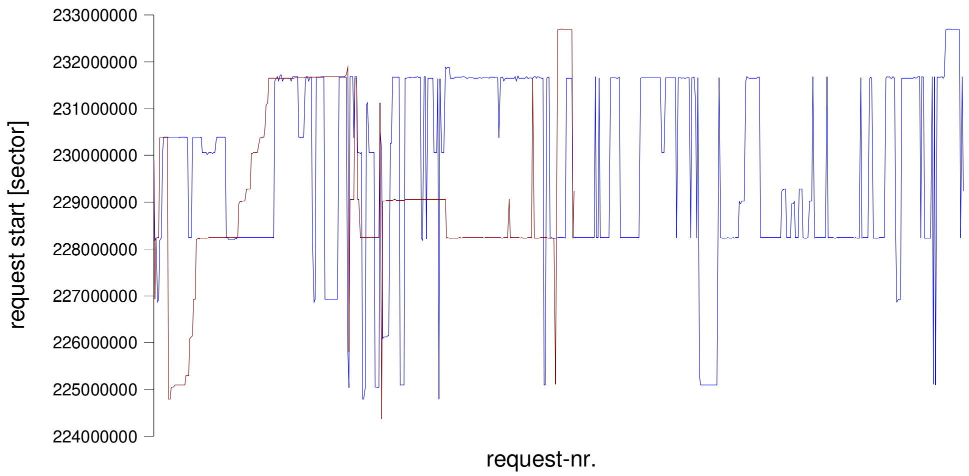


Abbildung 3.5: Desktop: Zugriffsfolgen beim Start von Mozilla 1.0.2

| | | real time | user time | system time | Kontext-Switches Zeitablauf | Kontext-Switches I/O |
|---|---|---|---|---|---|---|
| laptop | emacs | 2.44 | 0.17 | 0.04 | 5 | 187 |
| | mozilla | 7.23 | 2.01 | 0.11 | 822 | 823 |
| | ooffice | 14.20 | 2.28 | 0.19 | 726 | 959 |
| desktop | emacs | 1.07 | 0.06 | 0.02 | 5 | 174 |
| | mozilla | 3.89 | 1.15 | 0.07 | 1301 | 857 |
| | ooffice | 7.18 | 2.14 | 0.12 | 846 | 1029 |

Tabelle 3.7: CPU- und Zeitverhalten der Testszenarien ohne PLS

Die aufsteigende Sortierung der Elemente einer Zugriffsfolge nach Sektoren ist optimal hinsichtlich Blockgeräte I/O Performance. Mit alternativen Sortiermethoden, welche mehr Rücksicht auf den Parallelisierungseffekt nehmen, erhält man daher immer eine schlechtere I/O Performance. Bei Anwendungen wie Mozilla, bei denen der Anteil der User Time an der Zeit für das Preloading entsprechend lang ist, kann eine alternative Sortiermethode von Vorteil sein. Durch die lange User Time liegt viel, durch die strenge Sektorsortierung nicht genutztes Potential in der Parallelausführung. Andererseits können sich Startzeiten von Anwendungen mit geringerer User Time durch die schlechtere I/O Performance einer alternativen Sortierung wieder verlängern.

## 3.4 CPU- und Speicherverhalten der Testszenarien

GNU Time liefert nicht nur Zeitinformationen, sondern auch Werte zum CPU- und Speicherverhalten der Testszenarien. Bei den Messungen handelt es sich um einmalige Messungen, welche nach Reboot und Login auf einem unbelasteten System durchgeführt wurden. Die Ergebnisse der Messungen ohne Verwendung von Pre-

| | | real time | user time | system time | Kontext-Switches Zeitablauf | Kontext-Switches I/O |
|---|---|---|---|---|---|---|
| laptop | emacs | 1.51 | 0.16 | 0.04 | 6 | 21 |
| | mozilla | 4.81 | 2.08 | 0.09 | 864 | 368 |
| | ooffice | 8.01 | 2.26 | 0.37 | 693 | 78 |
| desktop | emacs | 0.71 | 0.06 | 0.01 | 3 | 10 |
| | mozilla | 2.63 | 1.09 | 0.07 | 1531 | 388 |
| | ooffice | 4.63 | 2.10 | 0.14 | 844 | 118 |

Tabelle 3.8: CPU- und Zeitverhalten der Testszenarien mit PLS

loading und Recording sind in Tabelle 3.7 angeführt.

Interessant an den Werten ist das Verhältnis von Real- zu User- und System Time. Es zeigt, dass die Prozesse die meiste Zeit des Anwendungsstarts mit Warten auf Blockgeräte I/O verbringen. Entsprechend hoch ist die Anzahl der Kontext-Switches bezüglich I/O. Dieses Verhalten ist am Desktop durch die im Vergleich zur Festplatte noch schnellere CPU ausgeprägter als am Laptop.

Tabelle 3.8 zeigt das CPU- und Speicherverhalten bei der Verwendung des PLS. Wie erwartet sinkt die Anzahl der Kontext-Switches wegen I/O. Bei rein sequentieller Ausführung von Preloading und des entsprechenden Testszenarios geht dieser Wert gegen null. Da zum Zeitpunkt des Anwendungsstarts wegen der Parallelausführung das Lesen der Seiten vom Blockgerät noch nicht beendet ist, kommt es weiterhin zu Kontext-Switches aufgrund von I/O.

Die Ergebnisse für Testsystem 1 zeigen, dass die Anzahl der I/O Kontext-Switches durch das Preloading bei Mozilla (-55%) nicht so stark gesenkt wurden, wie bei GNU Emacs (-88%), bzw. OpenOffice (-91%). Daraus kann geschlossen werden, dass die Sektorsortierung für den Startvorgang von Mozilla günstig ist. Dies zeigt auch die im Vergleich zur Zeit für das Preloading relativ lange User Time bei der Ausführung des Szenarios. Allgemein zeigt sich ein Zusammenhang zwischen der Minimierung der I/O Kontext Switches durch Preloading und den Grad der Optimierung der Ausführungszeit der Testszenarien (Tabelle 3.4, Spalte 8) durch den Einsatz von Preloading.

### 3.4.1 Requestanzahl und Transferrate

Bei den in Tabelle 3.9 zusammengefassten Messungen wurde versucht, einen Zusammenhang zwischen der Requestauslastung, bzw. Requestanzahl und der beim Preloading einer Zugriffsfolge erzielten I/O Performance zu zeigen. Dazu sind ab Spalte drei die Anzahl der Elemente der Zugriffsfolge (Umfang ZF), die Dauer für das Einlesen der Zugriffsfolgen (Pld), die Menge der eingelesenen Daten (Pld daten), die daraus errechnete I/O Performance (I/O Perf.), sowie die Anzahl der benötigten Requests (Requests) und die Auslastung derselben (Request Auslastung) angegeben.

Die Anzahl der für das Laden der Zugriffsfolge benötigten Requests fällt am Testsystem Desktop deutlich niedriger aus, als am Testsystem Laptop. Grund dafür ist die Fähigkeit des Desktop-Blockgeräts Requests im Umfang von 2048 Sektoren

| | | Umfang ZF | Pld [s] | Pld-daten [MB] | I/O Perf. [MB/s] | Requests | Request-kapazität [Sektoren] | Request Auslastung [%] |
|---|---|---|---|---|---|---|---|---|
| laptop | emacs | 1467 | 1.427 | 5.73 | 4.02 | 164 | 256 | 27.9 |
| | mozilla | 3891 | 2.006 | 15.20 | 7.58 | 224 | | 54.3 |
| | ooffice | 11386 | 4.137 | 44.48 | 10.75 | 502 | | 70.9 |
| desktop | emacs | 1465 | 0.389 | 5.72 | 14.70 | 77 | 2048 | 7.4 |
| | mozilla | 4498 | 0.383 | 17.54 | 45.87 | 337 | | 5.2 |
| | ooffice | 11665 | 2.264 | 45.57 | 20.13 | 252 | | 18.1 |

Tabelle 3.9: Zusammenhang zwischen der Requestanzahl, Auslastung und Transferrate

zur je 512B durchführen zu können, während Testsystem Laptop nur Requests von 256 Sektoren zu ebenfalls je 512B bearbeiten kann. Daher sind für den Preload der annähernd gleichen Datenmengen am Desktop wesentlich weniger Requests notwendig. Die Ausnahme bildet die Anzahl der benötigten Requests für das Preloading der Zugriffsfolge von Mozilla am Desktop. Dies kann einerseits mit dem grösseren Umfang der Zugriffsfolge und andererseits mit einer ungünstigeren Anordnung der Daten am Blockgerät erklärt werden.

Die Messergebnisse lassen darauf schliessen, dass mit dem Umfang der Zugriffsfolge die Auslastung der Requests und somit die Transferrate steigt, was mit dem grösseren Optimierungspotential umfangreicherer Zugriffsfolgen erklärt werden kann. aufgrund der höheren Maximalkapazität der Requests des Blockgeräts am Testsystem 2 konnte die I/O Performance beim Preloading gegenüber Testsystem 1 mindestens verdoppelt werden.

Gleichzeitig sank aber, verglichen mit Testsystem 1, die Auslastung der Requests am Testsystem 2. Das zeigt doch, dass selbst durch die Optimierung der Zugriffsfolgen die viel grössere Kapazität der Requests am Testsystem 2 nicht vollständig genutzt werden kann.

Betrachtet man z.B. OpenOffice so bedeutet eine durchschnittliche Auslastung der Requests am Testsystem Laptop von 70.9% eine durchschnittliche Requestgrösse von ca. 23 Page Frames. Am Desktop bedeutet eine durchschnittliche Auslastung von 18.1% eine durchschnittliche Requestgrösse von 46 Page Frames. Auch wenn die Maximalkapazität von 256 Page Frames am Desktop selten genutzt wird, so ist der durchschnittliche Umfang der Requests doch deutlich höher wie am Laptop.

Beim Preloading der Daten der verschiedenen Testszenarios konnte, wie vermutet, die in Tabelle 3.2 gelistete Blockgeräte I/O Performance nicht erreicht werden, wurden doch bei den Messungen in Tabelle 3.2 der sequentielle Zugriff gemessen. Selbst die optimierten Zugriffsfolgen weisen einen zu hohen Grad an Asequentialität auf, um die I/O Performance bei sequentiellen Zugriff zu erreichen.

## 3.5 Zugriffsverhalten der Testszenarien

Ziel dieser Messung ist es, das Zugriffsverhalten einer Anwendung auf ein Blockgerät auf Ebene des low level Treibers, also unterhalb der Request Queue eines Blockgeräts, zu ermitteln. Aufgrund von Request Clustering und Request Reordering, erhält man nur unterhalb der Request Queue ein unverfälschtes Bild der Zugriffsfolge einer Anwendung auf ein Blockgerät.

### 3.5.1 Implementierung des Messmoduls

Zur Aufzeichnung der Zugriffsfolge wurde ein Modul implementiert, das Requestparameter bei der Entnahme der Requests durch den low level Treiber aus der Request Queue des Blockgeräts in einem Array speichert.

Aufgezeichnet werden dabei die PID des Prozesses, der den Request gestellt hat, den Startsektor des Requests, sowie die Länge des Requests in Sektoren. Die Schnittstelle zum Kernel ist wie beim Preloading und Recording ein globaler Zeiger, welcher auf die Funktion zur Aufzeichnung der Parameter der Requests zeigt, wenn das Modul geladen ist. Dieser Zeiger wurde in der generischen IDE-Treiberimplementierung des Kernels, genauer in der Funktion `start_request` [2], definiert.

### 3.5.2 Messverfahren

Die Aufzeichnung beginnt mit dem Laden des Moduls. Beim Entladen des Moduls werden die aufgezeichneten Daten nach `stdout` ausgegeben. Um die Messung nicht zu verfälschen, darf kein anderer I/O als der des zu beobachtenden Prozesses zum Zeitpunkt der Aufzeichnung aktiv sein. Daher sollte z.B. syslog deaktiviert werden. Im Zuge einer Messung wurden folgende Schritte durchgeführt:

1. Laden des Moduls
2. Entladen des Moduls
3. Erneutes Laden des Moduls
4. Ausführen des Testszenarios
5. Entladen des Moduls

Das Laden und Entladen des Moduls verursacht ebenfalls I/O. Darum wird das Modul vor der eigentlichen Messung einmal ge- und entladen. Die Ausgabe beim Entladen des Moduls wurde in eine Datei umgeleitet. Die Messung wurde dreimal für jedes Testszenario durchgeführt. Nur Requests, die in allen Messungen vorkamen, wurden berücksichtigt. Für alle Testszenarios wurden die Zugriffsfolgen einmal mit und einmal ohne Verwendung von Preloading aufgezeichnet.

[2] siehe `Kernel 2.6.0 drivers/ide/ide-io.c`

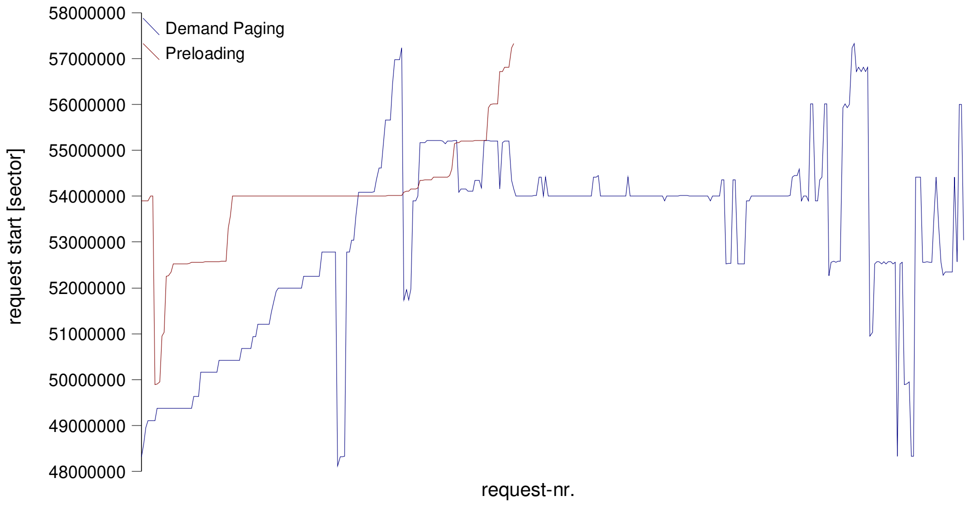


Abbildung 3.6: Laptop: Zugriffsfolgen beim Start von GNU Emacs 21.2.1

### 3.5.3 Messergebnisse

Die Ergebnisse sind in den Abbildungen 3.6, 3.4, 3.7, 3.8, 3.5 und 3.9 visualisiert. Die Ordinate repräsentiert den Datenbereich der Zugriffsfolgen in Sektoren. Die Abszisse stellt die Reihenfolge der Requests dar, hat also die Einheit 1. Es ist zu beachten, dass die Requests unabhängig von ihrer Grösse dargestellt sind, da dies mit der zur Visualisierung der Daten verwendeten Anwendung nicht möglich war. Jeder waagrechte Abschnitt eines Graphen repräsentiert einen Zugriff, während jeder nicht waagrechte Abschnitt als Head Movement des Blockgeräts interpretiert werden kann. In jeder Abbildung ist die nicht optimierte Zugriffsfolge des Demand Pagings als blauer und die optimierte Zugriffsfolge des Preloadings als roter Graph dargestellt.

Die Daten der Messung wurden weiters in der Tabelle 3.10 zusammengefasst. In je zwei Zeilen wurden die Daten für ein Szenario pro Testsystem je einmal mit und einmal ohne Preloading dargestellt.

Gut zu erkennen ist der Zusammenhang zwischen den Werten in den Spalten Requestanzahl und Gesamtauslastung. Durch die bessere Auslastung der Requests bei Verwendung des PLS sinkt die Anzahl der zum Laden der Daten notwendigen Requests. Dies bewirkt auch ein günstigeres Congestion Verhalten der Request Queue.

In Spalte Movement zeigt summiert für alle Requests den Weg in Sektoren vom Ende eines Requests zum Anfang des nächsten Requests. Diese Werte können zwar als Head Movements in Sektoren interpretiert werden, wobei jedoch die unbekannte physikalische Geometrie des Blockgeräts beachtet werden muss. Dennoch sinkt der Gesamtweg in Sektoren durch die Sektorsortierung beträchtlich.

Dies ist auch in den den Abbildungen 3.6, 3.4, 3.7, 3.8, 3.5 und 3.9 zu sehen. Bei allen Testszenarien ist der rote Graph der optimierten Zugriffsfolge aufgrund

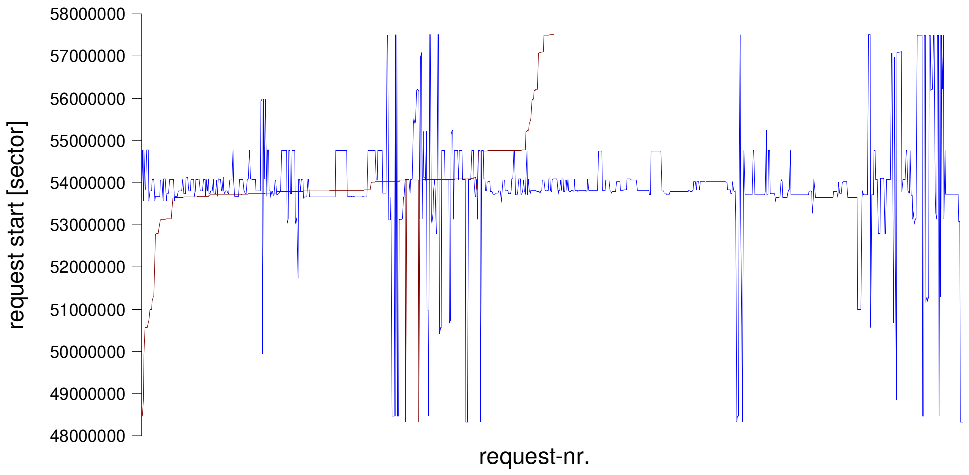


Abbildung 3.7: Laptop: Zugriffsfolgen beim Start von OpenOffice 1.0.1

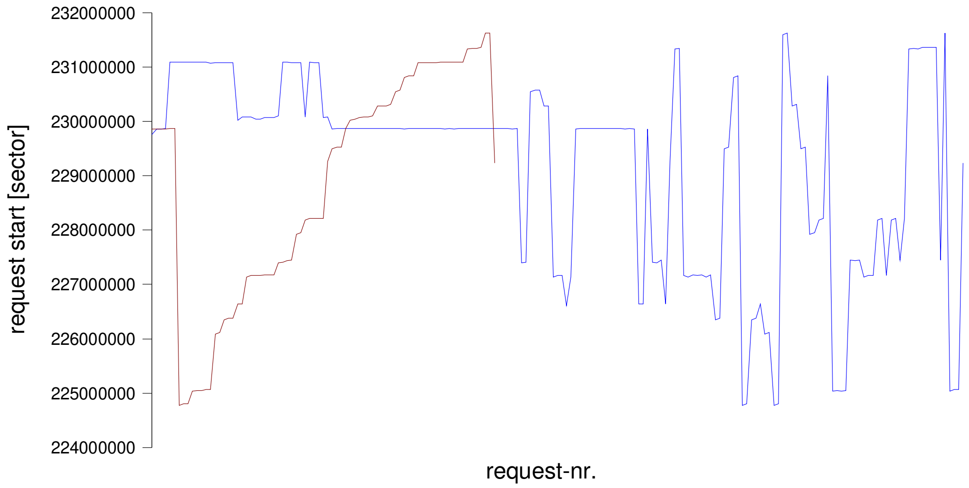


Abbildung 3.8: Desktop: Zugriffsfolgen beim Start von GNU Emacs 21.2.1

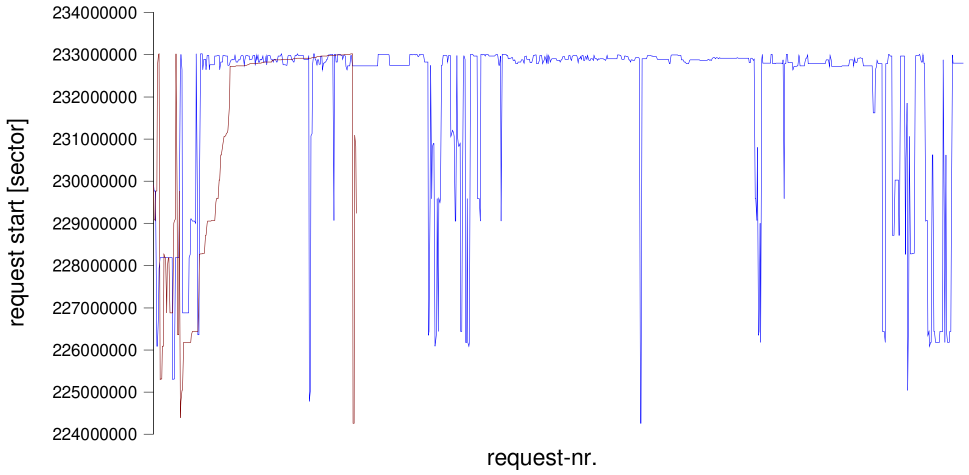


Abbildung 3.9: Desktop: Zugriffsfolgen beim Start von OpenOffice 1.0.1

| | | PL | Sektor-grösse [B] | Request-kapazität [Sektoren] | Anzahl Requests | Gesamt-Auslastung [%] | Movement [sektoren] | Faktor Reduktion |
|---|---|---|---|---|---|---|---|---|
| laptop | emacs | N | 512 | 256 | 361 | 14.3 | 105979490 | |
| | | J | | | 170 | 27.8 | 11623808 | 9.1 |
| | mozilla | N | | | 740 | 18.1 | 162168816 | |
| | | J | | | 226 | 51.7 | 27882672 | 5.8 |
| | ooffice | N | | | 999 | 36.3 | 449362664 | |
| | | J | | | 502 | 71.1 | 32453944 | 13.8 |
| desktop | emacs | N | | 2048 | 181 | 3.2 | 102786088 | |
| | | J | | | 77 | 7.5 | 14325784 | 7.1 |
| | mozilla | N | | | 648 | 2.9 | 334611120 | |
| | | J | | | 337 | 5.7 | 74284968 | 4.5 |
| | ooffice | N | | | 1006 | 4.6 | 333734728 | |
| | | J | | | 252 | 18.7 | 67017792 | 4.9 |

Tabelle 3.10: Auswerung des Recordings der Zugriffsfolgen auf Ebene des low-level Treibers

der besseren Auslastung deutlich kürzer als jener der nicht optimierten Zugriffsfolge. Weiters zeigen die roten Graphen der sektorsortierten Zugriffsfolgen einen erheblich sequentielleren Charakter als die blauen Graphen der nicht optimierten Zugriffsfolgen.

Trotz der theoretisch optimalen Sektorsortierung zeigt die Visualisierung der Zugriffsfolgen aller Testszenarien nicht immer einen optimal sequentiellen Verlauf. Vor allem zu Beginn und gegen Ende der Aufzeichnung der Zugriffsfolge kommt es immer wieder zu Ausreissern ans obere und untere Ende des Sektorenspektrums. Bei allen Testszenarien werden aber 99% der Daten in den sequentiellen Abschnitt der Zugriffsfolge geladen.

Ein Grund für die asequentiellen Bereiche zu Beginn und am Ende der Zugriffsfolgen könnte I/O sein, der durch das Laden und Entladen des Messmoduls verursacht wird. Eigentlich sollten aber die Daten des Messmoduls zu Beginn der Aufzeichnung bereits im Speicher sein, da das Messmodul vorher einmal geladen und wieder entladen wird.

Eine weitere Erklärung könnte die Unvollständigkeit der aufgezeichneten Zugriffsfolgen sein. Ein Vergleich der I/O Aktivitäten beim Anwendungsstart nach Preloading der Zugriffsfolgen mit den I/O Aktivitäten beim Warmstart zeigt aber keinen zusätzlichen I/O bei Verwendung des PLS. Daraus kann geschlossen werden, dass die aufgezeichneten Zugriffsfolgen vollständig sind.

# Kapitel 4

# Verwandte Arbeiten

In diesem Kapitel werden fünf verwandte Arbeiten beschrieben. Arbeiten wie Prefetching oder Prelink verfolgen wie das PLS das Ziel, den Anwendungs- bzw. Systemstart zu beschleunigen. Die anderen Arbeiten verfolgen einen breiteren Ansatz und beabsichtigen die Blockgeräte I/O Performance im Allgemeinen zu steigern. Bis auf Native Command Queuing sind alle Arbeiten in Software realisiert.

## 4.1 Log-Structured Filesystems

Wie das PLS versucht ein Log-Structured Filesystem (LFS) die Sequentialität von Blockgerätezugriffe zu optimieren [SL93]. Dadurch sollen die mechanischen Aktivitäten einer Festplatte minimiert und die I/O Performance optimiert werden.

Das Ziel der Dateisystementwicklung ist die effiziente Nutzung der Festplatte. Die Entwicklung tendiert zur Verwendung immer grösserer Caches. Dadurch werden zwar read Requests reduziert, aber kaum write Requests. Als Flaschenhals der Dateisysteme stellte sich daher immer mehr die write Performance heraus. Der Schreibvorgang auf eine Festplatte ist dann effizient, wenn grosse kontinuierliche Datenbereiche auf einmal geschrieben werden.

Die ersten Unix Dateisysteme führten keine Optimierung der Blockanordnung durch [TH78]. Das führte mit der Zeit zu einer Fragmentierung der Festplatte, welche sich negativ auf die I/O Performance auswirkte. Der nächste Schritt war daher die Entwicklung des Fast File Systems (FFS) [MCKU84]. FFS versucht logisch zusammenhängende Daten physikalisch nahe beieinander am Blockgeräte anzuordnen um so Seeks zu minimieren, was zu einer dramatischen Performancesteigerung der Dateisysteme geführt hat.

Dennoch zeigt auch das FFS Schwächen speziell beim Zugriff auf viele kleine Files, wie in Unix Umgebungen üblich. Die dazu notwendigen vielen Seeks wirken sich negativ auf die Performance des FFS aus. Dies war der Grund für die Entwicklung des Log-Structured Filesystems.

### 4.1.1 Prinzip eines LFS

Ein LFS schreibt alle Daten und Metadaten des Dateisystems sequentiell in eine theoretisch unendliche Datei, das *Log*, fort. Werden Daten geändert, so wird die aktuelle Version der Daten ans Log angehängt, während die alten Daten als ungültig markiert werden.

Ein Vorteil dieses Ansatzes ist, dass zwischen zwei write Operationen kein Seek notwendig ist, da die Daten sequentiell im Log fortgeschrieben werden. Lesezugriffe profitieren von der sequentiellen Anordnung aller Files am Blockgerät. Zugriffsfolgen auf mehrere Files profitieren auch von der zeitlichen Lokalität der Daten.

#### Free Space Management

Das Hauptproblem bei der Implementierung eines LFS ist die Abbildung des theoretisch unendlichen Logs auf ein Blockgerät mit endlicher Kapazität. Erreicht das Log das Ende der Festplatte, so wird es am Anfang fortgesetzt. Aus alten Daten muss daher Platz für neue Daten des Logs gewonnen werden. Ein Free Space Management in Form eines `Cleaner` Prozesses sucht dazu laufend nach alten Files und optimiert den freien Platz durch Zusammenfassen und Umordnen der Datenblöcke. Der Cleaner unterscheidet zwischen oft und selten geschriebenen Files. Da die aktuelle Version eines oft geschriebenen Files bald wieder ungültig wird, konzentriert sich der Cleaner auf die Optimierung der Anordnung von wenig benutzten Files, wie z.B. Executables.

#### Performance

Wie Messungen zeigen, erreicht ein LFS beim Schreiben 62-82% der maximalen Bandbreite, was fast die zehnfache Leistung des FFS ist [SL95]. Leseoperationen skalieren mit der Prozessorleistung und sind von der verfügbaren Bandbreite unabhängig.

Als Problem erwies sich der Overhead für das Free Space Management, welcher einen Leistungsabfall von bis zu 40% verursachen kann. Dieser Cleaner Overhead ist hauptsächlich vom Füllgrad einer Festplatte abhängig. Messungen zeigten, dass ein Füllgrad von 50% einen Leistungsabfall von 33% durch Cleaner Overhead bewirken kann.

### 4.1.2 Vergleich mit dem PLS

Bei der Konzeption des LFS konzentrierte man sich auf die Optimierung der Schreiboperationen, während das PLS ausschliesslich lesende Zugriffe des Anwendungsstarts auf ein Blockgerät optimieren will.

Beide Systeme optimieren die Blockgerätezugriffe hinsichtlich Sequentialität. Das LFS verfolgt dabei den konsequenteren Ansatz. Schreibvorgänge sind durch das Log optimal sequentiell. Da alle Files sequentiell im Log abgelegt werden, ist auch der Lesevorgang einer Datei optimal sequentiell. Das PLS erreicht hingegen keine optimale Sequentialität, da nur die Reihenfolge der Zugriffe auf die für den Anwendungsstart benötigten Daten optimiert wird.

## 4.2 Logisches Prefetching in Windows XP

Eine der Neuerungen des Microsoft Betriebssystems Windows XP ist ein Dienst, der als Prefetching bezeichnet wird. Prefetching soll die Performance von System- und Anwendungsstart verbessern. Dazu sollen Delays bei der System- und Treiberinitialisierung zum Laden (Prefetching) später im Startprozess benötigter Blockgerätedaten genutzt werden [ML01].

### 4.2.1 Arbeitsweise von Prelink

Wie beim PLS werden Parameter von Blockgeräterequests beim System- und Anwendungsstart aufgezeichnet. Diese Informationen werden genutzt, um bei späteren Starts ein asynchrones Prefetching der für den Start benötigten Daten parallel zum Startprozess durchzuführen. Auf Basis der aufgezeichneten Daten erfolgt weiters eine Optimierung des Blocklayouts hinsichtlich Maximierung der I/O Performance beim Prefetching. Aufgezeichnete Zugriffsfolgen werden in Szenarien organisiert.

Wird ein Szenario erneut gestartet, öffnet der Kernel die entsprechende Datei mit den Prefetchanweisungen und generiert die asynchronen Requests. Ein Prefetching eines Szenarios erfolgt nur bei der ersten Ausführung nach dem Systemstart.

#### Organisation der Szenarien

Jedes Szenario wird in einer Datei verwaltet. Grundsätzlich werden zwei verschiedene Arten von Szenarien unterschieden:

Systemstarts werden in einen eigenen Szenario behandelt. Die Monitoring Informationen der letzten acht Systemstarts werden in einer Datei gespeichert, welche eine Minute nach Ende des Systemstarts angelegt oder aktualisiert wird.

Die zweite Art von Szenarien sind die der Anwendungsstarts. Die Prefetch-Instruktionen dieser Szenarien sind in Files mit der Benennung `appname-xxxxxx` gespeichert. Für jeden Anwendungsstart wird eine eigene Datei angelegt.

Die Zuordnung von Anwendungsstarts zu ihren Szenarien erfolgt also mit Hilfe des Dateinamens der ausführbaren Datei. Der Systemstart wird gesondert behandelt.

#### Abbruch des Monitorings eines Szenarios

Das Monitoring eines Anwendungsstarts ist idle-time gesteuert, d.h. es wird dann beendet, wenn der Zeitabstand zwischen den Page Faults ein bestimmtes Intervall überschreitet. Im Gegensatz dazu ist das Recording des Systemstarts nicht idle-time gesteuert. Das Ende des Systemstarts wird dem Monitoring explizit mitgeteilt.

#### Optimierung der Szenarien

Wurde das Prefetching beendet, erfolgt der System- oder Anwendungsstart. Dabei wird das startende Szenario vom Task Scheduler weiter beobachtet. Die vom Task Scheduler aufgezeichneten Informationen werden zur Optimierung der Prefetch-Instruktionen genutzt. Dazu wird beobachtet, wie erfolgreich das Prefetching war, d.h. welche Seiten nach dem Prefetching nachgeladen werden müssen, bzw. auf welche durch das Prefetching in den Speicher geladenen Seiten beim Start des Szenarios nicht zugegriffen wurde.

#### Optimierung der I/O Performance beim Prefetching

Zur Maximierung der Performance des Prefetchings werden die Informationen des Monitorings zur Optimierung der Datenanordnung auf der Festplatte verwendet. Dazu wird in idle-Phasen des Systems versucht, die Daten des System- und Anwendungsstart sequentiell am Blockgerät anzuordnen. Der Prefetcher analysiert dazu die aufgezeichneten Parameter in den Szenarien und schreibt in die Datei `layout.ini` Anweisungen an den Defragmentierer für ein optimiertes Blocklayout. Die Datei wird nach 32 Anwendungsstarts erstellt. Der Defragmentierungsprozess in der ersten längeren idle-Phase des Systems vom Prefetcher gestartet.

### 4.2.2 Vergleich mit dem PLS

Von allen in diesen Kapitel vorgestellten Arbeiten weist das Prefetching die grössten Parallelen zum PLS auf. Das Recording des PLS ist mit dem Monitoring und das Preloading mit dem Prefetching gleichzusetzen. Wie die Zugriffsfolgen des PLS werden auch die Szenarien des Prefetchings persistent gespeichert.

Die aufgezeichneten Szenarien des Monitorings werden in je einer Datei gespeichert. Beim Prefetching werden die Parameter zur Generierung der Requests aus diesen Files gelesen. Im Gegensatz dazu hält das PLS alle Zugriffsfolgen im Speicher. Beim Systemhalt werden alle Zugriffsfolgen in eine Datei geschrieben.

Wie beim PLS angedacht, aber noch nicht implementiert, werden die beim Monitoring gewonnenen Szenarien laufend optimiert. Bevor ein Szenario zum Prefetching freigegeben wird, muss ein Monitoring von 10 Starts der Anwendung erfolgt sein. Ein Prefetching erfolgt nur beim ersten Start einer Anwendung nach dem Systemstart.

Der Systemstart wird beim Prefetching gesondert behandelt. Alle Blockgerätezugriffe während des Systemstarts werden einem Szenario zugeordnet. Das PLS speichert hingegen für jeden Anwendungsstart im Zuge des Systemstarts eine eigene Zugriffsfolge. Zukünftig sollen die Zugriffsfolgen aufeinanderfolgender Anwendungsstarts allerdings automatisiert zusammengefügt werden.

Zur Maximierung der I/O Performance beim Prefetching wird ein Blockreordering auf Basis der Daten des Monitorings durchgeführt. Prefetching geht damit einen Schritt weiter als das PLS, bei dem die physikalische Anordnung der Daten am Blockgerät nicht verändert wird.

## 4.3 Native Command Queuing

Das Native Command Queuing (NCQ) [AH04][AHJC03] des Serial ATA II Standards versucht durch out-of-order Ausführung der Blockgerätezugriffe den Overhead durch die Aktuatorenbewegungen einer Festplatte zu minimieren.

### 4.3.1 Arbeitsweise von Native Command Queuing

In einem System, das weitgehend aus elektronischen Komponenten besteht, haben die mechanische Komponenten des elektromechanischen Gerätes Festplatte die grössten Auswirkungen auf die Systemperformance. NCQ soll daher die mechanischen Abläufe einer Festplatte optimieren. Dazu werden die Kommandos an eine Festplatte in einer internen Queue gesammelt. Mit den Kommandos in der Queue kann ein dynamisches ein Rescheduling, bzw. ein Reordering durchgeführt werden. Am meisten profitieren daher Multithreading Umgebungen von NCQ, bei denen mehrere Threads zugleich asynchronen I/O verursachen.

#### Seek- und Rotation Latency

Programme tendieren dazu, auf die Daten einer Festplatte nicht in der Reihenfolge zuzugreifen, in der diese auf der Festplatte abgelegt sind. Der durch Aktuatorenbewegung verursachte mechanische Overhead wird in Seek- und die Rotation Latency eingeteilt. Die Seek Latency beschreibt die Zeit, die für die Positionierung des Aktuators über den gewünschten Track notwendig ist, während die Rotation Latency die Zeit angibt, die gewartet werden muss, bis die Daten unter den Lesekopf rotiert sind.

#### Optimierungsstrategie

Ältere Ansätze minimierten nur die Seek Distanz und somit die Seek Latency. Die maximale Performance erhält man aber nur dann, wenn sowohl Seek- als auch Rotation Time minimiert werden. Eine minimale Seek Time bedeutet nicht automatisch maximale Performance. So kann der Gesamt-Overhead hauptsächlich aus Rotation Latency bestehen. NCQ versucht daher neben der Seek- auch die Rotation Time zu minimieren. Dazu wird je nach Position der Aktuatoren die Reihenfolge der Kommandos hinsichtlich minimaler Seek- und Rotation Latency umgeordnet.

#### Tagging

Muss eine Festplatte bei ungünstiger Reihenfolge der Anfragen und daraus resultierender Position der Daten mehrmals rotieren, um sämtliche Anfragen auszuführen, kann dies durch eine richtige Reihenfolge theoretisch auf eine Rotation minimiert werden. Damit kann der Durchsatz der Festplatte optimiert werden. Auf

Grund der out-of-order Bearbeitung der Kommandos müssen diese eindeutig bezeichnet werden (tagging). Der Host weiss damit, welches Kommando gerade bearbeitet wurde.

### 4.3.2 Vergleich mit dem PLS

NCQ ist hauptsächlich in Disk-Firmware realisiert, während das PLS eine reine Softwarelösung ist. Im Gegensatz zum PLS versucht NCQ die Reihenfolge der vom Treiber generierten Kommandos zu optimieren. Das PLS hat hingegen keine Kenntnis vom physikalischen Layout einer Festplatte, und optimiert daher eine Ebene höher die Reihenfolge der Blockgeräterequests. Grundsätzlich könnte das PLS vom NCQ profitieren, da beim Preloading eine grosse Anzahl asynchroner Requests generiert wird.

Ebenso wie das PLS Blockgeräterequests aus Sicht der Anwendung out-of-order ausführt, so führt auch das NCQ Kommandos an die Festplatte out-of-order aus. Jedoch wird beim NCQ der Host durch tagging von der Reihenfolge der Bearbeitung explizit informiert. Zwischen dem PLS und der Anwendung erfolgt ein indirekter Informationsaustausch über die Reihenfolge der Requests durch die Synchronisierung von Preloading und Anwendungsstart. Wurden die von der startenden Anwendung benötigten Daten durch das Preloading noch nicht geladen, so wird die Anwendung solange blockiert, bis die Daten im Speicher sind.

Im Gegensatz zum PLS arbeitet das NCQ nicht spekulativ. Das NCQ optimiert über die in der Queue vorhandenen Kommandos und nutzt dabei seine Kenntnis von der physikalischen Geometrie der Festplatte. Das PLS nimmt beim Preloading an, dass eine startende Anwendung wieder auf genau die selben Daten zugreift, wie beim Recording aufgezeichnet.

## 4.4 Prelink

Wie auch das PLS hat Prelink das Ziel den Anwendungsstart zu beschleunigen. Prelink verfolgt dazu jedoch einen völlig anderen Ansatz. Im Gegensatz zum PLS versucht es nicht die Blockgeräte I/O Performance während des Anwendungsstarts zu optimieren, sondern den Overhead durch dynamisches Linken zu minimieren.

### 4.4.1 Linking in ELF und a.out

Das aktuelle Format für ausführbare Files und Shared Libraries in Linux ist das *Executable and Linking Format* (ELF) [TIS95], welches 1995 das *Assembler Output Format* (a.out) ablöste. Die beiden Formate unterscheidet vor allem die Plattformunabhängigkeit ELF-Formats. Die Symbole einer ausführbaren Datei im ELF Format werden nicht wie a.out über statische Adressen, sondern über ihren Namen angesprochen.

### ELF

Shared Libraries im ELF Format bestehen aus positionsunabhängigen Code, der an beliebige Positionen im virtuellen Adressraum eingebunden werden kann. Um ein Programm starten zu können, muss jeden undefinierten Symbol einer Shared Library eine statische Adresse zugewiesen werden. Dieser Vorgang wird *Relocation* genannt.

Ein undefiniertes Symbol ist ein vom Programm referenziertes Symbol, welches nicht im eigenen Programmcode definiert wurde. Solche Referenzen werden entweder durch statisches Linken zur Compile-, oder dynamisches Linken zur Laufzeit aufgelöst.

Der Preis für die Flexibilität von positionsunabhängigen Code ist Relocation Overhead durch den Run Time Dynamic Linker (RTLD) beim Anwendungsstart. Der Overhead hängt von der Anzahl der verwendeten Shared Libraries eines Programms und der Anzahl der externen Referenzen in dieses Shared Libraries ab.

### a.out

Im Gegensatz zum ELF Format musste das Mapping von a.out Shared Libraries durch den Linker an genau definierte, für jede Library eindeutige Slots im Adressraum erfolgen, was die Erstellung von Shared Libraries im a.out Format erschwerte. Die Slots wurden in einem zentralen Adressregister in Form einer Textdatei zur Verfügung gestellt. Das ist heute nicht mehr möglich, da mittlerweile tausende Shared Libraries in Umlauf sind, deren Version, bzw. Art und Anzahl der exportierten Objekte sich laufend ändert. Durch das statische Layout des Adressraums war der Arbeitsaufwand für den dynamischen Linker beim Laden einer Shared Library im a.out Format gering, da Relocation und Namensauflösung bereits zur Linkzeit durchgeführt werden konnten.

## 4.4.2 Das Prinzip von Prelink

Prelink versucht die Flexibilität des ELF-Formats mit den Performance-Vorteilen von a.out zu kombinieren, in dem es Shared Libraries im ELF-Format wieder an definierte Ports im Adressraum einbindet. Der wesentliche Unterschied zu a.out ist der, dass die Eindeutigkeit nicht über alle Installationen, sondern nur über die an Prelink übergebenen ausführbaren Files und ihren Shared Libraries gefordert ist.

Anschliessend führt Prelink mit Hilfe des dynamischen Linker des Systems die Relocation durch und speichert das Ergebnis im ELF Header der ausführbaren Datei. Prelink speichert weiters eine Liste der verwendeten Shared Libraries und deren Checksummen.

Zur Laufzeit überprüft der dynamische Linker, ob das Mapping aller Shared Libraries an die von Prelink zugewiesenen Adress-Slots erfolgreich war. Anhand der Checksumme überprüft der dynamische Linker, ob die Libraries seit dem Prelinking unverändert geblieben sind. Sind alle Voraussetzungen erfüllt, so kann der Linker die Relocationergebnisse aus dem ELF-Header verwenden. Da so viele

Suchoperationen durch Relocation vermieden werden, kann die CPU-Zeit des Anwendungsstarts um bis zu 50% reduziert werden.

### 4.4.3 Vergleich mit dem PLS

Wie auch das PLS führt Prelink ein Monitoring von Aktivitäten während des Anwendungsstarts durch, zeichnet Resultate auf und verwendet diese zur Beschleunigung zukünftiger Anwendungsstarts.

Prelink zeichnet dabei Ergebnisse des dynamischen Linkers auf und speichert diese im ELF-Header der ausführbaren Datei. Anders als das PLS modifiziert Prelink die ausführbare Datei einer Anwendung. Das hat vereinzelt zu Problemen mit Anwendungen geführt, welche z.B. die Systemintegrität mit Hilfe von Checksummen wie MD5 prüfen.

Im Gegensatz zum PLS arbeitet Prelink fast ausschliesslich im User Space. Bei der Implementierung von Prelink wurde vor allem der Runtime Linker modifiziert. Weiters beschleunigt Prelink auch den Warmstart, da die Optimierungen hauptsächlich die CPU-Zeit des Anwendungsstarts betreffen. Beim I/O-lastigen Kaltstart dürfte daher das PLS die besssere Startperformance erreichen. Dennoch ergänzen sich das PLS und Prelink und empfehlen sich daher für den gemeinsamen Einsatz: das PLS optimiert das I/O Verhalten und Prelink reduziert die CPU-Zeit beim Anwendungsstart.

Mittlerweile ist Prelink etabliert und wurde in beinahe alle Linux Distributionen integriert.

## 4.5 Read Ahead

Der Read Ahead Mechanismus [PPC04] des Linux Kernels soll zukünftige Blockgeräterequests anhand vergangener Zugriffsmuster vorhersagen und asynchron ausführen. Dadurch sollen major Page Faults und synchrone Read Requests minimiert und die Requestauslastung maximiert werden. Um eine Verschmutzung des Page Caches mit nicht benötigten Daten zu vermeiden, soll die Vorhersage zukünftiger Zugriffe einen möglichst hohen Treffergrad aufweisen.

### 4.5.1 Elemente des Read Ahead Mechanismus

Im Linux Kernel 2.6.0 wurde der Read Ahead Algorithmus auf der Ebene des Page Caches implementiert. Zur Analyse der Zugriffsmuster werden alle Requests an den Page Cache verwendet. Auf Basis dieser Analyse generiert der Read Ahead Algorithmus die entsprechenden Requests an die Blockgeräte des Systems.

Abbildung 4.1 zeigt die Elemente von Read Ahead. Der Algorithmus definiert ein *Current Window*, das die Page Frames des letzten Requests umfasst. Das *Readahead Window* definiert einen Page Frame Bereich, in den Zugriffe in naher Zukunft wahrscheinlich sind. Die *Next Window Size* gibt die Grösse des nächsten Readahead- oder Current Windows an.

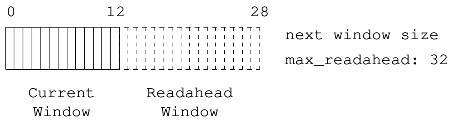


Abbildung 4.1: Kernel 2.6.0 - Komponenten von Read Ahead

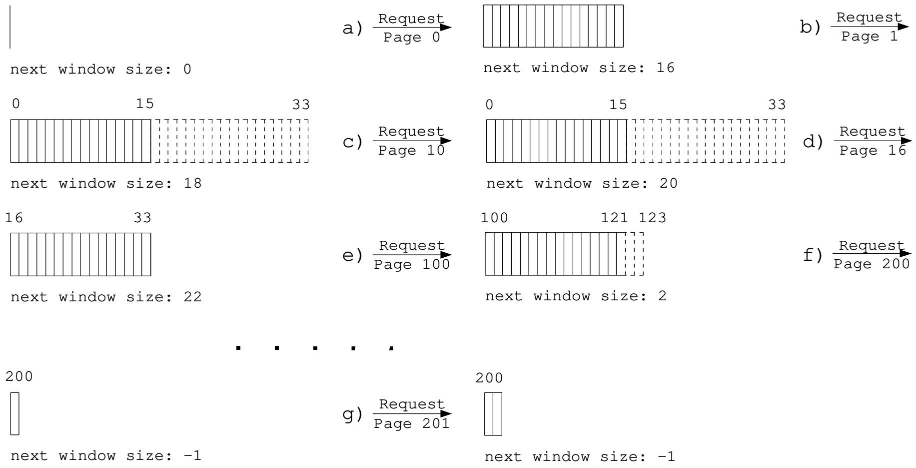


Abbildung 4.2: Kernel 2.6.0 - Beispiel Read Ahead

### 4.5.2 Arbeitsweise von Read Ahead

Am in Abbildung 4.1 gezeigten Beispiel soll die Arbeitsweise von Read Ahead gezeigt werden.

Beim ersten Zugriff auf eine Datei werden `max_readahead`/2 Page Frames gelesen. Diese Page Frames bilden das initiale Current Window. Die Next Window Size wird auf die Grösse des Current Windows initialisiert (Abb.4.2a).

Erfolgt der zweite Zugriff auf den Page Frame sequentiell zum ersten, so wird ein Readahead Window der Grösse Next Window Size anschliessend an das Current Window definiert(Abb.4.2b). Jedesmal, wenn ein Request das Current Window betrifft, wird die Next Window Size um zwei inkrementiert (Abb.4.2c), andernfalls um zwei dekrementiert (Abb.4.2e-f). Die obere Schranke der Next Window Size für jedes Device global mit 32 Page Frames definiert.

Überschreitet ein Request die Grenze des Current Windows hin zur ersten Page Frame des Readahead Windows, so wird das Readahead Window zum neuen Current Window. Das Readahead Window wird zurückgesetzt (Abb.4.2d).

Fällt ein Request weder in das Current-, noch in das Readahead Window, so werden beide Fenster zurückgesetzt. Ein neues Current Window im Umfang des letzten Readahead Windows wird eingelesen (Abb.4.2e).

Wird durch eine Reihe asequentieller Requests die Grösse des Readahead Windows null (Abb.4.2g), so wird Read Ahead deaktiviert, womit jeder Page Frame mit einen eigenen Request eingelesen wird (Abb.4.2h). Read Ahead wird erst wieder aktiviert, wenn die Zugriffsfolge wieder ausreichend sequentiellen Charakter aufweist. Im ext2 und ext3 Dateisystem ist für die Wiederaufnahme des Read Ahead eine Grösse des Current Windows von 32 Page Frames gefordert.

### 4.5.3 Vergleich mit dem PLS

Im Gegensatz zum PLS verfolgt Read Ahead einen allgemeineren Ansatz. Read Ahead versucht die Effizienz aller Blockgerätezugriffe zu steigern und ist nicht nur auf den Anwendungsstart fixiert.

Das PLS versucht einen Zugriffsfolge möglichst genau und vollständig zu erfassen und einen Anwendungsstart zuzuordnen. Dazu ist ein Recording früherer Anwendungsstarts und eine umfangreiche Datenhaltung notwendig.

Read Ahead basiert hingegen auf einen wesentlich spekulativeren Ansatz. Es verwendet zur Vorhersage nur wenige, bereits durchgeführte Blockgerätezugriffe und führt keine persistenten Aufzeichnung früherer Zugriffsmuster durch. Naturgemäss irrt Read Ahead dadurch auch öfter, was zu einer erhöhten Belastung des Page Caches mit nicht benötigten Page Frames führt.

# Kapitel 5

# Zusammenfassung und Ausblick

## 5.1 Zusammenfassung

Das Ziel dieser Arbeit ist die Verbesserung der Performance des Anwendungsstarts. Die Zugriffsfolge eines Anwendungsstarts soll dazu so umgeordnet werden, dass der mechanische Overhead der Blockgeräte minimiert und so die I/O Performance maximiert wird. Diese optimierte Zugriffsfolge soll dann konzentriert zu Beginn des nächsten Kaltstart der Anwendung durchgeführt werden.

Zur Prüfung dieses Konzepts wurde das Preload Service (PLS) als Kernelmodul für Linux 2.6.0 implementiert. Es besteht aus einer Recording Einheit, einer Datenbank und einer Preloading Einheit.

Um eine Zugriffsfolge optimiert durchführen zu können, muss diese Zugriffsfolge vor Beginn des Anwendungsstarts bekannt sein. Das Recording zeichnet dazu die Zugriffsfolgen startender Anwendungen auf, indem es die bio Requests in die einzelnen Elemente des bio Vektors zerlegt.

Die Parameter eines Vektorelements werden in einen Page Objekt in der Datenbank gespeichert. Die Page Objekte eines Anwendungsstarts werden einem Zugriffsfolgenobjekt zugeordnet. Durch Startsektorsortierung entsteht eine neue, maximal sequentielle Zugriffsfolge. Die Zuordnung einer Zugriffsfolge zu einem Anwendungsstart erfolgt mit Hilfe des Pfades der ausführbaren Datei, sowie den Kommandozeilenparametern. Eine Import- und Exportfunktion garantiert die Verfügbarkeit einer aufgezeichneten Zugriffsfolge beim nächsten Kaltstart der entsprechenden Anwendung.

Das Preloading führt eine optimierte Zugriffsfolge zu Beginn eines Anwendungsstarts aus. Dazu konstruiert das Preloading aus den Daten der Zugriffsfolge bio-Requests. Durch die Sektorsortierung und die konzentrierte Durchführung der Zugriffsfolge wird die Requestauslastung optimiert und der mechanische Overhead der Blockgeräte minimiert.

Wie die Messungen gezeigt haben, hat das im PLS implementierte Konzept Potential. So konnte die Ausführungszeit der drei Testszenarien um den Faktor 1.5 bis 1.8 beschleunigt werden. Die Startzeit von OpenOffice am Laptop wurde durch den Einsatz des PLS von 14.2 auf 7.9 Sekunden reduziert.

Das Optimierungspotential durch Verwendung des PLS hängt massgeblich von der physikalischen Anordnung der für einen Anwendungsstart benötigten Daten am Blockgerät ab. Je sequentieller der Charakter einer optimierten Zugriffsfolge ist, desto besser ist die Blockgeräte I/O Performance beim Preloading. Eine optimal sequentielle Zugriffsfolge kann aber selbst durch Sektorsortierung nur zufällig erreicht werden.

In letzter Zeit sind interessante Konzepte und Anwendungen zur Beschleunigung des System- und Anwendungsstarts vorgestellt worden. So wird derzeit Prelink von allen wichtigen Linux Distributoren integriert und das Betriebssystem Windows XP verwendet ein dem PLS ähnliches System zur Optimierung von Anwendungs- und Systemstarts.

Abschliessend bleibt zu hoffen, dass die Linux-Community das in dieser Arbeit vorgestellte Konzept aufgreift und weiterentwickelt.

## 5.2 Ausblick

Soll die Performance des Anwendungsstarts weiter gesteigert werden, so ist das mit dem in dieser Arbeit gewählten Ansatz kaum mehr möglich. Der nächste Schritt wäre ein Reordering der Daten am Blockgerät um eine optimal sequentielle Zugriffsfolge beim Preloading zu erreichen. Vergleicht man die errechnete maximal mögliche Performancesteigerung mit den gemessenen Ausführungszeiten der Testszenarien, so erkennt man das Potential von Blockreordering zur weiteren Beschleunigung von Anwendungsstarts.

Wie eingangs erwähnt, ist das in dieser Arbeit vorgestellte PLS eine Konzeptstudie. Um das gezeigte Potential dieses Konzeptes in einer einsatzfähigen Version nutzen zu können, müssen noch einige Features implementiert werden. In den folgenden Abschnitten werden die wichtigsten beschrieben.

#### Validierung aufgezeichneter Zugriffsfolgen

Derzeit erfolgt keine Überprüfung auf Vollständigkeit und Qualität einer aufgezeichneten Zugriffsfolge.

Erfolgt ein Anwendungsstart und ist eine Zugriffsfolge für diese Anwendung in der Datenbank gespeichert, so wird derzeit diese vorhandene Zugriffsfolge durch weitere Recordings ergänzt. Um die Anzahl der Zugriffsfolgen in der Datenbank gering zu halten, erfolgt die Zuordnung einer Zugriffsfolge zu einer Anwendung mit einer gewissen Unschärfe. Diese Unschärfe der Zuordnung macht eine Qualitätskontrolle der aufgezeichneten Zugriffsfolgen bei zukünftigen Anwendungsstarts notwendig.

Als Beispiel kann das Kommando `tree` betrachtet werden. Das Kommando ergibt abhängig vom aktuellen Arbeitsverzeichnis unterschiedliche Zugriffsfolgen. Bei der Zuordnung einer Zugriffsfolge zu einer Anwendung fliesst aber das aktuelle Arbeitsverzeichnis nicht in den Schlüssel ein. Bei der derzeitigen Implementierung wächst daher die Zugriffsfolge von `tree` mit jeden Aufruf des Kommandos in einen neuen Verzeichnis.

Da nur Requests nach nicht im Speicher vorhandene Daten den bio Layer erreichen, müsste eine Instanz zur Validierung vorhandener Zugriffsfolgen z.B. im Page Cache implementiert werden. Man könnte für jede Page einer Zugriffsfolge einen Zähler verwalten, welcher von der Validierungsinstanz bei Zugriff inkrementiert wird. Weiters erfolgt nur dann ein Preloading für eine Page einer Zugriffsfolge, wenn bei 80% aller Starts der entsprechenden Anwendung nach dem Preloading auf diese Page zugegriffen wird. Alle anderen Pages werden aus der Zugriffsfolge entfernt.

### Automatisches Zusammenfassen von Zugriffsfolgen

Das Zusammenfassen von Zugriffsfolgen ist derzeit nur manuell über das `ioctl` Interface des virtuellen Gerätetreibers möglich.

Bewirkt der Start einer Anwendung den Start weiterer Anwendungen, so können die Zugriffsfolgen dieser Anwendungen zu einer Zugriffsfolge zusammengefasst werden. Die resultierende Zugriffsfolge wird der ersten Anwendung der Gruppe zugeordnet. Durch den grösseren Umfang der zusammengefassten Zugriffsfolge steigt das Potential zur Optimierung der Blockgeräte I/O Performance beim Preloading. Ein Anwendungsbeispiel wäre der Start des X-Servers mit anschliessenden Start von z.B. Gnome, wiederum bestehend aus vielen Einzelanwendungen, wie z.B. das Gnomepanel. Denkbar ist auch die Anwendung beim Systemstart.

Zur Realisierung könnte man die Folge der Anwendungsstarts aufzeichnen und den Zeitabstand der Starts analysieren. Erfolgt ein weiterer Anwendungsstart innerhalb eines definierten Zeitraums nach Ende des Preloadings, so werden zwei Zugriffsfolgen zusammengelegt. Die Zusammenlegung erfolgt erst dann, wenn die Hintereinanderausführung eine definierte Anzahl mal erkannt wurde.

### Überwachung von Systemparametern beim Import einer Datenbank

Systemparameter, welche die Korrektheit der aufgezeichneten Zugriffsfolgen beeinflussen, sollten vor dem Import einer Datenbank kontrolliert werden.

Ein Beispiel hierfür ist die Blockgrösse eines Dateisystems. Da das Recording unterhalb des Blockmappings des Dateisystems arbeitet, würden durch eine Änderung der Blockgrösse die aufgezeichneten Zugriffsfolgen wertlos werden. Ähnlich verhält es sich mit Defragmentierung des Dateisystems. Wird die Blockanordnung auf der Festplatte verändert, so würde die in der Datenbank gespeicherte Abbildung einer Page Frame auf eine Sektornummer des Blockgeräts nicht mehr stimmen. Schlimmstenfalls könnten so Daten des Preloadings beim Zurückschreiben korrumpiert werden.

**Verlagern des Recordings**

Derzeit wird das Recording eines bio-Requests in der Funktion `submit_bio` durchgeführt. Die Funktion ruft `generic_make_request` auf, welche wiederum `blk_chapterition_remap` aufruft. In dieser Funktion wird ein Request an die Partitionierungstabelle des Blockgeräts angepasst: erfolgt ein Zugriff auf den Sektor $n$ einer bei Sektor $m$ beginnenden Partition, so ändert die Funktion den Startsektor des Requests auf $n + m$.

In der aktuellen Version des PLS wird also der Sektor $n$ zur Sektorsortierung der Zugriffsfolge verwendet und nicht der physikalisch richtige Sektor $n + m$. Sind die Pages einer Zugriffsfolge über mehrere Partitionen verteilt, so kann diese Sortierung zu einer nicht optimal sequentiellen Zugriffsfolge und somit zu einer suboptimalen I/O Performance beim Preloading führen.

**Anpassen des Preloadings an die verfügbare Speicherausstattung**

Derzeit nimmt das Preloading keine Rücksicht auf die Speicherausstattung eines Systems. Beim Preloading sehr grosser Zugriffsfolgen kann der Umfang der einzulesenden Datenmenge den verfügbaren Speicher überschreiten. Das Preloading sollte dann abgebrochen werden um Swapping zu verhindern.

## 5.3 Danksagung

In erster Linie möchte ich Professor Anton Ertl für die gute Zusammenarbeit und Betreuung danken. Mein Dank gilt weiters meinen Angehörigen für die Unterstützung, allen voran meiner Frau Petra, meinen Söhnen Felix und Lorenz Johann, sowie meinen Eltern.

# Literaturverzeichnis

[AH04] Amber Huffman, uva. *Serial ATA II: Extensions to Serial ATA 1.0a*. Dell Computer Corporation, Intel Corporation, Maxtor Corporation Seagate Technolog, Vitesse Semiconductor Corporation, August 2004. http://www.sata-io.org/docs/S2Ext_1_2_Gold.pdf

[AHJC03] Amber Huffman, Joni Clark. *Serial ATA Native Command Queuing*. Intel Corporation Colorado Springs Colorado, Seagate Technology Scotts Valley California, July 2003. http://www.seagate.com/content/docs/pdf/whitepaper/D2c_tech_paper_intc-stx_sata_ncq.pdf

[BBDKMSV99] Michael Beck, Harald Böhme, Mirko Dziadzka, Ulrich Kunitz, Robert Magnus, Claus Schräter, Dirk Verworrner. *Linux Kernelprogrammierung: Algorithmen und Strukturen der Version 2.2*, 5. Auflage. Addison-Wesley-Longman Verlag GmbH, Bonn, 1999. ISBN.: 3-8273-1476-3

[BC03] Daniel P. Bovet, Marco Cesati. *Understanding the Linux Kernel*, second Edition. O'Reilly, USA, 2003. ISBN.: 0-596-00213-0

[JN93] Jakob Nielsen. *Usability Engineering*. Morgan Kaufmann Publishers Inc., San Francisco California, 1993. ISBN.: 0-12-518406-9

[MCKU84] Marshall Kirk McKusick, William Joy, Leffler Sam, R.S. Fabry. *A Fast File System for UNIX* August 1984. ACM Transactions on Computer Systems, 2(3).

[MG04] Eduard Stupening. *Studie zum Einsatz von Open Source Software in deutschen Unternehmen 2004*. META Group Deutschland GmbH, Ismaning, 2004. http://www.metagroup.de/presse/2004/pm34_21-10-2004.htm

[ML01] Matjaž Ladava. *Fast System Startup for PCs Running Windows XP*. Microsoft Corporation, Washington Redmont, 2001. http://www.microsoft.com/whdc/system/sysperf/fastboot/fastboot-winxp.mspx

[NC04] Netcraft. *November 2004 Web Server Survey*. Netcraft LDT, Wiltshire, 2004. http://news.netcraft.com/archives/2004/11/01/november_2004_web_server_survey.html

[PH98] David A. Patterson, John L. Hennessy. *Computer Organization and Design: the Hardware/Software Interface*, second Edition. Morgan Kaufmann Publishers Inc., San Francisco California, 1998. ISBN.: 1-55860-428

[PPC04] Ram Pai, Badari Pulavarty, Mingming Cao. *Linux 2.6 performance improvement through readahead optimization*. Linux Symposium, Ottawa Ontario Canada, July 2004. http://oss.software.ibm.com/linux/papers/ols2004/Pai-OLS2004.pdf

[RC01] Alessandro Rubini, Jonathan Corbet. *Linux Device Drivers*, second Edition. O'Reilly, USA, 2001. ISBN.: 0-596-00008-1

[RL04] Robert Love. *Linux Kernel Development*. Sams Publishing, Indiana Indianapolis, 2004. ISBN.: 0-672-32512-8

[SL93] Margo Seltzer, Keith Bostic, Marshall Kirk McKusick, Carl Staelin. *An Implementation of a Log-Structured File System for UNIX*. Havard University, 1993. http://www.scriptics.com/people/john.ousterhout/seltzer.html

[SL95] Margo Seltzer, Keith A. Smith, Hari Balakrishnan, Jacqueline Chang, Sara McMains, Venkata Padmanabhan. *File System Logging Versus Clustering: A Performance Comparison*. Havard University, 1995. http://www.eecs.harvard.edu/ margo/usenix.195/usenix.195.ps.gz

[STA98] William Stallings. *Operating Systems: Internals and Design Principles*, third Edition. Prentice Hall International, New Jersey USA, 1998. ISBN.: 0-13-917998-4

[TH78] K. Thompson. *Unix Implementation*. August 1978. Bell Systems Technical Journal, 57(6), part 2.

[TIS95] Tool Interface Standard Group. *Executable and Linking Format (ELF) Specification*, Version 1.2. Free Standards Group (FSG), San Francisco California, 1995. http://www.linuxbase.org/spec/refspecs/elf/elf.pdf

[WM04] Wolfgang Mauerer. *Linux Kernelarchitektur - Konzepte, Strukturen und Algorithmen von Kernel 2.6*. Carl Hanser Verlag München Wien (www.hanser.de) ISBN.: 3-446-22566-8