technische universität
dortmund

Bachelorarbeit

Simulationsbasierte Bewertung von maschinellen Lernverfahren für Vorablade-Strategien in Linux

Alexander Lochmann
23. Februar 2011

Betreuer:
Prof. Dr.-Ing. Olaf Spinczyk
Dr. Michael Engel

Technische Universität Dortmund
Fakultät für Informatik
Lehrstuhl Informatik 12
Arbeitsgruppe Eingebettete Systemsoftware
http://ess.cs.tu-dortmund.de

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst, keine anderen als die angegebenen Quellen und Hilfsmittel verwendet sowie Zitate kenntlich gemacht habe.

Dortmund, den 23. Februar 2011

Alexander Lochmann

Zusammenfassung

Eingebettete Systeme unterliegen trotz des technologischen Fortschritts noch immer Ressourcenbeschränkungen, z.B. hinsichtlich des Arbeitsspeichers oder der Rechenleistung. Trotzdem möchte man bei solchen Systemen keine größeren Wartezeiten beim Zugriff auf Informationen, die auf lokalen Speichermedien abgelegt sind, inkaufnehmen. Daher muss anhand neuer Methoden versucht werden die gewünschten Informationen schnell verfügbar zu machen. Hier setzt der Sonderforschungsbereich 876 „Verfügbarkeit von Informationen durch Analyse unter Ressourcenbeschränkung" an. Es wird versucht mit Hilfe maschineller Lernverfahren ein gegebenes System zu analysieren, um rechtzeitig die nötigen Informationen zu laden, bevor sie angefordert werden. Dabei steht jedoch immer die Ressourcenbeschränktheit im Vordergrund.
Im Rahmen dieser Arbeit soll zunächst eine Evaluationsplattform entwickelt werden, mit der diese neuen Ansätze bewertet werden können. Hierbei sollen Teile des Betriebssystems Linux nachempfunden werden, da es sich bereits bei Smartphones in Form von Android [1] etabliert hat. Zusätzlich müssen Vergleichskriterien ermittelt werden, um eine adäquate Bewertung vornehmen zu können. Anschließend soll ein erster Ansatz mit Hilfe der Plattform evaluiert werden. Dabei dient die Linux-Vorabladestrategie als Referenzmaß.

Inhaltsverzeichnis

1 Einleitung

Die vorliegende Arbeit befasst sich mit dem Thema „Simulationsbasierte Bewertung von maschinellen Lernverfahren für Vorabladestrategien unter Linux". Ziel ist es herauszufinden, ob mittels maschineller Lernverfahren [2] die Performanz des Linuxkerns verbessert werden kann. Zunächst muss einmal klar definiert werden, was es zu optimieren gilt. Von den vielen Facetten eines Betriebssystems sollen hier lediglich der Ein- und Ausgabe-Stapel betrachtet werden. Jedoch sind auch hier einige Ansätze möglich.
Betrachten wir daher erst mal den Ablauf einer Ein- und Ausgabeoperation. Sie wird von einem Programm im Benutzerbereich initiiert - dies könnte z.B. der Systemaufruf *read*()(vgl. „man 2 read") sein. Zu aller erst wird überprüft, ob die Daten bereits im Zwischenspeicher (siehe Abbildung 1.1) vorhanden sind. Im Zuge dessen wird die Linux-Vorablade-Strategie [3.3.1] aufgerufen, um ggf. weitere Daten vorauszuladen. Liegen die angeforderten Informationen nicht vor, wird die Anfrage an das Dateisystem weitergegeben. Dieses bestimmt alle zu lesenden Sektoren und fertigt daraus Blockanfragen, die an den Treiber übergeben werden. Hier werden die einzelnen Anfragen an das jeweilige Speichermedium übermittelt. Die Abbildung 1.1 verdeutlicht diesen Ablauf grafisch.

Wie bereits erwähnt, bieten sich einige Ansatzpunkte für Optimierungen. Eine sehr einfache Möglichkeit wäre, das Speichermedium zu verbessern. Dies lässt sich z.B. durch den Einsatz von Solid-State-Disks [3],[4] erreichen. Hierdurch würden offensichtlich die Antwortzeiten der Speichermedien gesenkt, wo durch die Reaktionszeit des Prozesses verkürzt würde. Jedoch ist es nicht möglich diese auf annähernd Null zu drücken. Liegt ein Datum nicht im Zwischenspeicher, muss die Anwendung immer auf die darunterliegenden Schichten warten.
Daher setzt diese Arbeit einige Ebenen höher im Zwischenspeicher an. Genauer gesagt bei der Vorabladestrategie. Das Vorgehen einer Vorabladestrategie sei an dieser Stelle nur kurz erläutert - Näheres findet sich dazu in [3.3]. Sie trifft anhand bestimmter Merkmale eine Vorhersage, auf welche Daten ein Prozess im Folgenden zugreifen wird. Daraus resultieren asynchrone Lesevorgänge, die eine Anwendung nicht blockieren. Es ist offensichtlich, dass je genauer die Vorhersage ausfällt, desto seltener muss ein Prozess suspendiert werden. Somit sinkt stark die Reaktionszeit.
Betrachtet man das Ganze nun im Kontext der Eingebetteten Systeme, lassen sich noch weitere Gründe für die Optimierung von Vorablade-Strategien ableiten. Ein Eingebettetes System sei kurz definiert als: „Ein infortmationsverabeitendes System, das in ein größeres Produkt integriert ist."[5]. In der letzten Zeit ist ein starker Trend hin zu sehr leistungsstarken Prozessoren zu beobachten(siehe jüngste Erscheinungen von nVidia oder Freescale). Dies ermöglicht viele neue Funktionen für z.B. Mobiltelefone. Mit großer Wahrscheinlichkeit benötigen neue Funktionen aber auch einiges mehr an Daten, was in

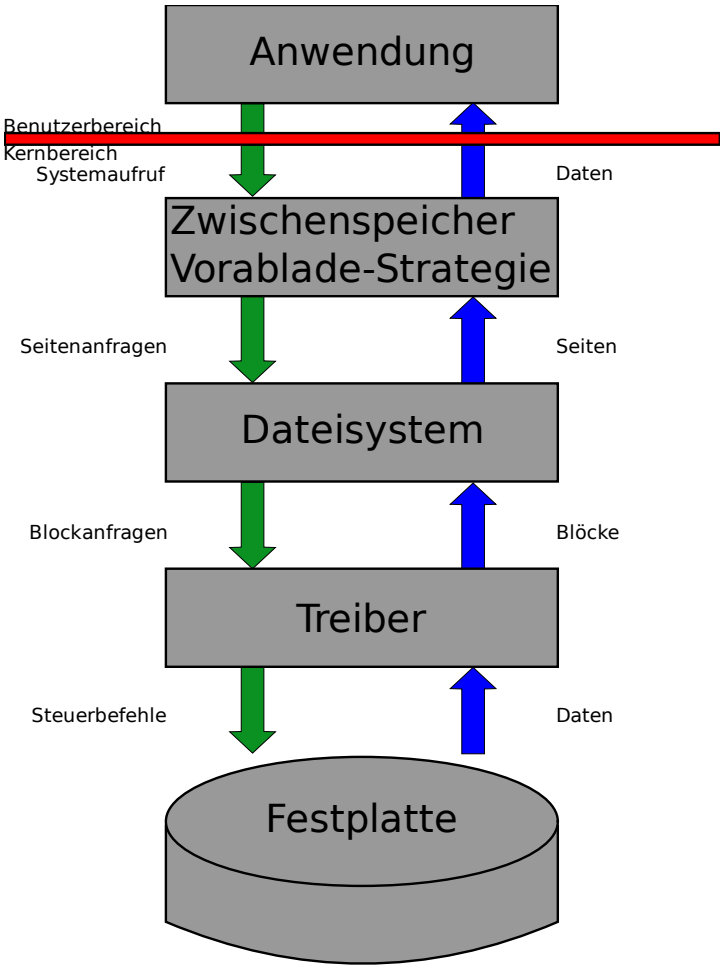


Abbildung 1.1: Linux E/A-Stapel

einer größeren Anzahl von Ein- und Ausgabeoperationen resultiert. Daraus ergibt sich wiederum ein erhöhter Energieverbrauch. Da jedoch die Leistungsfähigkeit von Akkus nicht im gleichen Maße [5] steigt wie die Rechenkapazität von Prozessoren, ist es immens wichtig, überflüssige Ein- und Ausgabevorgänge zu vermeiden. Hieraus leitet sich offensichtlich der Anspruch einer möglichst präzisen Vorablade-Strategie ab. Denn ungenaue Vorhersagen führen zu unnötigen Zugriffen auf das Speichermedium, was wiederum den Energieverbauch steigen lässt. Darüber hinaus belegen überflüssige Daten unnötig viel Arbeitsspeicher, der bei Eingebetteten Systemen ohnehin schon nicht sehr groß ausfällt. Diese Arbeit versucht daher zu bewerten, wie sich die Effizienz von Vorablade-Strategien mit Hilfe von maschinellen Lernverfahren steigern lässt. Diese wurden gewählt, da sie in der Lage sind ihr Verhalten im Betrieb anzupassen - sie lernen. Die Lernverfahren gibt es in verschiedenen Varianten, die unterschiedliche Vorteile mit sich bringen. Es seien hier nur zwei Vertreter genannt: das Naive Bayes Verfahren und die Conditional-Random-Fields - kurz CRFs ([2] und [6]).

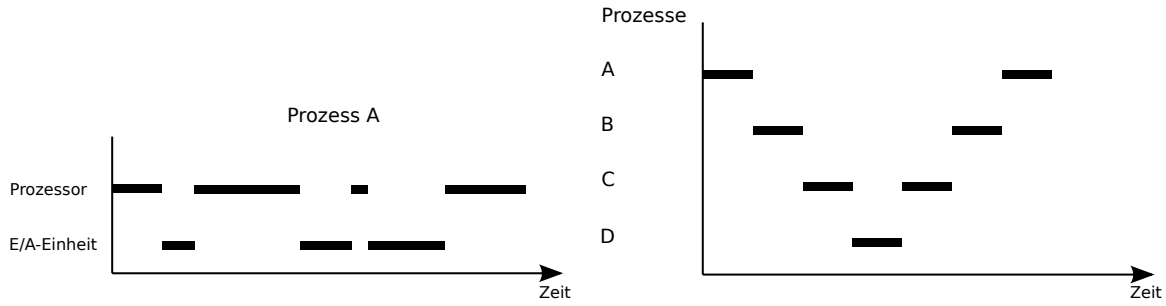
Da absolute Messwerte nur eine geringe Aussagekraft haben, ist es nötig, diese mit einer Referenz zu vergleichen. Hier bietet sich die Linux-Vorablade-Strategie an. Sie ist bereits seit längerem Bestandteil des Linuxkerns und über die Jahre immer weiter entwickelt worden. Ein weiterer Bestandteil der vorliegenden Arbeit ist die Ausarbeitung geeigneter Bewertungskriterien. Ein wichtiges Merkmal ist mit Sicherheit die Trefferquote im Zwischenspeicher. Die Quote allein genügt jedoch nicht, um die Verfahren adäquat zu bewerten. Daher ist es unabdingbar, weitere, sekundäre Merkmale zu bestimmen, um eine möglichst präzise Bewertung vorzunehmen. Darüber hinaus soll ein passender Simulator entworfen und implementiert werden. Hierbei werden sich noch einige Schwierigkeiten ergeben, die es zu bewältigen gilt. Mit Sicherheit lässt sich hinterfragen, warum unbedingt ein Simulator nötig ist. Man kann die Lernverfahren doch direkt in einen Betriebssystemkern integrieren. Zunächst einmal ist die Integration nicht trivial. Da die Strukturen im Linuxkern sehr auf Performanz optimiert sind, fällt es nicht leicht dort Änderungen einzubringen. Um die relevanten Messgrößen im Betrieb zu bestimmen, sind weitere Veränderungen erforderlich. Z.B. muss der Zwischenspeicher jeden Treffer bzw. Zugriff protokollieren. Außerdem müssen diese Werte in regelmäßigen Abständen ermittelt werden. Hierfür geht im schlimmsten Fall entsprechend viel CPU-Zeit verloren, so dass das System komplett unbrauchbar wird. Abgesehen davon hat das Messen in einem realen System Einfluss auf die Resultate - vgl. Heisenbergsche Unschärferelation. Hinzu kommt, dass die Testbedingungen zwischen zwei Simulationsläufen derart variieren, dass die Ergebnisse nicht mehr vergleichbar sind. Diese Probleme entfallen bei einem Simulator fast vollständig. Einzig das Hindernis der divergierenden Bedingungen muss gelöst werden. Aber diese Arbeit hat auch hierfür eine adäquate Lösung erarbeitet. Im Kapitel Evaluation wird beschrieben, wie diese aussieht. Eine Simulationsumgebung bietet außerdem den Vorteil, marginale Änderungen - z.B. an den Lernverfahren - sofort testen zu können. Ein vollständiges Neu-Übersetzen des Linuxkerns entfällt.

2 Anforderungen

Bei einem klassischen Ansatz wird nur der Zwischenspeicher und die Zugriffe darauf betrachtet. Als die Messgröße - leider auch die einzige - wird die Trefferquote betrachtet. Vorabladestrategien fügen hierbei weitere Seiten hinzu, die sofort vollständig verfügbar sind.
Da das Vorausladen jedoch nicht in $O(1)$ geschieht und hierfür möglicherweise mehrere E/A-Anfragen nötig sind, vernachlässigt der klassische Ansatz den Einfluss auf die Performanz des gesamten Systems. Beispielsweise können weitere E/A-Anfragen durch eine Vorabladestrategie initiiert die Anfragen eines anderen Prozesses verdrängen, so dass dieser länger als nötig blockiert ist. Daher ist es sinnvoll den kompletten Stapel - von der Anwendung hinunter bis zur Festplatte - zu betrachten. Allerdings sollen nur Teile des Betriebssystemkerns simuliert werden. Ein Programm dient hier nur als Lieferant von Systemaufrufen. Der Horizont des Simulators endet somit an der Grenze zum Benutzerbereich. Was jenseits dieser Grenze abläuft interessiert hier nicht.
Daraus leiten sich weitere Erfordernisse ab: Um den Dateioperationen Dateien zuordnen zu können, ist eine Dateideskriptortabelle vonnöten. Diese wiederum ist nur prozessweit eindeutig, daher müssen auch die Prozesse abgebildet werden. Aus dieser Schlussfolgerung ergibt sich die Notwendigkeit eine eigene Prozessverwaltung sowie ein eigenes Prozesszustandsmodell zu entwickeln. Beide Punkte werden in diesem Kapitel abgehandelt. Des Weiteren müssen für eine adäquate Simulation die einzelnen Schichten geeignet nachgebildet werden. Der letzte Abschnitt befasst sich mit der Spezifikation der Anforderungen an den E/A-Stapel.

2.1 Simulations- und Zeitmodell

Elementarer Bestandteil eines Simulators ist die Definition eines Simulationsmodells. Hierbei gilt es herauszustellen, welche Teile eines betrachteten Systems nachzubilden sind und welche vernachlässigt werden können.
Im vorliegenden Fall gibt es zwei Ressourcen, die abgebildet werden müssen. Zum einen die CPU und zum anderen die E/A-Einheit, z.B. eine Festplatte. Beide Betriebsmittel können zu einem Zeitpunkt nur von einem Faden bzw. einer E/A-Anfrage belegt werden - sie sind unteilbar. Zudem arbeiten beide unabhängig voneinander: auf einem ein-Prozess-System wechseln sich daher Rechenzeiten und E/A-Zeiten ab. Eine Ressource muss immer auf Beendigung der Aktivität der anderen warten. Abbildung 2.1(a) stellt dies graphisch dar. Richtet man den Blick nun auf ein aktuelles Betriebssystem, so gibt es mehrere Prozesse, die um das Betriebsmittel CPU bzw. Festplatte kämpfen. Während vorher entweder die CPU oder die E/A-Einheit blockiert war, kann nun die betreffende



(a) Uni-Prozess-System (b) Multi-Prozess-System

Abbildung 2.1: Auslastung der Betriebsmittel CPU und E/A-Einheit

Ressource abgegeben werden, so dass andere diese belegen können. Dies führt zu echter Parallelverarbeitung.

In Abbildung 2.1(b) sind vier Prozesse dargestellt, die abwechselnd die CPU belegen und diese wieder abgeben bis ihre E/A-Aktivitäten beendet sind. Hieran wird deutlich, dass beispielsweise, während Prozess A rechnet, die E/A-Operationen der anderen drei Prozesse durchgeführt werden können.

Zu einer vollständigen Beschreibung des Simulationsmodells fehlt noch die Festlegung eines Zeitmodells. Bei den oben genannten Ressourcen kommt es insbesondere auf die Parallelität an. Zudem soll die Zeit unaufhörlich voranschreiten, ohne dass ein Ereignis die Simulation beendet. Daher bietet sich ein prozessorientiertes Modell an. Allerdings lassen sich die Aktivitäten von Prozessen nicht kontinuierlich erfassen. Deshalb muss hier eine Quantisierung stattfinden, deren Auflösung noch näher bestimmt zu ist. Solch ein Sample[1] fasst die Aktivität zu einem bestimmten Zeitpunkt zusammen; alternativ kann man auch den Begriff Ereignis verwenden.

Offenbar führt dieses Hindernis zu einem ereignisorientiertes Modell, denn ein Prozess kann nur dann die CPU bekommen, wenn es auch Ereignisse zu bearbeiten gibt. Festzuhalten ist also, dass ein Weg zu finden ist, beide Modelle miteinander zu kombinieren.

Nähere Informationen zu den Grundlagen über ereignis- bzw. prozessorientierte Modelle sind in [3.1] zu finden.

2.2 Prozesszustandsmodell und Prozessverwaltung

Zunächst sei ein Prozess definiert als ein Programm in Ausführung [7]. Wobei jeder Prozess sein eigenes Text- und Datensegment sowie einen eigenen Halde und einen eigenen Stapel besitzt. Außerdem verfügt ein Prozess über einen Zustand. Dieser beschreibt eindeutig die aktuelle Aktivität. Alle Zustände sind exklusiv; zu einem Zeitpunkt kann ein Prozess nur einen Zustand annehmen. Ein simples Zustandsmodell ist in Abbildung 2.2 dargestellt. Es weist folgende Zustände auf:

[1] engl. für Abtastwert, Begriff aus der Signalverarbeitung

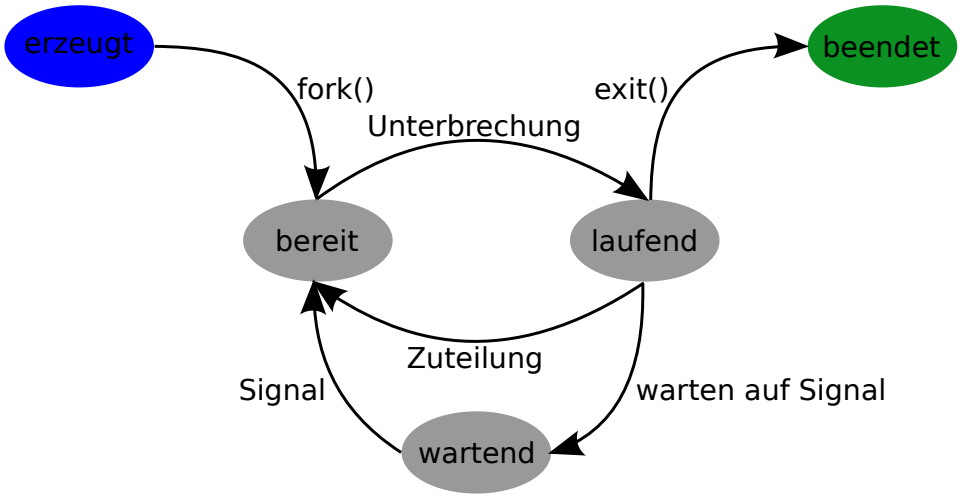


Abbildung 2.2: Prozesszustandsmodell aus [8]

- **neu:** Der Prozess wurde mittels $fork()$[2] erzeugt.
- **beendet:** Der Prozess hat terminiert, z.B. durch den Aufruf von $exit()$[3]
- **laufend:** Der Prozess hat den Prozessor bekommen und rechnet gerade.
- **bereit:** Der Prozess wartet auf die CPU.
- **wartend:** Der Prozess wartet auf ein Signal, z.B. auf die Beendigung einer E/A-Anfrage oder auf die Terminierung eines Kindprozesses.

Zwischen „bereit" und „wartend" kann ein Prozess nur dann wechseln, wenn er von der Prozessablaufsteuerung die CPU erhält bzw. entzogen wird. Möchte ein Prozess Daten von einem Speichermedium, wie einer Festplatte, lesen, wechselt er in den Zustand „wartend". Sind die Daten vollständig gelesen worden, wird er wieder auf „bereit" gesetzt.
Ein Prozess kann Aufgaben nur sequentiell erledigen. Es ist z.B. nicht möglich bei einer Anwendung - wie einer Textverarbeitung -, die nur aus einem Prozess besteht, gleichzeitig Text einzugeben und die Rechtschreibprüfung zu starten (angelehnt an [8]). Er besitzt nur einen Ausführungsfaden. Dieser hat exklusiv Zugriff auf alle Daten und Variablen. Bei Systeme, die multithreadfähig sind, wird diese 1:1 Zuordnung aufgebrochen, so dass ein Prozess mehrere Ausführungsfäden haben kann. Diese heißen dann Threads[4]. Jeder Faden besitzt einen separaten Programmzähler und Stapel. Die übrigen Bestandteile eines Prozesses teilt er sich mit den anderen Fäden. Daher werden diese Fäden auch als leichtgewichtige Prozesse bezeichnet.
Um zu entscheiden, welcher Prozess als nächstes die CPU bekommt, verwendet man

[2] siehe „man 2 fork"
[3] siehe „man 2 exit"
[4] engl. für Fäden

eine Prozessablaufsteuerung. Dieser entscheidet nach verschiedensten Kriterien, wer als nächster an der Reihe ist. Ein prominentes und einfaches Beispiel ist die RoundRobin-Strategie[7]. Hierbei bekommt jeder Prozess für eine bestimmte Zeit die CPU. Diese Dauer ist im System festgelegt und nennt sich Zeitscheibe. Wird eine Zeitscheibe vollständig ausgenutzt, so wird dem Prozess die CPU entzogen. Blockiert der Prozess vor Ablauf seiner Zeitscheibe, so gibt er die CPU ab und darf das nächste Mal die verbleibende Zeit abarbeiten.
Da ein Prozess potentiell mehrere Ausführungsfäden besitzt, unterscheidet die Prozessablaufsteuerung unter Linux nicht zwischen einzelnen Prozessen, sondern zwischen Fäden [9]. Ist ein Programm nicht multithreaded, so hat dennoch genau einen Faden nämlich den Prozess selber.
Aufgabe dieser Arbeit ist es nun solch ein Prozesszustandsmodell sowie eine Prozessverwaltung zu entwerfen.

2.3 E/A-Stapel

Betrachten wir nun die Anforderungen an die einzelnen Schichten, warum sie wichtig sind und was sie leisten sollen.

- **Anwendung**
  Da es für diesen Anwendungsfall nicht von Interesse ist, was im Benutzerbereich geschieht, muss man sich auf die Interaktion der Anwendung mit dem Betriebssystemkern beschränken. An der Schnittstelle muss eine noch zu bestimmende Menge von Systemaufrufen mitgeschnitten werden. Darüber hinaus sollte der Mitschnitt abgespeichert werden, damit man ihn zu einem späteren Zeitpunkt als Eingabe für den Simulator verwenden kann. Folglich gilt es ein geeignetes Format zu definieren, in dem die Informationen abgelegt werden.

- **Zwischenspeicher / Vorabladestrategie**
  Im Zwischenspeicher werden die einzelnen Seiten, aus denen eine Datei besteht, abgelegt. Hier sollen vorwiegend Messwerte wie die Trefferquote oder die Anzahl vorausgeladener Seiten erhoben werden - Näheres zu den einzelnen Größen findet sich in [7.1].
  Ebenfalls auf dieser Ebene ist die Vorabladestrategie zu finden. Ihre Aufgabe besteht darin, je nach konkreter Strategie über die nächsten, vorauszuladenen Seiten eine Entscheidung zu treffen.

- **Dateisystem**
  Ein Dateisystem ordnet einer Datei physikalische Sektoren auf dem jeweiligen Speichermedium zu. Eine triviale Lösung wäre bei jedem Zugriff auf die Festplatte eine randomisierte Zuordnung der Datei zu Sektoren zu erzeugen. Jedoch entspricht dies in keinerlei Hinsicht der Realität. Wird eine Datei mehrfach von dem Speichermedium gelesen, so muss sichergestellt sein, dass - sofern die Datei zwischendurch nicht

gelöscht wurde - jedes Mal die identische Menge Sektoren gelesen wird. Außerdem fragmentieren Dateien je nach gewähltem Dateisystem unterschiedlich stark, so dass beim Lesen der Lesekopf öfters neu positioniert werden muss. Dateisysteme wie Ext2 [10] verwenden außerdem als kleinste Zuordnungseinheit oftmals ein vielfaches der physikalischen Sektorgröße - Blockgröße genannt. Hierdurch entsteht möglicherweise zwischen zwei hintereinanderliegenden Dateien eine Lücke, die ebenfalls eine Neu-Positionierung des Schreib-/Lesekopfes zur Folge hat.
Es ist also offensichtlich, dass ein Dateisystem Einfluss auf die Performanz des gesamten Systems hat und damit nicht zu vernachlässigen ist.

- **E/A-Ablaufsteuerung**
  Eine E/A-Ablaufsteuerung ist auf der Ebene des Treibers anzusiedeln. Sie ist zwar nicht Bestandteil des Treibers, sondern Teil des Betriebssystem, jedoch interagiert sie mit dem Treiber. Da bei einem Simulator kein Treiber notwendig ist, um eine Festplatte korrekt anzusprechen, kann man die E/A-Ablaufsteuerung an die Stelle des Treibers setzen. Ihre Aufgabe ist es, die Sektoranfragen der höher gelegenen Schichten entgegen zu nehmen und an das entsprechende Speichermedium weiterzuleiten. Hierbei darf sie die Reihenfolge der Anfragen verändern, um den Mehraufwand durch Positionierung des Lesekopfes zu verringern - Elevatoralgorithmus [7]. Außerdem dürfen Anfragen auf adjazente Sektoren zusammengeführt werden. Je nach gewählter Strategie sind auch Priorisierungen von Anfragen bestimmter Prozesse möglich, so dass diese für eine möglichst kurze Zeit blockiert sind [11]. Die Notwendigkeit der Nachbildung dieser Schicht leitet sich aus dem Einfluss auf die Gesamtsystemperformanz ab [12],[13].

- **Festplatte**
  Die Festplatte ist nicht außer Acht zu lassen, da je nach Ausführung E/A-Anfragen schnell oder langsam beantwortet werden. Hiermit hat sie den größten Einfluss darauf, wie lange ein Prozess blockiert ist. Wählt man beispielsweise eine moderne Festplatte [14] oder gar eine Solid-State-Disk [15], dann sinken die Antwortzeiten auf einzelne Millisekunden bzw. in den Bereich von Mikrosekunden. In solchen Fällen muss ein Prozess nur kurz auf die Beendigung des E/A-Vorgangs warten.
  Dies macht die Simulation einer Festplatte essentiell, um quantitative Werte, wie die Leerlauf- bzw. Lastzeit, zu erhalten.

Abgesehen von der Anwendungsschicht gilt für alle darunterliegenden Schichten, dass ihre Schnittstellen korrekt definiert sein müssen, damit problemlos Komponenten ausgetauscht werden können. Hinzu kommt, dass die Schichten in ihren jeweiligen Verhalten dem Linuxkern nachempfunden werden sollen.

3 Grundlagen

3.1 Simulation

Die Simulation ist die Analyse eines Systems anhand einer modellhaften Abbildung anstatt des Originalsystem. Die Gründe für eine Simulation sind vielfältig. Sie wird angewandt, wenn ein System z.B. zu komplex ist, um es in der Realität zu untersuchen. Ebenso wird sie benutzt, um Abläufe zu untersuchen, die in der Wirklichkeit zu schnell ablaufen [16]. Bevor eine Simulation möglich ist, muss erst mal ein Modell des zu untersuchenden Systems erstellt werden. Bei der Modellbildung kommt es darauf an, von den unnötigen Aspekten zu abstrahieren und die wesentlichen Eigenschaften abzubilden. Wo die Schwerpunkte liegen, hängt von dem Blickwinkel ab, unter dem man ein reales System betrachtet.
Modelle lassen sich nach unterschiedlichsten Kriterien erstellen. Beispielsweise gibt es zeitkontinuierliche Modelle, die sich in der Physik oder Elektrotechnik über Differentialgleichungen abbilden lassen. Hierbei schreitet die Zeit stetig voran. Im Gegensatz dazu gibt bei zeitdiskreten Modellen eine kleinste, unteilbare Einheit, um die die Zeit voranschreitet.
Zeitdiskrete Modelle können noch weiter unterteilt werden in ereignisorientierte, prozessorientierte, transaktionsorientierte und aktivitätsorientierte Modelle. Die beiden erstgenannten Varianten sollen im Folgenden erläutert werden. Zu den anderen Ansätzen finden sich weiterführende Informationen in [16].
Die Stärken einer Simulation sind offensichtlich die Möglichkeit die Zustandsübergänge in einem System Schritt für Schritt zu untersuchen sowie die Abbildung von komplexen Strukturen durch ein einfaches Modell. Nachteilig ist jedoch der höhere Entwicklungsaufwand für ein Modell im Vergleich zum Aufstellen einer Differentialgleichung.
Anwendung findet die Simulation z.B. in der Flugzeugindustrie, um neueste Entwicklungen zunächst an einem Modell zu testen, bevor Geld aufgewendet wird, um einen Prototypen zu erstellen. Außerdem wird sie beispielsweise in der Informatik eingesetzt, um den Entwurf von Schaltkreisen zu vereinfachen [17].

3.1.1 Ereignisorientierte Simulation

Bei einem ereignisorientieren Modell liegt das Hauptaugenmerk auf den Zustandsübergängen in einem System. Es werden die Übergänge zwischen zwei Zuständen simuliert. Die Zeit, die dazwischen vergeht, wird vernachlässigt. Somit werden nur noch einzelne Punkte auf der Zeitachse betrachtet. Dies sind genau die Zeitpunkte, zu denen die jeweiligen Ereignisse stattfinden. Die simulierte Zeit schreitet immer sprunghaft von Ereig-

nis zu Ereignis voran. Sie wird beim Ausführen eines Ereignisses auf dessen Zeitpunkt vorgestellt.
Alle Ereignisse werden in einer Liste verwaltet, die aufsteigend nach der Zeit sortiert ist. Es wird immer das erste Element aus der Liste entnommen und abgearbeitet. Dabei wird zuerst die simulierte Zeit auf den Zeitpunkt des Ereignisses vorgestellt. Anschließend wird je nach Ereignistyp die passende Behandlungsroutine ausgewählt. Diese führt die nötigen Zustandsänderungen durch und erhebt ggf. noch einige statistische Werte. Während der Behandlung von Ereignissen ist durchaus möglich, dass weitere Ereignisse generiert werden, z.B. erzeugt eine Fertigungsmaschine beim Erhalten eines neuen Auftrages das entsprechende „Auftragsende-Ereignis" [16]. Die Simulation terminiert sobald entweder bei einem Zustandsübergang ein entsprechendes Ereignis generiert wurde oder zu Beginn direkt ein *Ende*-Ereignis mit einem festen Zeitpunkt in die Liste eingefügt wurde.

3.1.2 Prozessorientierte Simulation

Beim prozessorientierten Ansatz werden alle Operationen auf einer Entität zu einem Prozess zusammengefasst. Ein Prozess hat aktive und passive Phasen. Während einer aktiven Phase führt er Zustandsübergänge durch; die simulierte Zeit bleibt währenddessen stehen. Er passiviert sich, indem er die Kontrolle an einen anderen Prozess abgibt. Dabei kann er vor der Abgabe der Kontrolle einen Zeitpunkt vormerken, zu dem er wieder reaktiviert werden soll. Dies ist z.B. der Fall, wenn ein Prozess die Arbeit einer Fertigungsmaschine darstellt. Sobald diese mit der eigentlichen Arbeit beginnt, merkt der Prozess einen Zeitpunkt vor, zu dem die Fertigung abgeschlossen ist, und gibt die Kontrolle ab. Alternativ kann er die Kontrolle abgeben, wenn keine Aufträge mehr zu bearbeiten sind. In diesem Fall legt er sich für eine unbestimmte Zeit schlafen und wartet auf ein Signal eines anderen Prozesses, dass wieder Aufträge verfügbar sind.
Der große Vorteil bei diesem Modell liegt in der simulierten Parallelität. Während ein Prozess, der die Fertigungsmaschine repräsentiert, seine Arbeit verrichtet, kann zur gleichen Zeit ein anderer Prozess sich um die Bearbeitung neuer Aufträge kümmern.

3.2 Maschinelle Lernverfahren

Um maschinelle Lernverfahren genau zu verstehen, muss erst einmal der Begriff „Lernen" genau definiert werden. Hierzu gibt es zwei Zitate von Witten und Frank [2]:

> [...] sich durch Studium, Erfahrung oder Lehre Wissen anzueignen;
>
> Etwas lernt, wenn es sein Verhalten so ändert, dass es in Zukunft eine bessere Leistung aufweist.

Hieraus wird deutlich, dass lernen sich aus zwei Dingen zusammensetzt: Zunächst mal muss jemand das Wissen erwerben; es muss verfügbar sein. Zum anderen beschreibt es die Fähigkeit das erworbene Wissen einzusetzen, um sich selber zu verbessern. Mit den

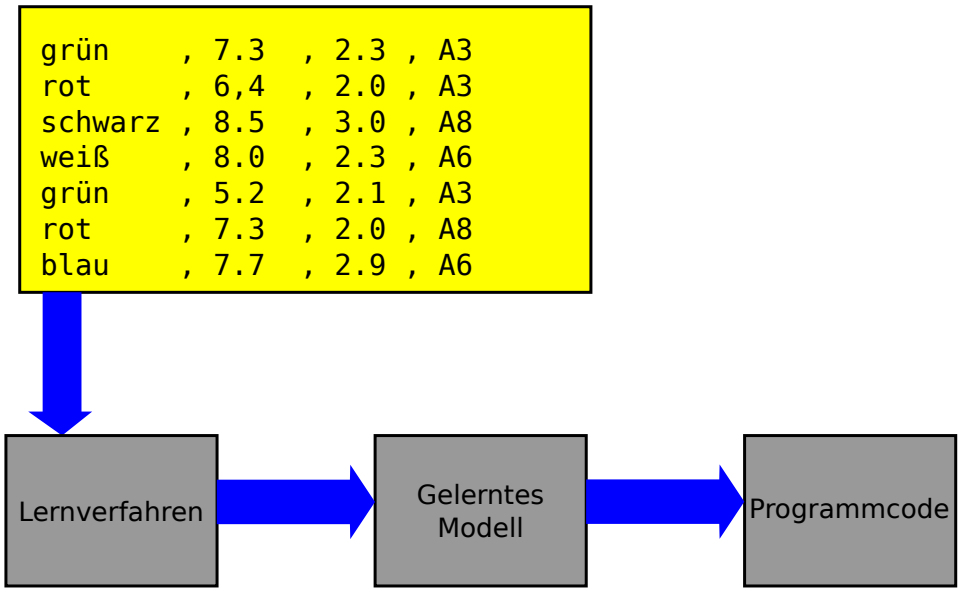


Abbildung 3.1: Trainieren eines Modells

beiden Bestandteilen lässt sich der typische Ablauf beim maschinellen Lernen einfacher verstehen.
Ein typisches Vorgehen sieht wie folgt aus:
Man benötigt einen Trainingsdatensatz - gelber Kasten in Abbildung 3.1. Dieser enthält alle Merkmale sowie die zugehörigen Klassifikatoren. In diesem Beispiel gibt es drei Merkmale: Farbe, Länge und Breite eines Autos. Die zugehörigen Klassifikatoren lauten: A3, A6 und A8. Mit Hilfe des Trainingsdatensatzes und einem Lernverfahren wird ein gelerntes Modell erzeugt. Das Lernen erfolgt durch die Betrachtung der einzelnen Merkmale in Verbindung mit den Klassifikatoren. Hierbei wird zunächst nur das Wissen erworben. Das resultierende Modell kann anschließend mit geeigneter Ansteuerung in ein Programm integriert werden.
Um das erlernte Wissen nun anzuwenden, müssen zunächst Daten zu allen Merkmalen gesammelt werden. Anschließend wird dieser Merkmalsvektor durch ein Programm geschickt, das mit dem erlernten Modell interagiert. Die Ausgabe ist der hoffentlich richtige Klassifikator. Im Folgenden sollen nun zwei Lernverfahren kurz vorgestellt werden, da sie im Rahmen dieser Arbeit evaluiert werden sollen.

3.2.1 Naives Bayes

Das Naive Bayes Verfahren vergibt für jedes Beispiel aus den Trainingsdaten einen Klassifikator. Dabei besteht jedes Beispiel aus n Merkmalen. Aufgrund des Trainingsdatensatzes errechnet sich das Verfahren Wahrscheinlichkeiten für die einzelnen Klassifikatoren. Es gibt an, wie groß die Wahrscheinlichkeit dafür ist, dass ein bestimmtes Label auf die gegebenen Merkmale zutrifft. Ausgewählt wird der Klassifikator mit der größten Wahrscheinlichkeit. Weitere Informationen finden sich in [18].

3.2.2 Conditional Random Fields

Die *Conditional Random Fields* - kurz CRF - können als Abwandlung des Naive Bayes Klassifikators betrachtet werden. Sie betrachten eine bedingte Wahrscheinlichkeit für eine Abfolge von Klassifikatoren und zugehörigen Merkmalen. Zur Bestimmung eines Klassifikators verwenden die *CRFs* nicht nur den aktuellen Merkmalsvektor, sondern auch die vorangegangenen n Vektoren. Dabei spielt es keine Rolle, ob die Merkmale paarweise unabhängig sind oder nicht. Näheres findet sich in [19].

3.3 Vorabladestrategien

Das Vorausladen bezeichnet abstrakt betrachtet, das Verfügbarmachen von Informationen, bevor diese benötigt werden. In einem Betriebssystem entspricht diese Information dem Inhalt von Dateien, der im Zwischenspeicher[1] zum wahlfreien Zugriff bereit gehalten wird. Welche Informationen vorabgeladen werden, entscheidet eine Heuristik. Sie versucht das zukünftige Programmverhalten vorherzusagen, um anhand dessen die nötigen Teile einer Dateien zu laden.
Man kann zwischen zwei Arten von Vorabladestrategien unterscheiden: zum einen die statische Strategie. Hierbei ist die Reihenfolge der Zugriffe vorher bekannt und unveränderlich. Daher können die Informationen frühzeitig geladen werden, damit der Prozess nicht auf Fertigstellung der E/A-Anfragen warten muss. Ein Beispiel hierfür ist der Start eines Betriebssystems. Zum anderen gibt es die dynamischen Strategien. Im Gegensatz zu der statistischen Variante ist hier das Programmverhalten variable. Es ist nicht festgelegt, welche Aktion als nächstes erfolgt. Aus diesem Grund gibt es unterschiedliche Heuristiken, um den nächsten Schritt vorauszusagen. Eine Möglichkeit ist die Linux-Strategie, die im Folgenden erläutert wird. Ein weiterer Ansatz, der im Rahmen dieser Arbeit evaluiert werden soll, versucht die Vorhersagen mittels maschineller Lernverfahren [3.2] zu treffen. Dieser wird im folgenden Abschnitt ebenfalls vorgestellt.

3.3.1 Linux-Vorabladestrategie

Die Linux-Strategie betrachtet nur lesende Zugriffe auf eine Datei, z.B. durch die Systemaufrufe *read*() oder *pread*() [2]. Dabei wird grundlegend zwischen zwei Arten des Vorausladens unterschieden: das synchrone Vorabladen wird aufgerufen, wenn eine Seite beim Zugriff nicht im Zwischenspeicher vorhanden ist. Die asynchrone Variante wird gestartet, sobald die Seite mit der Markierung *READAHEAD* gelesen wird. Der Zustand des Vorausladens für eine geöffnete Datei wird durch fünf Variablen repräsentiert:

- **start:** Position in Seiten, an der das Vorausladen gestartet wurde.

[1] im Englischen „Page Cache"
[2] siehe „man 2 pread"

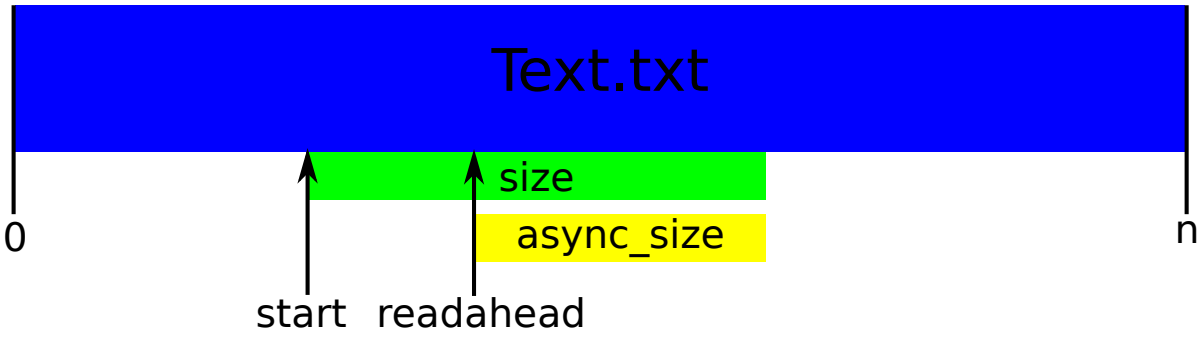


Abbildung 3.2: Linux-Vorabladestrategie

- **size:** Anzahl der Seiten, die vorabgeladen wurden. Man Spricht auch vom *readahead*-Fenster.
- **async_size:** Anzahl verbleibender Seiten, bevor das asynchrone Vorausladen beginnt.
- **ra_pages:** Maximale Anzahl an vorausgeladenen Seiten (geräteabhängig). Ist dieser Wert null, ist das Vorausladen deaktiviert.
- **prev_pos:** Gibt die zuletzt gelesene Position in Bytes an.

Damit die Anwendung bei sequentiellen Zugriffen nicht blockiert, wird das asynchrone Vorausladen gestartet, sobald nur noch *async_size* Seiten übrigbleiben, die betreffende Seite wurde vorher als *READAHED* markiert - siehe Abbildung 3.2. Für die maximale Parallelität wird *async_size* normalerweise auf *size* gesetzt.
Im Folgenden sollen die wichtigsten Funktionen der beiden Varianten vorgestellt werden; Spezialfälle werden nicht betrachtet.
Befindet sich eine durch ein *read*() angeforderte Seite nicht im Speicher, aktiviert das Betriebssystem das synchrone Vorausladen. Zu allererst prüft dies, ob die Datei mit dem Flag *F_RANDOM*[3] geöffnet wurde. Ist dies der Fall, deutet es auf einen randomisierten Zugriff hin, was zum Abschalten der eigentlichen Heuristik führt. Es wird lediglich die eigentliche Anfrage bearbeitet.
Im allgemeinen Fall jedoch wird versucht den Kontext zu ermitteln. Hierzu wird der Zwischenspeicher bzw. die Parameter des Systemaufrufes untersucht, um herauszufinden, ob und wie die Datei vorher bereits gelesen wurde. Lässt sich keinerlei Historie ermitteln, so wird von einem kleinen, zufälligen Zugriff ausgegangen. Es erfolgt kein weiteres Vorausladen. Andernfalls sowie bei Zugriffen, die am Dateianfang beginnen, werden die Zustandsvariablen auf ihre jeweiligen Standardwerte gesetzt. Die Variable *start* wird auf die aktuelle Position gesetzt. Die Größe des Fensters (*size*) wird in Abhängigkeit der Anzahl von der Anwendung angeforderter Seiten und der maximalmöglichen Anzahl vorauszuladener Seiten gewählt. Bei einer großen Anfrage entspricht das Fenster annähernd der Anzahl geforderter Seiten. Handelt es sich hingegen um eine kleine Anfrage wird das

[3] siehe linux-2.6.35.7/include/linux/fs.h Zeile 91

Fenster bis zu viermal so groß gewählt. Schlussendlich werden die entsprechenden Seiten von der Festplatte gelesen.
Das asynchrone Vorausladen wird - wie bereits erwähnt - aufgerufen, wenn im Verlaufe eines *read()* die als *READAHEAD* markierte Seite gelesen wird. Zu allererst muss geprüft werden, ob das darunterliegende Gerät evtl. bereits ausgelastet ist. Falls ja, bricht die Strategie ab. Andernfalls wird der Start des Fensters auf die Position unmittelbar hinter dem Ende des alten Fensters gesetzt. Die Größe des Fensters wird ebenfalls neubestimmt. Hierbei geht die Strategie zunächst aggressiv vor und lässt es schnell anwachsen. Sobald die maximale Größe erreicht wird, flacht das Wachstum ab [20].

3.3.2 Open-Vorabladestrategie

Bei dieser Strategie wird eine Entscheidung über das Vorausladen von Dateiinhalten beim Öffnen der Datei gefällt - sprich dem Systemaufruf *open*()[4]. Dabei gibt es drei unterschiedliche Vorhersagen: *FULL*,*READ* und *ZERO*. Bei *READ* wird randomisiert aus der Datei gelesen. *FULL* hingegen weist darauf hin, dass die Datei von der Anwendung vermutlich vollständig sequentiell gelesen wird. Der Bezeichner *ZERO* bedeutet, dass keinerlei Inhalt gelesen wird.
Die Vorhersagen werden von maschinellen Lernverfahren gemacht. Dabei interagiert das jeweilige Verfahren mit dem gelernten Modell. Hierbei kommen sowohl Naive Bayes [3.2.1] als auch CRF [3.2.2] zum Einsatz. Beide Verfahren machen ihre Vorhersage aufgrund des Anwendungsnamens und der Endung der Datei, die gerade geöffnet wird.
Der Bezeichner *FULL* veranlasst das vollständige Einlesen der Datei in den Zwischenspeicher; bei *ZERO* und *READ* geschieht jedoch nichts. Während auf der Datei operiert wird, finden keine weiteren Aktionen statt.

[4]siehe „man 2 open"

4 Verwandte Arbeiten

4.1 Ein weiterer Teilsystemsimulator

Bereits 1993 haben sich Ganger et al. [12] mit dem Einfluss von E/A-Performanz auf das Gesamtsystem befasst. In ihrem Artikel stellen sie ein Prozessmodell vor, das viele Vorgänge in einem System bereits berücksichtigt. Es wird unter anderem zwischen Kontextwechseln, Unterbrechungen sowie dem Wechsel zwischen Benutzer- und Kernelbereich unterschieden. Die zur Simulation notwendigen Daten haben sie auf einem SVR4 MP UNIX erhoben. Dies wurde von der Firma NCR entwickelt und instrumentiert, um die nötigen Daten aufzuzeichnen. Das Ziel war verschiedene Verfahren zur Sortierung von E/A-Anfragen zu evaluieren hinsichtlich ihres Einflusses auf das gesamte System. Dazu haben sie zwischen drei Klassen von E/A-Anfragen unterschieden: zeitkritisch, zeitlich begrenzt und zeitlich unbeschränkt. Eine Anfrage ist zeitkritisch, wenn ein Prozess auf ihre Fertigstellung wartet. Als zeitlich begrenzt ist eine Anfragen einzuordnen, wenn es einen zeitlichen Rahmen gibt, in dem sie fertiggestellt werden kann, bevor sie zeitkritisch wird, z.B. Anfragen einer Vorabladestrategie. Ein Beispiel für zeitlich unbeschränkte Anfragen sind Schreiboperationen, um die Seiten im Arbeitsspeicher mit der Festplatte synchron zu halten. In ihrer Evaluation haben sie nachgewiesen, dass sich die Ausführungszeit mit einer geeigneten Strategie durchaus reduzieren lässt, jedoch zum Preis einer größeren Anzahl Kontextwechsel. Außerdem haben sie nachgewiesen, dass es sich positiv sowohl auf die Performanz als auch auf die Anzahl Kontextwechsel auswirkt, wenn ein Prozess nicht die CPU entzogen wird, sobald die E/A-Anfrage eines anderen Prozesses fertig ist.
Dieser Artikel weißt sehr viele Parallelen zu dieser Arbeit auf; er kann als regelrechtes Vorbild dienen. Denn im Rahmen dieser Arbeit wurden Kontextwechsel sowie die Wechsel zwischen Benutzer- und Kernelbereich vollkommen außer Acht gelassen. Der bedeutendste Unterschied ergibt sich jedoch aus dem Alter des Artikels. Die vorliegende Arbeit bildet den E/A-Stapel eines aktuellen Linuxkerns nach.

4.2 AMP

Zhou et al. [21] stellen in ihrem Artikel einen dynamischen Zwischenspeicher vor, der je nach Zugriffsmuster unterschiedliche Ersetzungsstrategien verwendet. Zur Erkennung von Zugriffsmustern kombinieren sie verschiede existierende Ansätze, um den größtmöglichen Nutzen zu erhalten. Je nach ermitteltem Muster wird dem Prozess ggf. ein eigener Teil des Zwischenspeichers zugewiesen. Diesem Teil wird passend zum Zugriffs-

muster eine Ersetzungsstrategie zugewiesen. Sie haben anhand des Indezierungswerkzeuges Glimpse [22] gezeigt, dass dieses Vorgehen die Fehlerrate im Zwischenspeicher sowie die Ausführungszeit senken kann. Darüber hinaus haben sie ihre Lösung in einen Linuxkern integriert und getestet. Die eigentliche Arbeit haben sie über ein Programm im Benutzerbereich realisiert. Jedoch liefert ihr Simulator nur qualitative Ergebnisse. Er berücksichtigt keinerlei Festplattenaktivitäten geschweige denn Zusammenhänge zwischen Prozessen. Er simuliert die bloßen Zugriffe auf den Zwischenspeicher. Das Gesamtsystem wird hierbei vollkommen außer Acht gelassen.

4.3 Der AccuSim

Butt et al. [23] haben sich bereits mit dem Einfluss von Vorabladestrategien auf die Performanz am Beispiel von Linux beschäftigt. Sie evaluieren unterschiedliche Ersetzungsstrategien, wie z.B. LRU, LIRS oder 2Q, hinsichtlich ihrer Performanz mit und ohne Vorausladen. Sie verwenden unterschiedliche Anwendungen, die für verschiedene Klassen von Zugriffen stehen. Als Repräsentanten für sequentielle Zugriffe haben sie die Indexzierungswerkzeuge Cscope und Glimpse und den GNU-Compiler gewählt. Die Vertreter für randomisierte Zugriffe sind die beiden Datenbankbenchmarks TPC-H und TPC-R. Leider sind all die Anwendung keine geeigneten Bespiele für Programme, die auf ubiquitären Systemen laufen.
Ihnen ist es durchaus gelungen nachzuweisen, dass ein Vorausladen einen signifikanten Einfluss auf die Performanz haben kann. Jedoch verwendet ihr eigens dafür entwickelter Simulator die aus der Aufzeichnung vorgegeben Prozessreihenfolge. Im Gegensatz zu dieser Arbeit haben sie keinen eigene Prozessablaufsteuerung implementiert. Des Weiteren fehlen dem Simulator die vollständige Nachbildung eines Dateisystems sowie einer E/A-Ablaufsteuerung.

5 Lösungsansätze

Dieses Kapitel legt den Fokus auf die Betrachtung der Lösungsansätze der in [2] ausgearbeiteten Anforderungen. Zunächst sollen das Prozessmodel sowie das gewählte Vorgehen bei der Integration einer Prozessablaufsteuerung erläutert werden. Darauf aufbauend wird das Zeitmodell vorgestellt. Den Abschluss bildet die Realisierung der einzelnen Schichten des E/A-Stapels.

5.1 Generelle Konzepte

Die Grundüberlegung beim Entwurf sieht vor, dass eine bestimmte Menge an Ereignissen bereits vorhanden sein müssen, damit er Simulator mit seiner Arbeit beginnen kann. Dies lässt sich durch zwei Tatsachen begründen: zum einen wird die Abfolge der Prozesse nicht aus der Eingabe übernommen, sondern mittels einer Prozessablaufsteuerung neu bestimmt. Zum anderen kann ein Prozess nur dann simuliert werden, sofern er über Ereignisse verfügt. Würden die Ereignisse Stück für Stück eingelesen und verarbeitet, gäbe es keinen Spielraum, um die Prozessreihenfolge zu verändern, da stets nur ein Prozess bereit wäre zum Rechnen. Folglich sollte im besten Fall die Eingabe vollständig im Speicher liegen. Jedoch wächst dann der Speicherverbrauch linear in der Größe der Eingabedateien.
Um dies zu unterbinden, wird mit zwei parallelen Fäden gearbeitet: Ein Faden arbeitet als Parser. Er liest die Eingabe und konvertiert sie in eine interne Repräsentation. Außerdem legt er die zugehörigen Prozesse in der Prozesstabelle an und ordnet ihnen die eingelesenen Ereignisse zu. Der zweite Faden übernimmt die eigentliche Aufgabe: Er ist für die Simulation des Betriebssystemkerns zuständig. Abbildung 5.1 verdeutlicht diesen Zusammenhang graphisch.
Damit sich eine ausreichend große Menge an Ereignissen im Speicher befindet, nehmen die Ausführungsfäden ihre Arbeit mit zeitlichem Versatz auf - in der Abbildung durch den grünen Balken kenntlich gemacht. Wie groß dieser Versatz ausfällt, hängt ganz von dem Anwender ab. Mit Kommandozeilenparameter $-k$ kann eine Zeitspanne in Sekunden angegeben werden, die der Parser Vorsprung vor dem eigentlichen Simulator erhalten soll. Ein Nachteil bei diesem Vorgehen ist, dass es stark von der Rechenleistung und der aktuellen Auslastung des Computers abhängt, wie viele Ereignisse während dieser Zeit eingelesen werden. Somit sind die Ergebnisse eines Simulationslaufes leider nicht reproduzierbar.
Verfügt der Computer jedoch über ausreichend Arbeitsspeicher, kann der Start des zweiten Fadens soweit hinausgezögert werden bis die Eingabe vollständig eingelesen wurde - der Parameter $-e$ bewerkstelligt dies. In Abbildung 5.1 entspräche die Größe des grünen

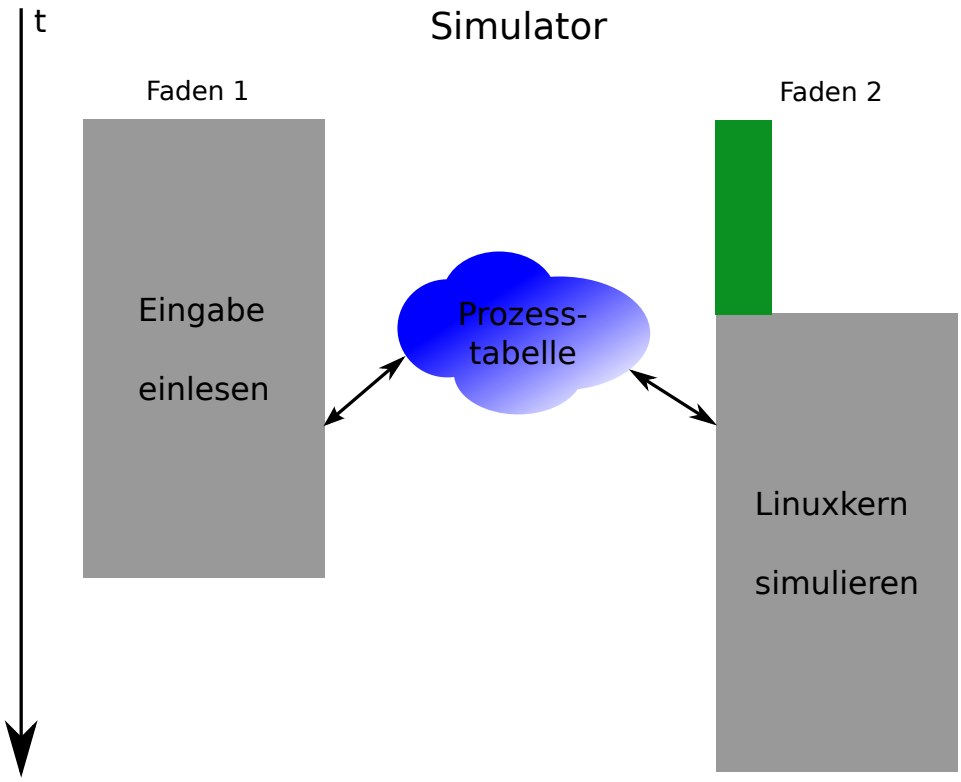


Abbildung 5.1: Aufbau des Simulators

Balkens dann genau der Lebensdauer des ersten Fadens.
Aufgrund des oben genannten Vorgehen ist es leider möglich, dass Prozesse in der Prozesstabelle erscheinen, bevor sie wirklich existieren - der Vaterprozess hat noch kein $fork()$ ausgeführt. Somit würde die relative Reihenfolge der Prozesse untereinander, wie z.B. die Abfolge von Befehlen in einem Bashskript, vollkommen verloren gehen. Um dies zu vermeiden, wird beim Einlesen ein kleiner Trick angewandt: der Parser filtert nach Systemaufrufen, die neue Prozesse($fork()$) bzw. neue Fäden($clone()$) erzeugen. Findet er einen solchen, wird das Kind in der Prozesstabelle angelegt und sofort in den Zustand „nicht ausführbar" (siehe 5.3) versetzt. Hier verweilt er bis im Verlaufe der Simulation der entsprechende Systemaufruf des Vaters bearbeitet wird.

5.2 Simulations- und Zeitmodell

Für die gestellte Aufgabe ist es notwendig ein prozessorientiertes Modell zu verwenden. Das bedeutet, die Zeit schreitet voran, unabhängig von den Geschehnissen im Simulator. Damit die ereignisgesteuerte Komponente nicht außer Acht gelassen wird, werden die einzelnen Anforderungen als Nebenbedingungen an das eigentliche Modell formuliert. Eine dieser Bedingungen ist, dass nur solange simuliert wird bis alle Ereignisse abgearbeitet sind. Außerdem werden - wie bereits erwähnt - nur Prozesse betrachtet, die noch ausstehende Ereignisse haben.

Zunächst einmal handelt es sich um ein zeitdiskretes Modell, das bis auf Mikrosekunden genau auflöst. Ist kein Prozess „rechenbereit", so schreitet die Systemzeit um einen Schritt voran. Die Schrittweite beträgt standardmäßig 100μs. Wenn ein Event abzuarbeiten ist, wird zunächst überprüft, ob der betreffende Prozess Rechenzeit im Benutzerbereich verbracht hat. Ist dies der Fall, so wechselt der Kern in den Benutzermodus und läuft die entsprechende Zeitspanne ab. Dabei schreitet die Zeit pro Durchlauf ebenfalls nur um einen Schritt voran. Im Anschluss wird der eigentliche Systemaufruf behandelt. Hierbei wird die Systemzeit lediglich um eine Zeitkonstante erhöht. Sie entspricht der mittleren Bearbeitungszeit eines Systemaufrufes. Diese wurde mittels eines Systemtapskriptes gemessen [A.1.2]. Der Grund für dieses Vorgehen ist simpel: Da man für Systemaufrufe die mittlere Dauer bestimmen kann, ist der exakte Wert, um den sich die Systemzeit erhöht, bekannt. Somit ist es nicht erforderlich eine Näherung von 100μs zu verwenden.
Nach einer Operation, sei es die Bearbeitung eines Systemaufrufes oder die Rechenzeit im Benutzerbereich und bevor die Systemzeit inkrementiert wird, setzt die E/A-Ablaufsteuerung die erste E/A-Anfrage aus seiner Liste an die Festplatte ab. Dadurch werden - wie gefordert - Rechenoperationen auf der CPU und E/A-Vorgänge parallel durchgeführt.

5.3 Prozesszustandsmodell und Prozessverwaltung

Das erarbeitete Prozesszustandsmodell erinnert sehr an das Standardmodell. Jedoch unterscheidet es sich in der Bedeutung der einzelnen Zustände. Wie in Abbildung 5.3 zu sehen ist, wird zwischen fünf Zuständen unterschieden. Der Zustand „beendet" ist zur Vollständigkeit eingezeichnet. Im Folgenden werden nun die einzelnen Zustände erläutert:

- **beendet:**
  Der Prozess ist beendet worden.
- **bereit:**
  Der Prozess ist bereit zum Rechnen, er befindet sich in der Warteliste der CPU.
- **blockiert:**
  Der Prozess wartet auf die Fertigstellung einer oder mehrerer E/A-Anfragen.
- **wartend:**
  Der Prozess wartet auf die Terminierung eines Kindprozesses.
- **nicht ausführbereit:**
  Der Prozess wartet auf das Signal des Vaters ($fork()$), dass er rechnen darf.

Zwischen den Zuständen „blockiert" und „bereit" wechselt ein Prozess, wenn er E/A-Anfragen abgesetzt hat bzw. diese fertiggestellt sind. Durch den Systemaufruf $wait()$

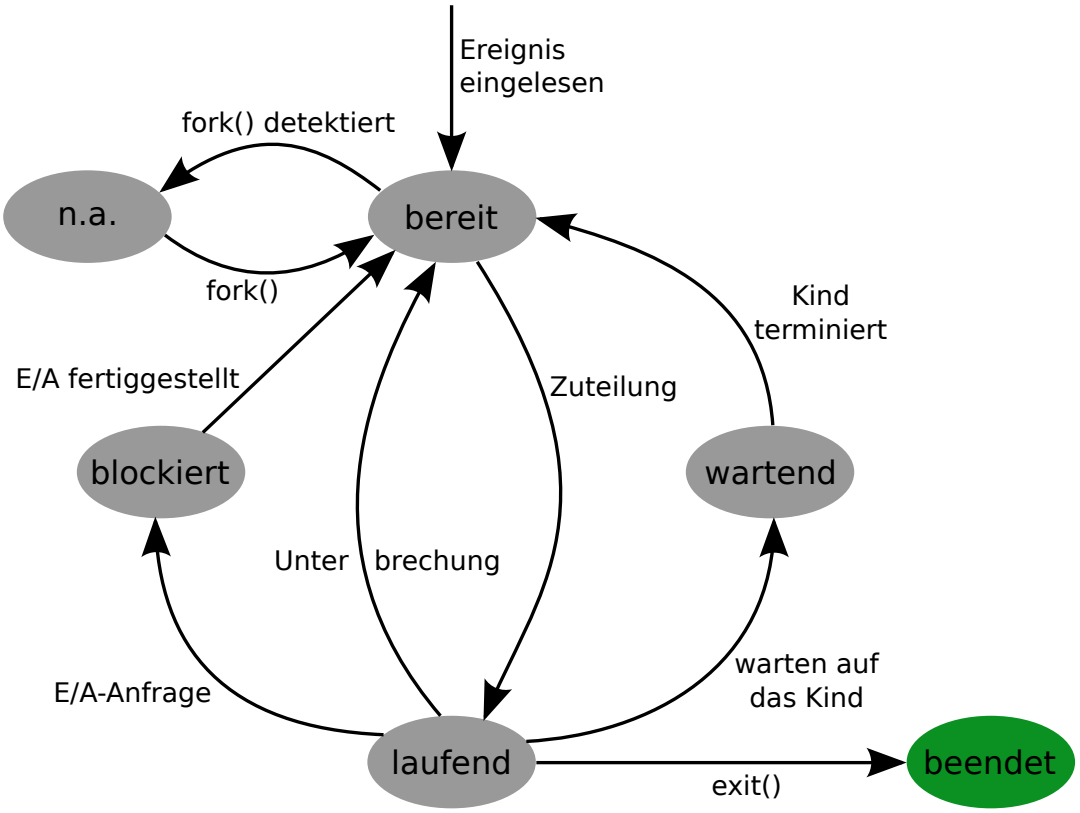


Abbildung 5.2: Prozesszustandsmodell des Simulators

bzw. *waitpid*()[1] wechselt ein Prozess in den Zustand „wartend". Hier verweilt er bis ein Kind terminiert, dann wird er durch ein Signal wieder auf „bereit" gesetzt. Ein Prozess gilt als „nicht ausführbereit", wenn der Parser bereits Ereignisse für diesen eingelesen hat, der Vaterprozess jedoch noch kein $fork()$ ausgeführt hat. Erst durch die Ausführung des passenden Systemaufrufes im Simulator wechselt das Kind in den Zustand „bereit". Somit wird die logische Abfolge der Prozesse - wie sie z.B. in einem Bashskript der Fall ist - eingehalten.
Da Änderungen am System, z.B. eine andere Vorabladestrategie, dafür sorgen, dass Prozesse länger oder kürzer blockiert sind, kann sich auch die Prozessreihenfolge ändern. Aus diesem Grund wird eine eigene Prozessablaufsteuerung eingesetzt, die die Abfolge der Prozesse regelt. Dieser bekommt bei jedem Durchlauf die Menge aller zu diesem Zeitpunkt rechenbereiter Prozesse vorgesetzt und muss daraus den nächsten Prozess ermitteln. Hier kann nach unterschiedlichen Strategien vorgegangen werden. Im Rahmen dieser Arbeit wurde die RoundRobin-Strategie [8] umgesetzt. Jeder Prozess erhält die CPU genau für ein Ereignis. Die Ablaufsteuerung wählt immer den Prozess aus, der die kleinste Prozess-ID hat und der noch nicht in seiner Historie geführt wird. Anschließend wird seine Prozess-ID in der Historie abgelegt. Sind alle Prozesse bereits in die Historie aufgenommen worden, so wird diese geleert. Nun wird erneut der Prozess mit der klein-

[1] siehe „man 2 waitpid"

sten ID ausgewählt.
Das entwickelte Prozessmodell sieht jedoch keine Synchronisation zwischen Prozessen bzw. Fäden vor. Dieser Information lässt sich aus den Aufzeichnungen nicht ermitteln. Dies hat zur Konsequenz, dass gewisse Konstellationen nicht simuliert werden können. Ein solches Szenario könnte wie folgt aussehen:
Ein Programm arbeitet mit einem Hauptausführungsfaden und mehreren Helferfäden. Der Hauptfaden öffnet die vom Benutzer gewünschte Datei und delegiert alle weiteren Aktionen auf dieser Datei an einen Helfer. Wird der Helfer nicht unmittelbar vor der Übergabe erst angelegt, sondern wesentlich früher, so laufen beide Fäden vollkommen unabhängig voneinander. Verarbeitet der Helfer seine Ereignisse schneller als der Hauptfaden, so werden auch die Dateioperationen eher abgearbeitet. Allerdings können sie nicht simuliert werden, da die entsprechende Datei noch gar nicht geöffnet ist.
Hier stößt das aktuelle Modell an seine Grenzen. Daher ist darauf zu achten, dass möglichst keine Anwendungen in einer Aufzeichnung auftauchen, die eben solches Verhalten aufweisen.

5.4 E/A-Stapel

Bei der Realisierung des E/A-Stapels wurde vermehrt Gebrauch von abstrakten Klassen[24] gemacht. Der Vorteil hierbei ist offensichtlich: Man erhält eine eindeutige Schnittstelle, die konkrete Realisierung bleibt jedoch verborgen. Des Weiteren ist es ohne Mehraufwand möglich eine andere Variante z.B. des Dateisystems zu integrieren. Man kann neue Ideen - sofern die vorhandene Schnittstelle ausreicht - sofort implementieren und testen. Über die Zeit kann der Simulator so zu einer Evaluationsplattform mit einer großen Vielfalt an Dateisystemen, Zwischenspeicherersetzungsstrategien etc. heranwachsen. Im Folgenden werden die Ergebnisse der Umsetzung der einzelnen Schichten dargelegt:

- **Anwendung**
  Zum Mitschneiden von Systemaufrufen im Betriebssystemkern wird das Werkzeug SystemTap [25] verwendet. Mit seiner Hilfe kann man auf verschiedene Ereignisse - sogenannte Probe Points - reagieren; dies kann beispielsweise der Aufruf einer Funktion sein. Alle relevanten Ereignisse werden in einem Skript in einer C-ähnlichen Syntax [26] zusammengefasst. Dieses wird dann in echten C-Code übersetzt und als Modul in den Kern eingehangen. Da es als Kernelmodul jegliche Rechte besitzt, gibt es gewisse Restriktionen bezüglich der Rechenleistung, die das Skript beanspruchen darf. Werden hierbei bestimmte Grenzwerte überschritten, wird zunächst die Behandlung eines Ereignisses abgebrochen. Genügt dies nicht, wird das gesamte Skript unterbrochen. Dies ist auch der größte Nachteil an SystemTap. Im Rahmen dieser Arbeit wurde versucht den Übersetzungsvorgang eines aktuellen Linuxkerns mitzuschneiden. Leider ist es nur gelungen das Laden der Bibliotheken sowie ein paar Aktionen das Assemblierers aufzuzeichnen. Die restlichen Operationen, wie z.B. das Übersetzen des Quellcodes, sind verloren gegangen, da

Name	Dateideskriptor	Dateigröße	Inode	Pfad	Mode	Flags	Rückgabewert	Offset	Whence	Anzahl
open/creat	X	X	X	X	X	X	-	-	-	-
fork/vfork	-	-	-	-	-	-	X	-	-	-
close	X	-	-	-	-	-	-	-	-	-
execve	-	-	-	X	-	-	-	-	-	-
exit	-	-	-	-	-	-	-	-	-	-
lseek/llseek	X	-	-	-	-	-	X	X	X	X
pread/write	X	-	-	-	-	-	X	X	-	X
read/write	X	-	-	-	-	-	X	-	-	X
unlink	-	-	X	X	-	-	-	-	-	-

Tabelle 5.1: Liste aller aufgezeichneten Systemaufrufe

sie zu schnell aufeinander folgten. Eine Alternative wäre das Programm Strace [27]. Jedoch beobachtet es nur ein Programm bzw. dessen Kinder. Da im Rahmen dieser Arbeit das gesamte System untersucht werden soll, ist es inakzeptabel.
Aufgabe des SystemTap-Skriptes [A.1.1] ist es die Systemaufrufe zur Prozessverwaltung sowie für Dateioperationen aufzuzeichnen. Hauptaugenmerk liegt dabei auf Funktionen wie *read*()[2] oder *write*()[3]. Eine vollständige Liste aller mitgeschnittenen Systemaufrufe ist in der Tabelle 5.1 zu finden. Pro Systemaufruf gibt es eine bestimmte Menge an Informationen, die immer protokolliert werden. Dies sind die Task- und Prozess-ID, die Prozess-ID des Vaterprozesses, der Name der Anwendung und des Systemaufrufes, die Benutzer- und Gruppen-ID, die Zeit in Mikrosekunden seit Start der Aufzeichnung und die Zeit in Millisekunden, die der Prozess seit dem letzten Systemaufruf im Benutzerbereich verbracht hat. Hinzu kommen Daten, die von dem jeweiligen Systemaufruf abhängen - eine Auflistung ist ebenfalls in der Tabelle 5.1 zu finden. Die Aufzeichnungen dienen anschließend dem Simulator als Eingabe.
Abgespeichert werden die Informationen in einer XML-Datei - die Schemadefinition befindet sich im Anhang [A.2]. XML wurde aufgrund ihrer Lesbarkeit und Erweiterbarkeit anderen Formaten wie z.B. kommaseparierten Varianten vorgezogen. Außerdem existieren hinreichend gute Bibliotheken, mit denen man eine XML-Datei einlesen kann. Darüber hinaus ist es sehr einfach mit einer XML-Schemadefinition die Eingabe auf Validität zu überprüfen.

[2] siehe „man 2 read"
[3] siehe „man 2 write"

- **Zwischenspeicher / Vorabladestrategie**
  Als Ersetzungsstrategien sind zwei einfach und bekannte Verfahren implementiert worden. Hierbei handelt es sich um Least Recently Used - kurz LRU - und um Least Frequently Used - kurz LFU. Bei LRU wird die Seite aus dem Speicher verdrängt, deren Zeitpunkt des letzten Zugriffs am weitesten in der Vergangenheit liegt. LFU hingegen verdrängt die Seite, die am wenigsten angefordert wurde.
  Im Zwischenspeicher werden außerdem wichtige Informationen zur Evaluation erhoben. Dies sind unter anderem: die Trefferquote, die Anzahl vorausgeladener Seiten und die Anzahl vorausgeladener Seiten, die nie angefordert werden - Näheres zu diesen Größeren findet sich in [7.1].
  Darüber hinaus wurden drei Vorabladestrategien umgesetzt. Als erstes ist die Linux-Strategie zu erwähnen. Hier ist 1:1 der Programmcode aus dem Linuxkern übernommen worden; abzüglich etwaiger Anpassungen an die Infrastruktur des Simulators. Damit ist garantiert, dass sie sich exakt so verhält wie in einem echten System. Wie bereits in [3.3.1] erwähnt, gibt es eine vom Gerät abhängige maximale Anzahl von vorausgeladenen Seiten pro geöffneter Datei. Diese lässt sich offenbar nicht aus dem Festplattensimulator ermitteln. Um hier keine willkürlichen Werte zu werden, ist mit Hilfe eines Systemtapskriptes [A.1.3] das Minimum und Maximum an Seiten in einem realen System bestimmt worden. Hiervon wurde dann der Mittelwert gebildet und in die Implementierung übernommen.
  Zusätzlich wurde die open-Strategie [3.3.2] mit zwei unterschiedlichen Vorhersagetechniken ([3.2.1] und [3.2.2]) implementiert.

- **Dateisystem**
  Da die gesamte Arbeit mit einem sehr großen Aufwand verbunden ist, wurden bei der Implementierung eines Dateisystems ein paar Abstriche gemacht. Außerdem ist nach der Begutachtung der Spezifikation des Extended Filesystem 2 [10] - kurz ext2 - klar geworden, dass es einige Arbeit benötigt, um dieses zu implementieren. Daher wurde ein einfaches Dateisystem - SimpleFilesystem genannt - realisiert. Dieses erfüllt die geforderte Programmierschnittstelle vollständig. Es wurde bewusst auf eine Gruppierung der physikalischen Blöcke zu logischen Blöcken, wie es moderne Dateisysteme tun, verzichtet, da der Aufwand für eine Bachelorarbeit zu hoch ist. Es bleibt somit noch Spielraum für Verbesserungen. Das SimpleFilesystem verwendet zur Allokation von physikalischen Sektoren eine First-Fit-Strategie.

- **E/A-Ablaufsteuerung**
  Zunächst wurde eine einfache E/A-Ablaufsteuerung realisiert. Diese verarbeitet Anfragen nachdem FiFo-Prinzip und verändert in keinster Weise die Reihenfolge der Anfragen oder führt adjazente Anfragen zusammen. Für zukünftige Arbeiten wurde die NOOP-Ablaufsteuerung implementiert. Sie ist der Variante im Linuxkern nachempfunden. Ihr Einsatzgebiet ist vor allem bei Speichermedien, die keinen oder einen geringen Positionierungsaufwand benötigen, um Informationen abzurufen. Damit ist sie die erste Wahl beim Einsatz von Flashspeichermedien.

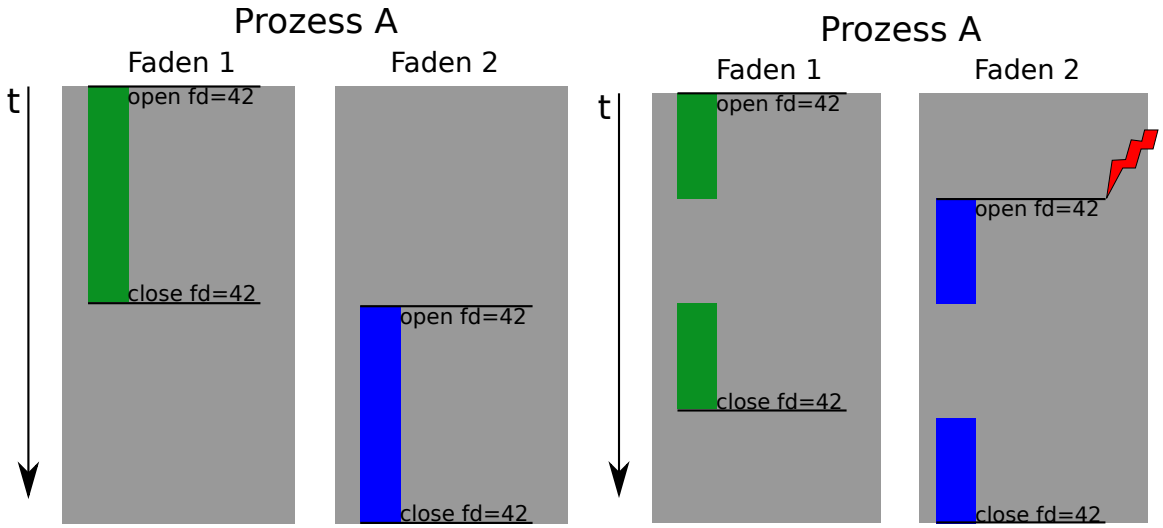
- **Festplatte**
  Auch bei der Simulation der Festplatte stehen zwei Varianten zur Verfügung. Bei dem ersten Modell handelt es sich um keinen echten Simulator. Viel mehr verwendet es für die Berechnung der Dauer einer Anfrage Mittelwerte für die Positionierungs-, Lese- und Schreibzeit. Diese Werte lassen sich nach Belieben zur Übersetzungszeit festlegen. Setzt man die Positionierungszeit auf null, so erhält man ein vereinfachtes Flashspeichermedium.
  Das zweite Modell ist ein sehr mächtiger und hochkonfigurierbarer Festplattensimulator. Der DiskSim ist - je nach Konfiguration - in der Lage vom Treiber bis zum physikalischen Gerät alles zu simulieren. Hierbei ist es möglich sehr genau die einzelnen Komponenten wie den Bus oder den Controller zu konfigurieren. Er ist an der Carnegie Mellon University von Pittsburgh entwickelt worden. Des Weiteren ist er bereits bei einigen Arbeiten über die Performanz von Speichersystemen zum Einsatz gekommen [12],[28],[13]. Außerdem spricht für den DiskSim, dass es eine Weiterentwicklung von Microsoft gibt, mit der auch Solid-State-Disks simuliert werden können.

6 Implementierung

Im nun folgenden Kapitel soll auf einige wichtige Implementierungsdetails eingegangen werden. Im vorherigen Kapitel wurde bereits deutlich, dass zur korrekten Simulation von Ein- und Ausgabeoperationen es einiges mehr nötig ist als die bloße Wiedergabe einer Abfolge solcher Operationen. Bei der Realisierung mussten einige Kompromisse und Umwege gemacht werden, um das Ziel zu erreichen. Im Folgenden soll Anhand von Beispielen aufgezeigt werden, welche Schwierigkeiten dabei entstanden sind und wie diese gelöst wurden. Die meisten, der hier vorgestellten Fälle, resultieren unmittelbar aus der Entscheidung, eine eigene Prozessablaufsteuerung einzusetzen.

6.1 Wettkampfbedingungen um Dateideskriptoren

Bevor konkret auf die Problematik eingegangen wird, soll zunächst ein einführendes Beispiel die Situation verdeutlichen:
Wir betrachten einen Prozess A, der z.B. aus zwei Fäden (1 und 2) besteht. Faden 1 öffnet eine beliebige Datei und bekommt dafür den Dateideskriptor 42 zugewiesen. Im Anschluss führt er eine Reihe von Operationen auf dieser Datei durch und schließt sie schlussendlich wieder. Für Faden 1 ist dies in der Grafik 6.1(a) durch den grünen Balken gekennzeichnet. Nach verstreichen eines beliebig großen Zeitintervalls öffnet Faden 2 eine Datei, ihm wird ebenfalls die Dateideskriptor 42 zugewiesen. Auch er führt Operationen durch und schließt die Datei - verdeutlicht durch den blauen Balken in Abbildung 6.1(a). Beide Vorgänge finden zeitlich nacheinander statt, daher gibt es keinerlei Probleme.
Verändert man nun in dem Simulator die Reihenfolge der Fäden, z.B. durch eine Prozessablaufsteuerung, ist nicht mehr garantiert, dass die Dateizugriffe nacheinander stattfinden. Stattdessen ist es sehr wohl möglich, dass Faden 2 seine Datei öffnen möchte, während Faden 1 seine noch geöffnet hält - oder andersherum. Abbildung 6.1(b) veranschaulicht dies nochmal. Ein naiver Ansatz wäre in diesem Fall Faden 1 abzubrechen und den Systemaufruf nicht auszuführen. Allerdings würden im Folgenden alle Operationen, die eigentlich auf der zweiten Datei ausgeführt würden, auf der noch geöffneten Datei durchgeführt. Im besten Fall ist die erste Datei groß genug, um auch die „fremden" Operationen zu bedienen, so dass wenigstens der Zugriff simuliert werden kann. Jedoch ist dies kein korrektes Systemverhalten.
Das Problem lässt sich umgehen, indem man beim Öffnen einer Datei den belegten Dateideskriptor auf einen Freien umgelenkt. Hierzu verwaltet jeder Prozess eine Abbildung von den belegeten Deskriptoren auf die Umgelenkten. Hierbei handel es sich um eine $1 : n$ Abbildung. Daher ist bei einem Zugriff auf einen umgeleiteten Dateideskriptor nicht möglich, den richtigen aus den n möglichen Deskriptoren zu bestimmen.



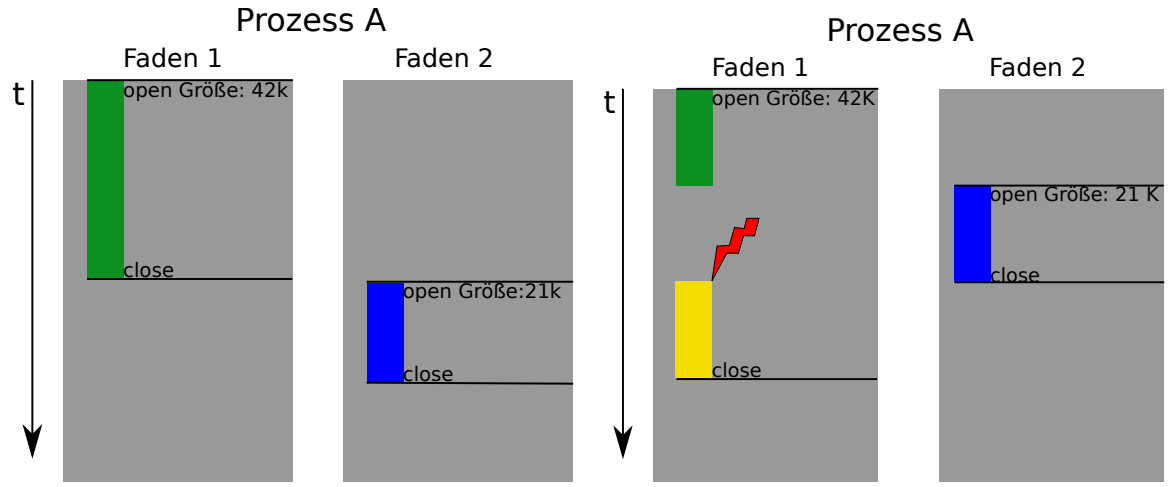
(a) Originaler Verlauf (b) Simulierter Verlauf

Abbildung 6.1: Wettkampfbedingungen um Dateideskriptoren

Offensichtlich ist eine weitere Information von Nöten, um eine Operation zweifelsfrei zu identifizieren.
Die Systematik hinter Dateioperationen ist auf einer sehr abstrakten Ebene immer gleich: Es beginnt mit einem Systemaufruf, der die Datei öffnet, gefolgt von beliebig vielen Aktionen auf selbiger. Abschließend wird die Datei durch den entsprechenden Systemaufruf geschlossen. Diese geschlossene Handlung kann man problemlos als eine Transaktion auffassen. Versieht man jede Transaktion mit einer eindeutigen Identifikationsnummer, kann man jede Operation über den Dateideskriptor und eben diese Nummer eindeutig einer geöffneten Datei zuordnen.
Abschließend sei kurz erläutert, was bei einem Zugriff geschieht: Als erstes wird die Menge aller umgelenkten Dateideskriptoren bestimmt. Hieraus wird derjenige auswählt, der dieselbe Identifikationsnummer besitzt wie der zu verarbeitende Systemaufruf.
Durch dieses Vorgehen ist es ohne Weiteres möglich sehr viele Dateien unter demselben Dateideskriptor zu öffnen.

6.2 Behandlung von Dateigrößen

Um Zugriffe auf Dateien zu simulieren, muss selbstverständlich die Größe der betreffenden Datei bekannt sein. Da potentiell jede Datei von einem Programm geöffnet werden kann, müsste auch jede Datei beobachtet werden. Es ist jedoch viel zu aufwendig, jede Operation, die die Dateigröße verändert, auf allen Dateien zu protokollieren. Stattdessen wird in dem in [5.4] beschriebenen Verfahren bei jedem open-Systemaufruf die Größe einer Datei bestimmt. Somit ist sichergestellt, dass die Sequenz aus lesenden Zugriffen, die höchstwahrscheinlich auf das Öffnen einer Datei folgt, erfolgreich simuliert werden kann. Durch eine Änderungen der Prozessreihenfolge im Vergleich zum beobachteten System



(a) Originaler Verlauf (b) Simulierter Verlauf

Abbildung 6.2: Veränderung der Dateigröße

ist es jedoch möglich, dass eine Datei, während sie von einem anderen Prozess gelesen wird, verkleinert wird. In Abbildung 6.2(a) ist der ursprüngliche Ablauf dargestellt. Prozess A führt Menge von Operationen auf der Datei „log.txt" mit der Größe von 42 KiByte aus. Im späteren Verlauf öffnet Prozess B ebenfalls diese Datei, jedoch mit dem Vermerk, dass sie auf null Bytes gekürzt wird.
Wird nun im Simulator die Abfolge der Prozesse verändert (siehe Abbildung 6.2(b)), können die in Gelb dargestellten Operationen nicht mehr ausgeführt werden, da sie über das Dateiende hinausgehen. Die Lösung hierfür ist recht simpel: die Dateigröße wird beim Öffnen der Datei nur dann verändert, wenn die neue Dateigröße über der alten liegt. Klar ist natürlich, dass dies in keinster Weise dem originalen Systemverhalten entspricht. Die Einbußen hierdurch sind jedoch zu vernachlässigen. Solche wären z.B., dass der Inhalt der Datei, der hinter der neuen Grenze liegt, noch existieren würde. Dies spielt jedoch keine Rolle, da es hier nicht auf den Inhalt, sondern nur auf die Zugriffe und deren Position ankommt. Der hierdurch verschenkte Speicherplatz ist ebenfalls zu vernachlässigen, da in der Regel entsprechend große Festplattenmodell verwendet werden. Im Vordergrund steht eher das Ziel, möglichst alle read-Systemaufrufe zu simulieren. Zudem wären die Ergebnisse zweier Simulationsläufe nicht mehr vergleichbar, wenn die zwischen beiden Läufen veränderten Komponenten Einfluss auf die Prozessreihenfolge hätten.

7 Evaluation

Das folgende Kapitel befasst sich mit Bewertung der unterschiedlichen Lernverfahren. Zunächst werden die betrachteten Messgrößen vorgestellt, sowie deren Beziehungen zueinander. Im Anschluss werden die zur Evaluation verwendeten Programme erläutert. Den Abschluss bildet die Auswertung der Messergebnisse.

7.1 Messgrößen

Zur Bewertung wurde nicht bloß die Trefferquote im Zwischenspeicher herangezogen sondern noch fünf weitere Werte. Diese werden im Folgenden vorgestellt:

- **Ausführungszeit**
  Sie beschreibt die simulierte Zeit vom Starten des Systems bis zum Beenden. Das System gilt als beendet, wenn keine Systemaufrufe mehr ausstehen, alle verbleibenden E/A-Anfragen abgearbeitet sind und alle als „dirty" markierten Seiten auf die Festplatte zurück geschrieben wurden. Dieser Wert beinhaltet die Zeiten für alle Aktivitäten. Er wird in Mikrosekunden angegeben.

- **Leerlaufzeit**
  Dies ist ein Maß dafür, wie lange die CPU sich im Leerlauf befand. Während dieser Zeit hätte man den Prozessor anhalten können. Es gibt weder Systemaufrufe noch Rechenoperationen im Benutzerbereich abzuarbeiten. Diese Größe wird ebenfalls in Mikrosekunden angegeben.
  Anzumerken ist, dass sowohl die Synchronisation als auch das Abarbeiten verbleibender E/A-Anfragen beim Herunterfahren des Systems sich nicht in diesem Wert niederschlagen, da währenddessen potentiell noch Rechenoperationen ausgeführt werden können.

- **Die Anzahl aller E/A-Anfragen**
  Hierbei handelt es sich um die Menge von Sektoranfragen, die an die Festplatte gestellt worden sind. Dieser Wert kann, je nach eingesetzter E/A-Ablaufsteuerung, variieren [11].

- **Gesamte E/A-Zeit**
  Es sei γ_i die Dauer der i-ten E/A-Anfrage an die Festplatte und μ die Anzahl alle E/A-Anfragen. Dann ist die gesamte E/A-Zeit definiert als:

$$\lambda = \sum_{i=1}^{\mu} \gamma_i$$

- **Rechenzeit im Benutzerbereich**
  Dieser gibt Auskunft, wie lange ein System im Benutzerbereich operiert hat. Abgesehen vom Absetzen einzelner E/A-Anfragen an die Festplatte, findet während dieser Zeit keinerlei Aktion im Betriebssystemkern statt. Auch diese Größe wird in Mikrosekunden angegeben.

- **Trefferquote im Zwischenspeicher**
  Dies ist der Quotient aus der Anzahl erfolgreicher Zugriffe und der Summe aller Zugriffe auf den Zwischenspeicher. Hier sind einige Dinge zu beachten:
  In die Statistik gehen nur lesende Zugriffe ein, wie es z.B. während des Systemaufrufes *read*() geschieht. Darüber hinaus können im Verlaufe eines Systemaufrufes mehrere Zugriffe auf den Zwischenspeicher erfolgen. Hier wird lediglich der Erste gewertet, da dieser in jedem Fall stattfindet. Er dient dazu, festzustellen, ob die angeforderten Daten bereits vorliegen. Alle weiteren Zugriffe, die stark von dem Inhalt des Zwischenspeichers und dessen Größe abhängen, werden nicht betrachtet. Dies würde sonst das Ergebnis verfälschen.
  Des Weiteren darf nicht außer Acht gelassen werden, wie eine Applikation eine Datei liest. Das kann entweder in einem Stück oder in mehreren, kleinen Teilen passieren. Bei letzterem ergibt sich folgendes Problem:
  Man betrachte eine Datei beliebiger Größe, die sequentiell in 256-byte-Blöcken gelesen wird. Der erste Zugriff auf eine neue Seite führt dazu, dass der Aufrufer blockiert wird, bis die Seite vollständig von der Festplatte gelesen ist. Die neun folgenden Lesevorgänge fallen ebenfalls in diese Seite, blockieren hingegen nicht und führen in der Statistik zu einem Treffer. Somit entsteht eine Trefferquote von bereits 90%, obwohl lediglich eine Seite gelesen wurde. Umgekehrt führt das Lesen in großen Stücken zu einer wesentlich geringeren Trefferquote, da mit einem Schlag mehrere Seiten angeforderte werden, die alle noch nicht im Speicher liegen.

- **Anzahl vorausgeladener Seiten**
  Es bezeichnet die Menge aller im Laufe der Simulation von einer Vorabladestrategie erzeugen Seiten. Wird eine Seite durch eine Vorabladestrategie erzeugt, später aus dem Speicher verdrängt und anschließend erneut durch das Vorabladeverfahren angelegt, so geht dies doppelt in die Wertung ein. Schließlich sind E/A-Anfragen notwendig, um die entsprechende Seite zu füllen. Somit liefert diese Kenngröße eine Obergrenze für Anzahl überflüssiger E/A-Operationen an.

- **Anzahl vorausgeladener, aber nie zugegriffener Seiten**
  Hiermit ist die Summe aller von der Vorabladestrategie geladenen Seiten, die im Laufe ihres Lebens nie zugegriffen wurden, gemeint. Eine Seite existiert solange im Speicher bis der Simulationslauf beendet ist. Es sei denn, sie wird durch eine andere verdrängt.
  Auch bei dieser Größe gilt, dass eine Seite doppelt in die Wertung einfließen kann, falls sie aufgrund eines vollen Zwischenspeichers mehrmals erzeugt werden muss.

7.2 Bedeutung der Messgrößen

Das Hauptaugenmerk bei der Bewertung liegt nicht - wie sonst üblich - auf der Trefferquote im Zwischenspeicher, sondern auf der Reduktion der Gesamt-E/A-Dauer. Diese ergibt sich aus der Summe der Einzelzeiten für E/A-Anfragen. Je größer dieser Wert ist, desto häufiger musste die Festplatte aktiv werden. Daher gilt: Je geringer die Aktivität eine Festplatte ist, desto häufiger kann sie in den Schlafmodus versetzt werden, was wiederum zu einem reduzierten Energieverbrauch führt. Lässt sich also mit einer Vorabladestrategie die Gesamt-E/A-Dauer senken, so deutet dies auf einen verminderten Energieverbrauch hin.
Dieses Maß allein genügt jedoch noch nicht. Lädt eine Vorabladestrategie eine große Zahl konsekutiver Seiten, führt dies zu einer geringeren E/A-Zeit, da wenig Positionierungsoperationen erforderlich sind. Dafür ist die Wahrscheinlichkeit sehr hoch, dass Seiten geladen werden, die nie angefordert werden. Andere Strategien hingegen laden vielleicht weniger, aber dafür gezielter Seiten voraus, was aufgrund der häufigen Positionswechsel zu einer größeren E/A-Zeit führt. Daher muss ein weiteres Qualitätsmaß definiert werden, dass diesem Punkt Rechnung trägt.
Betrachtet wird der Quotient aus der Anzahl aller vorausgeladenen Seiten und der Anzahl vorausgeladener Seiten, die aber nicht angefordert wurden. Je geringer dieser Wert ausfällt, desto besser ist es, da kaum oder gar keine Seiten unnötig geladen werden. Besser ist jedoch das Komplement des Quotienten zu betrachten. Dies ist ein direktes Gütemaß für eine Vorabladestrategie:

$$\alpha = 1 - \frac{\#\text{vorausgeladene Seiten}}{\#\text{unnötig vorausgeladene Seiten}}$$

Ziel ist es also, die Gesamt-E/A-Dauer zu minimieren unter der Bedingung, dass eine möglichst hohe Güte bei der Vorabladestrategie erreicht wird.

7.3 Benchmark

Ein Benchmark ist Maßstab zur Vergleichbarkeit von Messwerten. Es ist ein festgelegter Testparcours, den alle Probanden durchlaufen müssen. Dadurch lassen sich die Messwerte direkt miteinander vergleichen.
Daher ist es auch sinnvoll einen Benchmark für Vorabladestrategien zu definieren. Ziel ist es dabei möglichst unterschiedliche Programme einzusetzen, um viele Testszenarien abzudecken. Jedoch ist zu beachten, dass die Strategien im Kontext ubiquitärer Systeme eingesetzt werden sollen. Daher sollten Anwendungen gewählt werden, die auf solchen Systemen, wie z.B. ein Smartphone, verwendet werden. Im Folgenden sollen die einzelnen Bestandteile des Testparcours vorgestellt werden:

- **VideoLANClient**:
  Der VideoLANClient [29], kurz VLC, ist ein einfachgehaltenes, aber leistungsstarkes

Abspielprogramm für Medien aller Art. Es wurde ausgewählt, da es ein sehr leichtgewichtiges Programm ist, dessen interne Abläufe sich vollständig simulieren lassen.
Andere Programme synchronisieren ihre Ausführungsfäden so, dass Blöcke[1] von Dateioperationen über Fadengrenzen hinweg stattfinden. Der VideoLANClient kapselt jeden Block in einen Faden.

- **Adobe Acrobat Reader:**
  Der Acrobat Reader [30] ist das Werkzeug zum Betrachten von pdf-Dateien. Er ist bereits für einige mobile Plattformen erhältlich. Daher wurde er für den Testparcours ausgewählt.

- **Eye of GNOME:**
  Eye of GNOME [31] - kurz Eog - ist ein Bildbetrachtungsprogramm unter Gnome. Es steht stellvertretend für alle Programme dieser Art.

Abschließend ist noch erläutern, wie der Benchmark genau aufgebaut ist. Zu allererst werden in Abständen von je einer Minute elf unterschiedliche pdf-Dateien geöffnet. Darauf wird eine Wiedergabeliste sequentiell abgespielt. Ist diese zu Ende, wird eine weitere Wiedergabeliste abgespielt. Sie wird in zufälliger Reihenfolge für die Dauer von 75 Minuten abgespielt. Hierdurch entsteht ein zufälliges Muster von geöffneten Dateien. Im Anschluss wird Eog gestartet, der für 30 Minuten eine Menge von Bildern in zufälliger Reihenfolge abspielt.
Abschließend ist noch auf einen wichtigen Punkt hinzuweisen:
Da die Aufzeichnungen mittels SystemTap im Betriebssystemkern stattfinden, werden die Aktivitäten aller Prozesse aufgezeichnet. Somit sind die Mitschnitte zweier Benchmarkdurchläufe nicht zwangsläufig identisch. Während der Aufzeichnung sind jedoch keine weiteren Programme im Vordergrund aktiv, so dass diese Schwankungen nur durch Hintergrundaktivitäten (Rauschen) beeinflusst werden. Zudem überwiegt die Menge der Systemaufrufe, die durch den Benchmark verursacht wurden, die Menge derer, die von Hintergrundprozessen stammen.

7.4 Auswertung

Zu Beginn sei klargestellt, welche Parameter fix sind:

Anzahl Iterationen	8
Startgröße des Zwischenspeichers	4 MiBi
E/A-Ablaufsteuerung	FiFo
Ersetzungsstrategie	LRU

Bei jeder Iteration ist die Größe des Zwischenspeichers verdoppelt worden.
Die erste Messreihe ist mit einem Festplattenmodell aus den 90er Jahren erstellt worden.

[1] Ein Block besteht aus dem Öffnen einer Datei, den anschließenden Operationen sowie dem Schließen der Datei.

Zwischenspeichergröße	E/A-Anfragen	Gesamt-E/A-Dauer	Trefferquote	Ausführungszeit	Leerlaufzeit	Vorausgeladene Seiten	Unnötige Seiten	Güte
4	50374	247508.6543	80.9686	514407313	220773800	0	0	0
8	49855	244446.4860	81.1218	510887896	217441400	0	0	0
16	49518	242051.8770	81.2420	508742239	215136400	0	0	0
32	49348	241536.0095	81.3004	508181515	214535400	0	0	0
64	49338	241881.0476	81.3038	508369683	214134500	0	0	0
128	48624	237406.4795	81.5652	503373058	210079500	0	0	0
256	48624	237405.4921	81.5652	503371988	207825400	0	0	0
512	48624	237405.4921	81.5652	503371988	207825400	0	0	0

Tabelle 7.1: Messreihe ohne Vorausladen

Außerdem wurden folgende Vorabladestrategien verwendet: keine, die Linux-Strategie und die open-Strategie in den Ausführungen CRF, Naive Bayes und Orakel. Bei dem Orakel handelt es sich um eine Variante, die zunächst die komplette Aufzeichnung betrachtet und anschließend für jedes Öffnen einer Datei sich den zugehörigen Klassifikator merkt. Hiermit ist es möglich, immer das richtige Verhalten vorherzusagen.
Die Zeile „keine Ergebnisse" in den folgenden Tabellen bedeutet, dass der Simulator den betreffenden Simulationsauflauf abgebrochen hat. Der Grund hierfür ist bei einer zu geringen Größe des Zwischenspeicher zu suchen. Falls eine Seiten neu angelegt werden muss, jedoch alle Seiten im Speicher eingelesen oder geschrieben werden, so gibt es keine Seite, die verdrängt werden kann. Der Simulator bricht ab.

Bei der Betrachtung der Messwerte [7.1] für den Durchlauf ohne ein Vorausladen von Seiten fällt sofort auf, dass bereits bei einer sehr geringen Größe des Zwischenspeicher von nur 4 MiBi die Trefferquote bei 80,96% liegt. Somit bleibt wenig Spielraum für Steigerungen. Logischerweise ist die Güte dieser Strategie null, da keine Seiten vorausgeladen werden. Des Weiteren ist zu sehen, dass bei einer massiven Zunahme (Faktor 128) der Größe des Zwischenspeichers die Gesamt-E/A-Dauer sich gerade mal um 4% reduzieren lässt. Im Mittel musste die CPU 41% der Ausführungszeit warten und hätte angehalten werden können.

Die Linux-Strategie weißt dagegen eine deutliche Verbesserung in allen Kategorien auf. Die Gesamt-E/A-Dauer ist im Mittel um 25% geringer als ohne Vorausladen. Ebenfalls gesunken ist die Leerlaufzeit der CPU; hier sind im Mittel Reduktionen von 21% zu beobachten - vgl. Tabelle 7.2. Zudem fällt die hohe Güte von ≈96% auf; es werden kaum Seiten unnötigerweise geladen. Der kleine Ausreißer bei einer Zwischenspeichergröße von 4 MiBi lässt sich durch den knappen Zwischenspeicher begründen: eine Seite wird zwar

Zwischenspeichergröße	E/A-Anfragen	Gesamt-E/A-Dauer	Trefferquote	Ausführungszeit	Leerlaufzeit	Vorausgeladene Seiten	Unnötige Seiten	Güte
4	kein Ergebnis							
8	57231	67692.3629	91.4392	349366554	55787300	55537	6630	88,06
16	53778	62776.6720	91.5859	345140798	51280200	52106	3436	93,40
32	52503	62218.2029	91.7792	344371703	50080000	50831	2511	95,06
64	50758	59702.2105	92.3277	339052482	44630000	49088	1880	96,17
128	49593	57275.5500	94.7228	326514399	30987500	47924	1019	97,87
256	49580	57105.4463	94.7372	326510788	30971400	47911	1019	97,87
512	49580	57105.4463	94.7372	326510788	30971400	47911	1019	97,87

Tabelle 7.2: Messreihe mit der Linux-Strategie

vorausgeladen mit hoher Wahrscheinlichkeit jedoch beim nächsten Systemaufruf durch die Erzeugung einer anderen Seite verdrängt.
Bei einem Blick auf die Tabellen zu Naive Bayes und CRF ([7.3] und [7.4]) fällt sofort auf, dass beide Verfahren erst ab einer Zwischenspeichergröße von 64 MiBi funktionieren. Vorher bricht der Simulator ab. Da die CRFs öfters *FULL* vorhersagen, ist die Anzahl E/A-Anfragen sowie die Gesamt-E/A-Dauer bei ihnen größer als bei Naive Bayes. Dies gilt ebenfalls für die Güte, auch hier ist zu sehen, dass die CRFs ein Stück aggressiver vorgehen: Durch die häufigere Vorhersage von *FULL* werden mehr Seiten vorausgeladen, allerdings werden auch mehr Seiten gar nicht benötigt. Was deutlich auffällt, dass beide Verfahren nicht in der Lage sind die Gesamt-E/A-Dauer zu senken; im Gegenteil sie ist sogar um einiges größer im Vergleich zu dem Durchlauf ohne Vorausladen. Zudem gelingt es nicht, eine annehmbare Güte zu erreichen - sie legt bei beiden unter der 10%-Marke.
Stellt man dem nun das Orakel gegenüber (Tabelle 7.5), das bekanntlich immer richtige Vorhersagen macht, fällt auf, dass es nicht merklich besser ist als die CRFs. Selbstverständlich liegt hier die Güte bei 100%, da nur dann *FULL* vorhergesagt wird, wenn die Datei auch vollständig gelesen wird. Die Gesamt-E/A-Dauer sinkt offensichtlich, jedoch fällt die Reduktion im Vergleich zur Linux-Strategie nur marginal aus. Die anderen Werte verringern sich ebenfalls bzw. steigen im Fall der Trefferquote. Die Änderung im Bezug auf die Werte ohne ein Vorausladen von Seiten ist sehr klein.
Abschließend ist festzuhalten, dass beide Varianten in keinster Weise mit der Linux-Strategie konkurrieren können. Selbst wenn in einem echten System ein Orakel zur Verfügung stünde, das alle Klassifikatoren korrekt vorhersagt, so ist dennoch nicht möglich die Werte der Linux-Strategie zu erreichen.

Zwischenspeichergröße	E/A-Anfragen	Gesamt-E/A-Dauer	Trefferquote	Ausführungszeit	Leerlaufzeit	Vorausgeladene Seiten	Unnötige Seiten	Güte
4	kein Ergebnis							
4	kein Ergebnis							
8	kein Ergebnis							
16	kein Ergebnis							
32	kein Ergebnis							
64	73809	267627.0032	81.5820	530260553	235872400	25941	24450	5,74
128	73098	262284.2631	81.8392	517373195	221838500	25745	24442	5,06
256	72023	265615.1663	81.8485	521625015	226044500	24689	23403	5,20
512	72023	265449.0633	81.8485	521472088	225927900	24689	23403	5,20

Tabelle 7.3: Messreihe mit der Open-Strategie (CRF)

Zwischenspeichergröße	E/A-Anfragen	Gesamt-E/A-Dauer	Trefferquote	Ausführungszeit	Leerlaufzeit	Vorausgeladene Seiten	Unnötige Seiten	Güte
4	kein Ergebnis							
4	kein Ergebnis							
8	kein Ergebnis							
16	kein Ergebnis							
32	kein Ergebnis							
64	60842	252358.0209	81.5205	519195912	224674300	12741	11508	9,6
128	60138	253110.1093	81.7702	514905018	219376200	12620	11500	8,87
256	60120	250804.2506	81.7798	512616488	217072300	12607	11500	8,78
512	60120	250804.2506	81.7798	512616488	217072300	12607	11500	8,78

Tabelle 7.4: Messreihe mit der Open-Strategie (Naive Bayes)

Zwischenspeichergröße	E/A-Anfragen	Gesamt-E/A-Dauer	Trefferquote	Ausführungszeit	Leerlaufzeit	Vorausgeladene Seiten	Unnötige Seiten	Güte
4	kein Ergebnis							
8	kein Ergebnis							
16	kein Ergebnis							
64	49308	195067.0294	86.3217	461928258	167460300	12362	0	100
128	48905	193512.5497	86.4677	461623962	166097300	12262	0	100
256	48615	192583.1772	86.5219	450268988	154727800	12090	0	100
512	48615	192583.1772	86.5219	450268988	154727800	12090	0	100

Tabelle 7.5: Messreihe mit der Open-Strategie (Orakel)

8 Zusammenfassung

8.1 Zusammenfassung

Um das eigentliche Ziel dieser Arbeit, die Bewertung maschineller Lernverfahren, erreichen zu können, bedurfte es einiger Vorarbeiten. Zunächst wurden Bewertungskriterien erarbeitet, nach denen eine Vorabladestrategie zu bewerten ist. Hierbei spielte eine traditionelle Größe, wie die Trefferquote im Zwischenspeicher, eine eher untergeordnete Rolle. Zudem wurden weitere, eigene Gütekriterien definiert [7.2]. Diese haben sich bei der Auswertung der Messdaten als durchaus hilfreiches Mittel erwiesen. Des Weiteren ist es gelungen zu zeigen, dass sich mit einer geeigneten Vorabladestrategie sehr wohl Energie bei Eingebetteten Systemen sparen lässt.
Darüber hinaus ist es im Rahmen dieser Arbeit gelungen eine Evaluationsplattform zu entwickeln, mit der sich verschiedene Schichten des Linux E/A-Stapels untersuchen lassen. Hierzu können diverse Kenngrößen [7.1] herangezogen werden. Als ein Vertreter sei hier die Leerlaufzeit des Prozessors genannt. Klar wurde jedoch auch, dass es nicht möglich ist Abläufe 1:1 zu simulieren. Es mussten an einigen Stellen Kompromisse gemacht werden [6], um eine Simulation zu ermöglichen. In einigen Fällen mussten gar größere Abstriche gemacht werden, da es Konstellationen in einem Betriebssystem gibt, die sich mit dem hier gewählten Konzept nicht abbilden lassen [5.3].
Aufgrund der oben genannten Umstände ist es dann kein Problem mehr gewesen sich auf das Hauptaugenmerk dieser Arbeit zu konzentrieren. Leider offenbarten die Ergebnisse der diversen Simulationsläufe [7.4] eklatante Schwächen des aktuellen Ansatzes. Dieser deckt aktuell nur die Extremfälle ab; entweder wird eine Datei gar nicht vorausgeladen, oder vollständig. Wie bereits erwähnt, gibt es viele Anwendungen, die mit mehreren Fäden parallel aus einer Datei lesen, so dass ein einzelner open-Systemaufruf nur zum Lesen eines Teils der Datei führt. Weiterhin sorgt ein aggressives Vorausladen bei großen Dateien für eine Sättigung des Zwischenspeichers [7.4]. Infolgedessen keinerlei Seiten mehr verdrängt werden können und die Simulation abbricht. Dennoch war es möglich Szenarien aufzuzeigen, die von dem aktuellen Ansatz leicht profitieren konnten. Im Ganzen betrachtet bleibt jedoch festzustellen, dass es noch einiger Forschungsarbeit bedarf, um die Vorabladestrategie, die mit maschinellen Lernverfahren arbeitet, effizienter zu machen.

8.2 Ausblick

Die Auswertung der Messergebnisse hat ganz klar gezeigt, dass der aktuelle Ansatz sich noch nicht mit der Linux-Strategie messen kann. Daher stellt sich die Frage, wie man

weiter vorgehen soll. Eine mögliche Richtung wäre, weg von der Betrachtung einzelner open-Systemaufrufe hin zu globalen Zusammenhängen zu gehen. Dabei versucht man ein früher beobachtetes Programmverhalten auf die aktuelle Situation anzuwenden. Ein naives Beispiel wäre: Prozess A hat in der Vergangenheit nach seiner Erzeugung immer erst Datei 1 und dann Datei 2 gelesen. So ist es möglich unmittelbar nach der Erzeugung beide Dateien in den Zwischenspeicher zu laden, damit sie beim anschließenden Lesen direkt verfügbar sind.
Statt in Konkurrenz mit der Linux-Strategie zu treten ist aber durchaus denkbar, dass man selbige unterstützt. Man versucht Linux bei den pathologischen Fällen zu verbessern, indem man genau dort mit maschinellen Lernverfahren ansetzt. Damit Ergebnisse einfach reproduzierbar sind, sollte hier möglichst eine kostenlose Anwendung zum Einsatz kommen.
Eine weitere Frage, die es in der Zukunft zu beantworten gilt, ist, wie lassen sich maschinelle Lernverfahren in einen echten Linuxkern integrieren. Hierzu muss erst mal geklärt werden, ob dies überhaupt möglich ist. Falls ja, welche Möglichkeiten gibt es und welchen Einfluss haben diese auf das Gesamtsystem.
Darüber hinaus gibt es noch genug Spielraum für Verbesserungen an der Evaluationsplattform. Es bedarf z.B. der Implementierung eines echten Dateisystems, wobei hier der Fokus auf eine Variante gelenkt werden könnte, die vermehrt im Umfeld der Eingebetteten Systeme zum Einsatz kommt.
Um die Bewertung von Vorabladestrategien zu verbessern, wäre es sicherlich sinnvoll den Benchmark um weitere Anwendungen zu ergänzen. Aktuell gibt es keinen echten Vertreter für kleine, randomisierte Zugriffe. Ein solcher Kandidat wäre z.B. der Datenbankbenchmark TPC-H. Dieser wird u.a. bei einem Artikel über den Einfluss der Linux-Strategie auf die Performanz von Zwischenspeichern verwendet [23]. Einziger Nachteil hierbei ist, dass es sich um ein kostenpflichtiges Produkt handelt.
Des Weiteren muss noch ein Weg gefunden werden, um die Verteilung der E/A-Aktivitäten zu erheben. Die Summe aller E/A-Zeiten gibt keinerlei Auskunft über die Verteilung der E/A-Anfragen. Im schlimmsten Fall sind die einzelnen Anfragen so verteilt, dass die Zeit zwischen zwei Anfragen zu kurz ist, um das Speichermedium in einen Schlafmodus zu versetzen und wieder aufzuwecken. In solch einem Fall vergrößert der Schlafmodus zusätzlich die Antwortzeiten der Festplatte, da Zeit für das Aufwecken benötigt wird.

Literaturverzeichnis

[1] Google: *Android*. http://www.android.com/. Version: 02 2011

[2] Witten, Ian H. ; Frank, Eibe: *Data-Minig*. Hanser, 2001

[3] Chen, Feng ; Koufaty, David A. ; Zhang, Xiaodong: Understanding intrinsic characteristics and system implications of flash memory based solid state drives. In: *Proceedings of the eleventh international joint conference on Measurement and modeling of computer systems*. New York, NY, USA : ACM, 2009 (SIGMETRICS '09). – ISBN 978–1–60558–511–6, 181–192

[4] Dirik, Cagdas ; Jacob, Bruce: The performance of PC solid-state disks (SSDs) as a function of bandwidth, concurrency, device architecture, and system organization. In: *Proceedings of the 36th annual international symposium on Computer architecture*. New York, NY, USA : ACM, 2009 (ISCA '09). – ISBN 978–1–60558–526–0, 279–289

[5] Marwedel, Peter: *Eingebettete Systeme*. Bd. korrigierter Nachdruck. Fakultät für Informatik Technische Universität Dortmund Otto-Hahn-Str. 16 44221 Dortmund peter.marwedel@tu-dortmund.de : Springer, 2008

[6] Fricke, Peter ; Jungermann, Felix ; Morik, Katharina ; Piatkowski, Nico ; Spinczyk, Olaf ; Stolpe, Marco: Towards Adjusting Mobile Devices to User's Behaviour, 2010

[7] Tannenbaum, Andrew S.: *Moderne Betriebssysteme*. 2. überarbeitete Auflage. Pearson Studium, 2003. – übersetzt von Prof. Dr. Uwe Baumgarten, TU München

[8] Silberschatz, Abraham ; Galvin, Peter B. ; Gagne, Greg: *Operationg System Concepts*. 8th Edition - International Student Version. John Wiley & Sons (Asia) Pte Ltd., 2010

[9] Bovet, Daniel P. ; Cesati, Marco ; Oram, Andy (Hrsg.): *Understanding the Linux Kernel*. Thrid Edition. O'Reilly Media, 2006

[10] Card, Rémy ; Ts'o, Theodore ; Tweedie, Stephen: Design and Implementation of the Second Extended Filesystem. In: *Proceedings of the First Dutch International Symposium on Linux*. – http://web.mit.edu/tytso/www/linux/ext2intro.html

[11] Love, Robert ; Taber, Mark (Hrsg.): *Linux Kernel Development*. Third Edition. Addison-Wesley, 2010

[12] Ganger, Gregory R. ; Patt, Yale N.: The Process-Flow Model: Examing I/O Performance from the System's Point of View. In: *Proceedings of the ACM Sigmetircs Conference*, 1993, S. 86–97

[13] Worthington, Bruce L. ; Ganger, Gregory R. ; Patt, Yale N.: Scheduling Algorithms for Modern Disk Drives. In: *Proceedings of the ACM Sigmetrics Conference*, 1994, S. 241–251

[14] Digital, Western: *Caviar Black - Product sheet*. http://www.wdc.com/wdproducts/library/SpecSheet/ENG/2879-701276.pdf. Version: 02 2011

[15] OCZ: *OCZ Vertex 2 Series - Product sheet*. http://www.ocztechnology.com/res/manuals/OCZ_Vertex2_3.5in_Product_sheet_3.pdf. Version: 02 2011

[16] Page, Bernd: *Diskrete Simulation Eine Einführung mit Modula-2*. Springer-Verlag, 1991

[17] Xilinx: *ISE Simulator*. http://www.xilinx.com/tools/isim.htm. Version: 02 2011

[18] Hasti, T. ; Tibshirani, R. ; Friedmann, J. H.: *The Elements of Statistical Learning*. Korrigierte Ausgabe. Springer, 2003

[19] Lafferty, J. ; McCallum, A. ; Pereira, F.: Conditional random fiels: Probabilistic models for segmenting and labeling sequence data. In: *Proceedings of the 18th International Conference on Machine learning*, 2001, S. 282–289

[20] linux-2.6.35.7/mm/readahead.c,Zeile294-331

[21] Zhou, Feng ; Behren, Rob von ; Brewer, Eric: AMP: program context specific buffer caching. In: *ATEC '05: Proceedings of the annual conference on USENIX Annual Technical Conference*. Berkeley, CA, USA : USENIX Association, 2005, S. 20–20

[22] Manber, Udi ; Wu, Sun: GLIMPSE: A Tool to Search Through Entire File Systems / The University of Arizona - Department of computer science. 1993. – Forschungsbericht

[23] Butt, Ali R. ; Gniady, Chris ; Hu, Y. C.: The performance impact of kernel prefetching on buffer cache replacement algorithms. In: *SIGMETRICS Perform. Eval. Rev.* 33 (2005), Nr. 1, S. 157–168. http://dx.doi.org/http://doi.acm.org/10.1145/1071690.1064231. – DOI http://doi.acm.org/10.1145/1071690.1064231. – ISSN 0163–5999

[24] Stroustrup, Bjarne ; Auflage, 4. aktualisiert und e. (Hrsg.): *Die C++-Programmiersprache*. Addison-Wesley, 2000. – Deutsche Übersetzung der „Special Edition" von Nicolai Josuttis und Achim Lörke

[25] Red, Hat ; Hitachi ; Oracle: *Systemtap Language Reference*. http://sourceware.org/systemtap/langref/. Version: 02 2011

[26] Red, Hat ; Hitachi ; Oracle: *SystemTap Tutorial*. http://sourceware.org/systemtap/tutorial/. Version: 02 2011

[27] *Strace*. http://sourceforge.net/projects/strace/. Version: 02 2011

[28] Ganger, Gregory R. ; Worthington, Bruce L. ; Hou, Robert Y. ; Patt, Yale N.: Disk Subsystem Load Balancing: Disk Striping vs. Conventional Data Placement. In: *Proceedings of the Hawaii International Conference on System Sciences*, 1993, S. 40–49

[29] *VideoLANClient*. http://www.videolan.org/vlc/. Version: 02 2011

[30] Adobe: *Acrobat Reader*. http://www.adobe.com/de/products/reader.html. Version: 02 2011

[31] Project, The G.: *Eye of Gnome*. http://projects.gnome.org/eog/. Version: 02 2011

Abbildungsverzeichnis

Tabellenverzeichnis

Anhang

A.1 SystemTap-Skripte

A.1.1 Aufzeichnung von Systemaufrufen

```
#! /usr/bin/stap

/*
 * Author: Jochen Streicher (http//ess.cs.tu-dortmund.de/~js)
 * Modified by Alexander Lochmann
 */

/* timestampt at trace start  */
global start

/* For some systemcalls, we need also the return value.
   To bring both call parameters and the return value together, we store
   the pretty-printed parameters at call time in a per-tid-list
   and output them together with the other stuff at return time.

   Mind, that the printed timestamp is actually that of the return time.
*/
global arguments[8192]
global output[8192]

/* for some reasons, we also remember the entry time */
global entry_time[16384]
/* timestamp(since start) at the last syscall return */
global last_sys_ret[32768]
/* to avoid a big usertime(since task start) last_sys_ret has to be activated */
global last_sys_ret_act[32768]

/* we need the return value of open,
   so keep the parameters in per-tid-lists */
global filemodes[8192]
global fileflags[8192]
global filenames[8192]
global filesizes[8192]
global fileinos[8192]
global llseek_result_uaddr[8192]

/* we also have to remember the file names to all filedeskriptors in all */
global filehandles[32768]
/* to look up a path we need the pointer to struct nameidata */
global nd[8192]
global open_counter

function timestamp:long()
{
        return gettimeofday_us() - start
}

probe begin
{
```

```
        start = gettimeofday_us()
        /* type initialization */
        filehandles[0,0] = 0
        open_counter = 0
        printf("<?xml version=\"1.0\" encoding=\"UTF-8\"?>\n<trace xmlns=\"http://www.
            w3.org/2001/XMLSchema\">\n");
}

probe end
{
        printf("</trace>\n");
}

function get_parent(pid:long)
{
        return task_pid(task_parent(pid2task(pid)))
}

/* heading information we print for every event */
function psyscall_entry(l_name)
{
        if (pid() != stp_pid() && pid() != get_parent(stp_pid()) && pid() != get_parent
            (get_parent(stp_pid())))
        {
                entry_time[tid()] = timestamp()

                if (last_sys_ret_act[tid()])
                        cputime = task_utime() - last_sys_ret[tid()]
                else
                        cputime = 0
                cputime = cputime_to_msecs(cputime)
                output[tid()] = sprintf("<event>\n<time>%u</time>\n<usertime>%u</
                    usertime>\n<syscallname>%s</syscallname>\n<pid>%d</pid>\n<tid>%d</
                    tid>\n<ppid>%d</ppid>\n<uid>%u</uid>\n<gid>%u</gid>\n<execname>%s</
                    execname>\n",
                entry_time[tid()],cputime ,l_name, pid(),tid(), ppid(), uid(), gid(),
                    execname())
        }
}

function psyscall_end()
{
        if (pid() != stp_pid() && pid() != get_parent(stp_pid()) && pid() != get_parent
            (get_parent(stp_pid())))
        {
                last_sys_ret[tid()] = task_utime()
                last_sys_ret_act[tid()] = 1
                printf("%s%s</event>\n",output[tid()],arguments[tid()])
        }
        clear_vars()
}

function clear_vars()
{
        delete entry_time[tid()]
        delete arguments[tid()]
        delete output[tid()]
}

function open_return_cleanup()
{
        delete filenames[tid()]
        delete fileflags[tid()]
        delete filemodes[tid()]
        delete filesizes[tid()]
        delete fileinos[tid()]
```

```
        delete nd[tid()]
}

function check_path:long(filename)
{
        subfname = substr(filename,0,5)

        if (subfname != "/proc" && subfname != "/sys/" && subfname != "/dev/")
        {
                return 1
        }
        else
        {
                return 0
        }
}

function get_inode_path(fdes:long)
{
        filp = @cast(task_current(),"task_struct")->files->fdt->fd[fdes]
        filesizes[tid()] = @cast(filp,"file")->f_path->dentry->d_inode->i_size
        fileinos[tid()] = @cast(filp,"file")->f_path->dentry->d_inode->i_ino
        filenames[tid()] = task_dentry_path(task_current(),@cast(filp,"file")->f_path->
            dentry,@cast(filp,"file")->f_path->mnt)
        /*filenames[tid()] = sprintf("%s",substr(path,0,strlen(path) - 1))*/
        imode = @cast(filp,"file")->f_path->dentry->d_inode->i_mode

        /* Based on 'linux-2.6.35.7/include/linux/stat.h': define S_ISDIR(m) (((m) &
            S_IFMT) == S_IFDIR) */
        if ( (imode & 00170000) == 0040000)
        {
                /* return '0 = false' if the file is a directory  */
                return 0
        }
        else
        {
                /* return '1 = everthing is fine' if the file represented by fdes is
                    not a direcotry */
                return 1
        }
}

function get_id(fdes:long)
{
        arguments[tid()] = sprintf("%s<id>%u</id>\n",arguments[tid()],filehandles[pid()
            ,fdes])
}

probe syscall.open* ?
{
        filenames[tid()] = filename
        filemodes[tid()] = mode
        fileflags[tid()] = flags
        psyscall_entry(name)
}

probe syscall.open*.return ?
{
        if ($return < 0 || !entry_time[tid()] || !get_inode_path($return) || !
            check_path(filenames[tid()]))
        {
                clear_vars()
                open_return_cleanup()
                next
        }
```

```
        open_counter = open_counter + 1
        filehandles[pid(), $return] = open_counter
        arguments[tid()] = sprintf("<fd>%d</fd>\n<fsize>%u</fsize>\n<inode>%u</inode>\n
            <path>%s</path>\n<mode>%u</mode>\n<flags>%d</flags>\n<id>%u</id>\n",
            $return, filesizes[tid()], fileinos[tid()], filenames[tid()], filemodes[tid
            ()], fileflags[tid()],filehandles[pid(),$return])
        psyscall_end()

        open_return_cleanup()
}

probe syscall.creat ?
{
        filenames[tid()] = pathname
        filemodes[tid()] = mode
        /* Based on linux-2.6.35.7/include/asm-generic/fcntl.h: O_CREAT|O_WRONLY|
            O_TRUNC */
        fileflags[tid()] = 577
        psyscall_entry(name)
}

probe syscall.creat.return ?
{
        if ($return < 0 || !entry_time[tid()] || !get_inode_path($return) || !
            check_path(filenames[tid()]))
        {
                clear_vars()
                open_return_cleanup()
                next
        }

        open_counter = open_counter + 1
        filehandles[pid(), $return] = open_counter
        arguments[tid()] = sprintf("<fd>%d</fd>\n<fsize>%u</fsize>\n<inode>%u</inode>\n
            <path>%s</path>\n<flags>%d</flags>\n<mode>%u</mode>\n<id>%u</id>\n",
            $return, filesizes[tid()], fileinos[tid()], filenames[tid()], fileflags[tid
            ()],filemodes[tid()],filehandles[pid(),$return])
        psyscall_end()

        open_return_cleanup()
}

probe syscall.close ?
{
        if (filehandles[pid(), $fd] == 0) next
        psyscall_entry(name)
        arguments[tid()] = sprintf("<fd>%d</fd>\n", $fd)
        get_id($fd)
}

probe syscall.close.return ?
{
        if (!entry_time[tid()]) next
        psyscall_end()

        delete filehandles[pid(), $fd]
}

probe syscall.write ?
{
        if (filehandles[pid(), $fd] == 0) next
        psyscall_entry(name)
        arguments[tid()] = sprintf("<fd>%d</fd>\n<count>%u</count>\n", $fd, count)
        get_id($fd)
}
```

```
probe syscall.write.return ?
{
        if (!entry_time[tid()]) next
        arguments[tid()] = sprintf("%s<ret>%d</ret>\n", arguments[tid()],$return)
        psyscall_end()
}

probe syscall.read ?
{
        if (filehandles[pid(), $fd] == 0) next
        psyscall_entry(name)
        arguments[tid()] = sprintf("<fd>%d</fd>\n<count>%u</count>\n", $fd, count)
        get_id($fd)
}

probe syscall.read.return ?
{
        if (!entry_time[tid()]) next
        arguments[tid()] = sprintf("%s<ret>%d</ret>\n", arguments[tid()], $return)
        psyscall_end()
}

probe syscall.pwrite* ?
{
        if (filehandles[pid(), $fd] == 0) next
        psyscall_entry(name)
        arguments[tid()] = sprintf("<fd>%d</fd>\n<count>%u</count>\n<offset>%u</offset
            >\n", $fd, count, offset)
        get_id($fd)
}

probe syscall.pwrite*.return ?
{
        if (!entry_time[tid()]) next
        arguments[tid()] = sprintf("%s<ret>%d</ret>\n", arguments[tid()], $return)
        psyscall_end()
}

probe syscall.pread* ?
{
        if (filehandles[pid(), $fd] == 0) next
        psyscall_entry(name)
        arguments[tid()] = sprintf("<fd>%d</fd>\n<count>%u</count>\n<offset>%u</offset
            >\n", $fd, count, offset)
        get_id($fd)
}

probe syscall.pread*.return ?
{
        if (!entry_time[tid()]) next
        arguments[tid()] = sprintf("%s<ret>%d</ret>\n", arguments[tid()], $return)
        psyscall_end()
}

probe syscall.lseek ?
{
        if (filehandles[pid(), $fd] == 0) next
        psyscall_entry(name)
        arguments[tid()] = sprintf("<fd>%d</fd>\n<whence>%d</whence>\n<offset>%u</
            offset>\n", $fd, whence, offset)
        get_id($fd)
}

probe syscall.lseek.return ?
{
        if (!entry_time[tid()]) next
```

```
        arguments[tid()] = sprintf("%s<ret>%d</ret>\n", arguments[tid()], $return)
        psyscall_end()
}

probe syscall.llseek ?
{
        if (filehandles[pid(), $fd] == 0) next
        llseek_result_uaddr[tid()] = result_uaddr
        psyscall_entry(name)
        arguments[tid()] = sprintf("<fd>%d</fd>\n<whence>%d</whence>\n<offset>%u</
            offset>\n", $fd, whence, (offset_high << 32) | offset_low)
        get_id($fd)
}

probe syscall.llseek.return ?
{
        if (!entry_time[tid()]) next
        ret = user_long(llseek_result_uaddr[tid()])
        arguments[tid()] = sprintf("%s<ret>%d</ret>\n", arguments[tid()],ret)
        psyscall_end()

        delete llseek_result_uaddr[tid()]
}

probe syscall.unlink ?
{
        psyscall_entry(name)
        filenames[tid()] = pathname
}

probe syscall.unlink.return ?
{
        if (!entry_time[tid()]) next
        /*
         * I don't know why, but the kernel could return on error values less than -1
             :-/
         */
        if ($return < 0 || !check_path(filenames[tid()]))
        {
                clear_vars()
        }
        else
        {
                arguments[tid()] = sprintf("<inode>%u</inode>\n<path>%s</path>\n",
                    fileinos[tid()], filenames[tid()])
                psyscall_end()
        }
        delete filenames[tid()]
        delete nd[tid()]
}

probe kernel.function("vfs_unlink")
{
        if (filenames[tid()] == "") next

        if ($dentry->d_inode == 0)
                fileinos[tid()] = -1
        else
                fileinos[tid()] = $dentry->d_inode->i_ino

        filenames[tid()] = sprintf("%s%s",d_path(nd[tid()]),d_name($dentry))
}

%( kernel_vr < "2.6.35"
%? probe kernel.function("__link_path_walk")
%: probe kernel.function("link_path_walk")
```

```
%)
{
        if (filenames[tid()] == "") next

        nd[tid()] = $nd
}


probe syscall.fcntl
{
        if (filehandles[pid(), $fd] == 0) next
        /* only log fcntl() if it wants to set FD_CLOEXEC */
        /* F_SETFD 'linux-2.6.35.7/include/asm-generic/fcntl.h' */
        if (cmd == 2)
        {
                psyscall_entry(name)
                nd[tid()] = fd
                arguments[tid()] = sprintf("<fd>%d</fd>\n<flags>%d</flags>\n<cmd>%d</
                    cmd>\n",fd,arg,cmd)
        }
}

probe syscall.fcntl.return
{
        if (!entry_time[tid()]) next
        if ($return < 0)
        {
                clear_vars()
                delete nd[tid()]
        }
        else
        {
                get_id(nd[tid()])
                psyscall_end()
                delete nd[tid()]
        }
}

probe syscall.fork ?
{
        psyscall_entry(name)
        arguments[tid()] = sprintf("<flags>%u</flags>\n",clone_flags)
}

probe syscall.fork.return ?
{
        if (!entry_time[tid()]) next
        arguments[tid()] = sprintf("%s<ret>%d</ret>\n", arguments[tid()], $return)
        psyscall_end()
}

probe syscall.execve, syscall.compat_execve ?
{
        psyscall_entry("execve")
        arguments[tid()] = sprintf("<path>%s</path>\n", filename)
        psyscall_end()
}

probe syscall.exit ?
{
        psyscall_entry(name)
        psyscall_end()
}

/*probe syscall.nanosleep, syscall.compat_nanosleep
{
```

```
        psyscall_entry("nanosleep")
        period = @cast(req_uaddr,"timespec","<sys/time.h>")->tv_sec * 1000000 + @cast(
            req_uaddr,"timespec","<sys/time.h>")->tv_nsec
        arguments[tid()] = sprintf("<sleeptime>%u</sleeptime>\n",period);
}

probe syscall.nanosleep.return, syscall.compat_nanosleep.return
{
        if (!entry_time[tid()]) next
        if ($return < 0)
        {
                clear_vars()
        }
        else
        {
                psyscall_end()
        }
}*/

probe syscall.wait4
{
        /* Got a waitpid()  */
        if (rusage_uaddr == 0)
        {
                psyscall_entry(name)
                arguments[tid()] = sprintf("<child>%d</child>\n<options>%d</options>\n"
                    , pid,options)
                psyscall_end()
        }
}
```

A.1.2 Messung der Systemzeit von Systemaufrufen

```
#! /usr/bin/stap

global count_syscalls
global total_time
global local_entry_time[8192]

global count_syscalls_subset
global total_time_subset
global local_entry_time_subset[8192]

function psyscall_entry()
{
        local_entry_time_subset[tid()] = cputime_to_msecs(task_stime())
}

function psyscall_end()
{
        total_time_subset = total_time_subset + (cputime_to_msecs(task_stime()) -
            local_entry_time_subset[tid()])
        count_syscalls_subset++

        delete local_entry_time_subset[tid()]
}

probe begin
{
        count_syscalls = 0
}

probe end
{
        printf("All␣systemcalls:\n----------------\n");
```

```
        printf("syscalls:%d, total time:%d ms\n\n", count_syscalls, total_time);
        printf("Our systemcalls:\n----------------\n");
        printf("syscalls:%d, total time:%d ms\n\n", count_syscalls_subset,
            total_time_subset);
}


probe syscall.* ?
{
        local_entry_time[tid()] = cputime_to_msecs(task_stime())
}

probe syscall.*.return ?
{
        if (!local_entry_time[tid()]) next

        total_time = total_time + (cputime_to_msecs(task_stime()) - local_entry_time[
            tid()])
        count_syscalls++

        delete local_entry_time[tid()]
}

probe syscall.open* ?
{
        psyscall_entry()
}

probe syscall.open*.return ?
{
        if (!local_entry_time_subset[tid()]) next
        psyscall_end()
}

probe syscall.creat ?
{
        psyscall_entry()
}

probe syscall.creat.return ?
{
        if (!local_entry_time_subset[tid()]) next
        psyscall_end()
}

probe syscall.close ?
{
        psyscall_entry()
}

probe syscall.close.return ?
{
        if (!local_entry_time_subset[tid()]) next
        psyscall_end()
}

probe syscall.write ?
{
        psyscall_entry()
}

probe syscall.write.return ?
{
        if (!local_entry_time_subset[tid()]) next
        psyscall_end()
}
```

```
probe syscall.read ?
{
        psyscall_entry()
}

probe syscall.read.return ?
{
        if (!local_entry_time_subset[tid()]) next
        psyscall_end()
}

probe syscall.pwrite* ?
{
        psyscall_entry()
}

probe syscall.pwrite*.return ?
{
        if (!local_entry_time_subset[tid()]) next
        psyscall_end()
}

probe syscall.pread* ?
{
        psyscall_entry()
}

probe syscall.pread*.return ?
{
        if (!local_entry_time_subset[tid()]) next
        psyscall_end()
}

probe syscall.lseek ?
{
        psyscall_entry()
}

probe syscall.lseek.return ?
{
        if (!local_entry_time_subset[tid()]) next
        psyscall_end()
}

probe syscall.llseek ?
{
        psyscall_entry()
}

probe syscall.llseek.return ?
{
        if (!local_entry_time_subset[tid()]) next
        psyscall_end()
}

probe syscall.unlink ?
{
        psyscall_entry()
}

probe syscall.unlink.return ?
{
        if (!local_entry_time_subset[tid()]) next
        psyscall_end()
}
```

```
probe syscall.fork ?
{
        psyscall_entry()
}

probe syscall.fork.return ?
{
        if (!local_entry_time_subset[tid()]) next
        psyscall_end()
}

probe syscall.vfork ?
{
        psyscall_entry()
}

probe syscall.vfork.return ?
{
        if (!local_entry_time_subset[tid()]) next
        psyscall_end()
}
```

A.1.3 Messung der maximalmöglichen vorauszuladenen Seiten

```
#! /usr/bin/stap

global count_opencalls
global local_entry_time[8192]
global filenames[32768]
global min_ra
global max_ra

function psyscall_entry(filename:string)
{
        local_entry_time[tid()] = task_stime()
        count_opencalls++
        filenames[tid()] = filename
}

function psyscall_end()
{
        delete filenames[tid()]
        delete local_entry_time[tid()]
}

probe begin
{
        count_opencalls = 0
        max_ra = 0
        min_ra = 13371337
}

probe end
{
        printf("open calls:%d, min:%d, max:%d\n\n",count_opencalls,min_ra,max_ra);
}


probe syscall.open* ?
{
        psyscall_entry(filename)
}

probe syscall.open*.return ?
```

```
{
        if (!local_entry_time[tid()]) next
        psyscall_end()
}

probe syscall.creat ?
{
        psyscall_entry(pathname)
}

probe syscall.creat.return ?
{
        if (!local_entry_time[tid()]) next
        psyscall_end()
}

probe kernel.function("fd_install").call
{
        if (filenames[tid()] == "") next

        if ($file->f_mapping->backing_dev_info == $file->f_path->dentry->d_sb->s_bdi)
        {
                pages = $file->f_mapping->backing_dev_info->ra_pages
                if (pages > max_ra)
                {
                        max_ra = pages
                }
                if (pages < min_ra)
                {
                        min_ra = pages
                }
        }
}
```

A.2 XML Schemadefinition

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema" targetNamespace="http://www.w3.org
    /2001/XMLSchema" elementFormDefault="qualified" attributeFormDefault="qualified">
        <element name="trace">
        <complexType>
                <sequence>
                <element name="event" type="eventType" minOccurs="0" maxOccurs="
                    unbounded"/>
                </sequence>
        </complexType>
        </element>
        <complexType name="eventType">
                <all>
                <element name="time" type="unsignedLong" minOccurs="1" maxOccurs="1"/>
                <element name="usertime" type="unsignedLong" minOccurs="1" maxOccurs="1
                    "/>
                <element name="syscallname" type="string" minOccurs="1" maxOccurs="1"/>
                <element name="pid" type="long" minOccurs="1" maxOccurs="1"/>
                <element name="tid" type="long" minOccurs="1" maxOccurs="1"/>
                <element name="ppid" type="long" minOccurs="1" maxOccurs="1"/>
                <element name="uid" type="unsignedLong" minOccurs="1" maxOccurs="1"/>
                <element name="gid" type="unsignedLong" minOccurs="1" maxOccurs="1"/>
                <element name="execname" type="string" minOccurs="1" maxOccurs="1"/>
                <element name="fd" type="long" minOccurs="0" maxOccurs="1"/>
                <element name="fsize" type="unsignedLong" minOccurs="0" maxOccurs="1"/>
                <element name="inode" type="unsignedLong" minOccurs="0" maxOccurs="1"/>
                <element name="path" type="string" minOccurs="0" maxOccurs="1"/>
                <element name="mode" type="unsignedLong" minOccurs="0" maxOccurs="1"/>
```

```
                <element name="flags" type="long" minOccurs="0" maxOccurs="1"/>
                <element name="offset" type="unsignedLong" minOccurs="0" maxOccurs="1"/
                    >
                <element name="whence" type="long" minOccurs="0" maxOccurs="1"/>
                <element name="count" type="unsignedLong" minOccurs="0" maxOccurs="1"/>
                <element name="ret" type="long" minOccurs="0" maxOccurs="1"/>
                <element name="child" type="long" minOccurs="0" maxOccurs="1"/>
                <element name="sleeptime" type="unsignedLong" minOccurs="0" maxOccurs="
                    1"/>
                <element name="options" type="unsignedLong" minOccurs="0" maxOccurs="1"
                    />
                <element name="cmd" type="long" minOccurs="0" maxOccurs="1"/>
                <element name="id" type="unsignedLong" minOccurs="0" maxOccurs="1"/>
                </all>
        </complexType>
</schema>
```