DIPLOMARBEIT

zum Thema

Entwicklung eines Block-Caches für das DROPS Echtzeit-Dateisystem

an der
Technischen Universität Dresden
Fakultät Informatik
Institut für Systemarchitektur
Lehrstuhl für Betriebssysteme

Eingereicht von: Robin Lutter
Geboren: am 03.10.1973
in Halle/Saale
Matrikel-Nr.: 2291749
Eingereicht am: 06. März 2000

Verantwortlicher Hochschullehrer:
Prof. Dr. H. Härtig

Betreuer:
Dipl.-Inf. Lars Reuther

Selbstständigkeitserklärung

Hiermit erkläre ich, Robin Lutter, das ich die vorliegende Arbeit selbstständig und nur mit den aufgeführten und zugelassenen Hilfsmitteln erstellt habe.

Dresden, 06. März 2000 Robin Lutter

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

Bei der Zusammenarbeit mehrerer unterschiedlich schneller Geräte ist es oft der Fall, daß das schnellere auf das langsamere wartet, z. B. der Prozessor wartet auf Daten aus dem Hauptspeicher oder von der Festplatte. Festplattenzugriffe (ca. 10ms) sind sehr viel langsamer als ein Speicherzugriff (ca. 50ns), das Lesen eines Speicherwortes ist also circa um Faktor 10^5 schneller als ein Lesezugriff auf einen Block der Festplatte. So bieten Caches die Möglichkeit, häufig benutzte Daten schneller zur Verfügung zu stellen. Der Cache stellt eine Sammlung von logisch mit einem Gerät verbundenen Speicherblöcken dar, die jedoch im Speicher gehalten werden. Er verdeckt somit für den Leser den Zugriff auf die Hardware und beantwortet die Anfrage aus seinem Pufferpool, wenn dies möglich ist. Wenn die Daten noch nicht im lokalen Speicher sind, leitet er die Zugriffe an die Hardware weiter. Die Übergabe der Antwort kann durch Kopieren in den Anwendungsspeicher oder durch Einblenden erfolgen.

Im Zusammenhang mit Echtzeit-Anwendungen stellen Caches jedoch eine stochastische Größe bei der Kalkulation des Rechenaufwands dar. Der *Worst Case* verschlechtert sich entsprechend durch die möglicherweise erfolglose Suche im Speicherbereich und anschließenden Lesebefehl. Häufig werden Caches nicht genutzt, um eine bessere Vorhersagbarkeit zu erzielen [But97]. Diese Vorgehensweise ist vor allem beim Einsatz in der Meß- und Steuerungstechnik verbreitet, da hier harte Echtzeitanforderungen existieren, d. h. es werden Garantien über die Reaktions- und Ausführungszeiten abgegeben. In Verbindung mit Multimedia – dem parallelen Abspielen unterschiedlicher Medienarten, z. B. von Audio- und Videoströmen – können diese harten Anforderungen an das System aufgeweicht werden. Diese Form wird auch als weiche Echtzeit bezeichnet. Es sind immer noch Zusagen notwendig, diese haben jedoch eine größere Toleranz als bei einer Anlagensteuerung.

Das DROPS-Projekt (*The Dresden Real Time Operating System*) beschäftigt sich mit der Entwicklung eines Betriebssystems, das – im Unterschied zu traditionellen Echtzeitsystemen – die Koexistenz von Echtzeit- und Timesharinganwendungen auf Systemebene unterstützt. Im Rahmen dieses Projektes wurde ein zusagenfähiges Speichersubsystem, bestehend aus SCSI-Treiber und Dateisystem, entwickelt. Das Dateisystem besitzt jedoch nur eine sehr einfache Pufferverwaltung, die kein Caching anbietet. Das bedeutet, daß jeder Zugriff auf Anwendungsdaten mehrere zusätzliche Festplattenzugriffe erfordert, um Verwaltungsdaten, z. B. Blocklisten oder i-nodes, zu lesen. Eines der Ziele des Projektes ist die Nutzung der freien Ressourcen durch UNIX Anwendungen als Timesharing-Applikationen. Dieses Ziel macht für die Arbeit mit L^4Linux ein Zugriff auf das Dateisystem über einen Cache wünschenswert. Durch Caching von Verwaltungsdaten – im weiteren auch als Metadaten bezeichnet – können weitere Plattenzugriffe eingespart werden. Das Ziel dieser Arbeit ist es, einen intelligenten Cache zur Verwaltung und Übertragung der Daten an unterschiedliche Klienten zu entwickeln.

Dieser Cache soll jedem Strom bzw. allen Anwendungen einen Pufferbereich in ausreichender Größe zur Verfügung stellen und mit Daten füllen. Die bereits im Rahmen von DROPS entwickelten Modu-

le (SCSI-Treiber [Meh98] und Echtzeitdateisystem [Reu98]) sollen dabei als Grundlage verwendet und unterstützt werden. Es soll untersucht werden, ob und wie sich ein Bereitstellen der Daten im voraus (*Prefetch* [MRSW97, Pat97]) in das System einbetten läßt. Aus den Ergebnissen sollen die Unterschiede bzw. Vor- und Nachteile der möglichen Verfahren analysiert werden. Das Vorauslesen der Blöcke kann pauschal, informiert oder adaptiv erfolgen.

Die Anforderung der Ressourcen soll dabei auf Grundlage von *Quality of Service (QoS)* Parametern geschehen bzw. unterstützt werden. Die benötigte Puffergröße kann von der Anwendung über eine globale Parameterliste bereitgestellt werden. Ein anderer Weg ist die Bestimmung der notwendigen Größe im Cache-Modul aus den QoS-Parametern, dabei soll die Größe möglichst genau den Anforderungen der jeweiligen Anwendung entsprechen.

Gliederung Im folgenden Kapitel soll ein Überblick über die Verwaltung von Dateisystem-Caches und eine Beschreibung der Verfahren gegeben werden. Davon ausgehend werden ausgewählte Dateisysteme und ihre Cache-Strategien betrachtet. Weiterhin wird eine kurze Einführung in DROPS und das Echtzeitdateisystem gegeben. In Kapitel 3 soll darauf aufbauend der Entwurf des Caches erläutert werden. Im darauf folgenden Kapitel wird auf einige Details der Implementierung eingegangen. Den Abschluß sollen eine kurze Bewertung der bereits umgesetzten Teile und ein zusammenfassender Ausblick bilden.

2 Stand der Technik

Dieses Kapitel dient zur Einführung in die Arbeit. Es werden die verwendeten Grundlagen wiederholt und ein Überblick über die Mechanismen gegeben. Im zweiten Teil des Kapitels werden Dateisystem-Caches verschiedener Betriebssysteme analysiert. Zum Abschluß soll ein kurzer Überblick über das Gesamtsystem und das Dateisystem gegeben werden.

2.1 Grundlagen

2.1.1 Zugriffsverhalten auf Dateisysteme

Es sind zwei Arten von Dateizugriffen auf das Echtzeitdateisystem zu erwarten: Echtzeit- und Nichtechtzeitzugriffe auf Multimediadateien (Ströme) durch L4-Tasks und nicht zeitkritische Zugriffe von Timesharing-Anwendungen von L^{4}Linux aus.

Multimedia (Echtzeit) Bei Echtzeitanwendungen wird meist auf sehr große Dateien zugegriffen, so daß ein sehr großer Speicherbereich notwendig wäre, um eine Wiederverwendbarkeit der Daten zu ermöglichen. Bei weniger verfügbarem Speicher würde ein Cache nur bei Anwendungen sinnvoll sein, bei denen in zeitlich kurzen Abständen die gleichen Ströme gelesen werden sollen. Weiterhin ist er zweckmäßig beim Auslesen von Verwaltungsstrukturen. Da der Cache auch als Pufferverwaltung für die zeitkritischen Applikationen dienen soll, kann dies in Verbindung mit der Pufferung der Metainformationen geschehen.

L^{4}Linux Die klassische Sichtweise auf die Verwendung von Daten wurde empirisch ermittelt. Dabei wird von vielen kleinen Dateien ausgegangen, die

- häufiger gelesen als geschrieben und noch seltener gelöscht und
- sequentiell gelesen werden [NS98].

Diese Betrachtungsweise trifft vor allem auf Verwaltungsdaten von Dateisystemen zu. Beim Echtzeitdateisystem werden Blocklisten, i-nodes und Bitmaps blockorientiert und Verzeichnisse in Dateien abgespeichert.

Der Cache soll dynamisch Speicher anfordern und – je nach Systemlast – freigeben können. Um neue Anfragen behandeln zu können, wenn der verfügbare Speicher belegt ist, muß eine geeignete Strategie gefunden werden, ein eingelagertes Element zu entfernen. Die Auswahl des zu entfernenden

Blockes entspricht dem Problem der Auslagerung von Speicherseiten. Die dafür entworfenen Algorithmen können also auch hier angewendet werden. Im folgenden sollen die für Caches verwendbaren beschrieben werden.

2.1.2 Ersetzungsstrategien

Caches moderner Dateisysteme puffern sowohl Lese- als auch Schreibzugriffe. Sie bieten unterschiedliche Mechanismen zum Zugriff auf die gepufferten Daten an. Die Anzahl der Puffer kann variabel oder fix sein. Unabhängig davon kommt es im Laufe des Betriebes dazu, daß alle verfügbaren Puffer belegt und keine neuen mehr angefordert werden können. Es muß ein Block entfernt und, falls er geändert wurde, neu auf Platte geschrieben werden. Diese Situation ist mit Verwaltung von virtuellem Speicher vergleichbar. Es können also die bekannten Speicher-Austauschalgorithmen, z. B. *first in first out (FIFO)*, *second chance (aging)*, *not frequently used (NFU)* oder *least recently used (LRU)*, angewendet werden. Ein wesentlicher Unterschied zwischen dem Seitenaustausch (*Paging*) und Caching ist, daß Referenzierungen auf einzelne Elemente im Cache relativ selten sind, so daß es sinnvoll ist, alle Einträge exakt in ihrer Referenzierungsreihenfolge (LRU) zu verketten.

Die optimale Austauschstrategie setzt die Kenntnis der gesamten Referenzfolge der zu speichernden Daten – auch der zukünftigen – voraus. Da die Referenz $i + 1$ aber frühestens zum Zeitpunkt i bekannt wird, ist die optimale Strategie für Dateisystemcaches nicht einsetzbar. Verbreitete Cacheblock-Ersetzungsstrategien sind *Least Recently Used* und *Not Frequently Used* Deshalb sollen beide im folgenden kurz erläutert werden:

Least Recently Used (LRU) Die LRU-Strategie ist die für Caches am häufigsten verwendete Seitenaustauschmethode, sie setzt die Kenntnis der Vergangenheit der gespeicherten Blöcke, aber nicht der Zukunft voraus. Bei Bedarf wird der am längsten nicht verwendete Block ausgelagert. Diese Strategie läßt sich einfach und effizient mittels einer doppelt verketteten Liste implementieren.

Not Frequently Used (NFU) Bei dieser Strategie wird die Vergangenheit jedes einzelnen Blockes betrachtet und es wird der am seltensten verwendete Block ausgelagert. Diese Vorgehensweise hat den Nachteil, daß Blöcke, die in der Vergangenheit oft referenziert wurden, lange Zeit nicht ausgelagert werden, obwohl sie nicht mehr gebraucht werden.

Da sich beide Algorithmen bei der Auswahl des zu entfernenden Blockes – wie bereits beschrieben – sehr extrem verhalten, erscheint eine Kombination beider Algorithmen naheliegend:

Least Recently/Frequently Used (LRFU) Diese Modifikation der Ersetzungsalgorithmen entscheidet auf Basis der Häufigkeit (*frequency*) und der Aktualität über den Austausch eines Blockes im Speicher. Auf Grundlage dieser Parameter erhält jeder Cacheblock einen zusätzlichen Wert. Zu dessen Berechnung gibt es mehrere Möglichkeiten unterschiedlicher Komplexität. Im Implementierungskapitel wird auf die gewählten Strategien näher eingegangen.

2.1.3 Suche im Cache

Da auf die gespeicherten Elemente im Cache möglichst direkt zugegriffen werden soll, gilt es, ein geeignetes Suchverfahren anzuwenden und die Daten mittels einer darauf angepaßten Strategie zu verwalten.

Die Daten im Cache sind unstrukturiert. Mögliche Strategien zum Wiederauffinden der Blöcke sind der Aufbau von Baumstrukturen oder Hashtabellen. Bei der Verwendung von Bäumen gibt es den Nachteil, daß die Komplexität der Suche schlecht skaliert, d. h mit zunehmender Größe des Baumes erhöht sich die Zeitkomplexität der Suche. Bei optimalen Bäumen wächst die Komplexität linear mit der Anzahl der Einträge [Knu73].

Die Verwendung von Hash-Tabellen hat sich für Caches als sinnvoll und nützlich erwiesen. Die Daten werden durch einen Schlüssel k als Suchindex in der Tabelle abgelegt. Die Bestimmung des Schlüssels geschieht über die Hashfunktion h aus dem Datum. Da die Tabellengröße meist kleiner als die Anzahl der möglichen Schlüssel ist, passiert es, daß unterschiedliche Daten auf dieselbe Adresse $h(k)$ abgebildet werden. Dies wird als Kollision bezeichnet und es muß eine Ausweichadresse bestimmt werden. Dafür gibt es verschiedene Strategien zur Auflösung:

Offenes Hashing hier bestimmt der Schlüssel die Suchfolge für die alternative Adresse:

1. Lineare Suche: Falls die Adresse $h(k)$ besetzt ist, wird mit konstantem Abstand c der nächste freie Platz ($h(k) + c$, $h(k) + 2c$, usw.) verwendet.
2. Doppeltes Hashing: Die Vorgehensweise ist analog zur linearen Suche, mit dem Unterschied, daß die Konstante c nicht fest gewählt wird, sondern vom Schlüssel k abhängt, d. h. $c = h_2(k)$. Dabei ist h_2 eine zweite Hashfunktion.

Nichtoffenes Hashing Hier wird die Suchfolge durch eine Zeigerkette bestimmt, die gleichzeitig mit der Tabelle abgelegt wird:

1. Bucket Search: jede Hashadresse stellt den Kopf einer verketten Liste dar, in die alle Daten mit dieser Adresse eingegliedert werden. Diese Liste wird getrennt gespeichert. Die Listen verschiedener Hashadressen sind disjunkt.

2. Listenverkettung: das erste Element jeder Liste steht an der zugehörigen Hashadresse. Bei einer Kollision wird ein freier Speicherplatz gesucht und mit dem zugehörigen Element verkettet. Der Aufbau der gefüllten Tabelle entspricht der Tabelle, die beim linearen Suchen entsteht.

Die nichtoffenen Adressierungsverfahren haben im allgemeinen eine günstigere Zugriffskomplexität, sind jedoch durch die Verwendung zusätzlicher Zeiger speicheraufwendiger als die offenen [Pfl86, Knu73].

2.2 Cache – Mechanismen

Strategie	Eigenschaft	Vorteil	Nachteil
write through	Änderungen werden sofort geschrieben	Konsistenz	viele Einzelzugriffe
nonwrite through	Änderungen werden beim Entfernen des Blockes geschrieben	Zugriffsverhalten	mögliche Inkonsistenz
nonwrite through mit „sync“	Änderungen werden periodisch geschrieben	bessere Integrität als ohne „sync“, vorhersagbares Zugriffsverhalten	

Tabelle 1: Vor- und Nachteile beim Behandeln des Schreibens durch den Cache

Bei der Modifikation von Blöcken entsteht die Situation, daß bei striktem LRU die veränderten Blöcke – je nach Größe des Caches – relativ lang veraltete Werte auf der Festplatte haben, d. h. der Zustand des Dateisystems ist inkonsistent. In Hinblick auf die Ausfallsicherheit ist also eine Unterscheidung der gespeicherten Blöcke wünschenswert:

1. wird der Block in Kürze wieder benötigt (z. B. *double indirect blocks*)?

2. ist der Block für die Konsistenz des Dateisystems relevant?

Anhand dieser Unterscheidung können die Blöcke in Kategorien aufgeteilt werden und man erhält eine modifizierte Variante von LRU. Blöcke, die voraussichtlich bald wieder benötigt werden, bleiben weiter vorn in der Liste und somit länger verfügbar. Falls ein Block wichtig für die Integrität des Systems ist, soll er schnell (unmittelbar) nach seiner Veränderung geschrieben werden. Durch das unmittelbare Schreiben wird die Wahrscheinlichkeit einer Zerstörung des Dateisystems durch einen Ausfall minimiert [Tan95].

In Tabelle 1 sind die Vor- und Nachteile der verschiedenen Möglichkeiten beim Behandeln modifizierter Blöcke in Hinblick auf Konsistenz und Geschwindigkeit dargestellt.

Ein weiteres Mittel, um Festplattenzugriffe einzusparen, ist das Lesen von Daten im voraus (*read ahead* oder *prefetch*). Dafür gibt es unterschiedliche Methoden. Die einfachste ist, linear weitere Blöcke zu lesen. Diese Variante führen moderne Festplatten aus, d. h. sie lesen immer die auf den angeforderten Block folgenden Blöcke in einen eigenen Puffer, um die Anzahl der Kopfbewegungen zu minimieren. Viele Cacheimplementierungen bieten diesen Mechanismus in gleicher oder abgewandelter Form an. Die nächste Möglichkeit beobachtet das Zugriffsverhalten auf die Blöcke und versucht Regeln zu finden und damit das Vorauslesen daran anzupassen (*adaptive prefetch* [MRSW97, CNMC]).

Applikation	Treffer
Relationale Datenbanken	50%
Spracherkennung	90%
Linker, Volltext-Suche	99%

Tabelle 2: Trefferrate durch informiertes Vorauslesen bei bekannten Anwendungen [Pat97]

Eine weitere, sehr erfolgversprechende Vorgehensweise ist das Informieren des Caches über die möglichen Folgezugriffe von Seiten der Anwendung (*informed prefetch*). Dies erfordert jedoch ein genaues Wissen über das Zugriffsverhalten und eine Anpassung der Software. In Tabelle 2 ist die Wahrscheinlichkeit eines Treffers für unterschiedliche, speziell optimierte Anwendungen dargestellt. Patterson vergibt – anhand der Information der Anwendung – Kosten für das Entfernen eines Blockes und das Lesen neuer Blöcke und leitet daraus die Reaktion auf eine Anforderung ab. Das Entfernen von Blöcken basiert auch hier auf LRU, jedoch wird vor dem Verwerfen eine Abschätzung über die Wahrscheinlichkeit der Wiederverwendung getroffen [Pat97].

2.3 Caches existierender Dateisysteme

In diesem Abschnitt sollen die Cachestrategien und -mechanismen verschiedener existierender Dateisysteme analysiert werden. Einen wesentlichen Unterschied gibt es bei der Art der Dateisysteme, sie können traditionell Anwendungs- und Verwaltungsdaten speichern oder zusätzlich Log-Informationen auf der Festplatte ablegen, d. h. es wird vor dem Schreiben der eigentlichen Daten der letzte konsistente Zustand abgespeichert. Diese Systeme werden als *Journaling File System* bezeichnet. Sie garantieren eine hohe Ausfallsicherheit.

Der Cache kann auf zwei unterschiedlichen Wegen in ein System integriert werden: als Bestandteil des jeweiligen Dateisystems oder als eigene Schicht. Die letzte Variante kann wiederum zwischen Dateisystem und Gerät[1] oder zwischen Anwendung und Dateisystem[2] angeordnet werden.

Es werden Caches unterschiedlicher Betriebsysteme betrachtet, die mit expliziten Multimedia- bzw. Servereigenschaften für sich werben. Die Caches werden im Zusammenhang mit den Standarddateisystemen analysiert. Das *Berkeley Fast File System (FFS)* und das *Second Extended File System (ext2)* wurden gewählt, da sie als Grundlage beim Entwurf des Echtzeitdateisystemes (*Real Time File System, RTFS*) dienten (siehe dazu auch Abschnitt 2.4).

2.3.1 Linux und FreeBSD

Beide Dateisystemimplementationen bieten einen Cache auf Blockebene an. Dieser stellt eine Schicht zwischen den Geräten und den Dateisystemen dar. Das Vorauslesen (*read ahead*) wird bei beiden Systemen linear durch den Puffercache realisiert. Beide Systeme bieten eine einheitliche Schnittstelle – das *Virtual File System (VFS)* – für den Zugriff durch Anwendungen auf verschiedene Dateisysteme an. Im folgenden sollen die Standarddateisysteme – das Fast File System und das Second Extended File System – betrachtet werden.

Berkeley Fast File System (FFS) Die Größe des Puffercaches bei FreeBSD (bzw. *Berkeley Software Distribution Operating System, BSD OS)* beträgt ca. 10% des verfügbaren Hauptspeichers. Daten im Cache können direkt in den Nutzeradreßraum eingeblendet werden. Neben dem Cache auf Blockebene bietet FreeBSD einen Namenscache an. Dieser speichert die Beziehungen zwischen Dateinamen und v-nodes. Metadaten werden beim FFS direkt (synchron) geschrieben, der Vorteil dieses Vorgehens ist die Garantie eines konsistenten Dateisystems. Der Nachteil des synchronen Schreibens ist, daß jede kleine Änderung einen physischen Schreibzugriff verursacht [MBKQ99].

[1] im weiteren als blockorientiert bezeichnet
[2] im weiteren als dateiorientiert bezeichnet

Second Extended File System (ext2) Die Linuximplementation unterteilt den Cache in Block- und Verzeichnispuffer, wobei der Verzeichniscache in älteren Kernelversionen noch Bestandteil des ext2-Dateisystems war und erst in aktuellen Versionen fester Bestandteil des *Virtual File System Switch (VFS)* ist. Somit ist die Abkürzung VFS in Verbindung mit Linux nicht mehr als bloße Schnittstelle zu verstehen. Metadaten werden im Linux-VFS solange im Speicher modifiziert und – im Gegensatz zum FFS – gehalten, bis dieser freigegeben werden muß. Dadurch können höhere Geschwindigkeiten erzielt, aber keine Konsistenzgarantien abgegeben werden. Es kann jedoch durch Setzen eines speziellen Flags beim Lesen eines Blockes eine unmittelbare Synchronisation gefordert werden.

Der Blockpuffer verwendet einen gemeinsamen Pufferpool für alle Dateisysteme und logischen Geräte. Linux benutzt ein dynamisches Cachesystem, das den ungenutzten Hauptspeicher als Puffer für die Blockgeräte verwendet. Bei Anstieg des Speicherbedarfs anderer Systemteile wird der Pufferraum verkleinert. Das Lesen eines angeforderten Blockes erfolgt immer synchron. Durch Verwendung eines bestimmten Systemaufrufes kann der Blockcache veranlaßt werden, asynchron weitere Blöcke zu lesen. Das bedeutet, es wird – je nach Puffergröße und physischen Geräteeigenschaften – eine feste Anzahl von weiteren Blöcken gelesen. Beide Caches wählen beim Ersetzen eines Blockes den am längsten nicht verwendeten aus (*least recently used*).

Das Linux-VFS bietet auch eine Schnittstelle zum direkten Speicherzugriff an, die auf dem Blockpuffer aufsetzt. Bis Kernelversion 2.2 fanden an dieser Stelle immer zusätzliche Kopieroperationen vom Pufferspeicher in den Anwendungsspeicher statt [BBD 95].

2.3.2 Windows NT (New Technology File System)

Das Dateisystem von Windows NT (*new technology File System, NTFS*) ist ein transaktionsorientiertes Dateisystem (*journaling*), welches alle Verwaltungsdaten als normale Dateien behandelt und speichert [Sol98]. Das wird auch beim Cache deutlich: im Vergleich zu anderen Dateisystemen besitzt Windows NT einen eigenständigen Cachemanager, der seine Daten auf Basis virtueller Blöcke verwaltet. Diese beinhalten Dateiblöcke, die mehrere Plattenblöcke enthalten. Im Unterschied zum Blockcache bei UNIX-basierten Systemen werden hier alle Daten dateiorientiert zwischengespeichert. Durch die Behandlung und Speicherung der Metainformationen als normale Dateien ist hier keine getrennte Betrachtung der unterschiedlichen Typen notwendig. Zu jeder Datei – auch für Verwaltungsdateien – existieren eine oder mehrere 64 Bit große Dateibeschreibungen (*filerecords*), welche die Informationen über die Datei enthalten. Der Cachemanager unterstützt das Verankern von Speicher (*pinning*). Dadurch darf der genutzte Speicher nicht ausgelagert werden. Im Normalfall kann auch der vom Dateicache verwendete Speicher ausgelagert werden [Rus98]. Der Cachemanager bietet vier verschiedene Schnittstellen für den Zugriff auf die Daten im Cache:

- Strommanipulation (*file stream manipulation*): Zugriffe auf kontinuierliche Daten (Ströme).

- Kopieren von Puffern in den Nutzeradreßraum (*copy interface*): diese Schnittstelle stellt die Möglichkeiten von Vorauslesen (*read ahead*) und verzögertem Schreiben zur Verfügung.
- Speicherlisten (Memory Descriptor List): dadurch kann direkt auf den Speicherbereich des Caches zugegriffen werden (*direct memory access, DMA*). Die Schnittstelle benutzt die gleichen Funktionsaufrufe für das Vorauslesen wie das Copy Interface.
- Pinning Schnittstelle: stellt einen nichtauslagerbaren Speicherbereich zur Verfügung [Nag97].

2.3.3 BeOS (Be-File System)

Da BeOS als Betriebssystem für Multimediaanwendungen entworfen und entwickelt wurde, lagen diesem Dateisystem beim Entwurf Multimediaanforderungen zu Grunde. Das Be-File System (BeFS) ist ein weiteres transaktionsorientiertes System. Die Daten werden auf Blockebene verarbeitet. Zu schreibende Blöcke werden sortiert, um die Anzahl der Schreiboperationen und die Positionierbewegungen zu minimieren. Im Falle eines *cache miss* liest BeOS immer 32 KByte Daten von der Festplatte. Dadurch wird ein einfaches Vorauslesen realisiert. Dieses „blinde“ *read ahead* ist beim Lesen kontinuierlicher Dateien sinnvoll und spart den zusätzlichen Verwaltungsaufwand durch eine Parametrisierung. Verwaltungstrukturen werden analog zu UNIX-Systemen in i-nodes abgespeichert. Für den Zugriff des Dateisystems auf i-nodes unterstützt der Cache DMA-Zugriffe. Ein Nachteil der BeOS Cacheimplementierung ist die feste Größe des Caches. Die Menge des als Pufferbereich verwendeten Speichers wird beim Start festgelegt und beträgt – unabhängig von der Anzahl der Plattenzugriffe – immer ein Achtel des verfügbaren physischen Systemspeichers. Der Cache unterstützt zusätzlich das Umgehen des Caches, es werden – fest – alle I/O-Aufträge durchgereicht, die größer oder gleich 64 KByte sind. Dies ist dann sinnvoll, wenn sehr große Dateien kopiert werden, es führt jedoch zu erhöhtem Verwaltungsaufwand, z. B. ein bereits gelesener Block wird auf diesem Wege modifiziert oder ein im Cache modifizierter Block wird unter Umgehung des Caches gelesen.

Zur Unterstützung des *journaling* gibt es eine explizite „callback“-Funktion. Diese unterstützt das Zurücknehmen von unvollständigen Änderungen des Dateisystems über Transaktionen. Eine Transaktion gilt erst dann als beendet, wenn alle betroffenen Blöcke geschrieben wurden [Gia99].

2.3.4 IRIX (Next Generation File System)

SGI-IRIX verwendet seit Version 5.3 standardmäßig das Next Generation File System (XFS) als Dateisystem. Dies ist der Nachfolger des IRIX-Dateisystems *Extent File System (EFS)* und basiert darauf. Es handelt sich um ein weiteres *journaling* Dateisystem. Jedoch umfassen EFS und XFS einen größeren Bereich als bekannte Dateisysteme, sie bieten zusätzlich eine Schnittstelle zur Anbindung anderer Dateisysteme (z. B. *Network File System (NFS)*) an. XFS arbeitet auf 64 Bit Basis. Es wurde

entwickelt, um sehr große Datenmengen (Datenbanken und digitale Medien) zu verwalten und bietet darüber hinaus garantierte I/O-Raten und Plattenverwaltung an. Zum Zeitpunkt der Anfrage nach garantierter Rate wird die verfügbare Bandbreite ab dem vorgesehenen, geplanten Startzeitpunkt im Dateisystem berechnet und reserviert, d. h. beim Öffnen einer Datei wird der absolute Startzeitpunkt als Parameter übergeben. Der Volume-Manager unterscheidet zwischen Echtzeit-, Meta- und „regulären“ Daten, diese werden physisch getrennt abgespeichert, es existiert ein *realtime-*, ein *log-* und ein *data sub-volume*. Auf dem Echtzeit-Volume werden auch Informationen über die verfügbaren Systemressourcen in Dateien abgelegt. Die angeforderten Garantien können verschiedene Eigenschaften besitzen: per file/file system, private/shared, striping und die Art des Schedulings. Für den Umgang mit kontinuierlichen Daten faßt XFS Gruppen von Blöcken zu einer Region, die sich über mehrere Platten verteilen kann, zusammen. Daten können synchron (ungepuffert) oder asynchron gelesen werden. Log-Informationen werden immer synchron geschrieben.

Der Cache ist, wie bei UNIX basierten Dateisystemen üblich, auf Blockebene angesiedelt und unterscheidet nicht zwischen den Datenarten. Alle Lese- und Schreiboperationen werden durch den Cache ausgeführt. Durch Setzen eines speziellen Flags kann dieser Mechanismus umgangen werden. Zusätzlich wurde ein Transaktionsmechanismus zum Cache hinzugefügt. Der dafür benötigte Log-Manager schreibt seine Daten, für das Dateisystem unsichtbar, mit auf die Festplatte. Ein weiterer Vorteil dieses Systems ist die Möglichkeit des parallelen Zugriffs auf Cacheinhalte. Es können mehrere CPUs gleichzeitig schreibend auf die Daten zugreifen [HD95, SGI99].

2.3.5 Symphony

Das Dateisystem Symphony wurde an der Universität von Texas in Austin entwickelt. Ziel bei der Entwicklung war es, sowohl kontinuierliche Daten (continuous media data) als auch allgemeine Dateien (z. B. Texte) auf einem physischen Dateisystem zu speichern. Metadaten haben eine feste Größe und sind an einer festen Position auf der Festplatte abgelegt. Weiterhin wird eine Teilung in datentypabhängige und -unabhängige Module unternommen. Der datenunabhängige Teil enthält neben dem Ressourcenmanager und dem Plattensubsystem ein Puffersubsystem. Die Pufferverwaltung unterstützt die parallele Verwendung unterschiedlicher Cache-Strategien, dafür wird für jede Strategie ein separater Pufferbereich bereitgestellt. Die Partitionierung des Caches wird in Abhängigkeit von der Systemlast dynamisch durchgeführt. Fällt die Anzahl der freien Puffer unter eine festgelegte Grenze, werden verwendete Puffer im voraus freigegen. Dazu besitzt jedes Pufferelement eine *time to reaccess*-Marke (TTR), welche die wahrscheinliche Zeit bis zum nächsten Zugriff darstellt. Es wird der Puffer mit der größten TTR verdrängt. Die Entscheidung, welche Elemente gespeichert oder verworfen werden, wird entsprechend der Datenart – kontinuierliche oder Textdaten – getroffen. So werden zum Beispiel Textdaten in ihrer Zugriffsreihenfolge (LRU) verkettet und kontinuierliche Daten per *intervall caching* gelesen [SGRV97].

2.4 Einordnung in *DROPS*

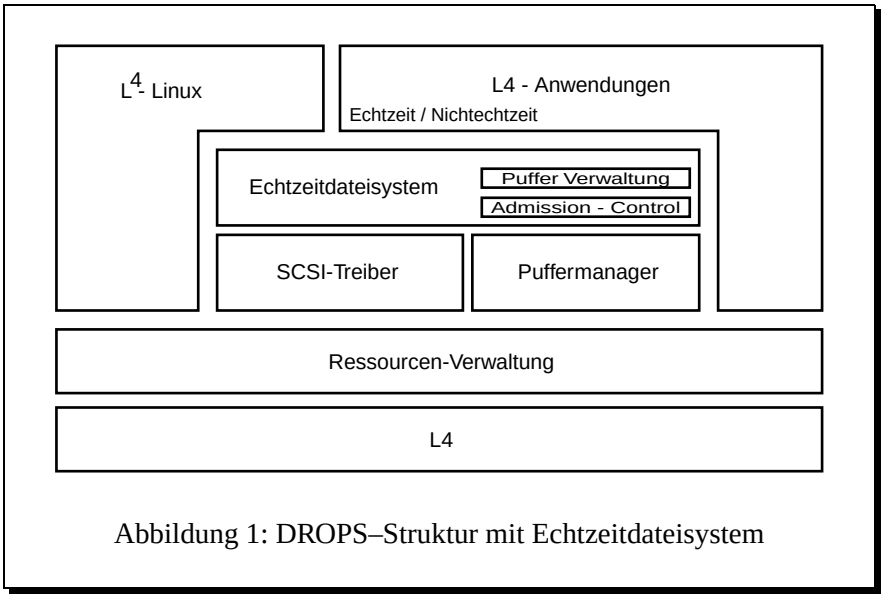

Abbildung 1: DROPS–Struktur mit Echtzeitdateisystem

Das *Dresden Realtime OPerating System* ist ein Projekt des Lehrstuhles für Betriebssysteme an der Technischen Universität Dresden. Ziel dieses Projektes ist ein System, welches Zusagen an Echtzeitanwendungen geben und die ungenutzten Ressourcen parallel UNIX-Anwendungen zu Verfügung stellen kann (siehe Abb. 1). Der dargestellte Puffermanager stellt eine einfache Speicherverwaltung für andere Komponenten zur Verfügung. Für die vorliegende Arbeit bildet das Echtzeitdateisystem mit seinen Komponenten die Grundlage. Im weiteren wird das Zusammenspiel der Komponenten SCSI-Treiber, Pufferverwaltung, Echtzeitdateisystem und des Caches betrachtet.

Als Grundlage dieses Systems dient der Mikrokern L4 [Lie96] bzw. der L4-kompatible Kern FIASCO [Hoh99]. Eine portierte Version des Linux-Kerns stellt die Basis für die UNIX-Anwendungen dar [Hoh96]. Die Hauptanwendungen sind Multimediasysteme, d. h. Systeme zum Verarbeiten und Bereitstellen von Audio- und Videoströmen.

2.4.1 Echtzeitdateisystem (Real Time File System)

Das DROPS Echtzeitdateisystem bietet sowohl eine Schnittstelle für kontinuierliche Daten als auch eine einfache POSIX-Schnittstelle für nichtzeitkritische Anwendungen. Die Verwaltungsstruktur ist an FFS und ext2 angelehnt, sie unterteilt sich in block- (Blocklisten, i-nodes) und dateibasierte (Verzeichnisse) Daten. Im Gegensatz zu Standarddateisystemen unterstützt das Real Time File System (RTFS) mehrere Blockgrößen. Wollen mehrere Anwendungen auf einen Strom vom Dateisystem zugreifen, muß die zugehörige Datei über entsprechend viele Festplatten verteilt werden, um eine Schachtelung der Zugriffe durchführen zu können.

Verwaltungsdaten verbrauchen oft weniger Platz als eine i-node anbietet, d. h. es können unterschiedliche Informationen in einer i-node abgelegt werden, so daß beim Lesen verschiedener Verwaltungsinformationen häufig dieselbe i-node während einem Zugriff mehrfach gelesen werden muß. Dieses Lesen geschieht zur Zeit streng synchron und völlig ungepuffert.

Kommando	Zugriff		
	Plattenblock		Art
Initialisieren	0	root superblock	lesen
	0	partition superblock	lesen
Löschen eines	512933	root i-node	lesen
Verzeichnisses	512933	root i-node	lesen
	512937	«.» root dir	Blockliste lesen
	509129	zu löschendes subdir	lesen
	509129		lesen
	512933		lesen
	512933		lesen
	512933		lesen
	512937		lesen
	512933		lesen
	512937	«.»	schreiben
	509129		lesen

Tabelle 3: Untersuchung des Zugriffsverhalten des Echtzeitdateisystems für ausgewählte Operationen

In Tabelle 3 sind zwei Operationen beispielhaft ausgewählt und die Lesezugriffe des Dateisystems dargestellt. Dieses einfache Beispiel zeigt die Möglichkeiten eines Caches auf Blockebene. Beim Initialisieren des Dateisystems mit einer Partition finden bereits zwei Lesezugriffe auf den selben physischen Block statt. Das zweite Beispiel illustriert, daß ein Cache mit nur 3 Elementen die Anzahl der Plattenzugriffe auf ein Viertel (3mal Lesen und 1mal Schreiben) reduzieren würde. Auf Seite I ff im Anhang sind die vollständigen Tabellen (Nummer 8 und 9) der Analyse von Nichtechtzeitzugriffen abgebildet. Dort ist auch ersichtlich, daß die Anzahl der Plattenzugriffe bei Daten in Unterverzeichnissen je nach Schachtelungstiefe drastisch zunimmt.

Folgendes einfaches Beispiel für eine Folge von Standardzugriffen auf eine Datei im Dateisystem läßt sofort erkennen, daß durch das Halten von Daten im Speicher Plattenzugriffe eingespart werden können (vgl. Tabelle 9 auf S. IV):

1. Datei suchen (Dir root: min. 6 Zugriffe, davon 4 auf root-Block)

2. Datei öffnen (min. 4 Zugriffe, davon 2 auf root-block)

3. Datei modifizieren (min. 7 Zugriffe, davon 5 auf Datei-Blocklisten-Block) und schließen

Die derzeitige Lösung erfordert mindestens 17 sequentielle Festplattenzugriffe auf 5 unterschiedliche Blöcke. Dabei geht das Beispiel von einer einzigen Datei im Wurzelverzeichnis (root) aus.

2.5 Zusammenfassung

Für Dateisysteme werden Caches sowohl von Standardbetriebssystemen (UNIX: FreeBSD und Linux; Windows NT), Systemen mit speziellen Multimedia-Anforderungen (BeOS und Symphony) als auch Systemen im Videoserverbereich mit Zusagen (SGI-IRIX) verwendet, sie werden entweder dateisystemunabhängig block- oder dateiorientiert oder als Bestandteil des Dateisystems verwendet. Der Einsatz von Blockcaches bietet die Möglichkeit, alle Daten untypisiert zu verarbeiten. Dateiorienterte Caches haben den Vorteil, daß Dateien mit ihrem Zusammenhang und Typ verwaltet werden und die Namensauflösung gepuffert werden kann. Dadurch kann für unterschiedliche Zugriffsarten eine angepaßte Strategie gefunden werden, z. B. werden für strombasierte Zugriffe separate Funktionen mit anderer Semantik angeboten als für Zugriffe auf Texte ohne garantierte Rate. Alle betrachteten Systeme verwenden LRU als Standard-Blockersetzungstrategie. Moderne Dateisysteme arbeiten transaktionsorientiert, dies wird auch bei der Verwendung des Caches, z. B. durch spezielle Funktionen, beachtet.

Die Verwendung einer festen Anzahl von Puffern (analog FreeBSD und BeOS) ist in Hinblick auf die verteilte Verwendung von Ressourcen nicht sinnvoll, da auf diese Weise entweder unbenötigter Speicher belegt wird oder benötigte Ressourcen nicht angefordert werden können.

3 Entwurf

In den folgenden Abschnitten sollen die im Abschnitt 2.3 dargestellten Ideen und Ansätze unter Beachtung der Rahmenbedingungen von DROPS im Allgemeinen und dem Echtzeitdateisystem im Speziellen analysiert werden. Aus den gewonnenen Forderungen soll eine angepaßte Umsetzung erzielt werden.

3.1 Entwurfsziele

Ziel dieser Arbeit ist die Entwicklung eines intelligenten Zwischenspeichers. Der Entwurf soll dabei die vom System gegebenen Randbedingungen berücksichtigen und unterstützen. Es wird versucht, alle relevanten Schnittstellen weitestgehend neutral zu definieren.

Weiterhin sind folgende Punkte für den Entwurf des Systems wichtig:

- Die pro Echtzeitverbindung benötigte Puffergröße soll aus parametrisierten Daten berechnet und zur Verfügung gestellt werden.
- Der Cache soll eine systemweite Datenübertragung unterstützen und daten- bzw. zugriffsunabhängig sein. Zur Verwaltung der einzelnen Echtzeit-Datenströme soll jeweils ein separater Pufferbereich pro Anforderung zur Verfügung gestellt werden.
- Alle Metadaten des Dateisystems sollen mit einer geeigneten Strategie gepuffert werden.
- Der Cache soll Nichtechtzeitzugriffe sowohl auf Echtzeitdateien als auch Nichtechtzeitdateien unterstützen, diese sollen sich nicht gegenseitig behindern.
- Dateizugriffe sollen durch ein Lesen im Voraus (*prefetch*) unterstützt werden.

3.2 Datentypen

Die zu puffernden Daten(-typen) werden analog zu den Zugriffsarten des – der Arbeit zugrunde liegenden – Dateisystems gewählt [Reu98]:

- Echtzeitzugriffe auf Echtzeitdateien
  - Kontinuierlicher Strom
  - Nichtkontinuierliche Echtzeitzugriffe

- Nichtechtzeitzugriffe auf Nichtechtzeit- und Echtzeitdateien

In Bezug auf die Bereitstellung von Puffern für das Lesen und Schreiben werden jedoch die Echtzeitdaten zu einer Zugriffsart zusammengefaßt, d. h. nichtkontinuierliche Zugriffe werden auf kontinuierliche abgebildet. Die Konsequenz dieser Vereinfachung ist, daß ein möglicher nichtkontinuierlicher Zugriff vorhersagbar – also einplanbar – sein muß oder es muß damit gerechnet werden, daß er abgewiesen wird. Ein Nachteil dieser Vorgehensweise ist die eventuelle Verschwendung von Ressourcen, da dadurch der Zugriff in jeder Periode eingeplant wird. Die nicht verwendeten Ressourcen können jedoch zeitunkritischen Anwendungen zur Verfügung gestellt werden.

Zusätzlich zu den genannten Typen sollen die Metadaten (Blockliste, i-nodes und Verzeichniseinträge) des Dateisystems in einem Puffer gehalten werden, da sich dadurch die Lesezeiten (Anzahl der Plattenzugriffe) verringern lassen (vgl. Anhang A). Daraus resultiert auch eine Beschleunigung beim Aufbau einer Echtzeitverbindung.

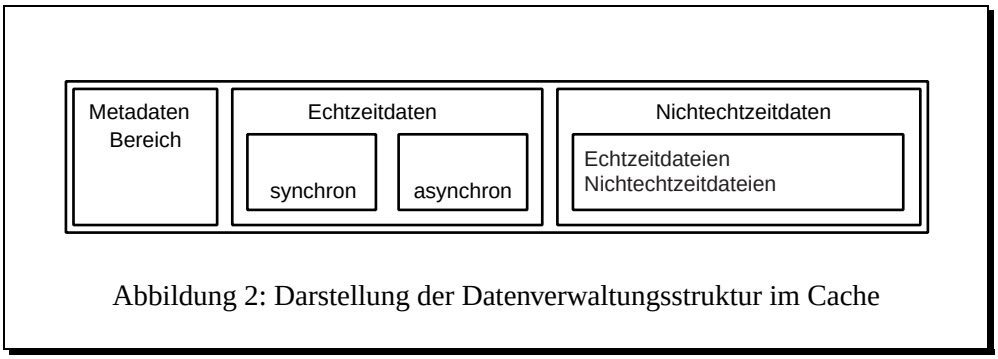

Abbildung 2: Darstellung der Datenverwaltungsstruktur im Cache

Wie in Abbildung 2 sichtbar ist, werden die oben genannten Datenarten auch in der Cache-Komponente getrennt verwaltet. Die unterschiedliche Behandlung von Echtzeit- und Nichtechtzeitdaten wird sofort plausibel, wenn man bedenkt, daß es sich bei Echtzeitströmen meist um große Dateien handelt, auf die sequentiell zugegriffen wird. Eine Pufferung erscheint nur dann sinnvoll, wenn ein sehr großer Speicherbereich zur Verfügung steht und/oder wenn mehrere Anforderungen auf einen Strom in kurzer zeitlicher Abfolge eintreffen. Bei Metainformationen findet keine explizite Trennung zwischen Echtzeit- und Nichtechtzeitzugriffen statt. Die Unterscheidung kann auf Ebene von Prioritäten durchgeführt werden.

3.3 Einordnung in die Systemstruktur

Im folgenden soll der Cache in das in Abschnitt 2.4 beschriebene System unter Beachtung des Komponentenmodells von DROPS eingeordnet werden [HRW 99]. Dadurch wird eine Abstraktion an den Grenzen der Komponenten notwendig, dies soll in den folgenden Abschnitten untersucht werden.

In Abbildung 3 sind die drei denkbaren Punkte dargestellt, an denen ein Cache das Dateisystem unterstützen kann. Die Schnittstelle 1 stellt die in Windows NT genutzte Möglichkeit dar, auf virtueller Blockebene zu cachen (vgl. Abschnitt 2.3.2). Diese Schnittstelle kann für den Nutzer transparent oder in Form einer eigenen Schicht mit zusätzlicher Funktionalität im System integriert werden. Die Vorteile dieses Vorgehens liegen bei einer vollständig dateisystemunabhängigen Komponente, die ihre Funktionalität systemweit unterschiedlichen Dateisystemen zur Verfügung stellen kann.

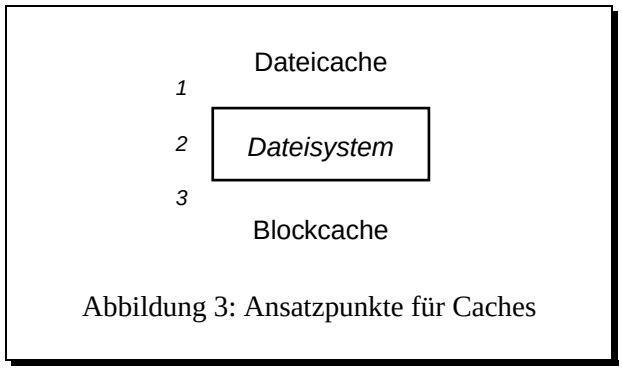

Abbildung 3: Ansatzpunkte für Caches

Eine verbreitete Variante ist es, den Cache zwischen Dateisystem und Hardware auf physischer Blockebene einzusetzen (Position 3 in Abb. 3). Dieser Ansatz hat den Nachteil, daß jegliche Informationen über die Beziehungen zwischen den Blöcken und ihre Eigenschaften zusätzlich übertragen werden müssen oder verloren gehen. Die Vorteile dieses Vorgehens liegen bei der untypisierten Pufferung von Verwaltungsdaten des Dateisystems und einem gerätespezifischen *read ahead*.

Ein erster Ansatz zur Implementierung war es, den Cache als untergeordnetes Modul in das Dateisystem zu integrieren (Variante 2 in Abb. 3). Dadurch wäre eine einfache und schnelle Integration in das bestehende Dateisystem und schnelle Kommunikation innerhalb eines Adreßraumes gewährleistet gewesen. Die notwendige Geräteunabhängigkeit ist auf diese Art durch die Schnittstelle des Dateisystems zum SCSI-Treiber gegeben. Dieses Vorgehen würde einen Teil der Vorzüge des dateiorientierten Cachings zur Verfügung stellen können, da die notwendigen Zusammenhänge zwischen den Blöcken im Dateisystem aufgelöst werden, also bekannt sind. Der Nachteil dieses Herangehens ist, daß jedes genutzte Dateisystem einen eigenen Cache benötigt. Dadurch würden die Größe und der Entwicklungsaufwand eines jeden Dateisystems anwachsen.

Im Laufe der Vorbetrachtungen erwies sich die Entwicklung eines Cache-Servers als eigenständiger Systembestandteil als zweckmäßiger, da er dadurch unabhängig vom jeweiligen Dateisystem und Hardwaretreiber ist. Dadurch können auch andere Klienten relativ einfach integriert werden. Der Server soll in Form einen logischen Dateisystemes oberhalb der physischen Dateisysteme – beziehungsweise anderer Systeme mit Dateischnittstelle – angeordnet werden. Dieser Entwurf sieht eine Abstraktion von den physischen Dateisystemen durch eine einheitliche Schnittstelle vor (Abbildung 7). Diese wird transparent, also sowohl in den physischen als auch im logischen Dateisystem verwendet. Es stellt also eine Schicht – ähnlich dem Linux *Virtual File System Switch* bzw. Windows NT Cachemanager – mit zusätzlicher Funktionalität dar. Die Pufferung und Verwaltung findet mittels logischer

Blöcke auf Dateiebene statt. Der Vorteil einer eigenen Schicht liegt in einer einheitlichen Schnittstelle zu beliebigen Dateisystemen und der Nutzung von Standardaktionen, die auf dieser Ebene realisierbar sind. Zum Beispiel können Merkmale wie *read ahead* wesentlich gezielter und effizienter eingesetzt bzw. implementiert werden, da der Dateizusammenhang bekannt ist. Weiterhin ergibt sich dadurch eine einfachere Wartung, Fehlersuche und Austauschbarkeit der einzelnen Teile des Systems (Mikrokernidee [Här98]).

Mit Einführung einer neuen Schicht stellt sich die Frage, welche Aufgaben sinnvoll auf dieser Ebene angeordnet werden können. So ist beispielsweise eine Integration grundlegender Aufgaben eines Dateisystems auf dieser Ebene, z. B. das Angebot verschiedener Dienste, wie Namensauflösung und der Aufbau von Namensräumen, denkbar.

Im Zusammenhang mit dem existierenden Dateisystem und dem Ziel, Caching für alle Metadaten des Dateisystems anzubieten, ist jedoch die Entwicklung eines reinen Dateicaches – analog zum NTFS – aus verschiedenen Gründen nicht sinnvoll bzw. nicht ausreichend. Da es sich beim RTFS um ein UNIX-orientertes Dateisystem handelt, kann nur ein Teil der Metadaten auf Dateiebene verwaltet werden (Verzeichniseinträge). Ein anderer Teil der Metainformationen wird auf Blockebene (i-nodes, Blocklisten) verarbeitet und ist auf Dateiebene nicht sichtbar. Wie bereits dargestellt, ist jedoch beim Echtzeitdateisystem ein Caching auch auf Blockebene sinnvoll (vgl. 2.4).

Die Einführung einer weiteren Schicht bzw. Serverkomponente zwischen Hardware und Dateisystem würde jedoch einen höheren Kommunikations- und Verwaltungsaufwand erfordern. Dadurch entstehen Performanceeinbußen durch zusätzliche Kontextwechsel, Kommunikation und Speicheraustausch. Weiterhin ist es durchaus denkbar, daß eine Weiterentwicklung des Dateisystems oder die Einbindung eines anderen Dateisystems den Blockcache unnötig macht.

3.3.1 Prinzipielle Organisation des Caches

Aus den vorangegangenen Abschnitten ergibt sich eine Teilung des Caches in einen Blockcache für die Verarbeitung der Metadaten und einen Dateicache für die Anwendungsdaten (Primärdaten).

Verbindungsarten Auf die in Abschnitt 3.1 beschriebenen Datenarten sind Zugriffe auf unterschiedlichen Wegen möglich: Echtzeitzugriffe und Nichtechtzeitzugriffe auf Echtzeitdateien. Weiterhin sollen Zugriffe auf Nichtechtzeitdateien gestattet werden. Daraus ergibt sich eine Dreiteilung der Verarbeitung: die Echtzeitzugriffe können synchron ablaufen, d. h. der Cache agiert nur als Puffermanager. Anwendung und Dateisystem kommunizieren über die DROPS-Stromschnittstelle direkt miteinander. Oder es wird eine asynchrone Verbindung aufgebaut, hierbei stellt der Cache die Puffer zur Verfügung und ist aktiv an der Datenübertragung beteiligt. Der dritte Teil verwaltet die Puffer

für die Nichtechtzeitzugriffe. Dieser Teil stellt auch die Cache-Funktionalität für Echtzeit-Metadaten bereit.

Im folgenden soll auf die beiden Komponenten – Blockcache und logisches Dateisystem – noch näher eingegangen werden.

Blockcachebibliothek Der vorliegende Entwurf sieht den Blockcache als fakultativen Bestandteil des Dateisystems (Variante 2 in Abbildung 3) vor, dadurch entfällt die sonst notwendige Synchronisation der Schreib- und Leseaufträge von Blockcache und dem zugehörigen Dateisystem. Das Dateisystem hat das Wissen über die Einordnung von Metadaten und die Anordnung dieser auf dem Gerät. Des weiteren werden alle Aufträge an die Hardware über eine gemeinsame Schnittstelle ausgeführt. Das Dateisystem kann also alle eigenen Schreib-/Leseaufträge verwalten und organisieren, z. B. durch Prioritäten und das Bündeln von Aufträgen.

Im Fall einer getrennten Lösung muß eine zusätzliche Komponente die Aufträge eines Dateisystems, also den Zugriff auf Verwaltungs- und Anwendungsdaten, koordinieren. Wird keine weitere Komponente verwendet, so muß das Dateisystem seine Aufträge entsprechend der Verwendung markieren, um ein doppeltes Speichern und damit verbundenes Kopieren zu verhindern. Jedoch würden die Abarbeitungszeiten aller Aufträge zunehmen. Bei der integierten Lösung kann die Funktionalität des Blockcaches an die Belange des Dateisystems – durch die Art und Weise der Verwendung – angepaßt werden. Ein Argument gegen die integrierte Lösung wäre das Vertrauen der Anwendungen untereinander. Bei getrennten Komponenten kann jede Anwendung entscheiden, ob sie der anderen traut. Im vorliegenden Fall müssen Blockcache und das Dateisystem einander trauen, da der Cache Operationen des Dateisystems ausführt. Weiterhin benötigen nicht alle Dateisysteme einen Blockcache.

Logisches Dateisystem Die Verwendung eines logischen Dateisystems bringt verschiedene Vorteile mit sich, z. B. ergeben sich bessere Möglichkeiten der Namensauflösung, -caching und des Prefetches als auf Blockebene, da auf Dateiebene die Struktur und der Zusammenhang bekannt sind. Dieses Wissen auf Blockebene abzubilden, würde einen höheren Verwaltungs- und Kommunikationsaufwand bedeuten, die Zusammenhänge müßten extra er- bzw. übermittelt werden. Das logische Dateisystem beinhaltet drei wesentliche Teile: den Dateicache, die Speicherverwaltung sowohl für Puffer- und Dateicache als auch für Echtzeitverbindungen und die Namensverwaltung für physische Dateisysteme. Die Pufferbereiche können verbindungsorientiert vergeben und verwaltet werden. Im Falle der asynchronen Übertragung und des Planungsmodells vom RTFS kommt ein weiterer Vorteil des Caches zum Tragen: die maximale Anzahl der Zugriffe auf einen Strom mit maximaler Bandbreite ist nicht mehr auschließlich von der Leistung der Festplatte(n) abhängig. Das bedeutet, daß aus mehreren Anforderungströmen für eine Datei im Idealfall ein Anforderungsstrom vom logischen an das physische Dateisystem generiert wird. Die Anwendungen nutzen dann einen größeren gemeinsamen

Speicherbereich. Das logische Dateisystem soll sich für die Anwendungen transparent in das System einfügen.

Nachfolgend sind die Vorteile einer logischen Dateisystem-Schicht zusammengefaßt:

1. Möglichkeit der Namensauflösung und eines Namenscaches innerhalb einer Schicht
2. Einheitliche Systemschnittstelle für blockorientierte (Datei-)Systeme (analog UNIX VFS)
3. Realisierung von Mounten und Elementaroperationen auf Dateien (Aufbau von Namensräumen)
4. Prefetch auf Dateiebene
5. Logging: Der Cache kann das Dateisystem nutzen, um vor dem Schreiben der Dateisystemverwaltungs- und Anwendungsdaten spezielle eigene Dateien zu schreiben, erst nach erfolgreichem Abschluß dieser Operation werden die eigentlichen Daten geschrieben.

Für kontinuierliche Daten wird sowohl der Daten- als auch der Metadatenbereich als Ringpuffer organisiert, die Anzahl der Puffer soll mindestens so groß sein, daß die Anwendung die Daten rechtzeitig vor Verfall, z. B. *timeout*, lesen kann. Der so aktivierte Bereich wird zum Füllen in den Adreßraum des Dateisystems (Produzent) und zum Lesen der Anwendung (Verbraucher) eingeblendet. Bei einer Zeitsynchronisation erhält die Anwendung (Verbraucher) nach einer Zeit t_p (Vorlauf) den geschriebenen Puffer zum Lesen für eine bestimmte Zeit t_c. Nach Ablauf von t_c wird der Inhalt ungültig und kann vom Dateisystem überschrieben werden. In Abbildung 4 ist dieses Vorgehen dargestellt.

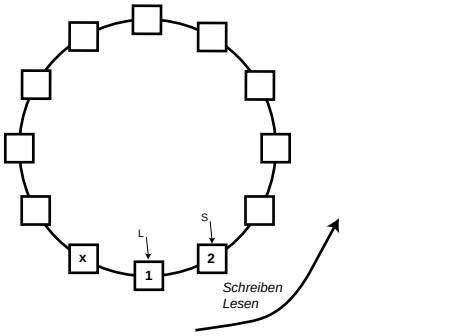

Abbildung 4: Ringpufferorganisation

Der Cache soll dabei zwei unterschiedliche Verbindungsarten unterstützen: synchron[3] und asynchron[4]. Bei der ersten Art sind die jeweiligen Pufferinhalte dem Cache während der Übertragung

[3] synchrone Übertragung: unter Umgehung des Caches

[4] asynchrone Übertragung: der Cache generiert die Aufträge an das Dateisystem

unbekannt, d. h. sie stehen der Anwendung exklusiv zur Verfügung und es können Mechanismen zum Schützen der Puffer und Aktualisieren der Verwaltungsinformationen im Cache entfallen. Der Vorteil der direkten Übertragung unter Umgehung des Caches ist die Einsparung zusätzlicher Kommunikation und Verwaltung. Da jedoch auch Nichtechtzeitzugriffe auf die Echtzeitdateien gestattet werden sollen, müssen die zum Strom gehörigen Pufferinhalte während der Übertragung vor Modifikationen geschützt bzw. bei Änderungen durch das Strominterface im Nichtechtzeitteil des Caches als ungültig markiert werden, da sonst Inkonsistenzen auftreten. Damit diese Prüfung wegfallen kann, darf die Datei während einer synchronen Echtzeit-Übertragung von anderen Anwendungen lediglich gelesen werden. Bei der asynchronen Übertragung werden die Puffer durch den Cache im voraus gefüllt. Das heißt, der Cache generiert die Aufträge an das Dateisystem und die Puffer können bis zur Wiederverwendung des Puffers anderen Anwendungen zur Verfügung stehen. Bei dieser Form der Datenübertragung ist ein feingranularerer Zugriffsschutz auf Dateiblockebene möglich als bei der Übertragung ohne Mitwirkung des Caches. So kann Caching auch für Echtzeitzugriffe unterstützt werden: für beide Anwendungen wird je ein hinreichend großer Puffer angefordert und diese werden intern als ein Bereich verwaltet.

Im folgenden werden die zu entwickelnden bzw. anzupassenden Schnittstellen und Komponenten beschrieben:

1. Das Blockinterface ist eine neue Schnittstelle im System, die eine Abstraktion der Hardware (SCSI, IDE u. ä.) ermöglicht. Über sie werden die Hardware-Ressourcen verwaltet und der Bedarf beim Verbindungsaufbau mit den registrierten Dateisystemen ausgehandelt.
2. Der Hardware-Treiber muß seinen Ressourcenbedarf und deren Verbrauch selbst kalkulieren. Die verfügbaren Ressourcen, z. B. die Bandbreite, werden als Parameter durch das Blockinterface übergeben. Diese Parameter stehen über eine Ressourcenschnittstelle zur Verfügung.
3. Die Stromschnittstelle soll eine systemweite Verarbeitung der Daten und deren einheitliche Beschreibung unterstützen.
4. Der Speichermanager verwaltet den verfügbaren Speicherbereich und stellt diesen anderen Anwendungen nach Bedarf zur Verfügung.
5. Das Echtzeitdateisystem muß an die neuen Schnittstellen angepaßt und die Verwendung des Blockcaches für Metadaten integriert werden.
6. Das logische Dateisystem verwaltet die Speicherbereiche (Puffer) für den Datenaustausch zwischen Hardware, Dateisystem und Anwendung. Die Puffergröße wird hier aus den Strombeschreibungsparametern berechnet. Der benötigte Speicher soll vom Speichermanager bezogen und an den Blockcache weitergegeben werden.
7. Die Blockcachebibliothek stellt die Funktionalität eines Caches auf Blockebene für die Metadaten dem Dateisystem zur Verfügung. Der benötigte Speicher kann wahlweise vom logischen Dateisystem oder dem Speichermanager angefordert werden.

8. Das Dateisystem muß einen geeigneten Algorithmus verwenden, um Aufträge für Meta- und Nutzdaten zu verwalten und sinnvoll zu synchronisieren. Zur Zeit basiert die Implementierung des RTFS auf dateisystemeigenen Puffern, es muß also die gesamte Pufferverwaltung ausgegliedert bzw. angepaßt werden.

3.3.2 Wichtung der Datenarten

Beim Rückfordern von Speicherseiten vom Dateicache durch den Speichermanager werden zuerst die ältesten ungenutzten Daten freigegeben. Dabei werden auch Daten, die im voraus gelesen wurden, entfernt. Allerdings muß unterschieden werden, ob die Daten aufgrund von Informationen der Anwendung oder durch erstellte Regeln gelesen wurden. Letztere erhalten ein Gütekriterium, z. B. die Trefferquote bereits verwendeter Daten. Für den Fall, daß eine vollständige Freigabe der zeitunkritischen Bereiche nicht ausreicht, werden danach – wenn möglich – die Pufferbereiche der Ströme unter Beachtung der QoS-Parameter und der jeweiligen Prioritäten verringert. Speicher bereits zugesicherter Ströme kann nicht entzogen werden, er wird jedoch nicht für neue Anfragen verwendet. Als letztes werden in obiger Reihenfolge cache-eigene Metadatenbereiche und – falls Blockcaches ihren Speicher vom Dateicache beziehen – Blockcachebereiche verkleinert und freigegeben.

Beim Schreiben von veränderten Daten wird in umgekehrter Reihenfolge gearbeitet, d. h. Metadaten werden so bald wie möglich nach dem erfolgreichen Abschluß des Schreibens der Logdateien geschrieben.

3.3.3 Bestimmung der Pufferanzahl

Die Puffergröße kann auf unterschiedliche Weise festgelegt werden. Am einfachsten ist es, einen festen – z. B. durch Messungen empirisch bestimmten – Wert vorzugeben. Diese Herangehensweise ist jedoch im Hinblick auf Portabilität und Verarbeitung von Strömen nicht praktikabel. Für die Blockcachebibliothek und die Daten im Dateicache, auf die nicht über die Stromschnittstelle zugegriffen wird, sollen die Anzahl und Größe variabel sein, d. h. eine Verdrängung findet erst statt, wenn Seiten zurückgefordert werden oder kein neuer Speicher mehr zur Verfügung steht. Für kontinuierliche Daten im Dateicache wird die Größe des Pufferbereiches beim Initialisieren einer Echtzeitverbindung festgelegt, auch hier sind parametrisierte Daten – z. B. die Datenart – sinnvoll.

Die Berechnung der Größe des Pufferbereiches beruht auf dem Modell für schwankungsbeschränkte Ströme. Dieses Modell geht von zwei periodischen Prozessen mit gleichen durchschnittlichen Raten $R = r_a = r_b$ aus: der Erzeuger A füllt ab einem Zeitpunkt $t_a = 0$ mit Periode T_A den Puffer P_{min} mit einer Menge von Datenpaketen konstanter Größe Q. Zum Zeitpunkt $t_b = 0$ beginnt der Verbraucher B mit der Periode T_B n Datenpakete mit variabler Größe D_i zu entnehmen. Am Ende jeder Periode

sind die angeforderten Daten $D = \sum_{i=1}^{n} > 0$ verbraucht, d. h. der Pufferinhalt wird ungültig. Dabei variieren die Ankunftszeitpunkte t_{a_i} der Daten, sie können zu früh (τ) oder zu spät (τ') eintreffen. Auf die vollständige Darstellung soll hier verzichtet werden, für den Cache ergibt sich die minimale Puffergröße $P_{min} = \lceil R(\tau + \tau') + Q \rceil$ (Berechnung siehe [Ham98, HR99]).

3.3.4 Zulassung neuer Aufträge *(Admission Control)*

Die Zulassung neuer Aufträge wird in der derzeitigen Implementierung im Dateisystem anhand SCSI-Spezifika durchgeführt. Wie in Abschnitt 2.1 erläutert, bringt ein von Hardware und Dateisystem unabhängiger Entwurf verschiedene Vorteile mit sich. Um eine vollständige Unabhängigkeit zu gewährleisten, sind jedoch einige Änderungen an den existierenden Teilen notwendig. Alle beteiligten Module müssen ihren spezifischen Ressourcenverbrauch selbst kalkulieren und die verfügbaren Eigenschaften bekanntgeben können. Ein gangbarer Weg ist, daß jede Komponente eine eigene Admission Control durchführt und alle eingehenden Aufträge selbst verwaltet und diese auch bei der jeweiligen Komponente anfordert. Der Nachteil dieser Herangehensweise ist, daß jede Komponente sehr ähnliche Berechnungen ausführen und Zugriffsmechanismen zur Verfügung stellen muß. Dadurch kann sich der Verbindungsaufbau verzögern. Ein weiterer Nachteil besteht im sequentiellen Reservieren von Ressourcen durch jede Komponente einzeln, dadurch kann das Fehlschlagen erst später registriert werden.

Eine andere Herangehensweise ist die Berechnung der freien Bandbreite in einer zentralen Komponente. Bei dieser registrieren sich alle Server[5] mit ihren verfügbaren Ressourcen und spezifischen Ressourcenverbrauch und die Clients[6] mit den gewünschten Eigenschaften. Die Admission Control entscheidet anhand der registrierten Ressourcen und Ströme über die Zulassung neuer Aufträge. Der Nachteil liegt in der höheren Ungenauigkeit bei der Reservierung von Ressourcen, es kann jedoch – im Falle mehrerer möglicher Wege zur Befriedigung der Anforderung – der günstigste Weg von der Admission Control ausgewählt und für die Anwendung transparent registriert werden. Die Anwendung erhält im Erfolgsfall eine Strom- und eine Komponentenidentifikation, damit kann sie eine Verbindung aufbauen.

Der Aufbau einer solchen zentralen Verwaltungskomponente stellt jedoch eine sehr komplexe Aufgabe dar, die mit einem geeignetem Verfahren zur Bildung von Namensräumen verbunden werden muß. Jede Komponente registriert eine eindeutige Identifikation und systemweite Dienstnamen, die sie anbietet bzw. benötigt. Jede Anwendung braucht dafür eine Ressourcenschnittstelle, mit der sie Auskunft über die zur Verfügung stehenden Ressourcen und den Verbrauch fremder Ressourcen geben kann. Diese Schnittstelle ist auch beim derzeitigen Prototyp für die Stromschnittstelle (*DROPS Streaming Interface* [RLG00]) für das Aushandeln von Verbindungen notwendig. Weiterhin ist bei

[5] Server: Komponente, die Dienste anbietet

[6] Client: Komponente, die einen Dienst verwenden möchte

der zentralen Lösung eine eindeutige Reservierung von Ressourcen anhand von Strom-IDs notwendig. Beteiligte Anwendungen beziehen ihre Ressourcen dann ausschließlich über diese.

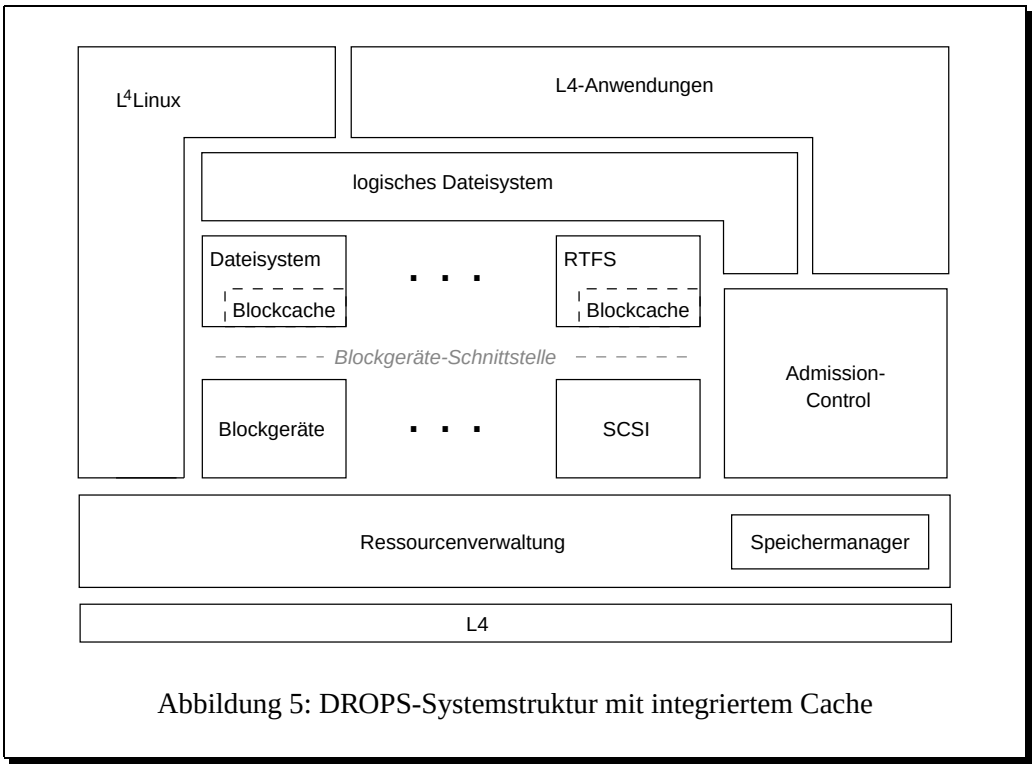

Abbildung 5: DROPS-Systemstruktur mit integriertem Cache

In Abbildung 5 ist die Systemstruktur dargestellt. Im folgenden sollen die Schnittstellen zwischen den einzelnen Komponenten definiert bzw. deren Funktionalität umrissen werden.

3.3.5 Schnittstellen

Datentypen Neben den Datentypen der Stromschnittstelle für DROPS ([Reu99, Lös99]) sind weitere Daten zur Verwaltung der Kommunikation zwischen den unterschiedlichen Komponenten notwendig. Diese sollen im folgenden mit ihren zugehörigen Schnittstellen definiert werden.

Blockgeräteschnittstelle Diese Schnittstelle soll die Kommunikation zwischen den Blockgeräten und den Dateisystemen gewährleisten und gleichzeitig eine Abstraktion der zugrundeliegenden Hardware ermöglichen.

Denkbare Zugriffe auf dieser Ebene sind blockorientierte Echtzeit- und Nichtechtzeitanforderungen. Die derzeitige Implementierung von Dateisystem und SCSI-Treiber realisiert die Auftragszulassung und -generierung (*Admission Control*) auf Dateisystemebene ([Meh98] und [Reu98]). In Hinblick auf die Unterstützung unterschiedlicher Hardware und mehrerer – verschiedener – Dateisysteme ist

eine Verlagerung dieser Aufgabe notwendig. Dafür gibt es zwei Möglichkeiten: in die Ebene des Hardwaretreibers (Blockschnittstelle bzw. Hardwareschicht) oder, wie in dieser Arbeit beschrieben, in eine eigene Komponente. Beide Wege erfordern eine Erweiterung um eine Ressourcenschnittstelle. Dadurch wird eine transparente Einbindung unterschiedlicher Hardware und/oder Emulationen möglich.

Allgemeine Beschreibung der Schnittstelle:

- Initialisieren der Verbindung (z. B. per Strom-ID)
- Registrieren eines Gerätes
- Anfordern der physischen Geräte IDs
- Verbindungsaufbau (Reservieren von Ressourcen über Admission Control)
- Reservieren von Blöcken (Preallokation)
- Lesen/Schreiben von Blöcken

Folgende Parameter sind für die Schnittstelle Blockgerät-Dateisystem notwendig:

- Allgemein:
  - Gerät (ID oder Name des Treibers)
  - Anfangsblocknummer (Blockliste)
  - Anzahl Blöcke
  - Adresse der übergebenen Puffer
  - Priorität
- Echtzeit:
  - Anforderung:
    - absoluter Startzeitpunkt
    - Dauer
    - benötigte Ressourcen (Speicher, Bandbreite, ...)
  - Timeout

Speichermanager Der Speichermanager ist nicht Bestandteil dieser Arbeit, es soll jedoch im folgenden die von logischem Dateisystem und Blockcache verwendete Funktionalität definiert werden. Der Speichermanager arbeitet mit einer minimalen Granularität von einer Speicherseite, d. h. es können Vielfache davon angefordert, freigegeben oder entzogen werden.
Der Speichermanagers soll die folgenden Funktionen anbieten:

- An-/Nachfordern von Seiten
- An-/Nachfordern von Seiten, die nicht ausgelagert werden (*pinning*)
- Einblenden von Seiten in den Adreßraum des Clients
- Freigabe von Seiten
- Freigabe von Seiten (nach Anfrage durch den Speichermanager)

Diese Funktionalität wird vom logischen Dateisystem genutzt und zusätzlich zur fakultativen Nutzung transparent an den Blockcache weitergeleitet. Der Vorteil dieser Herangehensweise liegt bei der Behandlung der Rückforderung von Speicherseiten. Da nur die Verbindung zwischen logischem Dateisystem und Speichermanager existiert, kann das logische Dateisystem entscheiden, den per Definition Metadaten enthaltenden – also kleineren – Blockcache vor zu frühem Entzug von Speicher zu kapseln. Durch die 1 zu 1 Verwendung der Schnittstelle ist eine problemlose Rücknahme dieser Entscheidung möglich.

Bei der Anforderung von Speicher wird der Bereich für Verwaltungsinformationen, also Namensauflösung und Dateisystemmetadaten, gepinnt angefordert, um garantierte Antwortzeiten für Standardaktionen zu erhalten. Weiterhin werden diese Informationen oft benötigt, so daß ein Mehraufwand durch häufiges Ein- und Auslagern dieser Daten entstehen würde. Bei Echtzeitdaten kann das Auslagern erlaubt werden. Allerdings wird für die Entscheidung, ob dies gestattet werden kann, eine genaue Abschätzung der Zeit, die zum Einlagern einer Seite verbraucht wird, benötigt.

Schnittstelle Cache – physisches Dateisystem Wie bereits in Abschnitt 3.3.1 beschrieben, sieht der Entwurf des Caches eine Trennung in zwei Teile vor: den dateisystemeigenen Blockcache zum einen und den Dateicache in der logischen Dateisystemschicht zum anderen.

Schnittstelle Blockcache – Dateisystem Der Blockcache wird als Bibliothek fest zum Dateisystem hinzugebunden. Diese Bibliothek kann – wie bereits beschrieben – ihren Speicher wahlweise direkt vom Speichermanager anfordern oder über das logische Dateisystem beziehen. Blockcache und Dateisystem verwenden eine gemeinsame, vom Dateisystem verwaltete Blockgeräteschnittstelle. Die Bibliothek bietet Funktionen zum Lesen und Schreiben von Blöcken an und stellt leere Puffer zum Schreiben neuer Blöcke bereit. Weiterhin stehen unterschiedliche Funktionen zur Behandlung von Echtzeit- und Nichtechtzeitanforderungen zur Verfügung.

Schnittstelle logisches – physisches Dateisystem Diese Schnittstelle läßt sich nicht transparent in das System einbinden, da Änderungen der Pufferverwaltung und der Schnittstellen des Dateisystems

notwendig sind. Die POSIX-kompatible Schnittstelle zwischen logischem und physischem Dateisystem unterscheidet sich von der Standardschnittstelle zur Anwendung durch die Übergabe des Pufferbereiches. Die Transparenz läßt sich für Nichtechtzeitzugriffe nur so realisieren, daß Anwendungen, die direkt auf das Dateisystem zugreifen, dann ohne Änderungen über den Dateicache zugreifen können. Anwendungen, die eigenen Speicher verwenden, können auch im Nichtechtzeitfall transparent zugreifen. Die Stromschnittstelle soll transparent eingefügt werden.

Schnittstelle logisches Dateisystem – Anwendung Die Schnittstelle, die das logische Dateisystem der Anwendung anbietet, untergliedert sich in drei Teile (Abbildung 6):

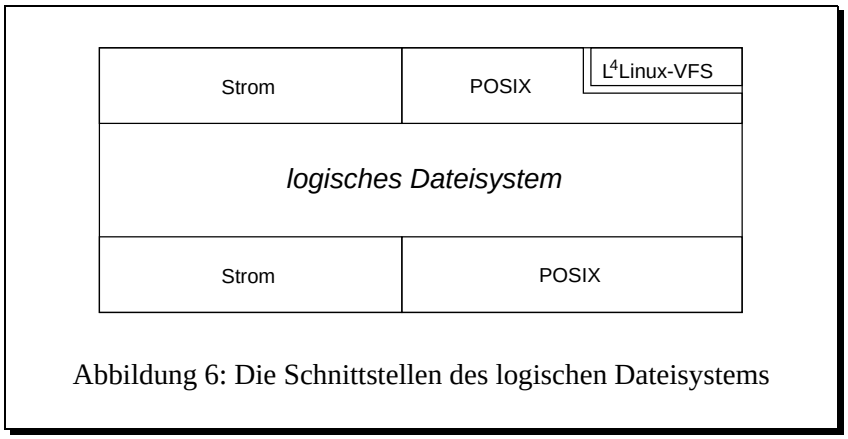

Abbildung 6: Die Schnittstellen des logischen Dateisystems

1. Die Stromschnittstelle:

   (a) dient dem Anfordern von Pufferbereichen für Echtzeitverbindungen. Dafür werden beim Verbindungsaufbau der Dateiname, die minimale und maximale Blockanzahl oder entsprechende Parameter und die geforderte Bandbreite übergeben. Die Anwendung kann auch eine Speicherbeschreibungsstruktur übergeben, um eigenen Speicher (ungepuffert) zu verwenden.

   (b) gestattet die Bestätigung oder Abweisung von Verbindungen.

   (c) ermöglicht die explizite Synchronisation einer Verbindung (start, stop, continue).

   (d) gewährleistet die (vorzeitige) Freigabe des Pufferbereichs.

2. POSIX-Schnittstelle für L4-Anwendungen [POS96] zum nichtzeitkritischen Zugriff auf Dateien.

3. Der VFS-Stub zur Anbindung an L^{4}Linux bildet die Funktionen der POSIX-Schnittstelle auf die VFS-Schnittstelle von L^{4}Linux ab.

L4-Echtzeitanwendungen Die benötigte Puffergröße kann von der Anwendung über eine Parameterliste bereitgestellt und bei der Admission Control registriert werden. Eine andere Vorgehensweise sieht ein Festlegen der Puffergröße durch die Admisson Control vor. Dabei werden die QoS-Parameter des Stromes durch die Anwendung beim Verbindungsaufbau übermittelt. Daraus werden die benötigten Puffergrößen P_{min} berechnet und verbindungsorientiert bereitgestellt. Die Berechnung basiert auf dem Modell für schwankungsbeschränkte Ströme (vgl. Abschnitt 3.3.3).

Der Dateicache fordert für jeden neuen Strom einen neuen Speicherbereich (Menge von Puffern einer Größe) vom Speichermanager an, bekommt er diesen, so wird die Anfrage angenommen. Im Falle des Abweisens einer Anfrage durch den Speichermanager muß der Cache bereits empfangenen Speicher (vorhandene Ressourcen) untersuchen und neu konfigurieren (siehe auch Abschnitt 3.3.2 auf Seite 22). Dabei werden zuerst Puffer der anfragenden Anwendung verwendet. Sollte kein Speicher reserviert werden können, muß die Verbindung abgewiesen werden.

Bei der asynchronen Übertragung kann die Anwendung zusätzlich ihr Zugriffsschema übergeben, um ein Prefetch durch den Cache zu unterstützen. Dieses Vorgehen ist ein Informieren durch die Anwendung (*Informed Prefetch* [Pat97]).

L4-Nichtechtzeitanwendungen Diese Anwendungen können über die POSIX Schnittstelle sowohl eigenen Speicher als auch Speicher des logischen Dateisystemes verwenden. Für nichtkontinuierliche und nichtzeitkritische Anwendungen wird das Zugriffsverhalten auf eine Datei protokolliert und aus den gewonnenen Informationen kann das Lesen im voraus variiert werden. Derartige Regeln können bereits nach der dritten Anforderung leicht aufgestellt werden (*Adaptive Prefetching* [MRSW97]).

L^{4}Linux Die Anbindung von Dateisystemen an L^{4}Linux erfolgt für jedes Dateisystem über einen eigenen Stub an das Linux-VFS. Mit Einführung des logischen Dateisystems, das eine dem Linux-VFS ähnliche Schicht in DROPS anbieten soll, ist diese Herangehensweise nicht mehr sinnvoll, da es dadurch zu doppeltem Caching kommen kann. Die Änderung der Daten im Dateisystem wäre auf unterschiedlichen Wegen möglich: über den Dateicache und durch Anwendungen im L^{4}Linux. Dies würde unweigerlich zu Inkonsistenzen und damit zu Datenverlusten führen, da gepufferte Daten direkt – unter Umgehung des Caches – im Dateisystem geändert werden könnten. Um dies zu verhindern, wären umfangreiche Synchronisationsmechanismen zwischen Dateicache und Linux-VFS notwendig, was wiederum nur über eine komplexe neue Schnittstelle im VFS realisierbar ist. Des weiteren wäre der Vorteil eines Caches für die L4-Dateisysteme durch L^{4}Linux-Anwendungen nicht nutzbar. Eine weitere Möglichkeit ist es, den Dateicache als Dateisystem in L^{4}Linux anzubieten, das hat den Vorteil, daß alle vom Dateicache erreichbaren Dateisysteme automatisch im VFS erreichbar wären. Der Nachteil dieser Lösung liegt im doppelten Caching der Verzeichniseinträge. Dieser ist jedoch nicht so gravierend, da die Gültigkeit vor dem Schreiben überprüft werden kann und die Verzeichnispuffergröße im VFS relativ klein ist, so daß alte Einträge schnell entfernt werden.

Der VFS Stub wird dabei über eine Bibliothek auf die POSIX-Schnittstelle des Dateicaches abgebildet, d. h. es findet keine Trennung zwischen L^{4}Linux- und L4 Anwendungen statt. Es werden alle Nichtechtzeitanforderungen aus einem Pufferpool beantwortet.

3.4 Zusammenfassung

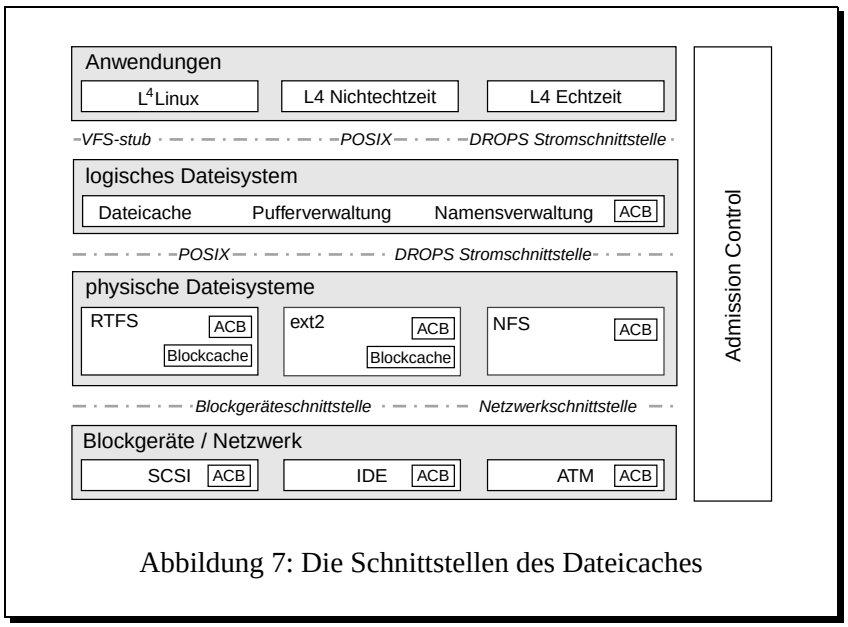

Abbildung 7: Die Schnittstellen des Dateicaches

In Abbildung 7 ist das Gesamtsystem mit integrierten Caches dargestellt. Die Darstellung entspricht dem Entwurf mit zentraler Auftragszulassung (*Admission Control*). Dafür besitzt jede Komponente eine entsprechende Schnittstelle zum Anfordern der Ressourceninformationen (*Admission Control Block, ACB*). Der Blockcache wird als Bibliothek in das Echtzeitdateisystem integriert. Er soll zur Pufferung der Verwaltungsdaten von Echtzeit- und Nichtechtzeitdateien dienen.

Das logische Dateisystem bietet je eine Schnittstelle für Echtzeit- und Nichtechtzeitanforderungen an. Die Anbindung von L^{4}Linux erfolgt über die Nichtechtzeitschnittstelle (POSIX kompatibel). Für Echtzeitanforderungen kann das logische Dateisystem als aktiver Bestandteil der Übertragung oder als passiver Puffermanager benutzt werden. Das logische Dateisystem soll die physischen Dateisysteme nutzen, um Transaktionsmechanismen zu integrieren und dadurch Konsistenzgarantien abzugeben.

Die Größe der Pufferbereiche und die Anzahl soll dynamisch zur Laufzeit festgelegt und geändert werden können, d. h. der Cache fordert dynamisch Speicher an und kann Ihn auch wieder freigeben. Die Entscheidung über das Verwerfen von Blöcken soll über ein modifiziertes *least recently used* erfolgen. Dabei wird die Referenzierungshäufigkeit beachtet.

4 Implementierung

Als Grundlage der Implementierung dient das von Lars Reuther entwickelte Echtzeitdateisystem [Reu98]. Die Implementierung wurde wie in Abschnitt 3.3.1 beschrieben in zwei Teilen ausgeführt. Der erste Teil umfaßt die Blockcachebibliothek, die zum Dateisystem hinzugebunden werden muß. Dieser Teil erforderte auch Änderungen am Echtzeitdateisystem, die im Rahmen der Tests der Bibliothek mit ausgeführt wurden. Der zweite Teil umfaßt das logische Dateisystem mit Pufferverwaltung, Dateicache und Namensdiensten.

Aufgrund der schnelleren Integrationsmöglichkeit in das Echtzeitdateisystem wurde mit dem Blockcache begonnen. Dadurch konnte gewährleistet werden, das dieser Teil im Rahmen der Arbeit getestet werden konnte. Das logische Dateisystem erfordert tiefgreifendere Änderungen im Dateisystem, die im Rahmen eines Redesigns vorgenommen werden sollten. Dadurch wurden nur Teile implementiert.

Die Implementierung basiert auf den in Kapitel 3 beschriebenen Funktionalitäten der Schnittstellen für Blockgeräte und Speicherverwaltung.

4.1 Blockgeräteschnittstelle

```
typedef struct dev_info {
dword_t id;                /* registrierte id/Name des Gerätes */
dword_t period;            /* Periodenlänge */
dword_t slots;             /* Anzahl Slots/Periode */
dword_t slot_len;          /* Slotlänge */
} id_dev_info_t;

typedef struct data_dsc {
word_t map_address;        /* Adresse des Schreib-/Lesepuffers */
block_t block;             /* zu lesender Block: */
                           /* Partition, Blocknummer, Länge */
} data_dsc_t;

typedef struct req_dsc {
dword_t req_id;            /* Strom id */
dword_t req_prio;          /* Priorität */
dword_t req_len;           /* Länge */
dword_t valid              /* absolute Startzeit */
dword_t deadline;          /* spätester Zeitpunkt */
data_dsc_t *data;          /* Beschreibung der zu lesenden Daten */
} req_dsc_t;
```

Abbildung 8: Die Datenstrukturen der Blockgeräteschnittstelle

Diese Schnittstelle ist – wie bereits beschrieben – als einheitliche Kommunikationsebene zwischen Dateisystemen und Blockgeräten gedacht. Sie basiert auf der SCSI-Schnittstelle des Echtzeitdateisystems. Die in den Abbildungen 8 und 9 gezeigten Funktionen und Datenstrukturen werden in der Implementierung des Caches auf die entsprechenden Funktionen der SCSI-Schnittstelle (`scsi_com` im RTFS) abgebildet.

```
dev_init();
dev_eject(devID);
dev_reset(devID);
dev_abort(devID);
dev_flush(devID);
dev_req_timeout(devID, cmd);                        /* restart, abort, ... */

getDevice_ids (ids[]);
getDevice_info(devID, dev_info);

init_request(devID, req_dsc);                       /* open */
sync_request(devID, req_dsc, signal);               /* start, contiune, stop */

allocate_blocks(devID, blockID, count, prio);
read_blocks(devID, blockID, count, buffer, prio);
write_blocks(devID, blockID, count, buffer, prio);
release_blocks(devID, blockID, count);
```

Abbildung 9: Die Blockgeräteschnittstelle

4.2 Puffercachebibliothek

Die Bibliothek bietet Cachefunktionalität auf Blockebene als Bestandteil des Dateisystemes an. Sie verwendet Funktionen des DROPS-Puffermanagers zum Anfordern und Einblenden von Speicherseiten. Diese Funktionen werden analog zur Blockgeräteschnittstelle (vgl. Abbildung 9) per *inline*-Funktionen verwendet. Entwurf und Implementierung der Schnittstellen wurden so allgemein wie möglich gehalten, um eine spätere Adaption an die DROPS-Geräteschnittstelle bzw. den Speichermanager zu ermöglichen.

4.2.1 Datenstrukturen

Die Datenverwaltung im Blockcache erfolgt unter Verwendung von drei Listen, je eine Liste für die freien, die verwendeten und die modifizierten Puffer. Die Liste für die modifizierten Puffer wurde eingeführt, um die Synchronisationsfunktion zu beschleunigen. Ohne diese Liste wäre bei jedem Aufruf der Funktion eine lineare Suche nach modifizierten Puffern durch die gesamte Liste notwendig. Die Elemente in den Listen für belegte Puffer sind doppelt verkettet, um ein schnelles Umketten innerhalb

einer Liste bzw. zwischen den Listen zu realisieren. Die Verkettung erfolgt strikt nach *least recently used*, das heißt das zuletzt genutzte Element steht vorn in der Liste.

Name	Typ	Beschreibung
def_prio	Integer	
blk_size	DWord	Puffergröße
max	DWord	max. Anzahl von Puffern
reserved	DWord	Anzahl von Puffern, die mit einmal angefordert oder freigegeben werden
count	DWord	Anzahl der aktuell im Cache vorhandenen Puffer
free	DWord	freie Puffer
lock	Semaphor	Variable um atomare Zugriffe zu koordinieren
wait	Zeiger auf Warteschlange	
hash	Zeiger auf Hashtabelle	
head	Zeiger auf den Kopf der Liste für verwendete Puffer	
tail	Zeiger auf das Ende der Liste für verwendete Puffer	
dirty	Zeiger auf modifizerte Puffer	
freelist	Zeiger auf freie Pufferliste	

Abbildung 10: Die Verwaltungsstruktur des Blockcaches

In Abbildung 10 ist die globale Verwaltungsstruktur einer Instanz des Blockcaches dargestellt. Der Zeiger auf das Ende der verwendeten Liste ist für die einfachere Implementierung anderer Ersetzungsstrategien interessant und wird in der vorliegenden Version von der Prefetch-Funktion genutzt, da im voraus gelesene Elemente derzeit an das Ende der Liste mit verwendeten Puffern (`used`-Liste) angehängt. Das bedeutet diese Elemente werden nach *most recently used* ersetzt.

Die Suche findet mittels einer Hashtabelle und einem nichtoffenen Adressierungsverfahren statt (*Bucket Search*). Dadurch sind besser vorhersagbare Antwortzeiten möglich als durch Listenverkettung oder einem offenem Verfahren (vgl. 2.1.3). Falls mehrere Daten auf eine Adresse fallen, wird das neueste Element immer als erstes Element verkettet. In Abbildung 11 ist der Aufbau der Verwaltungsstruktur des Blockcaches von der Hashtabelle ausgehend dargestellt. Die Hashfunktion operiert auf dem 32 Bit Wert für die Block-ID (Partition und Blocknummer), sie kann wahlweise direkt über die BlockID zugreifen oder einen Schlüssel daraus berechnen. Die Untersuchung der Anzahl der Kollisionen bei einer Umrechnung der Block-ID im Vergleich mit der Verwendung der Block-ID als Hashschlüssel zeigte in der Testumgebung (eine 500 MByte Partition, vgl., Abschnitt 5.1), daß die berechnende Funktion optimaler ist, d. h. es wird bei gleicher Tabellengröße eine annähernde Gleichverteilung im Schlüsselraum erzielt. Jeder Tabelleneintrag enthält einen Zeiger auf den Cache-Eintrag und ein Zeigerpaar auf evtl. Vorgänger und Nachfolger. Optional kann zum Zeitpunkt der Übersetzung das Führen einer Statistik über die Anzahl der Kollisionen und die Ausnutzung der Tabelle festgelegt werden.

Die Hashverwaltungsfunktionen erhalten einen typlosen Zeiger auf den gesuchten Wert (im vorliegenden Fall also die BlockID) und die Länge des Schlüssels als Parameter, sie sind somit nicht auf

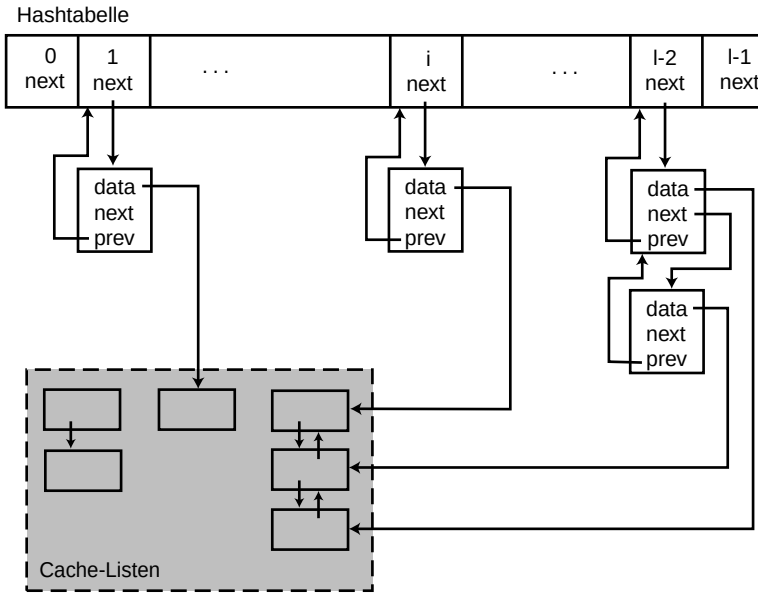

Abbildung 11: Aufbau der Hashtabelle

einen Datentyp oder eine Länge fixiert. Im Falle größerer Blöcke muß nur die Hashfunktion selbst angepaßt werden.

Die Entscheidung über das Verwerfen von belegten Puffern wird mittels einer abgewandelten Form von LRU getroffen. Dabei wird der im Zeitfenster (`LFU_FRAME`) am seltensten referenzierte Puffer entfernt. Die Größe des Zeitfensters läßt sich zum Zeitpunkt der Übersetzung festlegen (der Wert 1 realisiert striktes LRU). Sollten alle Elemente im Fenster gesperrt sein, d. h. es sind alle Elemente in Verwendung, wird ein Fehler zurückgegeben.

Es werden zwei verschiedene Funktionen zum Initialisieren des Blockcaches angeboten, die erste, `init_cache`, verwendet Standardwerte und ruft mit diesen Werten `init_cache_no` auf. Diese Funktion läßt eine direkte Einstellung der Werte für die Pufferanzahl zu. Dabei kann die max. Größe des Caches eingeschränkt und die Anzahl der Puffer, die gleichzeitig vom Speichermanager angefordert werden sollen, festgelegt werden. Mehrfache Aufrufe dieser Funktionen generieren voneinander unabhängige Instanzen von Blockcaches. Dies kann zum Beispiel für verschiedene Partitionen genutzt werden, ist aber nicht Bedingung. Beim Initialisieren werden die Listenköpfe initialisiert und alle diese Cache-Instanz betreffenden Informationen in einer Tabelle abgelegt. Um parallele Zugriffe auf die Puffer zu ermöglichen, werden die kritischen Abschnitte mittels Semaphoren geschützt. Kritische Abschnitte im Blockcache:

```
int init_cache        (dword_t cache_size, int priority);
int init_cache_no     (dword_t cache_size, dword_t reserved,
                       dword_t max, int priority);
int sync_cache        (dword_t instance);
int c_read_blk        (dword_t instance, dword_t partition,
                       dword_t block, dword_t count,
                       byte_t **buffer, int priority);
int c_prefetch_blk    (dword_t instance, dword_t partition,
                       dword_t block, dword_t count, int priority);
int c_get_empty_blk   (dword_t instance, dword_t partition,
                       dword_t block, dword_t count,
                       byte_t **buffer, int priority);
int c_write_value     (dword_t instance, dword_t partition,
                       dword_t block, dword_t count,
                       byte_t *buffer, int priority);
int c_release_blk     (dword_t instance, dword_t partition,
                       dword_t block, unsigned flags);
int c_invalidate_blk  (dword_t instance, dword_t partition,
                       dword_t block, dword_t count);
void c_flushd         (dword_t instance, dword_t min, dword_t max,
                       int intervall, int priority);
```

Abbildung 12: Die Funktionen der Puffercachebibliothek

- Änderungen des Instanzzählers
- Synchronisation des Caches
- Umketten von Pufferelementen (inkl. Freigabe oder Einketten neuer Puffer)

Alle weiteren Funktionen erwarten die Instanz des Caches (`instance`) als Parameter. Der referenzierte Block (`partition,` block und Folgeblöcke `block+count`) werden in dieser Instanz bearbeitet (gesucht, eingefügt oder entfernt) werden soll. Dabei kann die Priorität für jeden Aufruf neu festgelegt werden (`priority`= 0 verwendet die Standardpriorität).

Die Bibliothek bietet dabei die in Abbildung 12 dargestellten Methoden für beliebige Dateisystemimplementationen an. Der Entwurf geht von einer ausschließlichen Verwendung des Blockcaches für Metadaten aus, die Klassifikation erfolgt durch die Einbindung in das jeweilige System. Für Anwendungsdaten ist der Dateicache vorgesehen. Der Blockcache verwendet die gleiche Geräteschnittstelle wie sein Dateisystem, d. h. alle Aufträge werden vom Dateisystem koordiniert und generiert. Das macht eine einfachere Verwaltung der dem Dateisystem zur Verfügung stehenden Ressourcen möglich. Entscheidungen über Prefetch, z. B. beim Lesen von Blocklisten, werden ebenfalls durch das jeweilige Dateisystem getroffen, indem Blöcke niederprior gelesen werden. Zu diesem Zweck steht bei der Verwendung die Funktion `c_blk_prefetch` zur Verfügung. Sie liest Elemente nur dann, wenn noch freie Puffer zur Verfügung stehen und kettet sie an das Ende der Liste ein. Um die Funktionalität auch für andere Dateisysteme optimal nutzbar zu machen, wurde auf eine an das RTFS

angepaßte Funktion in der Bibliothek verzichtet. Ein festes *read ahead* läßt sich mit den Funktionen der Bibliothek realisieren, ist aber bei der vorgesehenen Nutzung und der Anordnung der Metadaten auf der Festplatte nicht sinnvoll.

Damit modifizierte Blöcke nicht zu lange einen vom Dateisystem abweichenden Wert besitzen, bietet die Bibliothek die Möglichkeit, einen Thread zu starten, der in regelmäßigen Abständen die modifizierten Blöcke schreibt. Dafür kann die Funktion `c_flushd` verwendet werden, sie schreibt alle modifizierten Blöcke einer Cache-Instanz zurück. Diese Funktion bietet auch die Möglichkeit der Begrenzung der Pufferanzahl. Wird `max` auf null gesetzt, arbeitet der Thread als Synchronisations-*Daemon*, d. h. im Abstand von *intervall* werden die modifizierten Blöcke geschrieben, aber nicht freigegeben. Der Thread muß vom Dateisystem gestartet werden, da z. Zt. keine flexible Vergabe der Task- und Thread-Ids in L4 möglich ist und somit zum Zeitpunkt des Erstellens der Bibliothek nicht von vornherein bekannt war, von welcher Task und als welcher Thread sie verwendet werden soll.

Zur expliziten Freigabe von einzelnen Blöcken existiert die Funktion `c_invalidate_blk`, sie entfernt den Block aus dem Cache. Diese Funktion ist nur interessant, wenn ein Block gelöscht wird und kurz darauf wieder geschrieben werden soll und der Nutzer sich auf einen unbenutzten Block verläßt.

Für die Arbeit mit kontinuierlichen Daten stehen die `c_*_cont_buff`-Funktionen zur Verfügung. Eine Verbindung wird mittels `get` aufgebaut und mit `release` geschlossen, während der Übertragung stehen die Puffer für andere Anwendungen nicht zur Verfügung. Das Dateisystem legt dabei die Anzahl der Puffer fest. Während einer Übertragung werden diese Puffer als Ringliste organisiert.

4.3 Das logische Dateisystem

Aufgrund der fehlenden bzw. unvollständigen Schnittstellen und des Aufwandes für dei Anpassung des Dateisystemes wurden die benötigten Teile des Dateicaches nur in Ansätzen implementiert. Deshalb soll hier nur die geplante Struktur erläutert werden. Des weiteren sind für die Anbindung des Dateisystemes an den Dateicache grundlegende Änderungen in der Verwaltungsstruktur des Dateisystems notwendig, um eine dem Entwurf entsprechende optimale Funktionalität zu gewährleisten und unnötige Kopieroperationen zu vermeiden.

In Abbildung 13 ist der prinzipielle Aufbau des Dateicaches dargestellt. Der Datenbereich wird – wie bereits beschrieben – den Anwendungen zum Lesen bzw. Schreiben, z. B. durch *Shared Memory*, zur Verfügung gestellt.

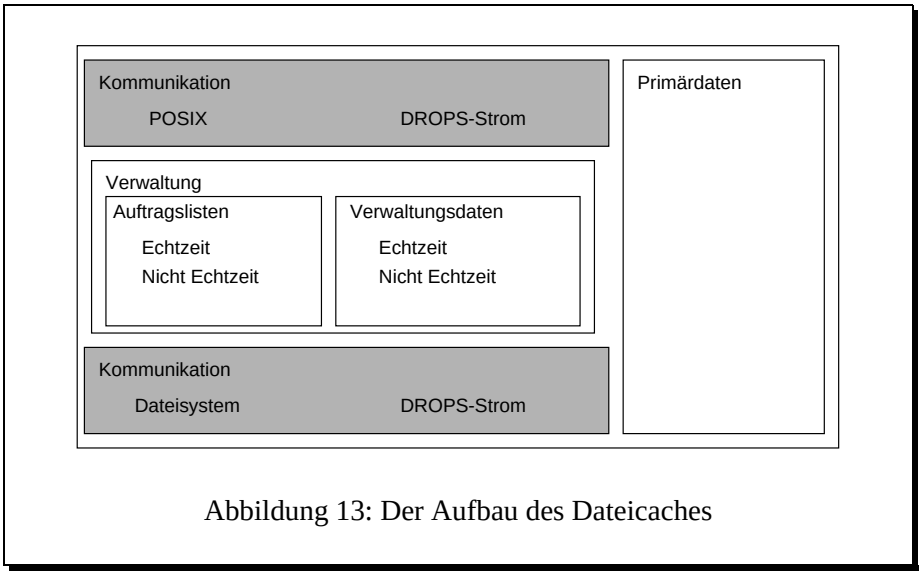

Abbildung 13: Der Aufbau des Dateicaches

4.3.1 Thread-Struktur

Für den Dateicache sind zwei verschiedene Klassen von Threads notwendig: Kommunikations- und Arbeitsthreads. Die Kommunikationsthreads sind für die Verbindungen zu den Dateisystemen bzw. den Anwendungen verantwortlich. Sie legen neue Aufträge gemäß ihrer Priorität in der Auftragsliste ab. Diese wird als Ringliste organisiert, d. h. die Synchronisation findet über je einen Schreib- bzw. Lesezeiger statt. Die Arbeitsthreads sind je ein Echtzeit- und ein Nichtechtzeitthread. Diese fordern Ressourcen an oder geben sie wieder frei und erstellen die Aufträge an das Dateisystem.

4.3.2 Datenstrukturen

Die Adressierung in der Hashtabelle ist – analog zu der des Blockcaches – mittels *Bucket Search* aufgebaut. Der Schlüssel wird aus dem String für den Dateinamen berechnet, d. h. pro Datei wird ein Eintrag der Hashtabelle belegt. Dieser zeigt auf eine weitere Liste, die im Falle kleiner Dateien Zeiger auf alle Puffer einer Datei enthält. Bei größeren Dateien enthält der Eintrag einen Zeiger auf eine Liste von Zeigern auf Dateiblocklisten *indirect blocks* (vgl. Abbildung 14). Die Hashtabelle wird dynamisch aufgebaut. Zum Schützen der Dateiblockpuffer enthält jede Pufferstruktur eine eigene Variable, die den Datei-Zeiger (`filehandle` oder `filedescriptor`) der schreibenden Anwendung enthält. Somit kann einfach garantiert werden, daß nur eine Anwendung die Puffer modifiziert. Die Anwendungsdaten werden in einem separaten Speicherbereich gehalten und wie in Kapitel 3 auf Seite 20 beschrieben, an die Anwendungen bzw. Dateisysteme freigegeben.

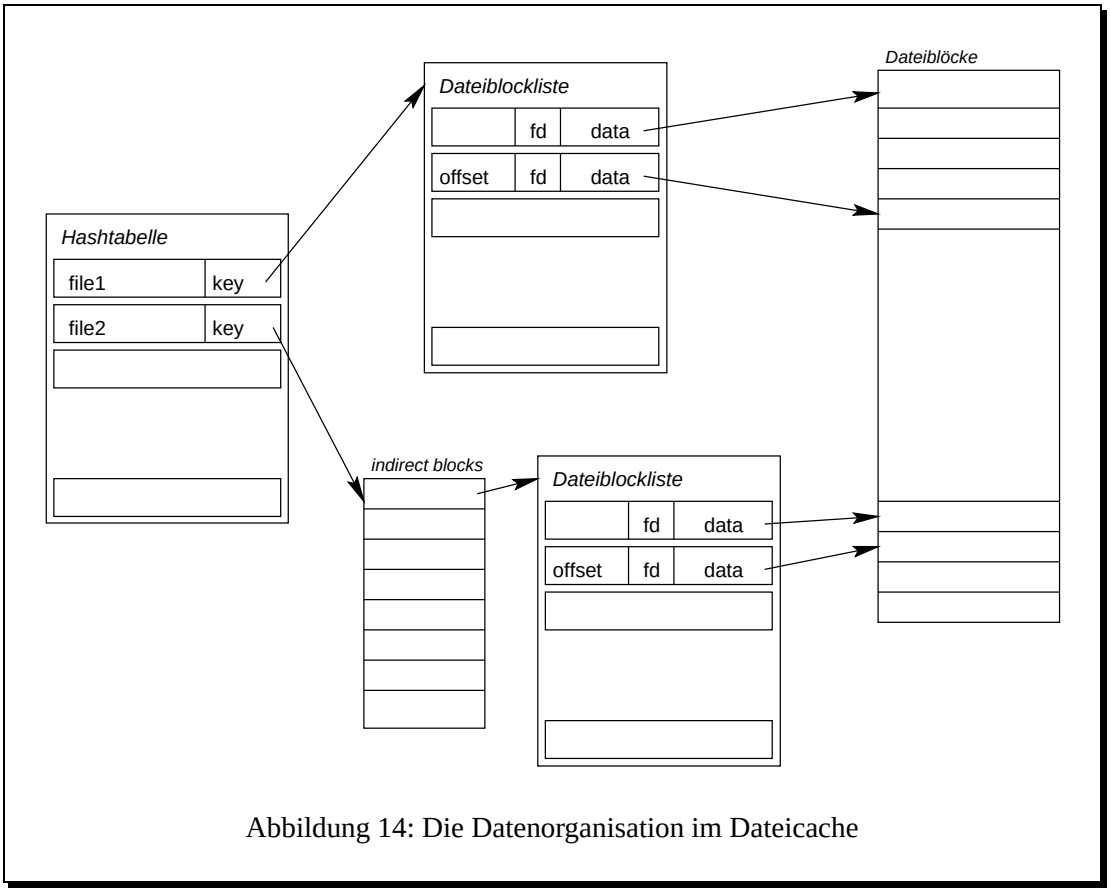

Abbildung 14: Die Datenorganisation im Dateicache

POSIX & Linux VFS Die Linux VFS-Schnittstelle basiert auf der POSIX Bibliothek. Zum Aufbau und zur Konsistenzprüfung sind jedoch zusätzliche Funktionen bzw. Funktionalitäten im L^{4}Linux-VFS Stub notwendig. Diese sind aber nur auf L^{4}Linux Seite nötig. Durch die einheitliche Schnittstelle werden alle Nichtechtzeitanfragen aus Sicht des Dateicaches garantiert gleich behandelt. Für Echtzeitanfragen ist die DROPS-Stromschnittstelle vorgesehen.

4.4 Stand der Implementierung

Die Blockcachebibliothek ist vollständig implementiert und wurde durch Integration in das Echtzeitdateisystem getestet. Zum Testen wurde ein eigenständiges Programm entwickelt, das auf der Dateisystembibliothek basiert.

Die Funktionalität des logischen Dateisystemes ist Ansätzen vorhanden, so existieren die Hashfunktionen und eine Umsetzung einer Ersetzungsstrategie (LRFU).

5 Bewertung

Die Funktionalität des Blockcaches wurde im Rahmen der Tests erfolgreich überprüft. Umfangreiche Tests des Gesamtsystems waren jedoch noch nicht möglich, da

1. die nachträgliche Integration des Blockcaches in das Dateisystem nicht optimal ist,
2. der Dateicache noch nicht vollständig implementiert werden konnte und
3. die Stabilität des Dateisystems für häufige Schreib-/Lesezyklen nicht ausreicht. So scheiterten mehrfach die Versuche, eine Vielzahl von Verzeichnissen und großen Dateien hintereinander zu schreiben.

Weiterhin läßt die derzeitige Implementation des Dateisystems keine zuverlässige Arbeit mit mehreren Partitionen und kontinuierlichen Daten zu [Reu00].

5.1 Testumgebung

Die Meßwerte wurden auf einem Standard-PC mit einem 133 MHz Intel Pentium Prozessor, 64 MByte Hauptspeicher, einem Symbios Logic Ultra Wide SCSI-Controller (NCR53c825) und einer 8,7 GByte IBM DGVS09U Festplatte mit einer 500 MByte RTFS-Partition unter L4 ermittelt. Die Partition befand sich in der äußeren Randzone der Festplatte. Um möglichst genaue Werte zu erhalten, wurde ein L4-Testprogramm implementiert, das die Funktionen der Dateisystem-Bibliothek verwendet.

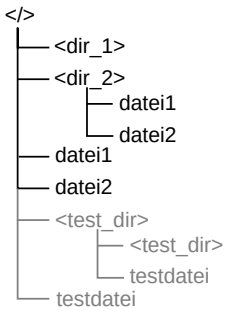

Abbildung 15: Die Dateisystemstruktur der Testumgebung

Die Struktur des Dateisystemes wurde dabei für jeden Durchlauf gleich angelegt (vgl. Abb. 15). Die grau dargestellten Teile wurden während des Tests erstellt, verwendet und anschließend wieder ge-

löscht. Dabei wurden die Messungen sowohl mit integriertem, auf 5 bzw. 10 Elemente beschränkten und unbeschränkten Blockcache, als auch ohne Blockcache ausgeführt. Bei dem Testprogramm handelt es sich um ein eigenständiges L4-Programm, das in einer Schleife die verfügbaren Dateisystemfunktionen aufruft und die bis zur Rückkehr benötigten Zeiten ausgibt bzw. speichert. Die benötigte Zeit eines Auftrages (Funktionsaufruf) wurde durch Auslesen des *Time Stamp Counter* des Prozessors bestimmt. Das Testprogramm verwendet zusätzlich das Verzeichnis „<Test>“ zum Schreiben der Meßdaten, diese werden nach Zugriffen getrennt in Dateien abgelegt.

5.2 Der Blockcache

Kommando	1. Zugriff		Durchschnitt		Minimum		Maximum	
Blockcache	ohne	mit	ohne	mit	ohne	mit	ohne	mit
Verzeichnis								
lesen «/»	17,656	13,577	12,258	0,707	12,189	0,700	17,656	0,746
lesen (subdir)	12,247	6,314	12,244	11,525	12,188	1,338	12,277	12,134
anlegen	62,015	49,986	32,158	32,309	32,046	32,201	62,015	49,986
löschen	32,782	7,586	37,375	7,520	37,296	7,443	37,409	7,586
Dateien in «/»								
anlegen	9,791	9,743	9,929	9,599	9,758	9,743	15,929	10,011
öffnen	1,792	1,117	1,757	0,700	1,742	0,696	1,792	1,117
lesen	1,023	0,689	1,012	0,675	1,001	0,669	1,071	0,726
Ende suchen	0,019	0,018	0,016	0,016	0,016	0,016	0,042	0,039
schreiben	4,069	4,087	4,081	4,082	4,011	3,987	10,178	10,141
schließen	0,014	0,014	0,014	0,014	0,014	0,014	0,037	0,033
Dateien in Unterverzeichnis								
anlegen	16,672	16,659	28,599	19,462	16,672	16,659	28,647	19,522
öffnen	14,758	15,610	15,941	4,881	13,716	2,272	17,620	15,610
lesen	6,878	7,141	6,873	7,205	6,818	7.119	6,923	7,246
Ende suchen	0,017	0,017	0,016	0,017	0,015	0,016	0,042	0,043
schreiben	15,398	15,404	15,418	15,417	15,279	15,275	15,451	15,465
schließen	0,015	0,015	0,012	0,014	0,012	0,012	0,032	0,034

Tabelle 4: Vergleich der Zugriffszeiten des Dateisystems (Zeiten in ms)

Die Untersuchung der Anzahl der Kollisionen bei der genutzten Hashfunktion im Vergleich mit der Verwendung der Block-ID als Hashschlüssel zeigte in der Testumgebung, daß die verwendete Funktion optimaler arbeitet, d. h. es wird eine annähernde Gleichverteilung der Schlüssel in der Tabelle erzielt.

In Tabelle 4 sind die Zeiten für Nichtechtzeitzugriffe auf das Dateisystem mit und ohne Blockcache gegenübergestellt. Die Zeiten für Lesen und Schreiben wurden mit einer Puffergröße[7] von 1 KByte ermittelt. Dadurch sind die Unterschiede nicht so signifikant, da hier schon weitere (Nichtmeta-)Blöcke gelesen wurden. Beim Lesen/Schreiben von 50 Byte entsprachen die Werte der Tendenz der anderen Befehle. Die Durchschnittswerte basieren auf den Meßwerten von mehreren Testdurchgängen mit jeweils 100-1000 Schleifendurchläufen. Wie bereits in Abschnitt 2.4.1 dargestellt, sind pro Dateisystemaufruf mehrere Zugriffe auf einen Block notwendig, so daß bereits beim ersten Aufruf – trotz Cache Misses – eine Beschleunigung im Vergleich zum Dateisystem ohne Cache festzustellen ist. „Ende suchen“ steht für die Ausführung des Befehls `fseek` zum Dateiende (1024 Byte). Die Befehle `fseek` und `fclose` sind nur der Vollständigkeit halber mit aufgeführt, da hier keine Leseoperationen auftreten. Bei letzterem Befehl sollten Einflüsse des Aufrufes `c_release_blk` untersucht werden. Durch die Schwankungen der Verarbeitungszeiten im Dateisystem waren auf diesem Wege keine Einflüsse festzustellen. Deshalb wurde in einer weiteren Untersuchung der Zeitbedarf der Cachefunktionen betrachtet.

Die Zeiten für den Best- bzw. Worst Case wurden ebenfalls unter Verwendung des *Time Stamp Counters* ermittelt. Die dafür benötigten Befehle wurden direkt in das Dateisystem integriert. Dadurch konnte der Zeitbedarf der betrachteten Funktionen genau vermessen werden. In Tabelle 5 sind die Ergebnisse für das Lesen von der Festplatte und die relevanten Cachefunktionen dargestellt. Für Cache-Misses und direkte Festplattenlesebefehle konnten auf diesem Wege keine sinnvollen Werte ermittelt werden. Die gezeigten Zeiten ($\approx$ 0,4 ms) sind wesentlich kleiner als die mittlere Zugriffszeit (6,5 ms) der verwendeten Festplatte im genutzten Bereich [Bög99]. Eine mögliche Erklärung ist, daß aufgrund der Verteilung der Plattenblöcke während der Untersuchung viele Cache-Hits im internen Festplattencache auftraten (Größe 1 MByte). Die genaue Ursache muß jedoch in einer weiteren Untersuchung mit anderem Versuchsaufbau erörtert werden. Die niedrigen Zeiten haben zur Folge, das auch kein Wert für die 85% der Aufrufe erzielt wurde. In der Tabelle ist aber ersichtlich, daß der Zeitverbrauch des Befehls `c_release_blk` sehr gering ist.

	Minimum	85%	Maximum
SCSI-Lesen	(370)		8460
Cache-Miss	(392)		8941
Cache-Hit	5	10-11	31
Block Release	5	7	9

Tabelle 5: Best und Worst Case Zeiten der Cache Befehle (in μs)

In Abbildung 16 sind die Durchschnittswerte des Dateisystemes mit den maximalen Abweichungen dargestellt. Es wird ersichtlich, daß sich die Durchschnittswerte bei einer großen Anzahl von Cache-Hits in der Nähe des Minimums bewegen. Das war auch zu erwarten, da im Verlaufe des Versuchs-

[7] hier ist der Lese-/Schreibpuffer der Anwendung gemeint

durchlaufes nur die ersten Zugriffe Maxima erzeugen, dies wird beim `mkdir` besonders gut sichtbar. Weiterhin wird deutlich, daß die Schwankungen bei dem Dateisystem ohne Cache relativ hoch sind. Dieses Verhalten kann durch Positionierzeiten der Platte hervorgerufen werden.

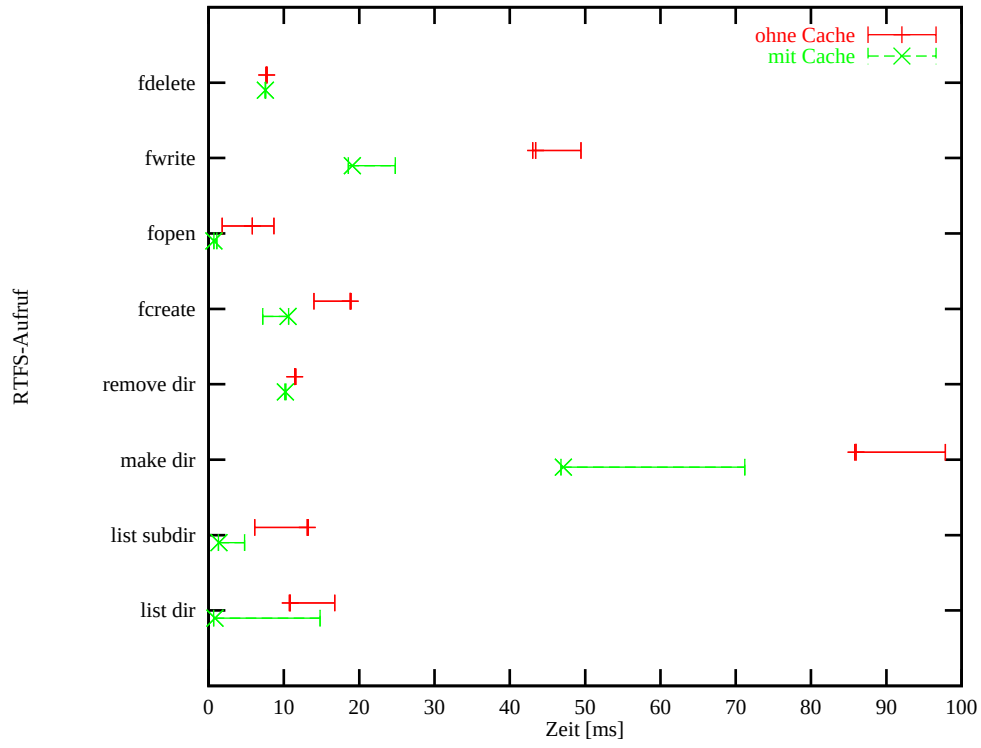

Abbildung 16: Durchschnittliche Bearbeitungszeiten für einen Dateisystemaufruf im Wurzelverzeichnis

Die Anpassung des Dateisystemes erfolgte auf sehr einfache Weise, d. h. es wurden alle relevanten Aufrufe 1 zu 1 ersetzt bzw. modifiziert. Eine Bearbeitung der Daten unter Verwendung des Wissens um einen Cache, z. B. Zusammenfassen von Leseaufrufen, wurde noch nicht beachtet. Eine weitere Möglichkeit für eine Beschleunigung ist das Speichern von logischen Dateisystemblöcken. Die derzeitige Implementierung im Dateisystem nutzt den Cache auf Ebene der physischen Plattenblöcke. Das bedeutet, es findet jedesmal eine Umrechnung der logischen Blocknummer in die physische statt. Durch Verlagerung dieser Berechnung in Richtung Blockgeräteschnittstelle können weitere Vorteile im Falle eines Cache-Hits erzielt werden. Die mögliche Zeitersparnis beläuft sich auf eine bis drei Mikrosekunden pro zu lesendem Block (vgl. Tabelle 6). Wie bereits betont wurde, besteht eine Dateisystemoperation aus mindestens drei Lesebefehlen. Bei der Untersuchung konnten gelegentliche Ausreißer bis zu 21 Mikrosekunden festgestellt werden (bei ca. 1000 Aufrufen je 1mal 21 und 18, 2mal 17 und 3mal 8 μs). Den Ursachen wurde nicht nachgegangen, sie könnten aber durch Umschalten der Task entstehen.

	Durchschnitt	Maximum	Minimum
einzelner Block	2.2 μs	3 μs	1 μs
Dateioperation			
8 Blöcke	16 μs	22 μs	15 μs
3 Blöcke	6.6 μs	8 μs	5 μs

Tabelle 6: Durchschnittlicher Zeitverbrauch bei der Umrechnung logischer in physische Plattenblöcke (Zeiten in μs)

In den Abbildungen 17 und 18 sind zwei ausgewählte Dateioperationen mit und ohne Cache gegenübergestellt. Diese Operationen wurden gewählt, da sie die Vorteile des Metadatencaches sehr gut illustrieren. Die beiden Kommandos zeigen auch sehr gut, wie stark die zu erwartende Verbesserung von der Anzahl der unterschiedlichen zu lesenden Blöcke abhängt. Beim Öffnen einer Datei tritt bei vier Lesebefehlen ein Cache-Hit auf, die Verbesserung der Zugriffszeiten beträgt rund Faktor 2,5. Bei dem Befehl zum Löschen eines Verzeichnisses treten sogar 9 Cache-Hits bei 12 Lesebefehlen auf, daraus ergibt sich eine Verkürzung der Zugriffszeit auf ein Fünftel. Weiterhin wird sichtbar, daß bei der Verwendung des Caches die Schwankungen wesentlich geringer und die Zugriffszeiten konstanter – also besser vorhersagbar – werden. Es wird auch deutlich, das die Anzahl der Zugriffe sich auf die Größe der Schwankungen auswirkt. Die größere Abweichung beim Löschen kann auf den Freigabealgorithmus von Festplattenspeicher im RTFS zurückgeführt werden.

Die Stabilität des Dateisystemes wurde durch die Verwendung des Blockcaches nicht negativ beeinflußt. Um die volle Leistungsfähigkeit auszunutzen, sind noch Optimierungen bei der Suche und dem Ersetzen alter Blöcke möglich bzw. nötig. Bei den Tests haben sich zehn Pufferelemente für den Blockcache als gutes *Working Set* ergeben. Wie bereits beschrieben, kann dieser Wert als Orientierung verwendet werden, um die Größe des Caches einzuschränken (z. B. mittels `c_flushd`). Jedoch sollte dieser Wert mit anderen Zugriffsmustern und kontinuierlichen Daten überprüft werden.

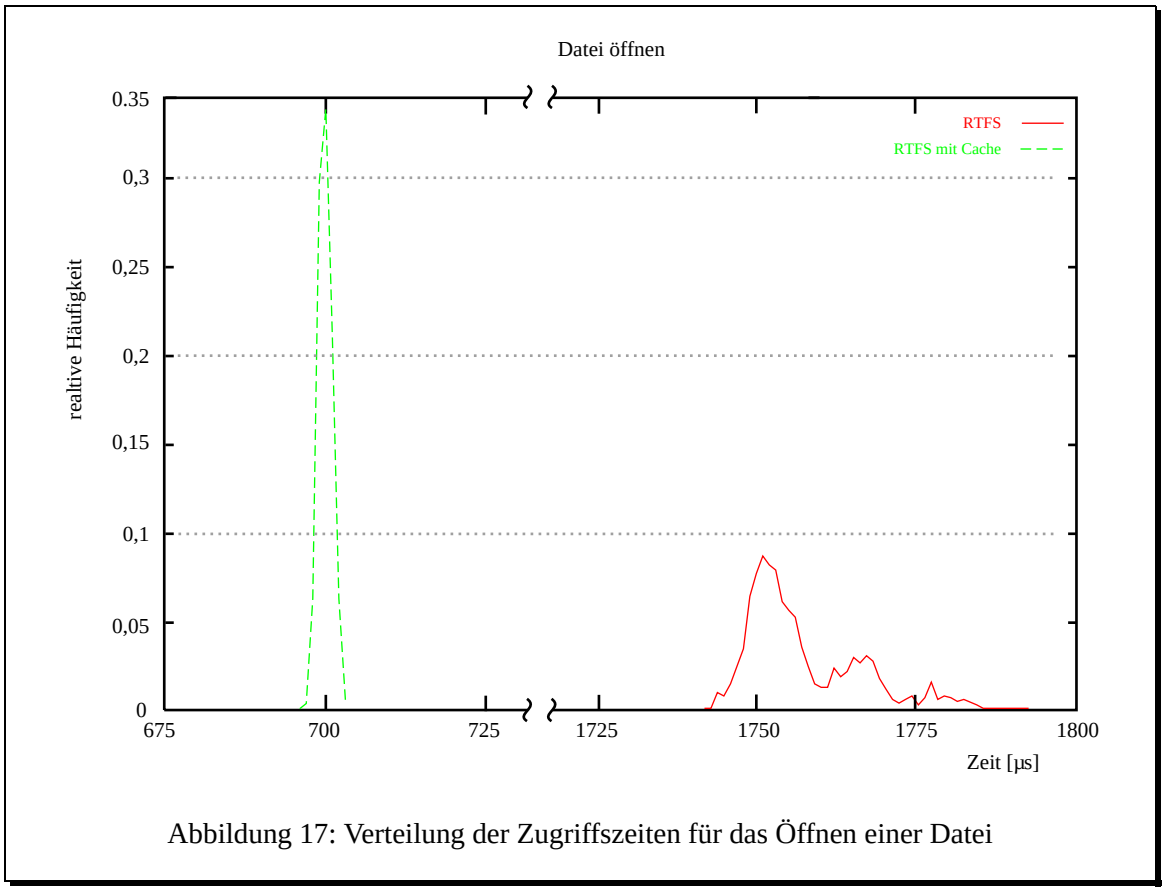

Abbildung 17: Verteilung der Zugriffszeiten für das Öffnen einer Datei

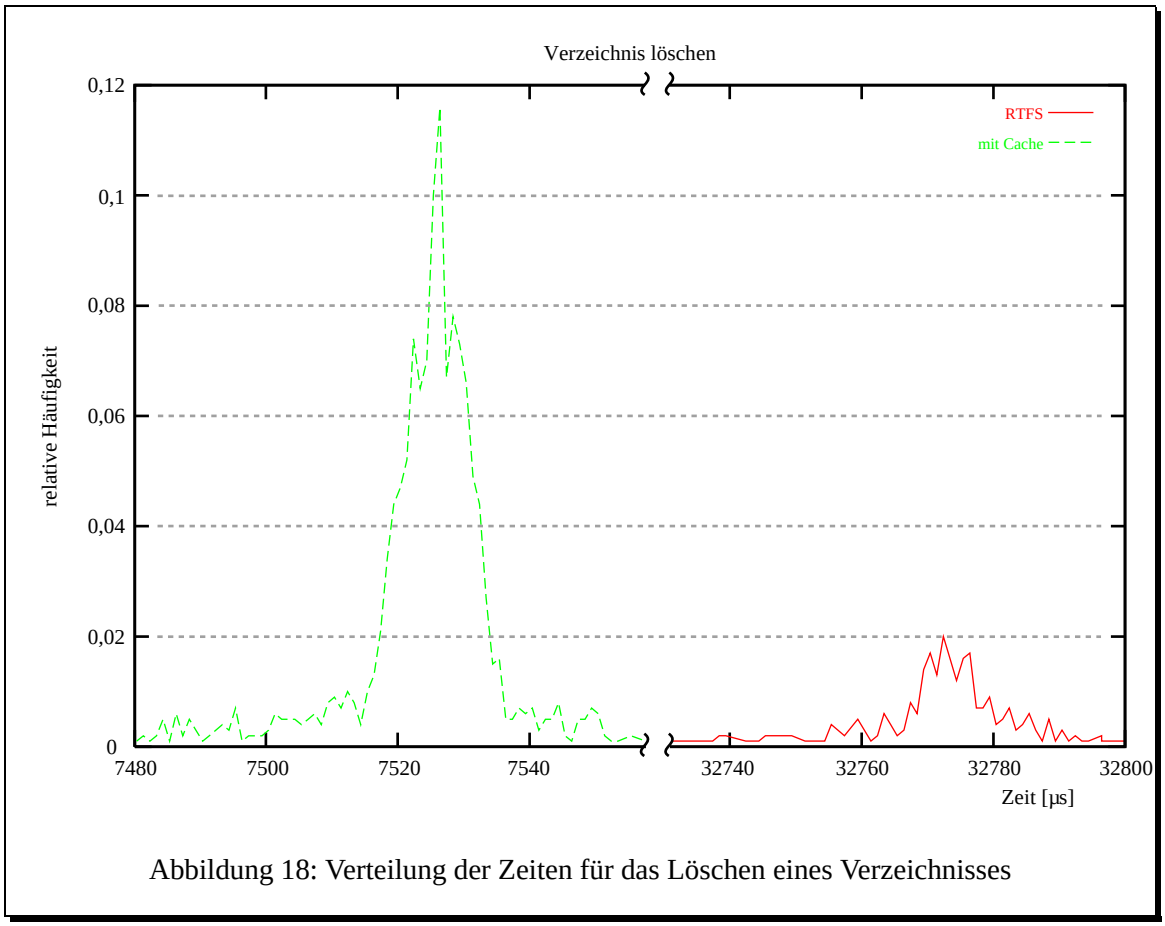

Abbildung 18: Verteilung der Zeiten für das Löschen eines Verzeichnisses

6 Zusammenfassung und Ausblick

Das Ziel der Arbeit, einen Blockcache für das Echtzeitdateisystem zu entwickeln und zu implementieren, wurde erreicht. Die vorliegende Implementierung wurde getestet und brachte schon bei einer einfachen Integration in das Dateisystem sichtbare Geschwindigkeitsvorteile. Der Blockcache ist dateisystemunabhängig und kann für weitere, zukünftig zu entwickelnde Dateisysteme eingesetzt werden. Nicht überprüft werden konnte die Funktionalität für kontinuierliche Daten, da die Anbindung an das Dateisystem fehlt. Allerdings lassen die Meßwerte der Verwaltungsdaten für Nichtechtzeitzugriffe auch dafür positive Ergebnisse erwarten.

Darüber hinaus wurde ein Cachesystem entworfen, das eine Abstraktion von physischen Dateisystemen und Standardaktionen anbietet. Dadurch kann eine für die Anwendungen transparente Anbindung und Unterstützung für unterschiedliche (Datei-) Systeme vorgenommen werden.

In der Arbeit wurde gezeigt, daß die Verwendung eines Caches auch für Echtzeitanwendungen sinnvoll sein kann und gerade bei der Verarbeitung von Verwaltungsstrukturen Vorteile bringt. Es wurde auch dargelegt, daß für Echtzeitströme ein Cache durch die begrenzt vorhandenen Ressourcen nur eingeschränkt Vorteile bei der Übertragung bringen kann.

Weiterhin wurde ein logisches Dateisystem für L4 entworfen und dessen Aufbau beschrieben. Die Implementierung des logischen Dateisystemes kann in einer weiterführenden Arbeit realisiert werden, da die DROPS-Stromschnittstelle inzwischen endgültig entworfen wurde und ein Prototyp in Arbeit ist. Erst dann werden Aussagen über die Vor- und Nachteile der in dieser Arbeit getroffenen Entwurfsentscheidungen möglich. Für die Verbindung vom RTFS zum logischen Dateisystem sind grundlegende Änderungen der Pufferverwaltung im RTFS notwendig. Dabei sollte auch die Nutzung des Blockcaches erweitert und optimiert werden.

Themen weiterführender Untersuchungen können außerdem die Bestimmung der Größe des *Working Sets* mit Hilfe anderer Testzyklen und eine Zeitabschätzung für die Synchronisation der Puffer sein. Weitere Möglichkeiten liegen bei der Nutzung des *informed prefetch* durch L4-Anwendungen.

A Zugriffsverhalten des Echtzeitdateisystems

Die folgenden Tabellen illustrieren das Schreib-/Leseverhalten des RTFS für nichtzeitkritische Anwendungen. Für die Untersuchung wurden für alle Funktionen die Blocklesebefehle des Dateisystems protokolliert und ausgewertet. In den Tabellen wird in der Spalte „Größe“ die Anzahl der Puffer ($min - max$) angegeben, min bezeichnet die Anzahl benötigter Puffer, um wiederholte Zugriffe während einer Operation aus dem Cache beantworten zu können und max die Anzahl, um alle Blöcke, die zur Beantwortung der Anfragen benötigt werden, im Puffer aufnehmen zu können. Die Symbole der Spalte in der „Hit“ haben folgende Bedeutung:

* Cache-Hit bei Anzahl der Puffer $\geq min$
\+ Beschleunigung des Zugriffs bei verzögertem Schreiben

Initialisieren des Dateisystems

Kommando	Zugriff			
	Plattenblock	Was	Größe	Hit
Initialisieren	0	root superblock		
	0	partition superblock	1	*

Tabelle 7: Initialisieren des Echtzeitdateisystems mit 1 Partition

Zugriffe auf Verzeichnisse In Tabelle 8 sind die möglichen Zugriffe auf Verzeichnisse bis zu einem Unterverzeichnis dargestellt. Bei tieferer Verzeichnisstruktur kommen entsprechende Zugriffe (Lesen der Blockliste in übergeordneten Verzeichnissen) hinzu.

Kommando	Zugriff			
	Plattenblock	Was	Größe	Hit
Initialisieren	0	root superblock		
	0	partition superblock	1	*
Verzeichnis anzeigen				
leeres root	512933	root inode lesen Blockliste		
	512933	lesen Blocklistendaten		*
	512937	«.»		
	512933		2	*
root mit	512933	root inode lesen Blockliste		
2 Unterverzeichnissen	*512933*	lesen Blocklistendaten		*

Fortsetzung nächste Seite

1 Datei	512937	root dir «.»		
	512933			*
	512933			*
	512805	i-node subdir1		
	512821	i-node subdir2		
	512829	file	2-5	
Verzeichnis anlegen				
im root	512933	root i-node lesen Blockliste		
	509129	schreiben neues Verzeichnis		+
	509129			*
	509129	schreiben Verzeichnisblockliste		+
	512933	schreiben Rootblockliste		+
	512933			*
	512937	«.»		
	512933			*
	512937	aktualisieren rootverzeichnis	5	+
im Unterverzeichnis	512933	root i-node lesen Blockliste		
	512933	lesen Blocklistendaten		*
	512937	«.»		
	512805	subdir i-node		
	509137	schreiben		+
	509137			*
	509137	schreiben Blockliste		+
	509141	schreiben subdir Blockliste		
	512805	subdir		*
	512809			
	512805			+
	512809	aktualisieren subdir	5	+
Verzeichnis löschen				
im root	512933	lesen root i-node Blockliste		
	512933	lesen Blocklistendaten		*
	512937	lesen Verzeichnis «.»		
	509129	subdir Blockliste		
	509129	lesen Blocklistendaten		*
	512933			*
	512933			*
	512933			*
	512937	«.»		*
	512933			*

Fortsetzung nächste Seite

	512937	schreiben «.»		+
	509129		3	*

Tabelle 8: Untersuchung des Zugriffsverhaltens des Echtzeitdateisystems auf Verzeichnisse

Zugriffe auf Dateien Beim Ausführen der Dateioperationen anlegen und öffnen kommen für Zugriffe auf Dateien in Unterverzeichnissen 3 Zugriffe (lesen Blockliste, Blocklistendaten, Verzeichniseintrag) pro Baumtiefe hinzu. Davon sind die ersten 2 auf dieselbe Blocknummer. Beim Lesen und Schreiben sind die Werte für Größe und Hit nur der Vollständigkeit halber angegeben, da in der Arbeit eine Trennung von Verwaltungs- und Nutzdaten umgesetzt wurde. Da die Untersuchung in Hinblick auf einen Metadaten-Cache erfolgte, sind nur die entsprechenden Zugriffe auf Verwaltungsdaten aufgezeichnet worden. Für den Dateicache sind Zugriffsmuster und eine Analyse des Verhaltens relevant, dort werden Plattenblöcke nicht beachtet.

Kommando	Zugriff			
	Plattenblock	Was	Größe	Hit
Datei anlegen				
im root	512933	root i-node lesen Blockliste		
	509121	schreiben neue Datei		+
	512933			*
	512933			*
	512937	«.»		
	512933			*
	512937	aktualisieren «.»	3	+
Datei öffnen				
im root	512933	root i-node lesen Blockliste		
	512933	lesen Blocklistendaten		*
	512937	«.»		
	509121	lesen Dateiblockliste	1-4	
Datei sequentiell lesen				
im root	509121	Datei Blockliste		(*)
	509122	Datei Block 1		
	...	Datei Block n	1-n	
Datei sequentiell schreiben				
im root	509121	Datei Blockliste		(*)
	509121	aktualisieren Blockliste		+
	509121			*
	509121			+
	509121			*
	509125			
	509125	schreiben Datum 1		+
	...			
	...	schreiben Datum n	1	+
Datei löschen				
im root	512933	root i-node lesen Blockliste		
	512933	lesen Blocklistendaten		*
	512937	«.»		
	509121	lesen Dateiblockliste		
	512933			*
	512933			*
	512937	«.»		*
	512933			*
	512937	aktualisieren «.»		+

Tabelle 9: Untersuchung des Zugriffsverhaltens des Echtzeitdateisystems auf Dateien

B Abkürzungsverzeichnis

BSD Berkeley Software Distribution

DMA Direct Memory Access

DROPS Dresden Realtime Operating System

EFS extent Filesystem (Dateisystem von SGI bis IRIX 5.3)

ext2 second extended Filesystem (Dateisystem von Linux)

FFS Berkeley Fast Filesystem (Dateisystem von BSD-OS bzw. FreeBSD)

IDE Intelligent Drive Electronics

FIFO First In First Out

LFU Least Frequently Used

LRU Least Recently Used

MRU Most Recently Used

NFU Not Frequently Used

NRU Not Recently Used

NTFS New Technology Filesystem (Dateisystem von Windows NT)

POSIX Portable Operating System Interface

QoS Quality of Service

RTFS Real Time Filesystem (DROPS Echtzeitdateisystem)

SCSI Small Computer System Interface

VFS Virtual Filesystem

XFS next generation filesystem (Dateisystem von SGI IRIX)

C Glossar

Admission Control entscheidet über die Zulassung neuer Aufträge in Abhängigkeit von der Systemlast.

Block Einheit, in der auf die Festplatte zugegriffen werden kann. Die minimale Größe wird durch die Festplatten-Hardware bestimmt. Diese können vom Dateisystem zu logischen Blöcken zusammengefaßt werden.

Blockliste Tabelle, die die zu einer Datei gehörenden Blöcke enthält.

Cache (aus dem franz. für verstecken/verdecken) Zwischenspeichern von Daten mittels schnellerer Hardware.

Cache-Hit gesuchtes Datum befindet sich bereits im Cachespeicher.

Cache-Miss gesuchtes Datum befindet sich nicht im Cachespeicher. Gegenteil von Cache-Hit.

Deadline Spätester Zeitpunkt, zu dem die Daten zur Verfügung stehen müssen.

Direct Memory Access (DMA) Zugriffsart auf den Hauptspeicher ohne Belastung der CPU.

double indirect blocks i-node Verwaltung, Blocklisten mit Zeigern auf indirekte Blocklisten. Verfahren um sehr große Dateien zu verwalten.

first in first out (FIFO) Strategie zum Seitenaustausch bei der immer das älteste Element entfernt wird.

FreeBSD frei verfügbares UNIX-Betriebssystem für PCs, entspricht 4.4 BSD Operating System.

indirect blocks i-node Verwaltung. Blockliste enthält Zeiger auf Blocklisten.

i-node Dateisysteminterne eindeutige Beschreibung einer Datei. Sie enthält Dateityp, Besitzer und eine Liste der belegten Plattenblöcke.

Journaling Filesystem Das Dateisystem führt Buch über alle Änderungen, die ausgeführt werden. Dadurch kann im Falle eines Absturzes der letzte konsistente Zustand des Systems schnell wiederhergestellt werden.

least frequently used (LFU) Speicheraustauschalgorithmus, es wird die am wenigsten verwendete Seite ausgelagert.

least recently used (LRU) Speicheraustauschalgorithmus, es wird die Seite ausgelagert, deren Verwendungszeitpunkt am weitesten zurückliegt.

Metadaten Verwaltungsdaten einer Datei, die zusätzlich verarbeitet werden müssen.

most recently used (MRU) zum LRU umgekehrter Speicheraustauschalgorithmus, es wird die zuletzt verwendete Seite ausgelagert.

not frequently used (NFU) analog LFU.

Paging Austausch von physischen und virtuellen Speicherseiten. Auslagern physischen Speichers auf Sekundärspeicher (z. B. Festplatten).

Pinning Verhinderung des Auslagerns von Speicher, Speicherbereiche an einer Adresse festhalten. Garantie das Speicheradresse mit physischem Speicher hinterlegt ist.

Portable Operating System Interface (POSIX) Standard zur Definition von Anwendungsschnittstellen zum Betriebssystem.

Prefetch siehe *Read Ahead*. Kann gestützt oder automatisiert erfolgen.

Quality of Service (QoS) minimale Eigenschaft („akzeptale“ Qualität) eines Stromes. Wird über Parametersatz beschrieben.

Read Ahead Vorauslesen der Daten von der Festplatte.

SCSI *Small Computer Systems Interface* Standard zum Anschluß von Peripheriegeräten.

second chance Seitenaustauschalgorithmus. Kombination aus LRU und FIFO. Es wird die älteste im Fenster nicht referenzierte Seite verdrängt.

timeout Zeitpunkt an dem das angeforderte Datum ungültig wird (siehe auch deadline).

worst case ungünstigster anzunehmender (Anwendungs-)Fall.

write throttling Schreibzugriffe werden verzögert, also nicht direkt ausgeführt.

Virtual File System (VFS) einheitliche Schnittstelle zum Zugriff auf unterschiedliche Dateisysteme. Im Linux steht VFS für *Virtual Filesystem Switch* und bezeichnet die Schicht im Kern, die zusätzlich zur Schnittstelle eine Dateiverwaltung (incl. Dateicache) bietet und Standardaktionen ausführt.

v-node objektorientierte Beschreibung einer geöffneten Datei. Enthält Dateibeschreibung und Dateisysteminformationen.

Literatur

[BBD 95] BECK, Michael ; BÖHME, Harald ; DZIADZKA, Mirko ; KUNITZ, Ulrich ; MAGNUS, Robert ; VERWORNER, Dirk: *Linux Kernel Programmierung – Algorithmen und Strukturen der Version 2.0*. 4. Bonn : Addison Wesley, 1995

[Bög99] BÖGEHOLZ, Harald: Platten-Karussell (Zusammenfassung). **In:** *c't Magazin für Computer Technik* 17 (1999), S. 152–162

[But97] BUTTAZO, Giorgio C.: *Hard Real Time Computing Systems*. 1. Kluwer Academic Publishers, 1997

[CNMC] CHOI, Jongmoo ; NOH, Sam H. ; MIN, Sang L. ; CHO, Yookun: Detection-based Adaptive Buffer Management Scheme. –
zu finden unter http://ssrnet.snu.ac.kr/ choijm/adaptive_cache.html

[Gia99] GIAMPAOLO, Dominic: *Practical File System Design with the BE File System*. 1. San Francisco : Morgan Kaufmann Publishers Inc., 1999

[Ham98] HAMANN, Claude-Joachim. Pufferberechnung bei schwankungsbeschränkten Strömen. Vortrag Klausurtagung Betriebssysteme. September 1998

[Här98] HÄRTIG, Hermann. Verteilte Betriebssysteme. Vorlesungsskript TU Dresden. 1998

[HD95] HOLTON, Mike ; DAS, Raij: XFS: A Next Generation Journalled 64-Bit Filesystem with Guaranteed I/O. 1995. –
zu finden unter http://www.europe.sgi.com/Technology/xfswhitepaper.html

[Hoh96] HOHMUTH, Michael: *Linux-Emulation auf einem Mikrokern*, Institut für Betriebssysteme, Datenbanken und Rechnernetze,TU Dresden, Diplomarbeit, 1996

[Hoh99] HOHMUTH, Michael: FIASCO – Ein L4 kompatibler Mikrokern. 1999. –
zu finden unter http://os.inf.tu-dresden.de/fiasco

[HR99] HAMANN, Claude-J. ; REUTHER, Lars. Pufferdimensionierung für schwankungsbeschränkte Ströme in DROPS. 1999

[HRW 99] HÄRTIG, H. ; REUTHER, L. ; WOLTER, J. ; BORISS, M. ; PAUL, T. Cooperating Resource Managers. 1999

[Knu73] KNUTH, Donald E.: *The Art of computer programming – Volume 3/Sorting and Searching*. Massachusetts : Addison Wesley, 1973

[Lie96] LIEDTKE, Jochen: L4 Reference Manual for 486, Pentium and Pentium Pro. 1996. –
zu finden unter http://os.inf.tu-dresden.de/L4/l4refx86.ps.gz

[Lös99] LÖSER, Jork. Streaming Schnittstelle für DROPS. 1999

[MBKQ99] MCKUSICK, Marshall K. ; BOSTIC, Keith ; KARELS, Michael J. ; QUARTERMAN, John S.: *The Design and Implementation of the 4.4 BSD Operating System*. 6. Massachusetts : Addison Wesley, 1999

[Meh98] MEHNERT, Frank: *Ein zusagenfähiges SCSI-Subsystem für DROPS*, Institut für Betriebssysteme, Datenbanken und Rechnernetze,TU Dresden, Diplomarbeit, 1998

[MRSW97] MCNAMEE, D. ; REVEL, D. ; STEERE, D. ; WALPOLE, J. Adaptive Prefetching for Device-Independent File I/O. 1997

[Nag97] NAGAR, Rajeev: *Windows NT File System Internals – A Developers Guide*. 1. Sebastopol USA : O'Reilly, 1997

[NS98] NEHMER, Jürgen ; STURM, Peter: *Systemsoftware: Grundlagen moderner Betriebssysteme*. 1. Heidelberg : dpunkt-Verlag, 1998

[Pat97] PATTERSON, Russel H.: Informed Prefetching and Caching. 1997. –
zu finden unter http://www.pdl.cs.cmu.edu/PDL-FTP/TIP/rhp_diss_abs.html

[Pfl86] PFLUG, Georg: *Stochastische Modelle in der Informatik*. 1. Stuttgart : Teubner, 1986

[POS96] ISO/IEC 9945-1:1996 Information technology – Portable Operating System Interface (POSIX) - Part 1: System Application Programming Interface (API) [C Language]. 1996

[Reu98] REUTHER, Lars: *Entwicklung eines echtzeitfähigen Dateisystems*, Institut für Betriebssysteme, Datenbanken und Rechnernetze,TU Dresden, Diplomarbeit, 1998

[Reu99] REUTHER, Lars. DROPS Komponenten Modell. Vortrag Klausurtagung Betriebssysteme. November 1999

[Reu00] REUTHER, Lars. nutzbare Funktionalität des Echtzeitdateisystemes und Einschränkungen. Mündliche Mitteilung. Februar 2000

[RLG00] REUTHER, Lars ; LÖSER, Jork ; GRÜTZMACHER, Lukas. Diskussion über den Aufbau und die Eigenschaften der Stromschnittstelle für DROPS. Februar 2000

[Rus98] RUSSINOVICH, Mark: Inside The Cache Manager. **In:** *Windows NT Magazine* 10 (1998), S. 67–75

[SGI99] SGI: IRIX Admin: Disks and Filesystems, Chapter 8: System Administration for Guaranteed-Rate I/O. 1999. –
zu finden unter http://techpubs.sgi.com:80/library/dynaweb_bin/ebt-bin/0650/nph-infosrch.cgi/infosrchtpl/SGI_Admin/IA_DiskFiles/@InfoSearch__BookTextView/15641;he=0?DwebQuery=xfs

[SGRV97] SHENOY, Prashant J. ; GOYAL, Pawan ; RAO, Sriram S. ; VIN, Harrik M. Symphony: An Integrated Multimedia File System. 1997

[Sol98] SOLOMON, David A.: *Inside Windows NT*. 2. Redmond : Microsoft Press, 1998

[Tan95] TANENBAUM, Andrew S.: *Moderne Betriebssysteme*. 2. München : Carl Hanser und Prentice-Hall International, 1995