Entwurf und Implementierung eines Block-Zwischenspeichers für verteilte Dateisysteme

Studienarbeit
Institut für Betriebs- und Dialogsysteme (IBDS)
Prof. Dr. Frank Bellosa
Fakultät für Informatik
Universität Karlsruhe (TH)

von

cand. inform.
Axel Sanwald

Betreuer:
Prof. Dr. Frank Bellosa
Dipl.-Ing. Kendy Kutzner

Tag der Anmeldung:	15. Mai 2006
Tag der Abgabe:	15. Juli 2006

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Karlsruhe, den 14. Juli 2006

Inhaltsverzeichnis

1. Einleitung

Heutzutage hält eine steigende Anzahl elektronischer Geräte Einzug in die privaten Haushalte. Egal ob mobiles Vielzweck-Kommunikationsgerät oder stationäre Unterhaltungselektronik, die kabel- oder funkgebundene Vernetzung steigt von Jahr zu Jahr. Gleichzeitig nimmt auch das Bedürfnis, Daten zwischen diesen Geräten austauschen zu können, zu. Was wäre hierfür besser geeignet als ein verteiltes Dateisystem?
Die Einsatzmöglichkeiten beginnen privat bereits bei der lästigen manuellen Synchronisation zwischen einem Desktop Computer und einem Notebook. Aber auch der Datenaustausch mit anderen Geräten wie beispielsweise MP3-Player, Digitalkameras oder Handys könnte einfach und effizient über ein gemeinsames Dateisystem erfolgen.

Darüber hinaus setzen heute viele Firmen bereits auf Netzwerke, um ihren Angestellten den Austausch von Daten aller Art zu ermöglichen und somit Teamarbeit zu unterstützen. Laufwerke in diesem Netz befinden sich heute zumeist in einem dedizierten Server und enthalten einen Datenträger mit einem lokalen, festen Dateisystem. Obwohl dieses Dateisystem anderen Geräten zugänglich gemacht wurde bleibt es unverteilt. Es ist lediglich von mehreren Benutzern gleichzeitig über ein gemeinsames Netzwerk erreichbar. Der große Nachteil eines solchen zentralisierten Servers wird schnell klar, wenn dieser einmal ausfällt, oder noch schlimmer, wenn er zerstört wird.

Eine natürlichere und insgesamt stabilere Form ist, wenn die Daten jedes Mitarbeiters auf dessen eigenen Computer gespeichert bleiben, während die Kollegen Kopien davon über eine gemeinsame Sicht erhalten können. Ist der Kollege einmal außer Haus und sein Notebook offline, sind dennoch die verteilten Kopien verfügbar. Auch könnten ganz gezielt Kopien ausgeteilt werden, um einerseits die Verfügbarkeit der Daten zu erhöhen und andererseits den physischen Verlust eines Gerätes, durch die redundant gespeicherten Inhalte, weniger dramatisch zu gestalten. Diese Entwicklung wird durch den rapiden Preisverfall bzw. dem Kapazitäts-Wachtum von persistentem Speicher weiter unterstützt. Notebooks mit 100 GB Speicherkapazität sind heute eher Standard als eine Seltenheit. Warum nicht diesen, oftmals zu großen Anteilen brach liegenden, Speicherplatz nutzen?

Als Basis für den gesamten Datenaustausch soll die in den letzten Jahren aufgekommene, bereits durch filesharing erprobte und bekannt gewordene Peer-to-Peer-Technologie dienen. Im Prinzip beschreibt der Begriff filesharing bereits die grundlegende Idee, nur, dass dafür in der Vergangenheit spezielle Programme für den Download von Daten und Internet basierte Register[1] für die Suche nach Einträgen nötig waren. All dieses gilt es in einem Dateisystem zusammenzufassen und mit einem gewohnteren und intuitiven Zugang zu versehen. Gleichzeitig sollte dabei der Benutzer nicht mit langwierigen Konfigurationsdetails gelangweilt werden.

Vor allem strukturierte Peer-to-Peer-Netze, wie beispielsweise *Chord* ([SMKK+01]) oder *Pastry* ([RoDr01]), zeichnen sich durch ihre Selbstorganisation, Dezentralität und eine nebenbei entstehende automatische Lastverteilung aus. Das unterliegende Peer-to-Peer-System soll also hauptsächlich die Grundlage für die Robustheit, Verfügbarkeit und Skalierbarkeit des geplanten Dateisystems bieten.

1.1 Zielsetzung der Arbeit

Ziel dieser Arbeit ist der Entwurf der Speicherkomponente[2] eines verteilten Dateisystems. Anders, wie in gewöhnlichen Dateisystemen, sieht diese Komponente jedoch keine persistente Speicherung der ihr anvertrauten Daten vor. Stattdessen werden alle Informationen in einem ausreichend großen Cache hinterlegt. Je nach Verdrängungsstrategie kann das zur Folge haben, dass Daten, die lange Zeit nicht mehr verwendet wurden, automatisch entfernt werden müssen, um Platz für Neuere zu schaffen.

Es gibt zwei wichtige Unterschiede zwischen dieser geplanten Speicherkomponente und den bekannteren Cache-Formen wie beispielsweise Prozessorcaches oder Festplattencaches: Zum Einen wäre da die für heutige Maßstäbe noch gigantische Größe von mehreren Terabytes. Darüber hinaus impliziert die Bezeichnung *Zwischenspeicher* eine nicht vorhandene *Endspeichereinrichtung*. Demnach sind Daten, die einmal aus dem Cache verdrängt wurden, nicht länger lokal verfügbar und müssten, sofern vorhanden und gefordert, von einem anderen Teilnehmer des Netzwerks neu bezogen werden. Allein schon die Größe von mehreren Terabytes, die durch das in 2.1 beschriebene Anwendungsszenario gerechtfertigt wird, erfordert eine sehr effiziente Verwaltung. Tatsächlich sieht die Zielsetzung dieser Arbeit jedoch keinerlei Größenbeschränkungen vor, was die Aufgabe der notwendigen Verwaltung keinesfalls erleichtert.

Ähnlich wie in echten Dateisystemen, die Dateien ggf. in mehreren Stücken[3] auf die physische Festplatte speichern (vergleiche mit EXT2 – [CaTT94]), soll auch diese Speicherkomponente auf solchen Dateifragmenten arbeiten. Diese Fragmente, auch Blöcke genannt, stellen gleichzeitig die Granularität dar, auf der das Peer-to-Peer-Netzwerk Daten austauschen wird. Die Blöcke werden dazu durch eindeutige Schlüssel, gewonnen aus einem kryptographischen Hash-Verfahren wie beispielsweise MD5 oder SHA1 ([X9.397]), gekennzeichnet. Diese Verfahren garantieren mit einer ausreichend hohen Wahrscheinlichkeit kollisionsfreie Ergebnisse.

[1] Zum Beispiel für das Peer-to-Peer-System *BitTorrent*.
[2] später auch Cache genannt
[3] in EXT2/3 und ReiserFS 'nodes' oder 'supernodes' genannt

Das Ziel dieser Arbeit ist also die Konstruktion dieses Block-Caches mit einer schnellen, effizienten und skalierenden Verwaltung ohne fest vorgegebenene Größeneinschränkungen.

1.2 Gliederung der Arbeit

Für die weitere Einführung in das Thema "verteilte Dateisysteme" beginnt Kapitel 2 mit einem Anwendungsszenario. Anschließend wird die grobe, bereits vorgegebene Architektur des geplanten Dateisystems vorgestellt. Im weiteren Verlauf des Kapitels geht es dann um die Analyse bereits bestehender, verwandter Systeme, um daraus weitere Anforderungen an den Zwischenspeicher abzuleiten.

Kapitel 3 beschäftigt sich dann ausführlich mit verschiedenen Wegen, wie der Block-Cache effizient konstruiert werden kann. Dabei werden auch mehrere Vorschläge für die geschickte Implementation der Verdrängungsstrategie gemacht.

Nachdem in Kapitel 3 eine entsprechende Konstruktion erarbeitet wurde, beschäftigt sich das vierte Kapitel detailliert mit deren Implementierung. Am Kapitelanfang findet sich zunächst die Beschreibung der Kommunikationstechnik mit den restlichen Komponenten des Dateisystems. Somit ist die Grundlage für die Umsetzung des Zwischenspeichers gegeben.

Für Kapitel 5 verbleibt die Evaluation der im vorigen Kapitel entstandenen Realisierung des Block-Caches. Da das Gesamtsystem derzeit noch nicht fertiggestellt ist, kann hierfür nicht auf bereits vorhandene Testprogramme für Dateisysteme zurückgegriffen werden. Stattdessen wird die einwandfreie Funktionalität der Komponente überprüft und anschließend deren Verhalten bei einer plötzlichen Flut an Anfragen untersucht.

Diese Arbeit endet mit Kapitel 6, das sich mit den erzielten Erfolgen und Erkenntnissen auseinander setzt. Darüber hinaus finden sich dort einige weiterführende Ideen, wie das geplante Dateisystem weiterentwickelt werden könnte.

2. Analyse

Dieses Kapitel beschäftigt sich mit der Analyse anderer, bereits existierender verteilter Massenspeichersysteme. Die Systeme sollen dabei hinsichtlich ihrer Speichertechnik und den damit gemachten Erfahrungen untersucht werden. Die daraus gewonnenen Erkenntnisse werden später in den Entwurf einfließen. Bevor jedoch mit der Auswertung begonnen wird, folgt zunächst ein Anwendungsszenario und die Vorstellung des geplanten *IGOR*-Dateisystems. Diese beiden vorgeschobenen Abschnitte sollen den Leser in die Problematik einführen und die bereits bestehenden Vorgaben skizzieren. Vor allem die Vorgaben sind sehr wichtig, da sich diese Arbeit ausschließlich einer einzigen Komponente des Dateisystems widmet: dem Block-Cache.

2.1 Anwendungsszenario

In der Forschungsabteilung eines Konzerns werden täglich große Datenmengen in der Größenordnung von Gigabytes produziert. Beispiele hierfür könnten die Entschlüsselung eines Genoms oder die auf atomarer Ebene stattfindende Kartographierung eines Gegenstandes sein. All diese Daten werden in einer Datenbank, möglicherweise eine Eigenentwicklung genau für diese Datentypen, abgelegt. Abgesehen von Sicherungskopien sollen auch andere Abteilungen des Konzerns und kooperierende Partner ständig auf dem neuesten Stand sein oder zumindest Zugriff darauf haben. Falls diese Datenbank nicht aus dem gemeinsamen Netzwerk abgefragt werden kann, so wäre ein naiver Ansatz, den Datenbestand täglich an alle Teilnehmer zu kopieren. Wenn man von den benötigten Bandbreiten absieht, so hat simples Duplizieren den entscheidenden Nachteil, dass man sich im Allgemeinen nicht auf neue oder geänderte Teile beschränken kann. Das heißt, die Aktualisierung eines einzelnen Datensatzes unter Millionen erfordert die erneute Kopie sämtlicher Datensätze. Aber auch wenn eine Netzabfrage möglich wäre, müssten dennoch rechenstarke Server und immer noch hohe Bandbreiten bereitgestellt werden, um die Verfügbarkeit sicherzustellen. Angenommen man möchte jetzt einen neuen Partner zusätzlich versorgen, so entstehen neue Kosten durch die nötige Vergrößerung der Server und der Bandbreite. Die Nachteile dieser Lösung sind offensichtlich schlechte Robustheit und Skalierbarkeit: Der Ausfall des Datenbankservers würde alle darauf angewiesenen Teilnehmer blockieren, im Falle neu hinzukommender Kooperationen entstehen enorme Kosten.

Ein Verbesserungsvorschlag könnte die Verwendung eines verteilten Dateisystems wie z.B. NFS[1] sein. Dies würde zwar die Rechenlast auf die Klientenseite verschieben, das Problem der Bandbreite aber nicht lösen. Unangenehmerweise entsteht dadurch zusätzlich die Frage, wer denn alles gestört wird, wenn ein Dateisystemknoten kurzzeitig nicht mehr erreichbar ist, oder schlimmer: Was passiert, wenn Teile von erzeugten Forschungsdaten bei einem Kooperationspartner abgelegt wurden, der aber die Kooperation plötzlich kündigt (ohne dabei diese Daten herauszugeben)? Die auf der Kehrseite neu entstandenen Nachteile und Gefahren dürften wohl zur Ablehnung des Vorschlags führen.

Diskutiert man die Evolution des Internets in den letzten Jahren, so werden schnell die Begriffe 'filesharing' und 'Peer-to-Peer' genannt. Daraus könnte sich die Idee entwickeln die große Datenbank via 'filesharing' täglich neu zu verteilen. Vor allem das Peer-to-Peer-Protokoll *BitTorrent*, ursprünglich dafür entwickelt neue Linux/Unix Pakete oder auch ganze Distributionen schnell zu verteilen, scheint sich durch seine Optimiertung für diesen Zweck besonders gut anzubieten. Zwar würde damit der Bandbreitenbedarf des publizierenden Servers durch Verteilung deutlich reduziert werden können, aber es wird auch weiterhin die komplette Datei anstatt nur deren Änderungen übertragen. Anzumerken bleibt, dass *BitTorrent* intern auf Dateifragmenten arbeitet, diese aber bisher nur als kleinste logische Einheiten für Übertragung und Verteilung dienen. Die Wiederverwendung von Fragmenten, die bereits einmal geladen wurden, ist derzeit nicht möglich.

An dieser Stelle kommt nun das *IGOR*-Dateisystem ins Spiel. Es kombiniert die Vorteile beider Verbesserungsideen miteinander, ohne sich dabei deren Nachteile einzuhandeln. Wie das genau funktioniert, zeigt der nächste Abschnitt, der sich detailliert mit dem bisherigen Entwurf beschäftigt.

2.2 Das *IGOR*-Dateisystem

Beim *IGOR*-Dateisystem handelt es sich um ein auf Peer-to-Peer-Technik basierendes und komplett im user-space implementiertes overlay-Dateisystem. Der Unterschied zwischen einem herkömmlichen Dateisystem und einem overlay-Dateisystem ist die Notwendigkeit unterliegender Hardware. Typische Dateisysteme wie FAT, EXT, Reiser usw. bilden die logische Sicht von Dateien und Verzeichnissen auf die physikalischen Gegebenheiten von Datenträgern ab. Den Benutzer jedoch interessiert im Allgemeinen nicht, auf welchen Sektoren seine Datei gespeichert wird, solange das Verzeichnis stimmt. Ein overlay-Dateisystem dagegen setzt auf ein bereits existierendes Dateisystem auf. Das heißt, anstatt auf Sektoren oder Ähnlichem zu arbeiten, verwendet ein solches Dateisystem ebenfalls die Abstraktion von Dateien als logische Einheit für seinen eigenen unterliegenden Speicher.

Die Implementation außerhalb des Betriebssystem-Kerns bringt vor allem einen Vorteil mit sich: die beliebte open-source Bibliothek *fuse*[2], die ein Fundament für den Aufbau eigener logischer Dateisysteme bildet, kann verwendet werden. Somit lässt sich die Entwicklung eines eigenen Kern-Moduls, was sehr zeit- und arbeitsaufwändig ist, einsparen. Nebenbei stellt die Implementation des Dateisystems außerhalb des Kerns ein zukunftsweisendes Konzept dar, dass bei aktuellen Mikrokern-Betriebssystem-Architekturen, wie z.B. *L4*, Pflicht ist.

[1] Network File System

[2] http://fuse.sourceforge.net/, 23.06.2006

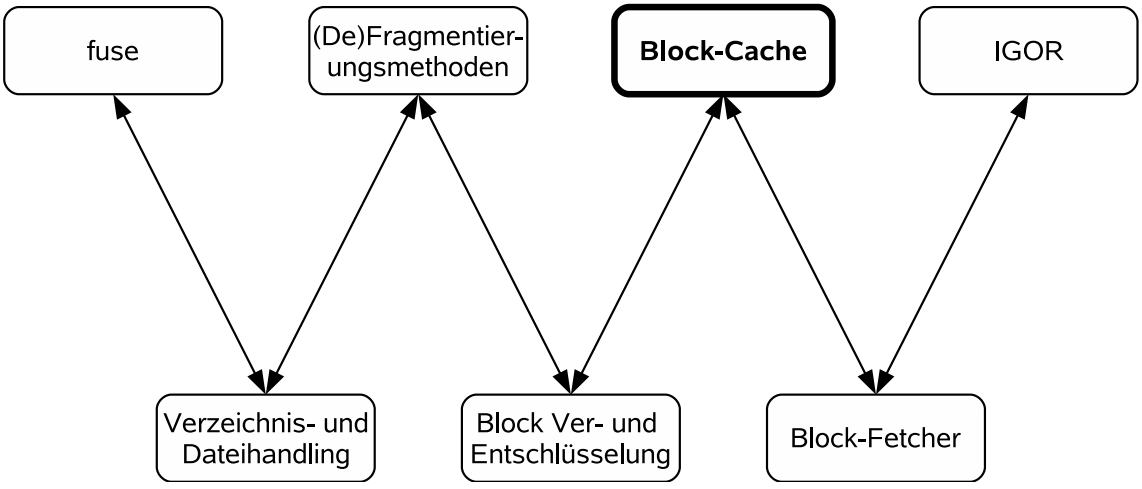


Abbildung 2.1: Aufbau des *IGOR*-Dateisystems

Auf der anderen Seite des Entwurfs befindet sich das von der Universität Karlsruhe (TH) entwickelte Peer-to-Peer-Modul *IGOR*. Dieses Modul dient der Kommunikation und dem Austausch von Dateifragmenten mit anderen *IGOR*-Dateisystem-Instanzen und zeigt sich gleichzeitig auch für den Namen des Dateisystems verantwortlich. Aus dieser Komponente heraus entsteht also der 'verteilt'-Aspekt des Dateisystems. Dateien, bzw. eindeutig gekennzeichnete Fragmente davon, werden durch *IGOR* anderen Teilnehmern des Peer-to-Peer-Netzwerkes zur Verfügung gestellt.

Offensichtlich müssen zwischen der Abstraktion von Dateien, wie sie das Betriebssystem erwartet, und den versandfertigen Fragmenten für das *IGOR*-Modul noch einige Zwischenschritte erfolgen. Abbildung 2.1 gibt zunächst einen Überblick über das als Prozess-Netzwerk konzeptionierte Dateisystem; die Funktionen der einzelnen Komponenten werden dann in den folgenden Unterabschnitten genauer beschrieben.

2.2.1 *fuse*

Wie schon erwähnt, ist *fuse*[3] eine bereits bestehende open-source Bibliothek, die das vom Betriebssystem-Kern definierte Interface des *Virtual File System Layers* implementiert. Auf der anderen Seite wird im Benutzer-Adressraum ein ähnliches Interface bereitgestellt, auf welches dann ein logisches Dateisystem aufgebaut werden kann. Während das native Interface in der Programmiersprache *C* verfasst ist, existieren mittlerweile auch Interfaces für *C++* und *Java*, sowie für diverse Skriptsprachen. Ab der Linuxkern-Version 2.6.14 ist das *fuse*-Modul dann auch offizieller Bestandteil des Linuxkerns sein; für eine breite Verfügbarkeit ist somit automatisch gesorgt.
Mit *fuse* programmierte Dateisysteme können wie gewohnt mit Hilfe des `mount`-Befehls in das virtuelle Dateisystem eingehängt werden und sind somit sämtlichen Applikationen zugänglich.

2.2.2 Verzeichnis- und Dateihandler

Bevor eine Applikation auf eine Datei zugreifen kann, muss sie diesen Wunsch an das Betriebssystem weiterleiten. Dieses lässt sich vom jeweiligen Dateisystem nach

[3]lang: 'filesystem in userspace'

erfolgreicher Fehler- und Sicherheitsüberprüfung ein so genanntes Handle geben und leitet es an die Applikation weiter. Das erhaltene Handle muss von der Applikation für alle zukünftigen Lese- und Schreibzugriffe auf die geöffnete Datei mit angegeben werden. Abgesehen von der Ausweisfunktion beim Lesen oder Schreiben, verknüpft das Dateisystem auch einen Dateizeiger daran, der für den nächsten Zugriff die Position innerhalb einer Datei angibt. Zwei Applikationen können also in der gleichen Datei schreiben, wobei durch die Handles unterschiedliche Positionen angegeben sein können.
Glücklicherweise konvertiert die *fuse*-Bibliothek diese Handles bereits in eindeutige Namen, so dass in der 'Verzeichnis- und Dateihandling'-Komponente auf das Management von Objekten im Dateisystem auf Namensbasis stattfinden kann. Beispielsweise sind Datein, Verzeichnisse und Verknüpfungen solche Objekte, die hier verwaltet werden.

2.2.3 De-Fragmentierungsmethoden

Ähnlich wie echte Dateisysteme einzelne Dateien auf ggf. mehrere Sektoren verteilen, arbeitet auch das *IGOR*-Dateisystem auf einer kleineren Einheit als Dateien: den so genannten Blöcken. Aber im Gegensatz zu Dateisystemen, die Dateien in gleich große Stücke schneiden, um sie z.B. auf Sektorgröße zu bringen, dürfen Blöcke in diesem Dateisystem unterschiedliche Größen haben. Genau genommen ist sogar geplant, diese Schnitte an logischen Positionen vorzunehmen. Dies bedeutet, dass beispielsweise eine Textdatei in ganze Seiten, eine Datenbank in Datensätze oder ein Video in Frames[4] zerschnitten wird. Jedes einzelne der daraus entstehenden Fragmente bekommt durch ein kryptographisches Hash-Verfahren eine pseudoeindeutige[5] Kennung zugewiesen. Ein Fragment mit Kennung wird dann Block genannt. Man beachte, dass nach diesem Schritt keinerlei Dateien mehr existieren, es gibt nur noch Blöcke. Das Beispiel eines Videofilmes, der auf zwei unterschiedliche Arten geschnitten ist, verdeutlicht den sich daraus ergebenden Vorteil am besten: die Videodaten des Filmes müssen nur ein einziges Mal gespeichert werden, während zwei unterschiedliche 'Wiedergabelisten' daraus die zwei Filme entstehen lassen.
Das (De)Fragmentierungs-Modul dient der geschickten Zerlegung und Wiedervereinigung von Dateien in Blöcke und umgekehrt. Alle folgenden Komponenten benötigen keinerlei Kenntnisse mehr über den Inhalt oder den Zweck dieser Blöcke.

2.2.4 Block Ver- und Entschlüsselung

Wie aus der Bezeichnung dieses Moduls bereits ersichtlich, ist es für eine Ver- und Entschlüsselung auf Blockebene zuständig. Das bedeutet, dass jeder Block einzeln verschlüsselt wird. Demnach kann jeder Block mit einem eigenen Schlüssel chiffriert werden. Ein diesbezglicher Vorschlag ist, zufällig einen bereits existierenden Block zu wählen und eine `xor`-Operation mit dem neuen Block durchzuführen.
Diese Komponente hat also die Aufgabe, Blöcke, die in Richtung Block-Cache wandern, zu verschlüsseln und Blöcke aus der Gegenrichtung zu entschlüsseln.

[4] Viele Videoformate bestehen aus einer Verkettung so genannter Frames (z.B. einzelner Bilder).
[5] Schlüssel, die mit einer so hohen Wahrscheinlichkeit einzigartig sind, so dass sie als einzigartig angenommen werden können.

2.2.5 Block-Cache

Das Block-Cache-Modul stellt die Speichereinrichtung dieses Dateisystems dar. Blöcke, die gerade verschlüsselt worden sind, oder die entschlüsselt werden sollen, werden hier ins unterliegende Dateisystem gespeichert oder von dort gelesen. Auch kann es sein, dass Blöcke von einer anderen Dateisystems-Instanz über das später folgende *IGOR*-Modul angefordert wurden und jetzt gelesen werden müssen. Der Block-Cache ist sozusagen der Mittelpunkt des Dateisystems. Anfragen kommen aus beiden Richtungen, vom lokalen System und entfernten Netzwerkteilnehmern, durch das Prozess-Netzwerk hierher. Wurde ein Block von einer entfernten Instanz geladen, so wird er zusätzlich in deren lokaler Block-Cache-Komponente abgelegt. Eine *on-demand*[6] Blockverteilung entsteht: Häufig benötigte Blöcke sind durch die Verteilung schneller und mit höherer Bandbreite verfügbar. Ein einmal heruntergeladener Block, sofern nicht aus dem Cache verdrängt, muss nicht erneut geladen werden.

Wie die Bezeichnung Cache bereits vermuten lässt, handelt es sich hierbei nicht um ein persistentes Speichersystem. Nicht mehr gebrauchte Blöcke werden stattdessen irgendwann aus dem Zwischenspeicher wieder entfernt. Hierbei gilt es zu beachten, dass lediglich Kopien gelöscht werden dürfen, nicht aber die Originaldatei bzw. deren zugehörige Blöcke. Unter anderem wird in Abschnitt 2.3 nach Lösungen für diese beiden Kernprobleme gesucht.

2.2.6 Block-Fetcher

Der Block-Fetcher wird benötigt, sobald eine Applikation einen Block lesen möchte, der sich nicht (mehr) im lokalen Cache befindet. Dieses Fragment muss dann zunächst über das Peer-to-Peer-Netzwerk geladen werden. Der Block-Fetcher speichert zu diesem Zweck Informationen dezentral und verteilt sie in das gesamte Netzwerk. Sie besagen, bei welchen Teilnehmern Kopien des gesuchten Blocks verfügbar sind. Letztendlich wird ein entsprechender Download-Auftrag generiert und an *IGOR* weitergeleitet.

2.2.7 IGOR

Wie bereits mehrfach erwähnt, handelt es sich bei *IGOR* um ein Peer-to-Peer-System, was auch in anderen Projekten zum Einsatz kommt und sich in ständiger Weiterentwicklung befindet. Beispielsweise ist das in [CrKF04] und [KuCF05] beschriebene, auf *IGOR* basierende *Videgor*[7] ein Plugin für einen Software-Videorekorder. Es ermöglicht ihm, die Aufzeichnungswünsche seiner Benutzer so im Netzwerk zu verteilen, dass mehrere, gleichzeitig ausgestrahlte Sendungen parallel aufgenommen werden können. Alle Mitglieder haben anschließend Zugriff auf die aufgenommenen Sendungen, wodurch bereits gezeigte Sendungen noch nachträglich aufgezeichnet werden können.
Im Falle des Dateisystems werden von *IGOR* Blocktransfers und Suchanfragen verwaltet und die Ergebnisse in aufbereiteter Form an den Anfragenden weitergeleitet. *IGOR* ist also für den Austausch von Blöcken und der Kommunikation innerhalb des verteilten Dateisystems zuständig.

[6] Blöcke werden nur bei Anforderung geladen.
[7] http://www.videgor.net, 13.07.2006

2.3 Analyse vergleichbarer Systeme

Nach der kurzen Vorstellung der Konzeption des geplanten *IGOR*-Dateisystems, kann nun mit der eigentlichen Aufgabe dieses Kapitels begonnen werden: Die Analyse vergleichbarer Systeme.

Dieser Abschnitt befasst sich mit bereits real existierenden oder geplanten Systemen, die eine Ähnlichkeit zum geplanten IGOR-Dateisystem haben erkennen lassen. Im Hinblick auf die Zielsetzung dieser Arbeit wird nach der Systembeschreibung hauptsächlich der Teil dieser Systeme genauer analysiert, der dem Block-Cache am nächsten kommt.

2.3.1 Freenet - A Distributed Anonymous Information Storage and Retrieval System

Das in [CMHS+02] beschriebene Freenet weist in vielerlei Hinsicht große Ähnlichkeit mit dem *IGOR*-Dateisystem auf. Zunächst einmal sei hier die Wahl eines Peer-to-Peer-Netzwerkes als gemeinsame Basis erwähnt. Freenet verspricht seinen Nutzern hohe Anonymität innerhalb des Netzwerkes. Dies geschieht dadurch, dass alle teilnehmenden Knoten auf genau die gleiche Art und Weise antworten, wenn sie nach Inhalten befragt werden. Dabei spielt es keine Rolle, ob der befragte Knoten die jeweilige Information selbst kennt oder sie von einem anderen Knoten vorher erfragen musste. Somit ist sichergestellt, dass der Ursprung einer Information nur schwer festgestellt werden kann.

Eine weitere deutliche Übereinstimmung findet sich in der Wahl des Schlüsseltyps. Genau wie das IGOR-Dateisystem verwendet auch Freenet pseudo-einzigartige Schlüssel, die aus einem kryptischen Hash-Verfahren gewonnen werden. Im Falle von Freenet handelt es sich um das als 'SHA1 Standard' publizierte Hashverfahren, wie es in [X9.397] nachgelesen werden kann. Im Gegensatz zum IGOR-Dateisystem, in dem zum jetzigen Zeitpunkt nur ein einziger Schlüsseltyp, nämlich der Blockidentifikator (kurz BlockId) vorgesehen ist, kommen in Freenet eine Vielzahl von unterschiedlichen Schlüsseln zum Einsatz. Unter anderem gibt es so genannte *Keyword signed keys (KSK)*, die aus einer beschreibenden Zeichenkette generiert werden, und die *content hash keys (CHK)*, die dem gleichen Zweck wie BlockIds dienen.

Die für diese Arbeit interessanteste Übereinstimmung ist jedoch folgende: Um im Freenet-Netzwerk Anfragen nicht jedesmal bis zur Informationsquelle weiterleiten zu müssen, besitzt jeder teilnehmende Knoten einen Cache zur Ablage gesammelter Informationen. Diese können selbst erfragte Informationen sein oder solche, die von anderen Knoten erfragt und durch diesen vermittelt wurden. Freenet verwendet für seinen Cache die *Least Recently Used (LRU)* Verdrängungsstrategie. Das bedeutet, dass diejenigen Informationen, die am längsten nicht mehr abgerufen wurden, entfernt werden, um Platz für neue Daten zu schaffen. Leider finden sich weder in [CSWH] noch [CMHS+02] genauere Informationen über die Implementation des Caches.
Glücklicherweise hat das Freenet-Projekt jedoch bereits eine lauffähige Version herausgebracht, in der ein Cache-Verzeichnis spezifiziert werden kann. Aus einem leicht gefüllten Cache lassen sich dann bereits folgende Informationen ableiten:

1. Es gibt 256 Unterverzeichnisse namens $00 - ff$. In jedes dieser Unterverzeichnisse werden Dateien, deren Namen unter anderem aus deren Schlüsseln gewonnen werden, einsortiert. Es ist jedoch nicht ersichtlich, aus welchem Teil des Schlüssels das Unterverzeichnis abgeleitet wurde.

2. Zusätzlich enthalten die Dateinamen am Ende vier Ziffern, die wohl der Identifikation des Schlüsseltyps (KSK, CHK usw.) dienen.

3. Im Cache-Verzeichnis selbst existiert eine Datei namens `index`, die Suchpfade auf Schlüssel zu enthalten scheint.

4. Zuletzt gibt es ein Verzeichnis `temp`, das ebenfalls eine Hand voll Schlüssel enthält. Die dort befindlichen Dateien scheinen gerade heruntergeladen zu werden.

2.3.2 OceanStore - An Extremly Wide-Area Storage System

Das OceanStore Projekt ([BCEG+00]) hat sich zum Ziel gesetzt ein kommerzielles System zu schaffen, das allgegenwärtig sein soll und persistenten Speicherplatz zur Verfügung stellt. Anstatt physisch gut geschützte und rechenstarke Server zu betreiben, wie diese Aufgabe noch vor wenigen Jahren angegangen worden wäre, wählten die Entwickler den Weg der Redundanz. Die Daten werden auf viele, über den ganzen Globus verstreute Server verteilt. Als Basis soll auch hier ein Peer-to-Peer-Overlay-Protokoll zum Einsatz kommen. Der OceanStore Prototyp *Pond*, wie in [REGW+03] beschrieben, verwendet hierfür das *Tapestry*-Protokoll. Zusätzlich zu den Servern soll, nach dem Willen der Entwickler, auf jedem beliebigen Gerät ungenutzter Speicher dazu verwendet werden können, Daten zwischenzuspeichern. Aus der Trennung von Informationen von festen Orten entstehen so genannte 'nomadic data' – mobile Daten, die an vielen Orten zwischengespeichert wurden. Neben der Datenverteilung auf den Servern ist dies eine weitere mächtige Stütze der Verlässlichkeit. Selbst wenn alle Server ausfallen sollten, bedeutet dies nicht zwangsweise einen Datenverlust.

Die Konsistenz aller Objekte dieses Dateisystems wird durch Versionierung erreicht. Genau wie beispielsweise im *Concurrent Versions System (CVS)* oder in *Subversion (SVN)*[8] besitzt jedes Objekt eine Version, die bei Änderung automatisch inkrementiert wird. Effektiv wird also ein neues Objekt erzeugt, anstatt ein bestehendes Objekt explizit zu überschreiben. Die nun veraltete Version verliert ihren Permanent-Status und darf gelöscht werden.

Ähnlich wie in Freenet und im geplanten *IGOR*-Dateisystem sind alle Objekte durch einen globalen pseudo-eindeutigen Hash-Schlüssel gekennzeichnet, hier *GUID (globally unique identifier)* genannt. Jedoch existieren auch hier unterschiedliche Objekttypen. Nur-Lese-Objekte, die sich z.B. in den Caches aufhalten, können durch Neuberechnung ihres Hashwertes und anschließenden Vergleich mit der GUID verifiziert werden. Um den Besitzer eines Objektes zu verifizieren, wird zusätzlich zum Objektinhalt auch noch der Objektname und öffentliche Teil des Besitzer-Codesschlüssels[9] mitgehasht. Wie bei Freenet so soll auch hier der SHA1 Standard als kryptographisches Hash-Verfahren verwendet werden.

[8] Beides open-source Systeme zur Versionsverwaltung von Dateien. (nach Wikipedia – http://de.wikipedia.org/wiki/Subversion_(Software) bzw. http://de.wikipedia.org/wiki/CVS – 13.06.2006)

[9] public key

Leider findet sich in den Publikationen [BCEG+00] und [REGW+03] kaum mehr als die Erwähnung der Existenz eines Caches. Eine Fußnote besagt jedoch, dass OceanStore genau wie Freenet die LRU-Verdrängungsstrategie implementiert.

2.3.3 Datagrid Systeme

Die beiden Hauptideen hinter so genannten Datagrid Systemen sind verteiltes Speichern großer Datenmengen und verteiltes Rechnen. An einem solchen Netzwerk nehmen also prinzipiell Speicher- und Rechenknoten teil. Letztere Knoten werden hier jedoch nicht diskutiert, da sie für diese Arbeit nur von untergeordnetem Interesse sind. Auch bleibt zu erwähnen, dass für derartige Massenspeichersysteme weiterhin häufig Magnetbänder verwendet werden. Dies ist auf die geringeren Kosten und die gute Altersbeständigkeit dieses Mediums zurckzuführen. Verglichen mit Festplatten macht die deutlich schlechtere Antwortzeit von Magnetbändern ein zusätzliches Cache System wünschenswert. Weltweit gibt es mittlerweile mehrere Implementationen von Datagrid Systemen in verschiedenen Stadien des Fortschritts: Die bekanntesten sind *apgrid* im asiatisch/pazifischen Raum und *GridKa*, ein Datagrid System für das Deutsche Forschungsnetzwerk (DFN) und die Kernforschungseinrichtung CERN. Bereits ab dem Jahr 2007 soll *GridKa* geschätzte vier bis acht Petabyte an Daten jährlich für das CERN speichern und angeschlossenen Forschern die Auswertung erleichtern.

In [TMSM+01], [TMSS+03] und [TMSM+04] wird das *apgrid* System vorgestellt. Als ehrgeiziges Ziel sind Datenverbindungen bis in den TB[10]-Bereich vorgesehen, die durch hohe Parallelität und redundantes Speichern erreicht werden sollen. Bei deterministisch erzeugbaren Daten sollen zusätzlich Programm und Aufrufparameter mitgespeichert werden, um solche Daten bei Bedarf neu generieren zu können. Die genannten Berichte enthalten allerdings keinerlei technische Daten über die verwendete Speichertechnik, was deren weitere Analyse hinfällig macht.

Jiang et al. schlagen in [JiZh03] eine dem Titel nach effiziente Cachestrategie für Datagrid Systeme vor: *least value based on caching time (LVCT)*. Ähnlich wie bei LRU wird eine nach letzter Zugriffszeit geordnete Liste von Dateien im Cache benötigt. Allerdings ist bei LVCT die so genannte *caching time* ein Wert, der durch das Akkumulieren der Dateigrößen von zwischenzeitlich referenzierten Dateien errechnet wird. Für die n-te Datei dieser Liste gilt also $\textit{caching time} = \sum_{i=0}^{n} sizeof(file_i)$. Eine Bewertungsfunktion, in die zusätzlich noch die Bereitstellungskosten[11] für eine Datei mit einfließt, trifft anschließend die Entscheidung, ob neue Dateien überhaupt in den Zwischenspeicher eingelagert werden sollten. Bedenkt man, dass bei der Referenzierung einer einzelnen Datei die *caching time*-Werte von $n - 1$ Dateien aktualisiert werden müssen, so entstehen ernste Zweifel an der Effizienz dieses Verfahrens.

Eine ähnliche Methode beschreibt [HaDK99], in der als Ausgangspunkt erneut die LRU-Verdrängungsstrategie herangezogen wird. *Object LRU (OLRU)* berücksichtigt zusätzlich zum Zeitpunkt der letzten Referenzierung noch die Anzahl der Referenzierungen, die Objektgröße selbst und in einer Menge von Objekten deren Position sowie deren Anzahl. All diese Werte werden zusammen mit gewichtenden Faktoren

[10] Terabytes pro Sekunde

[11] Beispielsweise zeitliche Kosten die durch das Wechseln, Spulen und Auslesen eines Magnetbandes entstehen.

in einer Bewertungsfunktion aufsummiert, die anschließend von einem genetischen Algorithmus auf Leistung optimiert wird. Anstatt wie bei LRU genau soviele Objekte vom Ende der Liste zu entfernen, bis ein neu einzulagerndes Objekt genügend Platz findet, selektiert jetzt die Bewertungsfunktion die zu verdrängenden Elemente. Laut den Autoren lassen sich mittels dieses Verfahrens Leistungssteigerungen bis zu 2% gegenüber herkömmlichem LRU erzielen.

Zusammenfassend lässt sich sagen, dass DataGrid Systeme anders als Freenet und dem geplanten *IGOR*-Dateisystem Daten dauerhaft speichern. Die verwendeten Caches dienen hierbei in erster Linie dem gewohnten Zweck ein langsameres Massenspeichersystem zu verdecken. Anstatt Fragmente von Dateien zu verwenden, arbeiten sowohl *apgrid* als auch *GridKa* auf ganzen Dateien als kleinste Granularität.

2.3.4 Fazit

Offensichtlich bestehen zwischen dem geplanten *IGOR*-Dateisystem und den in 2.3.1, 2.3.2 und 2.3.3 beschriebenen Systemen *OLRU* und *LVCT* einige Gemeinsamkeiten. So setzten Freenet und OceanStore genau wie das *IGOR*-Dateisystem auf Peer-to-Peer-Netzwerke auf und verwenden kryptographische Hash-Verfahren, um pseudo-eindeutige Schlüssel für Blöcke zu erzeugen. In den unterschiedlichen Systemen sind Blöcke entweder reine Dateifragmente oder sie enthalten zusätzliche Metadaten wie Besitzerangaben, Zugriffsrechte etc. Während Freenet und Oceanstore vor allem technische Gemeinsamkeiten aufweisen, kommt die Zielsetzung der Datagrid Systeme dem *IGOR*-Dateisystem am Nächsten: In einer *write-once, read-many*[12] Umgebung sollen enorme Mengen an Daten gespeichert werden, um sie gleichzeitig effizient in einem Netzwerk zu verteilen. Die dadurch entstehende Redundanz soll für hohe Transferraten sorgen und gleichzeitig für hohe Verfügbarkeit und Ausfallsicherheit stehen.

Die auf der technischen Seite gewonnenen Erkenntnisse sind sehr dürftig. Mit *OLRU* und *LVCT* wurden in [HaDK99] und [JiZh03] zwei viel versprechende Alternativen zur klassischen *Least-Recently-Used*-Verdrängungsstrategie eingeführt. Im Vergleich zu *LRU* konnten in den Experimenten der Autoren bessere Cache-Trefferraten erzielt werden. Allerdings sind diese Verbesserungen mit einem nicht zu vernachlässigenden Rechenaufwand, der zur Laufzeit entsteht, erkauft. Aus diesem Grund wird das *IGOR*-Dateisystem, wie auch Freenet, Oceanstore und GridKa, zunächst auf die klassische *LRU*-Verdrängungsstrategie setzen.

2.4 Anforderungen an den Block-Cache

Bevor mit dem Entwurf der Cache-Komponente begonnen werden kann, sollten zunächst einmal die Anforderungen genau spezifiziert werden. Aber auch nicht weniger wichtig ist die ständige Erinnerung an die große Menge zu verwaltender Daten. Daher versucht der folgende Abschnitt 2.4.1 zunächst die Größenordnungen abzustecken, um dann später in 2.4.2 Vorgaben für die Komponente aufstellen zu können.

[12] Dateien werden ein einziges Mal geschrieben, danach nur noch von vielen gelesen.

2.4.1 Dimensionierung

Es wurde bereits öfters erwähnt, dass der Cache für heutige Maßstäbe sehr groß ist, ohne genaue Angaben darüber zu machen. Im Idealfall muss der Cache mindestens den Gesamtumfang der gemeinsamen Daten haben, um tatsächlich das vorgespielte persistente Speichersystem sein zu können. Im Hinblick auf das in 2.1 beschriebene Anwendungsszenario wird schnell klar, dass es sich hierbei wenigstens um Gigabytes, eher aber um Terabytes handelt. Als Größenordnung für diese Arbeit sei also der einstellige Terabytes Bereich genannt, allerdings ohne dabei eine feste obere Grenze vorzugeben. Der Grund dafür ist das Entwurfs-Ziel den Cache ausreichend effizient zu gestalten, so dass er auch für zukünftige Speicherkapazitäten nutzbar bleibt. Denn ähnlich wie das als Moore's Law (siehe Veröffentlichung [Moor65]) bekannte Wachstum der Prozessorleistung, gibt es auch ein ständiges Kapazitäts-Wachstum der Speichermedien. Wo es technisch möglich ist, sollte also auf feste Grenzen verzichtet werden. Als Beispiel wie man es nicht machen sollte, sei hier das bekannte Dateisystem FAT mit seinen diversen Versionen[13], die es immer wieder auf den neusten Stand bringen mussten, genannt.

Eine weitere wichtige Kennziffer ist die Größe der Blöcke. Wie bereits erwähnt, dienen sie unter anderem als Einheit bei der Übertragung, die im Fehlerfall wiederholt werden müsste. Auch sollte bei der Änderung einiger Bytes einer Datei nicht automatisch immer die ganze Datei erneut übertragen werden müssen. Kleinere Blockgrößen sind hier also zu bevorzugen, allerdings steigen damit die Verwaltungskosten. Als Kompromiss gelte daher eine durchschnittliche Größe von 64 Kilobytes, die aus Gründen der Flexibilität erneut nicht in feste Grenzen gezwängt wird.
Demnach muss der Cache mit einer Block-Anzahl im dreistelligen Millionen-Bereich problemlos zurechtkommen. Außerdem sollten die Cache-Operation, wie z.B. Re-Referenzierung, deutlich unter linearen Aufwand liegen.

2.4.2 Anforderungen und Eigenschaften

Neben den bekannten Hauptaufgaben eines Caches, Einträge hinzuzufügen, zu aktualisieren und zu verdrängen, kommt vor allem ein auf Prioritäten gestütztes Auftragsmanagement hinzu. Alle den Cache erreichenden Anfragen und Befehle sind mit Prioritäten versehen, die bei deren Abarbeitung über die Reihenfolge entscheiden. Jeder dieser Aufträge soll letztendlich durch einen Block quittiert bzw. durch eine Fehlernachricht abgeschlossen werden. Auch verfügt jeder neu eingehende Block über eine Speicherpriorität, mit der er in den Cache eingetragen werden soll. Diese Zusatzangabe soll später von der Verdrängungsstrategie berücksichtigt werden. Zum Beispiel könnten so Elemente markiert werden, die nicht automatisch verdrängt werden dürfen.

In allen Aufgabenbereichen ist die Geschwindigkeit die wichtigste Eigenschaft. Abgesehen von kurzen Startzeiten, gilt es in erster Linie die Cache-Operationen "Finden", "Referenzieren", "Hinzufügen" und "Verdrängen" so zu gestalten, dass sie möglichst schnell ablaufen. Aus dem Abschnitt 2.4.1 geht bereits hervor, dass linearer Aufwand für die erwartete Blockmenge voraussichtlich viel zu langsam sein wird, $O(log_2(n))$ sollte also nach Möglichkeit nicht überschritten werden.

[13] Die Versionsgeschichte von FAT ist bei Wikipedia (http://de.wikipedia.org/wiki/File_Allocation_Table, 13.07.2006) verfügbar.

Aber auch hohe Fehlertoleranz ist von großer Bedeutung. Da es sich beim *IGOR*-Dateisystem um ein Dateisystem im Benutzer-Adressraum handelt, ist es nicht auszuschließen, dass andere Anwendungen die Cache-Daten im lokalen Dateisystem ändern. Diese Änderungen sollten spätestens beim Abruf des Blocks aus dem unterliegenden Dateisystem erkannt und korrigiert werden. Korrigieren bedeutet an dieser Stelle, den defekten Block neu aus dem Peer-to-Peer-Netzwerk zu laden. Neben diesen von unwissenden oder maliziösen Programmen verursachten Schäden gilt es zusätzlich auch Systemabstürze zu berücksichtigen, denn sie könnten die Konsistenz von Cache-Datenstrukturen beeinträchtigen. Letztlich kommt demnach noch ein Konsistenz-Prüfmodus ähnlich dem Dateisystem-Prüfprogrammen `fsck` für Instandsetzungs- und Wartungsaufgaben hinzu.

Die in diesem Abschnitt gefundenen Anforderungen gilt es beim Entwickeln der Block-Cache-Komponente zu berücksichtigen. Vor allem die hohen Geschwindigkeits-Anforderungen werden dabei die zentrale Rolle spielen. Daher wird sich das folgende Kapitel 3 intensiv mit dem Entwurf dieses Moduls beschäftigen.

3. Entwurf

Nachdem in Kapitel 2 vergleichbare Systeme analysiert und daraus Anforderungen an den Block-Zwischenspeicher herausgearbeitet wurden, geht dieses Kapitel der Frage nach, wie ein solcher Cache am besten zu implementieren ist. Bevor jedoch damit begonnen werden kann, wird in 3.1 zunächst ein Einblick in die bereits bestehende Gesamtarchitektur des *IGOR*-Dateisystems gegeben. Später untergliedert sich dann der Entwurf in zwei Teile: Als erstes beschäftigt sich Abschnitt 3.2 mit der Umsetzung der LRU-Verdrängungsstrategie. Anschließend werden in 3.3 verschiedene Ideen zur Konstruktion des Block-Zwischenspeichers und dessen Einbettung in das Gesamtsystem analysiert.

3.1 Architektur des Gesamtsystems

Im Hinblick auf die heute bereits verfügbaren Dualcore- und die in naher Zukunft erwarteten Multicore- oder Cell-Prozessoren ist ein wachsender Grad an echter Parallelverarbeitung zu erwarten. Tatsächlich scheint auch für das *IGOR*-Dateisystem mit seinen verschiedenen Komponenten, durch die einzelne Blöcke durchgereicht werden sollen, ein hohes Potential für Parallelverarbeitung zu bestehen. Aus diesen Gründen wurde das Gesamtsystem als "Multithreading"-Applikation vorgesehen, in dem jede Komponente durch einen oder mehrere Threads betrieben werden soll.

Mit einer solchen Entscheidung handelt man sich im Allgemeinen die Erschwerung von Fehlersuchen und, wenn Semaphoren oder Locks[1] hinzukommen, eine ergiebige Fehlerquelle ein. Das gilt vor allem dann, wenn es sich dabei um ein Projekt handelt, an dem mehrere Entwickler beschäftigt sind. Wenn diese Entwickler zusätzlich, auf Grund der Komplexität, nur selten über vollständige Systemkenntnis verfügen, steigt die Wahrscheinlichkeit von Fehlern erneut deutlich an.

Um diesem Effekt entgegen zu wirken, basiert das Gesamtsystem auf einem so genannten Prozess-Netzwerk, einem Modell, das erstmals vor über dreißig Jahren in [Kahn74] und [KaMa77] formuliert wurde. Dieses Modell sieht vor parallel ablaufende Prozesse durch Nachrichten führende Kommunikations-Kanäle miteinander zu

[1] Synchronisationsmechanismen für Threads

verbinden. Die hier verwendete Variante basiert auf Threads. Jeder dieser Threads besitzt eine eigene, nach Zeit sortierte Warteschlange an Nachrichten, die abgearbeitet werden sollen. Außerdem kann er selbst Nachrichten in den Warteschlangen der anderen Threads ablegen. Alle Nachrichten enthalten die Adresse einer Zielkomponente für die sie bestimmt sind. Letztendlich benötigen die Komponenten lediglich eine Methode, genannt `handleMessage(...)`, die von den Threads zur Verarbeitung eingehender Nachrichten durchlaufen werden muss. Diese Technik ist ähnlich der globalen Ereignisliste, wie sie in Diskreten Event Simulatoren, beispielsweise OMNet++[2], verwendet wird. Allerdings mit dem Unterschied, dass hier jeder Thread für seine Module eine eigene Ereignis- bzw. Nachrichtenliste besitzt. Tatsächlich ist es durch einen Aufrufparameter möglich, alle Komponenten einem einzigen Thread zuzuweisen, wodurch wieder der gewohntere sequentielle Ablauf entsteht, was z.B. für die Fehlersuche sehr praktisch ist.

Diese Architektur kommt weitgehend ohne Semaphoren aus, denn Synchronisation kann durch einfachen Nachrichtenaustausch geschehen und erleichtert deutlich die Zusammenarbeit mehrerer Entwickler. Diese müssen jetzt lediglich die Schnittstellen von kooperierenden Komponenten kennen. Die somit entstehende hohe Modularität wirkt sich positiv auf die Wartbarkeit und Wiederverwendbarkeit aus.

Es bleibt anzumerken, dass sich alle Threads in einem gemeinsamen Adressraum befinden und es daher möglich ist, auch Zeiger im Prozess-Netzwerk zu versenden. Hieraus könnten sich allerdings inkonsistente Speicherzustände ergeben, die durch eine einfache Konvention vermieden werden können: Prozesse, die einen Zeiger verschicken, dürfen anschließend nicht mehr auf die entsprechende Struktur schreiben.

3.2 Umsetzung der *Least Recently Used* Strategie

Es gibt diverse denkbare Wege einen Cache der geplanten Größenordnung zu implementieren. Längst nicht immer lassen sich jedoch auch die geforderten Eigenschaften erfüllen. Daher beschäftigt sich dieser Abschnitt mit mehreren Konstruktionsideen und bewertet diese im Anschluss.

3.2.1 Zielsetzung

Zur Durchsetzung von reinem LRU müssen mindestens ein Blockidentifikator (eine BlockId) sowie ein Zeitstempel pro eingelagertem Block gespeichert werden. Der verwendete kryptographische Hash-Algorithmus wird voraussichtlich 160Bit-Kennungen erzeugen; gerade noch ausreichend präzise Zeitinformation könnten bereits mit 4 Bytes auskommen. Ein "Datensatz" besteht also wenigstens aus 24 Bytes. Berechnet man entsprechend den Hauptspeicherbedarf für 100 Millionen Blöcke, wie in 2.4.2 als untere Grenze gefordert, erreicht man bereits $2,2$GB! Man beachte, dass in dieser Rechnung noch keinerlei Zuschläge durch dynamische Datenstrukturen oder Speicheranordnungen[3] berücksichtigt wurden.
Abgesehen von den damit verbundenen langen Ladezeiten scheitert eine simple Blockliste oder Blocktabelle am hohen Hauptspeicherbedarf. Nebenbei wurde in 2.4.2 auch noch die Speicherung von Prioritäten gefordert, was etwa ein weiteres

[2] verfügbar unter http://www.omnetpp.org, 13.07.2006

[3] Zur Optimierung der Ausführungsgeschwindigkeit werden in heutigen Systemen oftmals 'Löcher' im Speicher gelassen, die aber dennoch Speicherplatz kosten (Stichwort 'alignment').

Byte Kosten pro Block nach sich zieht. Zusätzlich wäre auch noch die Größe eines Blockes eine weitere wünschenswerte Kennziffer, die zur Auswahl von zu verdrängenden Einträgen berücksichtigt werden könnte.

Berücksichtigt man den Verlust der gesamten Tabelle bei einem Absturz, so ist es offensichtlich ungeschickt, alle Daten im Hauptspeicher zu halten – andere Modelle sind gefragt. Diese sollten wenigstens Hash-Kennung, Zeitstempel und Priorität speichern, die Blockgröße ist optional. Zumindest die ältesten Einträge der Datenstruktur sollten nach Zeitstempeln sortiert sein, um LRU durchsetzen zu können. Hierbei ist die Priorität nicht zu vergessen. Denn es muss möglich sein, Blöcke mit bestimmter Priorität für die Verdrängung zu selektieren. Bei n eingelagerten Blöcken ist für alle Cache-Operationen $O(log_2(n))$ eine gerade noch akzeptable Kostengrenze.

3.2.2 Datenbank

Eine erste Idee ist die Verwendung einer bereits bestehenden Datenbank Software. Neben "Einfügen", "Suchen" und "Entfernen" gehört auch das "Sortieren" zu der angebotenen Funktionalität heutiger Datenbanksysteme. Alle benötigten Cache-Operationen lassen sich also direkt auf eine Datenbank abbilden. Folgende Tabelle 3.1 beschreibt die minimale Form, wie sie für einen Block-Zwischenspeicher nötig wäre.

BlockId	Zeitstempel	Priorität
...	...	...

Tabelle 3.1: Tabelle für eine LRU-Datenbank

Ein weiterer Vorteil ist die oftmals bereits integrierte Unterstützung mehrerer paralleler Anfragen, wodurch die geplante Nebenläufigkeit im *IGOR*-Dateisystem weiter unterstützt wird. Auch im Hinblick auf die Konsistenz sind Datenbanken von Vorteil: Im Falle eines Absturzes ist möglicherweise die Datenbank nicht betroffen, die Daten bleiben also konsistent. Um die Konsistenz wirklich komplett durch eine transaktions-basierte Datenbank schützen zu lassen, müsste zusätzlich auch noch der gesamte Block-Inhalt in die Tabelle mit aufgenommen werden. Ob ein handelsübliches Datenbanksystem eine mehrere Terabytes große Tabelle noch effizient handhaben könnte, ist allerdings äußerst fraglich.

Gegen eine Datenbank spricht vor allem die Länge der Tabelle. Diese ist mit guten 100 Millionen Zeilen sehr lang, was das Suchen eines bestimmten Datensatzes evtl. sehr aufwändig machen könnte. Denn beim Suchen müsste ein arbeitsintensiver Vergleich der BlockIds, die in der Datenbank als Zeichenkette modelliert werden müssten, durchgeführt werden. Auch kann im Allgemeinen eine Datenbank nicht in einer sortierten Form abgelegt werden. Das heißt, dass für jeden zu verdrängenden Block die gesamte Tabelle erneut nach Zeitstempeln sortiert werden müsste!

Durch die geforderten Prioritäten bietet sich ein Verbesserung geradezu an: Anstatt alle Blöcke in einer einzigen Tabelle abzulegen, könnte für jede Priorität eine eigene Tabelle erzeugt werden. Das geforderte Ziel, einen Block bestimmter Priorität für die Verdrängung auszuwählen, wäre somit sogar noch vereinfacht. Solange allerdings keine Komplexitätsabschätzungen verfügbar sind und die Verteilung der Blöcke über die verschiedenen Prioritäten unbekannt ist, hat es keinen Sinn über weitere Optimierungen zu spekulieren.

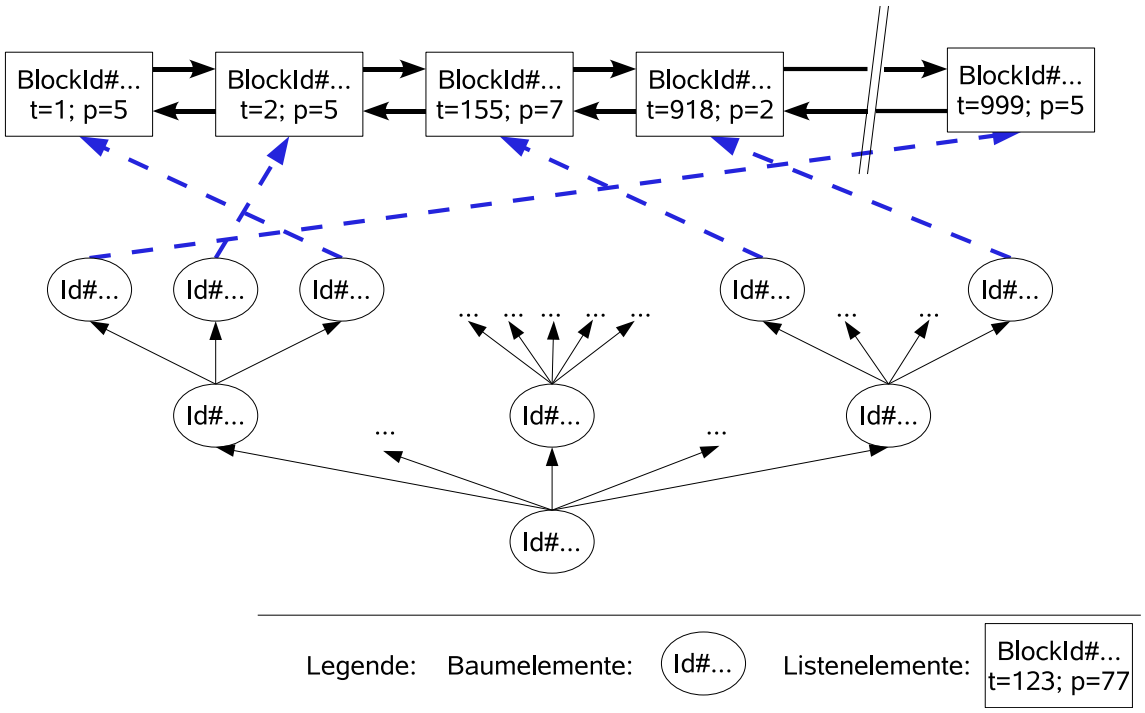


Abbildung 3.1: Datenbank-Vorschlag: B-Baum zeigt auf doppelt verkettete Liste

Fazit

Obwohl alle benötigten Cache-Operationen direkt auf eine Datenbank abgebildet werden könnten, was eine Menge Arbeit einsparen würde, ist hauptsächlich der Speicherbedarf und das Zeitverhalten eines Datenbankservers unklar. Es müsste demnach zunächst mit Hilfe eines Benchmarks der Aufwand für die vier benötigten Funktionen ermittelt werden. Da jedoch allgemeine Datenbanksoftware nicht speziell für dieses Problem optimiert wurde, ist es unwahrscheinlich, dass die geforderten Komplexitäten eingehalten werden könnten.

3.2.3 Eigene Datenbank basierend auf einem B-Baum

Von den wenigen Datenstrukturen, die überhaupt logarithmischen Aufwand garantieren können, sind Bäume die bekanntesten. Die erste Idee einer eigenen Datenbank sieht vor, einen B- oder B*-Baum[4] mit BlockIds als Schlüssel zu verwenden. Zeiger, die auf Elemente einer doppelt verketteten Liste deuten, sind hier die Nutzdaten. Der B-Baum garantiert eine BlockId-Suche in $O(log_{Grad(Baum)}(n))$, was für das Re-Referenzieren wichtig ist. Genau wie “Suchen” lassen sich auch “Einfügen” und “Löschen” mit dem gleichen Aufwand realisieren. Die unterliegende doppelt verkettete Liste enthält neben den BlockIds Zeitstempel und Prioritäten. Indem man diese Liste nach Zeitstempeln geordnet hält, lassen sich zu verdrängende Blöcke in einem einzigen Schritt identifizieren. Neue oder referenzierte Blöcke werden ggf. gelöscht und anschließend mit aktuellem Zeitstempel vorne angehängt. Die benötigten Operationen werden also alle in der gewünschten Schranke fertig! Zur graphischen Veranschaulichung dient Abbildung 3.1.

[4]Beides sind balancierte Bäume mit Grad größer 2. Bei B* sind die Nutzdaten ausschließlich in den Blättern gespeichert.

Wie bereits in 3.2.1 erwähnt, lassen sich die benötigten Datenstrukturen, hier doppelt verkettete Liste und B-Baum, kaum im Hauptspeicher unterbringen. Da spätestens mit dem Herunterfahren des Systems diese Daten sowieso auf die Festplatte gesichert werden müssten, bietet es sich an die Datensätze direkt auf der Festplatte zu manipulieren. Zwar würde dieser Schritt das Hauptspeicherproblem lösen, jedoch ist dann zu befürchten, dass die damit verbundenen Festplattenzugriffe die Latenz deutlich erhöhen. Zu deren Dämpfung könnte ein auf *memory mapped I/O*[5] basierendes Cache-System für Baumknoten und Listenelemente eingeführt werden. Der Vorteil dieser "Cache-im-Cache"-Idee ist, dass je nach der physisch verfügbaren Menge an Hauptspeicher unterschiedliche Größen einstellbar sind und somit der Speicherbedarf sehr fein justiert werden kann.

Beim Löschen von Datensätzen der doppelt verketteten Liste taucht jedoch eine weitere Schwierigkeit auf. Manipuliert man diese Liste direkt auf der Festplatte (die Zeiger beziehen sich also auf Adressen innerhalb einer Datei) entstehen durch Löschvorgänge ungenutzte Stellen. Unangenehmerweise benötigen diese Abschnitte nach wie vor Speicherplatz. Denn nach einmal erfolgter Speicherplatz-Zuordnung kann verwaister Speicherplatz nicht mehr abgegeben werden. Es entsteht also eine interne Fragmentierung, die durch zusätzlichen algorithmischen Aufwand abgebaut bzw. verhindert werden muss. Beispielsweise könnte hier eine frei-Liste oder ein frei-Bitmap hilfreich sein.

Die bisher entwickelte Datenstruktur erfüllt zwar alle Geschwindigkeitsvorgaben jedoch ohne Prioritäten bei der Auswahl eines Blockes zu berücksichtigen. Da die Verteilung der Blöcke über die Prioritäten unbekannt ist, muss der Aufwand hierfür mit $O(n)$ angegeben werden, denn die Liste muss gegebenenfalls komplett durchlaufen werden, um festzustellen, dass es keinen Block entsprechender Priorität gibt. Wie schon beim Datenbank-Ansatz in 3.2.2, so kann auch hier diese Aufgabe durch Segmentierung gelöst werden. Anstatt den Baum auf Elemente einer einzigen doppelt verketteten Liste zeigen zu lassen, kann für jede Priorität eine eigene Liste verwendet werden, ohne dass dabei Mehrkosten entstehen.

Fazit

Der Vorschlag verspricht sehr gute Ergebnisse: einen sehr schnellen Zwischenspeicher, dessen Geschwindigkeit sich hauptsächlich durch die für "Cache-im-Cache" zur Verfügung gestellte Hauptspeichermenge und der Festplatte definiert. Auf der Gegenseite steht einzig und allein die sehr aufwändige und komplizierte Implementierung. Leider ist anzunehmen, dass eine Realisierung den für diese Arbeit vorgesehenen Zeitrahmen sprengen würde.

3.2.4 Aufwandsverlagerung ins unterliegende Dateisystem

Nachdem sich die B-Baum-Lösung aus Abschnitt 3.2.3 als zu aufwändig für die Implementation innerhalb dieser Arbeit herausgestellt hat, soll als nächstes eine sehr simple Lösung erarbeitet werden. Die grundlegende Idee hierbei ist die Verlagerung so vieler Aufgaben wie möglich in das unterliegende Dateisystem. Im ersten Schritt wird der Block selbst gespeichert, dessen Dateiname sich aus seiner Hash-Kennung

[5] Teile von Festplatteninhalten werden als Speicherseiten in den Adressraum eingeblendet, eventuelle Änderungen werden automatisch übernommen.

```
/block/13/9C/139CA8EEFF432D1872D6138AEC9B2A5A3F9A6056
/block/A2/41/A241F4B6BADDE075CC68E56B3C23CAA63649C3F5
/block/DF/F4/DFF498A4A13FA92FB8142218F5C6E0598521C315

/time/1/2006/june/12/17/24/139CA8EEFF432D1872D6138AEC9B2A5A3F9A6056
/time/1/2006/june/13/14/47/DFF498A4A13FA92FB8142218F5C6E0598521C315
/time/1/2006/june/13/17/15/A241F4B6BADDE075CC68E56B3C23CAA63649C3F5
```

Abbildung 3.2: Verzeichnisbäume für Blöcke und minutengenaue Zeitstempel

ableitet. Im Fall von 160-Bit-Kennungen wären das 40 Hexadezimal-Symbole. Vernünftigerweise sollte hier zusätzlich eine Verzeichnishierarchie eingeführt werden, damit später nicht Millionen Blöcke in einem einzigen Verzeichnis abgelegt werden müssen. Beispielsweise lässt sich durch eine zweistufige Hierarchie, jeweils durch ein Byte der Hash-Kennung[6] erzeugt, die Menge der Blöcke pro Verzeichnis in den unteren vierstelligen Bereich[7] drücken. Eine ensprechende Veranschaulichung findet sich in Abbildung 3.2

Auf diesem Weg kann bereits die Existenz eines Blockes in einem einzigen Schritt ermittelt werden – der Aufwand hierfür liegt also bei $O(1)$[8]. Mit den gleichen Kosten lassen sich auch neue Blöcke in die Hierarchie einfügen.
Zur Durchsetzung der LRU-Strategie müssen jedoch noch Zeitinformationen für alle Blöcke mitgeführt werden. Eine erste Idee hierfür ist die Erweiterung der Block-Dateinamen um diesen Zeitwert. Ungeschickterweise wären dann aber die Block-Dateinamen nicht mehr exakt bekannt, für eine Re-Referenzierung müsste also zunächst mit einer teuren Suche begonnen werden. Alternativ kann der Zeitstempel in die Block-Datei integriert werden. Um den Zeitpunkt der letzten Referenzierung zu bekommen, muss nun die Datei geöffnet, der Stempel gelesen und ggf. modifiziert werden. Da die Datei bei einer Referenzierung sowieso geöffnet und gelesen werden muss, wäre der zusätzliche Aufwand akzeptabel.
Bedenkt man, dass für die Verdrängung der älteste Block gefunden werden muss und dessen Suche das Öffnen und Untersuchen aller Blöcke erfordert, so wird schnell klar, dass diese Technik für diese Aufgabe ungeeignet ist. Es wird sich später zeigen, dass sie dennoch gebraucht wird.

Zur Auffindung alter Blöcke wird ein zweiter Verzeichnisbaum eingeführt, der über mehrere Baumebenen hinweg Zeitstempel Stück für Stück ablegt. Auch hier könnten wieder Bytes den Weg durch den Baum weisen. Allerdings ist eine günstig zu realisierende, menschen-lesbarere Form diesem Weg vorzuziehen. In jedem Fall entsprechen dann die Blätter des entstehenden Baumes Zeitabschnitten, die je nach gewählter Granularität nur wenige Minuten breit sind. Diese Abschnitte sind gleichzeitig die größte Genauigkeit mit der das System die ältesten Blöcke identifizieren kann. Für jeden neu hinzukommenden Block wird ein Eintrag in diesem Zeitstempel-Baum er-

[6] kann als gleichverteilt angenommen werden

[7] Verteilt man 100 Millionen Blöcke auf 256^2 Verzeichnisse, so ergeben sich ca. 1525 Dateien pro Verzeichnis.

[8] $O(1)$ aus Sicht vom *IGOR*-Dateisystem. Je nach Implementation des unterliegenden Dateisystems kann es sich hierbei auch um logarithmischen oder schlechteren Aufwand handeln.

zeugt, dessen Pfad durch den Zeitstempel selbst beschrieben wird und dessen Name sich erneut aus der hexadezimalen Repräsentation seiner BlockId ergibt.
Zur Verdeutlichung zeigt Abbildung 3.2 die Namen von drei gespeicherten Blöcken, sowie deren Einträge im Zeitstempel-Baum. Es fällt auf, dass diese Einträge leere Dateien sein können, da lediglich deren Namen wichtig sind. Alternativ ist eine Verknüpfung mit der Block-Datei denkbar. In diesem Fall sind allerdings Hardlinks[9] zu bevorzugen, da sie keinen zusätzlichen Overhead erzeugen. Unabhängig davon können auf diesem Weg nun leicht zu verdrängende Blöcke identifiziert werden, denn die ältesten Blöcke sind stets mit logarithmischem Aufwand auffindbar. Dazu muss lediglich so lange der älteste aller abgehenden Äste besucht werden, bis das zugehörige Blatt gefunden ist.

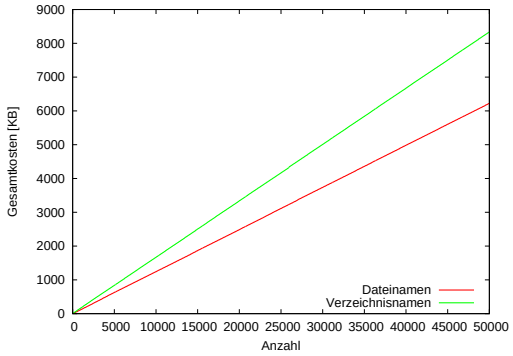
Als einzige noch offene Aufgabe verbleibt das Re-Referenzieren, denn hierfür muss der Zeitstempel einer gegebenen BlockId aktualisiert werden. Die nötige Suche nach einer BlockId kann allerdings im Zeitstempel-Baum nicht effizient durchgeführt werden. Daher müssen diese beiden Bäume auch in die andere Richtung miteinander verknüpft werden. Wie bereits oben erwähnt, lässt sich dieses Problem durch die Integration des Stempels in die Block-Datei ausreichend effizient lösen. Eine Re-Referenzierung durchläuft demnach folgende Schritte:

1. Block-Datei lesen $\rightarrow O(1)$
2. neue Zeitstempel-Verknüpfung erstellen $\rightarrow O(1)$
3. Block-Datei modifizieren $\rightarrow O(1)$
4. alte Verknüpfung löschen $\rightarrow O(1)$

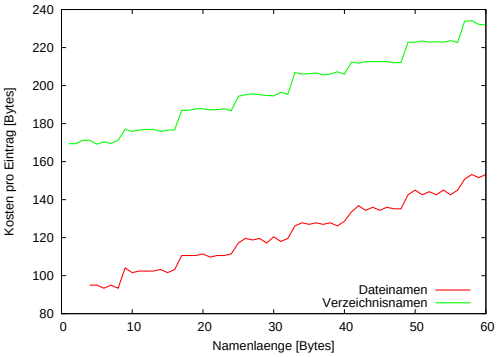
Nachdem nun alle Cache-Operationen ausreichend schnell konstruiert wurden, verbleibt noch die Frage, wo Prioritäten abzulegen sind. In Abbildung 3.2 sind sie bereits als erste Verzeichnisebene im Zeitstempel-Baum berücksichtigt. Entsprechend müssen die in die Block-Dateien eingebetteten Zeitstempel ebenfalls mit der Priorität versehen werden. Somit bleiben die Kosten für Cache-Funktionen unverändert. Darüber hinaus kann die Prioritätsmarkierung frei im Zeitstempel verschoben werden. Beispielsweise könnte man sie ganz ans Ende setzen, was die Frage nach dem ältesten Block jedweder Priorität geringfügig erleichtert.

Wie schon in den anderen Umsetzungsideen, so verbleibt auch hier ein Konsistenzproblem: Die strikte eins-zu-eins-Verknüpfung der beiden Verzeichnisbäume kann durch Abstürze in ungünstigen Momenten oder auf Benutzerbefehl korrumpiert werden. Im unangenehmsten Fall existiert im Zeit-Baum kein Eintrag mehr für einen Block. Der Block-Verdrängungs-Algorithmus würde diesen Block niemals wählen und ihn demzufolge auch niemals löschen. Die Kapazität des gesamten Zwischenspeichers würde abnehmen. Das gilt jedoch nur so lange, wie genau dieser Block nicht referenziert wird. Denn spätestens mit einer Referenzierung muss eine neuer Link angelegt werden, der Fehler würde behoben! Zusätzlich können solche Fehler behandelt werden, indem man aktiv beide Bäume durchsucht und die Verknüpfungen untereinander prüft. Hierfür ist allerdings linearer Aufwand nötig.

[9] Hardlinks zeigen auf die Verwaltungseinheit (INODE) eines Datenträgers, der die entsprechende Datei beinhaltet.



(a) Kosten für Datei-Anzahl bei konstanter Namenslänge



(b) Kosten für Datei-Namen bei konstanter Dateimenge

Abbildung 3.3: Verwaltungskosten im unterliegenden Dateisystem

Im ganzen Abschnitt gilt es zu beachten, dass sich alle Kostenangaben nicht länger auf Prozessor-Kosten beziehen, sondern als Festplatten-Lese/Schreibe-Kosten zu verstehen sind. Daher ist kein direkter Kostenvergleich mit den bisherigen Ideen durchführbar.

Kostenabschätzung im schlimmsten Fall

Die Verlagerung allen Aufwands in das unterliegende Dateisystem ist zwar ein praktischer und einfacher Weg, jedoch dürfen dabei entstehende Kosten für zusätzliche Dateien und Verzeichnisse nicht gänzlich außer Acht gelassen werden. Auch wenn ein null Bytes großer Link scheinbar keinen Platz benötigt, so fällt dennoch für die Speicherung seines Namens ein Platzbedarf an.
Um die bisher abstrakten Kosten durch eine konkrete Zahl ausdrücken zu können, wurden in einem heutigen Dateisystem[10] der Platzbedarf für Verwaltungseinträge gemessen. Die Graphen 3.3(a) und 3.3(b) zeigen die Ergebnisse dieser Messungen. Geschickterweise sind die Kosten für Verzeichnis-Namen im benötigten, unteren Bereich näherungsweise konstant, womit man sich auf die Zählung der vorkommenden Verzeichnisse beschränken kann. Der schlimmste Fall sieht vor, dass jeder Block in einem, von der Wurzel an anderen Zweig des Zeitstempel-Baumes abgelegt wird. Konstruiert man also Dateinamen und Verknüpfungen nach dem Muster aus Abbildung 3.2 und bringt sie mit den in Abbildung 3.3 gemessenen Kosten zusammen, so ergeben sich pro Block folgende Verwaltungskosten:

$$6 \cdot \text{Verzeichnis} + 1 \cdot \text{Link}_{40} + 1 \cdot \text{Stempeleinbettung} =$$
$$6 \cdot 170 \text{ Bytes} + 129 \text{ Bytes} + 18 \text{ Bytes} = 1170 \text{ Bytes}$$

Bei einer durchschnittlichen Blöckgröße von 65Kb, wie in 2.4.1 vorgegeben, entstehen also selbst im schlimmsten Fall nur ca. $1,78\%$ Verwaltungskosten. Realistischerweise teilen sich mehrere Blöcke die gleichen Blätter und mehrere Blätter die gleichen Zweige des Baumes. Daher wird dieser ohnehin schon akzeptable Mehraufwand nochmal deutlich reduziert!

[10] Hierfür wurde ReiserFS Version 3.6 verwendet.

Fazit

Dieses Verfahren benötigt praktisch keinen Hauptspeicher, da alle anfallende Arbeit direkt auf dem unterliegenden Dateisystem durchgeführt werden. Die zu implementierenden Zwischenspeicher-Operationen erfordern nur geringen, meist konstanten Aufwand[11] – eine gute Skalierbarkeit ist sichergestellt. Auch können Fehler in der Konsistenz sehr leicht behoben werden und es muss zu vielen Abstürzen bzw. massivem Benutzereingriff kommen, bevor Konsistenzfehler eine spürbare Auswirkung auf das Gesamtsystem haben können. Alles in allem verspricht dieses Verfahren also eine einfache Implementation unter sehr guten Nebenbedingungen.

3.2.5 Verfahrenswahl

Aufgrund der engen Zeitvorgaben dieser Arbeit haben wird der letzte Ansatz 3.2.4 für die Umsetzung gewählt. Er verspricht im Gegensatz zu 3.2.3 eine relativ einfache Implementation ohne dabei auf hohe Geschwindigkeiten zu verzichten oder schlechte Eigenschaften einzubringen. Darüber hinaus kann durch diese Wahl auf bereits vorhandene Standard-Datenstrukturen zurückgegriffen werden, während die Fusion eines B-Baums mit einer doppelt verketteten Liste enorme Zeit kosten würde.

3.3 Architektur des Block-Caches

Nachdem in 3.2.4 eine geeignete Implementierungsform für den Zwischenspeicher selbst gefunden wurde, beschäftigen sich die folgenden Abschnitte mit der Gesamt-Architektur des Block-Caches. Denn außer der Verdrängung wird auch noch eine Schnittstelle für die übrigen Komponenten des *IGOR*-Dateisystems benötigt. Um ein nahtloses Einfügen in das Gesamtsystem zu gewährleisten, soll für diese Aufgabe die in 3.1 vorgestellte Prozess-Netzwerk-Variante verwendet werden.

3.3.1 Anforderungen an die Architektur

Wie für alle Komponenten üblich, so soll auch der Block-Cache natürlich ein Musterbeispiel an Geschwindigkeit, Effizienz, Skalierbarkeit, Fehlertoleranz usw. sein. Betrachtet man die Hauptaufgabe des Block-Caches – das Lesen und Schreiben von Blöcken – sowie die Organisation der Verwaltungsstrukturen direkt auf dem unterliegenden Dateisystem, so fällt auf, dass zur Laufzeit hauptsächlich Eingabe/Ausgabe-Operationen durchgeführt werden müssen. Traditionell sind solche Funktionen blockierend und aus Sicht des Prozessors sehr zeitaufwändig[12]. Zwar ist es auf heutigen Betriebssystemen möglich, diese I/O-Systemaufrufe asynchron zu gestalten, der Implementationsaufwand dafür ist allerdings hoch. Und da durch das Prozess-Netzwerk-Design sowieso bereits "Multithreading" vorgegeben ist, ist an dieser Stelle synchrones I/O mit mehreren Threads zu bevorzugen. In beiden Fällen kann ein Strom an E/A-Aufträgen für den Datenträger erzeugt werden, während fertig gelesene Daten parallel weiterverarbeitet werden können.
Bedenkt man, dass heutige Festplatten noch nicht die geforderen Terabyte Kapazitäten bereitstellen können, so ergibt sich aus diesem Thread-Ansatz ein weiterer

[11] Erneut aus Sicht des *IGOR*-Dateisystems.

[12] Auf heutigen Rechnern kosten Prozessorbefehle Nanosekunden oder weniger, E/A-Operationen mit einer Festplatte im Allgemeinen mehrere Mikrosekunden.

Vorteil. Denn es müssen mehrere Festplatten beispielsweise als *Raid-System*[13] oder *Logical Disk Array*[14] miteinander verbunden werden, um dieses Fassungsvermögen zu erzeugen. Somit sind mehrere Datenträger beteiligt, wodurch echte Parallelverarbeitung ermöglicht werden könnte.
Wie zu erwarten, bringt die Parallelität auch ihre Schattenseiten mit sich: Da nun mehrere Threads gleichzeitig auf den selben Blöcken arbeiten könnten, kann es zu Inkonsistenzen kommen. Obwohl dieses Risiko bei Millionen von Blöcken relativ gering ist, kann es nicht gänzlich ausgeschlossen werden. Beispielsweise könnten neu publizierte Blöcke sehr häufig von verschiedenen Quellen abgefragt werden. Glücklicherweise sind einzelne Blöcke nicht voneinander abhängig, so dass dieses Problem durch atomaren Umgang mit einzelnen Blöcken bzw. BlockIds lösbar ist. Dabei versteht es sich von selbst, dass für diese Aufgabe nicht jede einzelne BlockId mit einer Semaphore oder einem "Lock" gesichert werden kann.
Eine andere Frage ist die nach dem Grad der Parallelität. Mit anderen Worten: Wie viele Threads sollen parallel arbeiten? Beispielsweise könnte ein Nutzer eine mehrere Megabytes große Datei speichern wollen, während mehrere Anfragen aus dem Netzwerk zu bearbeiten sind. Würde für jeden Block ein eigener Thread gestartet werden, könnten schnell tausende Threads zusammenkommen. Dabei dürfen auch nicht die im Betriebssystem für die Thread-Erzeugung entstehenden Laufzeit-Kosten vernachlässigt werden.

Darüber hinaus können manche Operationen sehr lange Zeit blockiert sein und auf Antwort warten. Man denke dabei z.B. an die Wartezeit, bis ein Block aus dem Netzwerk geladen wurde. Ein Wert im Sekundenbereich ist hier durchaus typisch.

Aus diesen beiden Beobachtungen heraus entsteht die Forderung nach einer asynchronen Aufgabenbearbeitung. Das bedeutet, falls lange Antwortzeiten zu erwarten sind, soll der bearbeitende Thread die entsprechenden Befehle geben, aber danach den Auftrag zur Seite legen und parallel einen weiteren bearbeiten. Hierfür ist eine Auftragswarteschlange nötig, in der die Auftäge in verschiedenen Stadien des Fortschritts abgelegt werden können.

Hinzu kommt die bereits in 2.4.2 geforderte Prioritätverwaltung für verschiedene Aufträge. Diese Anforderung kann auf verschiedenen Wegen implementiert werden. Da es keine exklusiven Ressourcen gibt, die einzelne Threads beanspruchen könnten, ist der Weg der Verdrängung einer niedrig priorisierten Aufgabe von einem Arbeiter-Thread denkbar. Somit könnte ein wichtiger Auftrag sofort bearbeitet werden. Ein nicht ganz so schnelles, aber deutlich einfacher zu implementierendes Verfahren ist die Erweiterung der vorhin ins Leben gerufenen Auftragswarteschlange. Anstatt neue Aufträge immer hinten anzufügen, könnte die Datenstruktur die Prioräten der Aufträge mit berücksichtigen und entsprechend sortieren. Ein Auftrag höchster Priorität müsste dann allerdings so lange warten, bis ein Arbeiter-Thread seine aktuelle Arbeit wegen langer Wartezeit aufgibt und sich nach neuer Arbeit umsieht. Je größer die Anzahl der Threads, desto schneller wird ein solcher Zeitpunkt erreicht sein.
Eine weitere sinnvolle Erweiterung der Sortier-Reihenfolge innerhalb der Auftragswarteschlange ist es, wenn, nachdem bereits eine Sortierung nach Priorität erfolgt

[13] Zusammenschluss mehrerer Festplatten eines Computers, um eine höhere Speicherkapazität oder einen höheren Datendurchsatz zu erhalten (nach Wikipedia, http://de.wikipedia.org/wiki/RAID, 13.07.2006).
[14] Sequentielle Kombination von Festplatten, um größeren Speicherplatz zu erhalten.

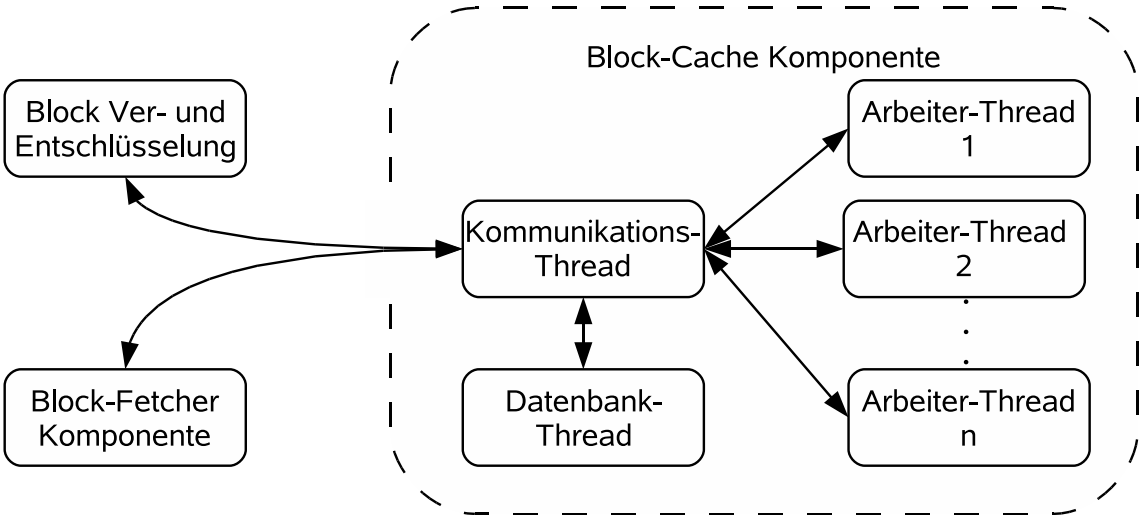


Abbildung 3.4: Architektur Block-Cache - Zentraler Ansatz

ist, noch eine Ordnung nach Fortschritt gemacht wird. Wichtige Aufträge kommen immer noch vor allen anderen, aber somit würden auch Aufträge bevorzugt abgearbeitet werden, bevor mit neuen begonnen wird.

3.3.2 Zentraler Ansatz

Zur Realisierung all der Vorgaben aus 3.3.1 bietet sich eine Aufteilung des Block-Caches in folgende Unterkomponenten an: einen Kommunikations- und Verwaltungs-Thread, einen Datenbank-Thread und diverse Arbeiter-Threads. Zur besseren Übersicht bietet das Blockdiagramm 3.4 eine graphische Veranschaulichung der Zusammenhänge. Der erstgenannte Prozess nimmt eingehende Aufträge von anderen Komponenten des *IGOR*-Dateisystems an und erzeugt daraus Arbeits-Aufträge. Diese werden, nachdem sichergestellt ist, dass derzeit noch kein aktiver Auftrag auf der gleichen BlockId arbeitet, an einen freien Arbeiter-Thread weitergereicht oder ggf. in die Auftragswarteschlange eingefügt.
Nachdem ein Arbeiter-Thread einen Auftrag erhalten hat, prüft er zunächst dessen gegenwärtigen Status, um dann an der richtigen Stelle fortsetzen zu können. Der Arbeiter führt die Aufgabe solange aus, wie keine Kommunikation mit anderen Komponenten erforderlich ist. Beispielsweise könnte ein Block nicht vorhanden sein und muss von der Peer-to-Peer-Komponente *IGOR* aus dem Netzwerk geladen werden, oder es gilt dem Datenbank-Thread eine Statusänderung mitzuteilen. Zu diesem Zweck gibt der Arbeiter den teilweise bearbeiteten Auftrag wieder an den zentralen Verwaltungs-Thread zurück, wodurch er gleichzeitig signalisiert, dass ihm neue Aufgaben zugewiesen werden können.
Bei Eingang einer teilweise bearbeiteten Aufgabe prüft der Verwaltungs-Thread den Fortschritt, verschickt ggf. Nachrichten an andere Komponenten und beantwortet, sofern möglich, den ursprünglichen Auftrag. In jedem Fall wird der Auftrag danach entweder gelöscht, weil er abgeschlossen ist, oder in den Zustand 'blockiert' versetzt, während er auf eine Antwort eines befragten Prozesses warten muss.
Der verbleibende Datenbank-Thread bekommt lediglich Nachrichten über durchgeführte Cache-Operationen, die in der Datenbank vermerkt werden müssen. Neben einem 'Neuer Block eingelagert' handelt es sich hierbei vor allem über Re-Referenzierungs-Meldungen. Jede bearbeitete Nachricht wird quittiert, um den Ar-

beiter-Threads die Vollendung bzw. Fortführung der Aufgabe zu ermöglichen.
Die Verdrängung alter Blöcke kann selbst als Auftrag modelliert werden. Der einzige Unterschied zu "normalen" Aufträgen ist, dass hierbei der Datenbank-Thread als Auftragsteller fungiert.

Bewertung

Die Vorteile dieses Ansatzes sind offensichtlich die einfache Konstruktion, denn bis auf das zentrale Verwaltungs-Modul, über das jegliche Kommunikation abgewickelt wird, müssen sich alle beteiligten Module nur einen einzigen Kommunikationspartner merken. Darüber hinaus ist dieses Modul über den Gesamtstatus des Block-Caches informiert, was gerade für Fehlersuchen sehr angenehm sein kann, zur Laufzeit aber keine Vorteile bringt.
Die Abwicklung der gesamten Kommunikation über ein einziges Modul bringt aber auch die Gefahr der Überlastung mit sich. Zwar führt diese hier nicht automatisch zum Absturz des gesamten *IGOR*-Dateisystems, dafür aber zu langen Antwortzeiten bzw. zu einem trägen System.
Indem auch die Arbeiter-Threads ihre gesamte Kommunikation über die Verwaltungs-Komponente abwickeln, muss dort immer der aktuelle Status analysiert werden um entsprechende Nachrichten erzeugen zu können. Wissen, bzw. Teilergebnisse, die den Arbeitern bereits bekannt waren, müssen ggf. neu berechnet werden. Somit entsteht unnötiger Mehraufwand.

3.3.3 Ansatz um mehr Eigenständigkeit zu erzielen

Um dem möglichen Überlastproblem des vorangegangen Vorschlags zu begegnen, wird in diesem Ansatz versucht die Kommunkationslast für den zentralen Verwaltungs-Thread zu senken. Zu diesem Zweck wird die Gesamtarchitektur aus 3.3.2 zunächst übernommen, die Semantik der Arbeiter-Threads jedoch modifiziert: Anstatt die Kommunikation mit anderen Komponenten durch den Verwaltungs-Thread zu leiten verschicken sie ihre Kommandos selbst. Bevor allerdings diese Nachrichten an das Prozess-Netzwerk übergeben werden dürfen, muss noch die Adresse für Antworten auf die Verwaltungs-Komponente gesetzt werden. Somit können die Arbeiter Befehle absetzen während der Verwaltungs-Thread nach wie vor als Rückkanal fungiert. Ebenfalls wie vorhin wird nach Absetzen eines Kommandos der Autrag in den Zustand 'blockiert' versetzt und an die Verwaltung zurückgeleitet. Dadurch wird Mehraufwand durch Zustandsprüfungen vermieden und die Kommunikationslast gesenkt.
Das Blockdiagramm 3.5 zeigt den neuen Verlaufsplan der Nachrichten. Hierbei ist zu beachten, dass die eingezeichnete 'Arbeiter-Gruppe' nur der Übersicht dient, aber keinerlei entwurfstechnische Relevanz hat. Diese Gruppe dient lediglich dem Zweck alle eingehenden und ausgehenden Verbindungen nur ein einziges Mal einzuzeichnen während sie tatsächlich für jedes Gruppenmodul extra existieren.

Bewertung

Indem es Arbeiter-Threads gestattet wird direkt Befehle an andere Komponenten abzusetzten, wird der Aufwand reduziert und die Kommunikationslast am Verwaltungs-Thread leicht verringert. Dieser Anstatz ist also offensichtlich besser und damit dem Vorschlag 3.3.2 vorzuziehen.

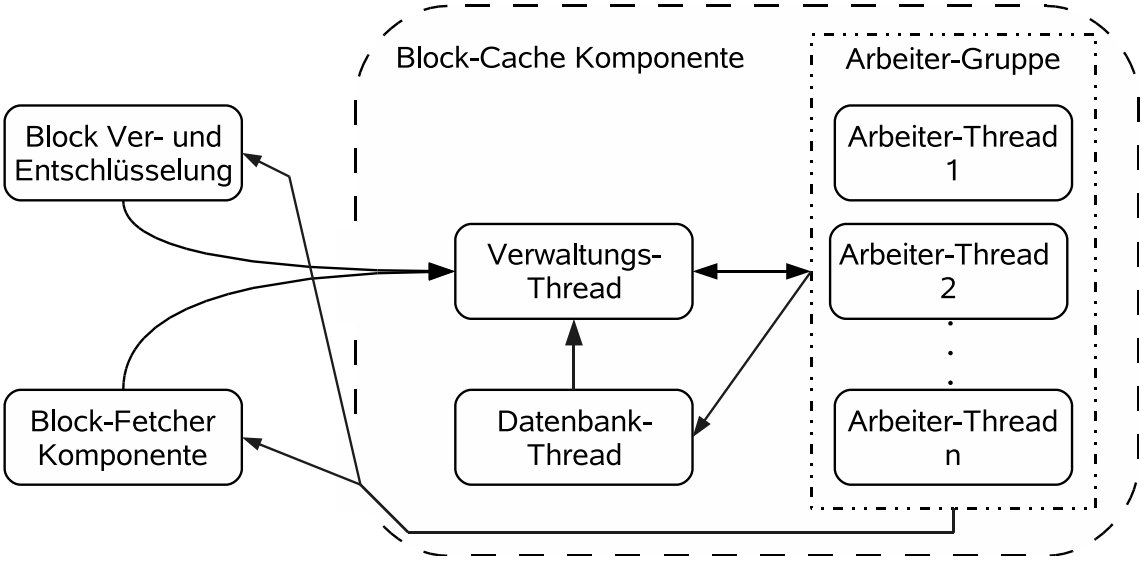


Abbildung 3.5: Architektur Block-Cache - Ansatz 3.3.3

Nach wie vor kann es jedoch zu Nachrichtenstaus am Datenbank-Thread und am Verwaltungs-Thread kommen. Der folgende Abschnitt diskutiert daher einen verteilten Ansatz, um zu prüfen, ob dieser besser geeignet ist.

3.3.4 Verteilter Ansatz

Um eine maximale Reduzierung der Kommunikation zu erreichen, dürfen Antworten auf Kommandos, die von Arbeiter-Threads erzeugt wurden, nicht mehr durch das Verwaltungs-Modul zurückgeleitet werden. Stattdessen muss jeder Arbeiter selbst Empfänger bleiben.
Das bedeutet allerdings, dass auch unfertige Aufträge, die aktuell 'blockiert' sind, erst nach Fertigstellung zum Verwaltungs-Thread geschickt werden dürfen. Somit müssen Arbeiter entweder eine Weile lang untätig warten, oder selbst über eine kleinere Auftragswarteschlange verfügen. Im ersten Fall ließe sich die nunmehr verringerte Abarbeitungsgeschwindigkeit, welche aus der geringeren Parallelität folgt, durch Erweiterung des Arbeiterpools kompensieren. Neben den entstehenden Hauptspeicherkosten nähert man sich auf diesem Weg allerdings wieder der schlechten Ein-Arbeiter-Pro-Auftrag-Lösung.
Die Aufrüstung von Arbeiter-Threads mit Auftragswarteschlangen kostet im Wesentlichen einen höheren Aufwand bei der Implementation. Jedoch müsste eine modulare Programmierung der in jedem Fall nötigen Warteschlange des Verwaltungs-Moduls diese zu großen Teilen auch für Arbeiter wiederverwendbar machen. Dieser Mehraufwand wäre also überschaubar. Auch ist auf diesem Weg die Parallelität wieder hergestellt, die Anzahl der Arbeiter muss nicht oder kaum erhöht werden.
Die Bereit-Liste[15] des Verwaltungs-Moduls kann nun strikt nach Priorität geordnet werden, ohne dabei den Fortschritt eines Auftrages zu berücksichtigen. Schließlich befinden sich alle Einträge nur im Zustand 'neu' bzw. 'unbearbeitet'. Der Verwaltungs-Aufwand wird somit bereits verkleinert und die starke Reduktion von Nachrichtentransfers durch diese Komponente macht Überlast-Situationen deutlich unwahrscheinlicher.

[15] besser als *readylist* bekannt

Auf der Gegenseite sind die Arbeiter-Threads durch ihre internen Warteschlangen nicht länger zustandsfrei. Die Verschiebung einzelner Arbeitsaufträge wird aufwändiger, was gleichzeitig auch die Veränderung der Arbeiter-Anzahl zur Laufzeit erschwert. Dies kann aber für verschiedene Last-Situationen sehr geschickt sein! Es ist beispielsweise denkbar, die Anzahl der Arbeiter, gemessen an der derzeitigen Systemlast, einzuregeln. Eventuell lassen sich auf diesem Weg stabilere Antwortzeiten erzeugen. Aufgrund der Komplexität schwingender Systeme soll diese Regelung allerdings nicht Teil dieser Arbeit sein, jedoch darf auch nicht die Möglichkeit hierfür verbaut werden.

Bewertung

Die zweite Variante – die Verwendung von Warteschlangen für einzelne Arbeiter – verspricht weiterhin geringe Antwortzeiten für einzelne Aufträge, ohne dabei die Parallelverarbeitung einzuschränken. Der Nachrichtenaustausch wird dabei auf ein Minimum reduziert, was einerseits den Kommunikations-Overhead minimiert, aber andererseits die Entwicklung durch kompliziertere Fehlersuche erschwert. Weitaus unangenehmer ist, dass die Arbeiter-Threads durch ihre internen Zustände nicht länger Aufträge zu anderen Arbeitern migrieren können. Es geht somit an Flexibilität verloren, die vorgesehene Implementation einer Regelung für die Anazhl der Arbeiter wird erschwert.

3.3.5 Verfahrenswahl

Für die Implementation innerhalb dieser Arbeit, wurde das Verfahren aus 3.3.3 gewählt. Denn verglichen mit dem ersten Ansatz, ist die Kommunikationslast geringer und die Verwaltung deutlich schlanker. Auch sind die Arbeiter, entgegen dem letzten Ansatz 3.3.4, immernoch zustandslos und damit flexibler.

3.4 Überprüfung des Dateisystems

In jedem der in 3.2 gemachten Vorschläge zur Umsetzung der *LRU*-Strategie gab es das Konsistenzproblem. Es basiert darauf, dass in einem Overlay-Dateisystem durch den Benutzer wissentlich oder unwissentlich die Blöcke selbst oder die Metadaten des Block-Caches korrumpiert werden könnten. In jedem Fall muss durch ein Prüfprogramm die Konsistenz wieder herstellbar sein.
Zu diesem Zweck bietet es sich an, den Block-Cache mit einem weiteren internen Modul zu versehen, welches diese Aufgabe durchführt. Verwendet man beispielsweise das in 3.2.4 vorgeschlagene Verfahren zu Speicherung der Metadaten in Verknüpfungsnamen, so ergäben sich folgende Aufgaben für ein solches Modul: Zunächst müsste der Block-Baum Stück für Stück durchsucht werden und die in jedem Block eingebetteten Links extrahiert werden, um letztlich die Existenz der Verknüpfungen zu verifizieren. In der zweiten Phase müsste der Baum der Verknüpfungen durchlaufen werden, um das Vorhandensein zugehöriger Blöcke zu bestätigen.
Es fällt auf, dass die bereits von Arbeiter-Threads durchgeführten Aufgaben zum Teil auch in den beiden Prüfphasen wiederverwendet werden könnten. Die Idee, die skalierbaren Arbeiter zusätzlich mit der Prüfung des Dateisystems zu belasten, liegt also nahe. Somit könnte die Prüf-Komponente Prüfaufträge generieren und diese von der zentralen Verwaltungs-Komponente an Arbeiter zuteilen lassen.

Allerdings ist abzusehen, dass die Bäume sehr viel schneller nach Blöcken oder Verknüpfungen durchsucht werden können, als einzelne Prüfaufträge durch Arbeiter-Threads verarbeitbar sind. Hierfür muss schließlich u.a. der gesamte Block aus dem unterliegenden Dateisystem gelesen und dessen Konsistenz durch das kryptographische Hash-Verfahren verifiziert werden. Um nicht Millionen Aufträge im Hauptspeicher halten zu müssen, ist also eine Art Staukontrolle nötig. Da jeder Prüfauftrag einzeln quittiert wird, kann das Prüf-Modul mittels einer simplen Zählvariable die Anzahl offener Aufträge gespeichert halten. Die einfachste Form einer Staukontrolle könnte also durch eine obere Schranke an gleichzeitigen Prüfaufgaben realisiert werden.

Ein beeindruckender Vorteil dieses Verfahrens ist, dass das Dateisystem auf diese Art und Weise zur Laufzeit prüfbar ist. Darüber hinaus können durch die Prioritäten im Verwaltungs-Modul wichtige Aufgaben während der Prüfzeit bevorzugt behandelt werden, während sich unwichtigere Aufgaben in die Prüf-Aufträge einreihen müssen. Letztendlich verbleibt nur noch das Problem der sinnvollen Zuordnung von Prioritäten.

4. Implementierung

Wie bereits aus dem Namen hervor geht, beschreibt dieses Kapitel detailliert, wie der Block-Cache als Programm umgesetzt wird. Dazu werden in 4.1 zunächst die verwendeten Nachrichtenformate vorgestellt, die zwischen den in 4.2 beschriebenen Komponenten ausgetauscht werden können. Für den Entwickler sind hier hauptsächlich die 'externen' Nachrichtenformate wichtig, mit deren Hilfe Kommandos an den Zwischenspeicher abgesetzt werden können und die darüber Auskunft geben, wie entsprechende Antworten zu interpretieren sind.
Die zweite Kapitelhälfte wird sich dann genauestens mit der Umsetzung der einzelnen Block-Cache-Komponenten befassen.

4.1 Nachrichtenformate

Dieser Abschnitt besteht im wesentlichen aus zwei Bereichen: Zunächst sind da die bereits 'extern' genannten Nachrichtentypen, welche von anderen, mit dem Block-Cache kooperierenden Komponenten verstanden bzw. gesprochen werden müssen. Der Vollständigkeit halber folgen anschließend weitere Nachrichtenformate, die für die interne Kommunikation zwischen den einzelnen Modulen verwendet werden.
Um den Leser mit der eventuell ungewohnten Form der Nachrichten nicht allein zu lassen informiert 4.1.1 zunächst über die Grundlagen, aus denen später alle Nachrichten in diesem Prozess-Netzwerk gebildet werden.

4.1.1 Allgemeine Design-Überlegungen

Zu allererst ist die Umgebung, in welcher die Nachrichten existieren bzw. verwendet werden sollen zu nennen: ein Prozess-Netzwerk oder zumindest eine naher Verwandter davon. Das bedeutet, dass sich, wie alle Komponenten, auch die Nachrichten in einem einzigen Adressraum befinden. Anstatt also alles in die traditionelleren Bitfolgen mit eventuell sogar festen Längen umwandeln zu müssen, kann dem Transportsystem eine ganze Klasse bzw. ein Zeiger darauf übergeben werden. Tatsächlich bedeutet ein Nachrichtenversand in unserer Umgebung lediglich die Übergabe eines Zeigers an einen in der Regel anderen Thread. Somit ist hier, anders wie in Netzwerken, die

Größe einer Mitteilung völlig irrelevant. Kosten entstehen lediglich für den Zeigeraustausch. Als Superklasse für alle Nachrichten im Prozess-Netzwerk dient `cMessage`[1], oder `cIFSMessage` für alle Mitteilungen innerhalb des *IGOR*-Dateisystems. Letztere kapselt bereits eine noch undefinierte Typ-Information sowie den Absender und einen Nachrichten-Identifikator.

Nun stellt sich die Frage, was denn eigentlich verschickt werden soll. Während beispielsweise Einträge über die Priorität einer Nachricht bzw. eines Auftrages von allen Modulen des Dateisystems genutzt werden können, gibt es offensichtlich auch Informationen, die lediglich von bestimmten Komponenten benötigt werden. Ein Beispiel hierfür wäre die Priorität mit der ein Block im Zwischenspeicher abgelegt werden soll, oder die Netzadresse, von der das Peer-to-Peer-Modul einen lokal nicht verfügbaren Block erfragen soll.

Natürlich genügt es einem Thread, lediglich die Informationen einer Nachricht zu berücksichtigen, die für das Erreichen seines Ziels benötigt werden. Man könnte also die erforderlichen Einträge aller Module in einer einzigen Nachrichtenklasse sammeln. Somit ist eine sehr einfache Klasse entstanden, die bei Bedarf immer noch nachträglich durch einen Aufzählungs-Typ[2] segmentiert werden kann. Jedoch bringt dieser Ansatz auch Nachteile mit sich: Jede Instanz verwendet nur ein Bruchteil der Variablen, der ungenutzte Rest benötigt trotzdem Hauptspeicher. Dieser Mehraufwand wächst mit der Weiterentwicklung des Systems sogar noch an: Neue Klassenmitglieder werden aufgenommen und bei wachsender Thread-Anzahl steigt zusätzlich auch noch die Nachrichtenmenge mit an.

Der radikalste Weg aus diesem Overhead-Problem ist die Aufsplittung einer großen Nachrichtenklasse in mehrere Kleinere. Beispielsweise könnte für jeden Kommunikationsweg, zumindest aber für jedes Modul, ein eigenes Klassenpaar erzeugt werden. Somit kann man sich auf die jeweils nötigen Informationen beschränken, es entstehen keine Speicher-Mehrkosten. Darüber hinaus beschreiben diese Klassen dann exakt die Schnittstellen der jeweiligen Komponenten. Ist bei einer Weiterentwicklung eine Änderung nötig, kann die jeweilige Klasse individuell angepasst werden. Allerdings darf hierbei nicht vergessen werden, dass dann alle kooperierenden Module ebenfalls geändert werden müssen. Auch muss jedes Modul zunächst den Typ einer eingehenden Mitteilung ermitteln, bevor sie verarbeitet werden kann. Dazu wären bei n vollvernetzten Modulen im schlimmsten Fall n^2 Möglichkeiten zu überprüfen. Allerdings sind hier $2n$ Klassen realistischer, wobei dann jedes Modul auf einen bestimmten Ein- und Ausgabetyp festgelegt ist. Neben dem Programmierer, der nun diverse Fallunterscheidungen implementieren muss, leidet vor allem die Wartbarkeit eines solchen Systems unter der unübersichtlichen Nachrichtenmenge.

Letztendlich fällt die Entscheidung zu Gunsten eines, durch die verwendete Programmiersprache *C++* ermöglichten Mittelwegs: die Mehrfachvererbung. Die Idee besteht darin, für die einzelnen Kommunikationswege nahezu *perfekte* Klassen zusammenzubauen. *Perfekt* bedeutet hierbei, dass möglichst wenige ungenutzte Variablen vorkommen sollen. Dazu werden oft benötigte Einträge verschiedener Aufgaben in einigen Klassen zusammengefasst. Beispielsweise kapselt die Klasse `cRequestTXFragment` Informationen welche für Anfragen jedweder Art benötigt werden: eine Priorität, einen Identifikator, der in die entsprechende Antwort integriert werden muss sowie

[1] Klassennamen sind stets mit einem initialen ‘c’ gekennzeichnet

[2] besser bekannt als *enum*

eine Adresse, die das Ziel der Antwort beschreibt. Je nach dem, was eine Nachricht alles beinhalten muss, können nun durch Vererbung Einträge verschiedener Oberklassen zusammengefügt werden. Mit Hilfe des `dynamic_cast`-Befehls lassen sich so kombinierte Klassen wieder ein einzelne Teile zerlegen, so dass ein Modul gar nicht den gesamten Nachrichtentyp kennen muss, um die benötigten Informationen extrahieren zu können. Somit können neue Komponenten oder Nachrichtenklassen generiert werden, ohne dafür alle bestehenden Module anpassen zu müssen.
Allerdings verbleibt die Frage, wie rechenaufwändig eine häufige Konsultation des *virtual method table*[3] - kurz *vtable* - ist, was durch den `dynamic_cast`-Befehl intern durchgeführt werden muss.

4.1.2 Nachrichtenformate von und zu externen Komponenten

Wie bereits mehrfach erwähnt, handelt es sich hierbei um die Schnittstelle des Block-Caches mit den anderen Komponenten des *IGOR*-Dateisystems. Da es in der verwendeten Prozess-Netzwerk-Umgebung nicht zu Nachrichtenverlusten kommen kann, genügen hier zwei Mitteilungstypen: Anfragen bzw. Aufträge oder Befehle und die zugehörigen Antworten. Diese werden in den folgenden Abschnitten 4.1.2.1 und 4.1.2.2 vorgestellt.
Da alle Nachrichtenklassen ähnliche Verwandschaftsverhältnisse haben bietet es sich an, alle Typen in ein einziges UML-Diagramm einzutragen. Abbildung 4.1 bietet eine solche Übersicht. In der oberen Hälfte sind die externen Nachrichtentypen eingezeichnet, in der Unteren die internen Klassen. Beide teilen sich die in Mitte befindlichen Fragment-Klassen.

4.1.2.1 Anfragen oder Befehle an den Block-Cache

Für alle Aufträge oder Befehle an den Block-Cache wird eine einzige Klasse verwendet: `cBCRequest`. Diese ererbt von den Fragment-Klassen `cBlockTXFragment` und `cRequestTXFragment` bereits alle Variablen, die für einen Blocktransfer und allgemeine Auftragsabwicklung benötigt werden. Dazu gehören unter anderem die Kennung und Priorität des Auftrags sowie eine BlockId und, falls vorhanden, der zugehörige Block-Inhalt. Neben der Priorität, mit der ein eingehender Block abgelegt werden soll, fehlt vor allem noch ein Kommando, um was für eine Art Auftrag es sich hierbei handelt. Die verschiedenen unterstützen Befehle sind im Aufzählungs-Typ `eBCOrders` gesammelt und umfassen folgende Kommandos: *load* und *store* für Blocktransfers, *changePriority* um die Priorität eines bereits gespeicherten Blockes zu verändern und *startFSCheck* um den Zwischenspeicher anzuweisen mit einer Konsistenzprüfung zu beginnen. Zur Beendigung gibt es zuletzt *shutdownAndFlush*, das dazu dient alle noch offenen Aufträge abzuarbeiten, um danach in einen konsistenten Zustand überzugehen. Nachdem ein solches Kommando empfangen wurde, werden alle neu eingehenden Anfragen abgewiesen.

4.1.2.2 Antworten des Zwischenspeichers

Alle Anfragen an die Block-Cache-Komponente werden, nachdem ein Ergebnis vorliegt, durch die Klasse `cBCResponse` beantwortet. Im Wesentlichen handelt es bei

[3] *C++* interne Datenstruktur, welche die Verwandschafts-Verhältnisse von Klassen aufzeichnet

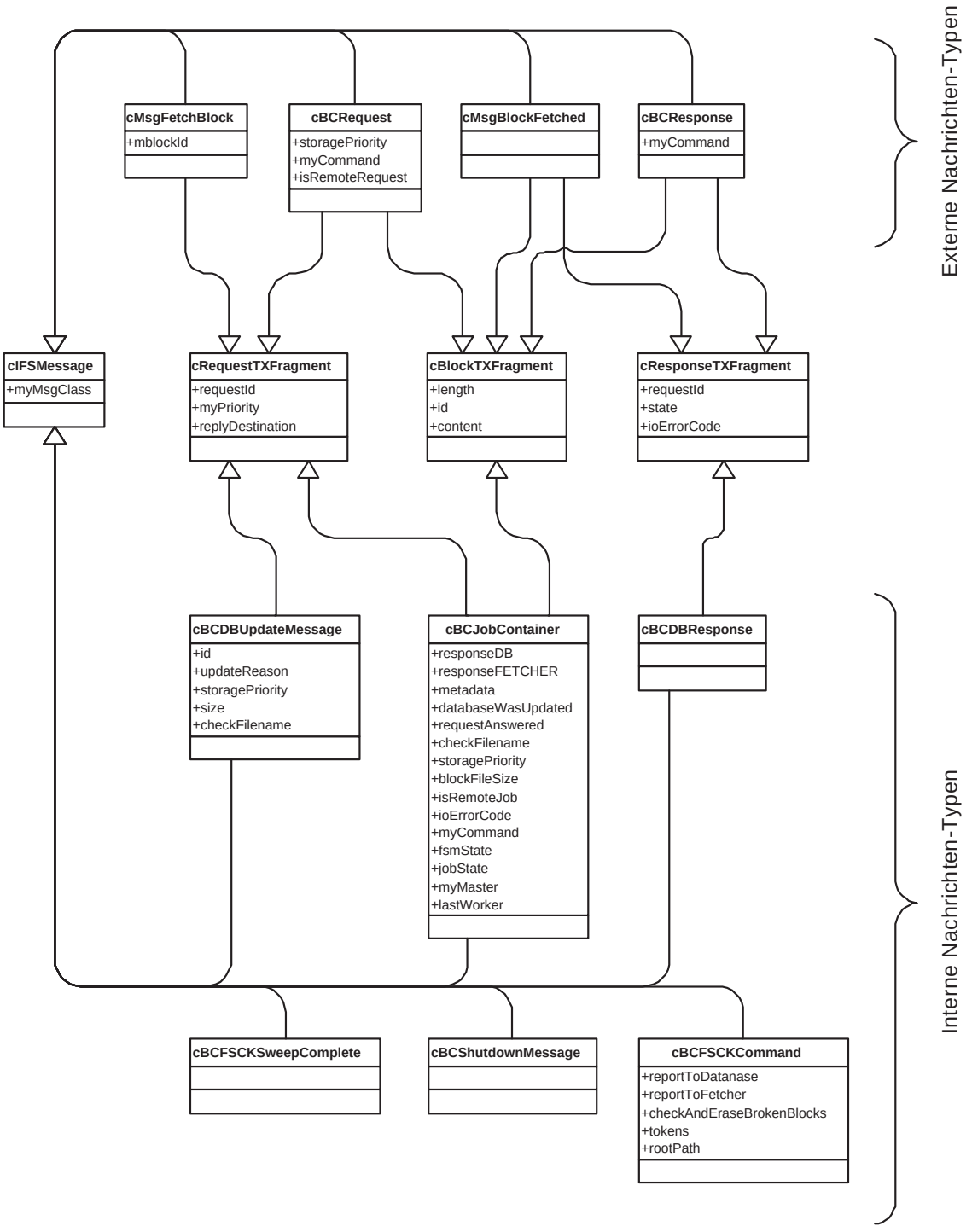


Abbildung 4.1: UML-Diagramm aller Nachrichtenklassen

den Antworten entweder um Blocktransfers (als Antwort auf einen *load*-Befehl) oder Erfolgs- bzw. Misserfolgs-Meldungen. In die Antwortklasse muss also ein Block, ein Status sowie ein Fehlercode eingebettet werden können. Diese Funktionalität wird bereits durch die beiden generischen Fragment-Klassen `cResponseTXFragment` und `cBlockTXFragment` gekapselt, so dass die Antwortklasse `cBCResponse` lediglich von diesen beiden Fragmenten erben muss.

4.1.2.3 Kommunikation mit der Block-Fetcher Komponente

Wie so viele Komponenten des Dateisystems, so ist auch die Block-Fetcher Komponente bisher noch nicht implementiert. Da der Block-Cache aber eine Schnittstelle zu diesem Modul benötigt, wurden bereits zwei einfache Klassen für die Kommunikation definiert. Damit wird bereits die Grundlage für einen später in Kapitel 5 durchgeführten Test gelegt.
Eine Anfrage an den Block-Fetcher benötigt neben den üblichen Nachrichtenkennungen lediglich eine BlockId, nach der gesucht werden soll. Die Anfrageklasse `cMsgFetchBlock` leitet sich daher wie zu erwarten von der `cRequestTXFragment` Fragment-Klasse ab und fügt selbst nur noch besagte BlockId hinzu. Entsprechend der Antwort vom Block-Cache, muss auch die des Fetchers, `cMsgBlockFetched`, einen Block und eventuelle Fehlercodes enthalten. Demnach ist sie aus den Fragmenten `cBlockTXFragment` und `cResponseTXFragment` zusammengestellt. Tatsächlich unterscheidet sich diese Klasse nur durch den Namen von `cBCRequest`.

4.1.3 Kommunikation mit internen Komponenten

Neben den essentiellen, externen Nachrichten zwischen dem Block-Cache und den anderen Modulen des geplanten Dateisystems, existiert auch innerhalb des Zwischenspeichers Kommunikationsbedarf. In diesem Bereich ist vor allem die Klasse `cBCJobContainer` wichtig, die im folgenden Abschnitt 4.1.3.1 vorgestellt wird.

4.1.3.1 cBCJobContainer

Die Klasse `cBCJobContainer` ist die Repräsentation eines Auftrages innerhalb des Zwischenspeichers und der dazu gehörigen Module. Daher leitet sich diese Klasse genau von den selben Fragmenten wie `cBCRequest` ab. Zusätzlich kommen jedoch noch einige weitere Klassenvariablen hinzu: der derzeitige Verarbeitungsfortschritt, der Zustand dieses Auftrages (bereit, laufend oder blockiert) sowie Informationen darüber, von welchem Arbeiter-Thread der Auftrag zuletzt bearbeitet wurde bzw. derzeit bearbeitet wird. Auf diesem Weg bleibt der Verwaltungs-Thread automatisch über den Zustand seiner Arbeiter-Threads informiert, ohne das dazu ein neuer Nachrichtentyp eingeführt werden muss. Darüber hinaus sind, ähnlich wie *Mount-Punkte*[4] im virtuellen Dateisystem, *Speicher-Punkte*[5] enthalten, an die eingehende Mitteilungen von kooperierenden Komponenten angehängt werden können. Bei einem Nachrichteneingang wird der entsprechende Auftrag wieder in den Zustand 'bereit' versetzt und an einen Arbeiter-Thread übergeben. Somit erhält auch er Zugang zu allen, diesen Auftrag betreffenden, bekannten Informationen.
Um einen weiteren Nachrichtentyp zwischen Verwaltungs-Thread und den Arbeiter-Threads einsparen zu können, erbt der Job-Container zusätzlich von `cIFSMessage`,

[4]Ein Verzeichnis über das ein weiteres Dateisystem zugänglich ist.
[5]Hierbei handelt es sich gerade um einen entsprechenden Klassen-Zeiger.

der Nachrichtenklasse des *IGOR*-Dateisystems. Somit lassen sich diese Container direkt über das Prozess-Netzwerk zwischen den einzelnen Cache-Modulen hin- und hersenden.

4.1.3.2 cBCDBUpdateMessage und cBCDBResponse

Der Nachrichtentyp `cBCDBUpdateMessage` dient dazu alle Ereignisse im Zwischenspeicher, die in der Block-Datenbank erfasst werden müssen, zu kapseln und an den Datenbank-Thread weiterzuleiten. Wann immer also ein Arbeiter-Thread einen Block liest, prüft, schreibt, dessen Priorität ändert oder entfernt wird eine solche Nachricht generiert und verschickt.
Wie bei allen Aufträgen, so wird auch hier, nachdem die Datenbank aktualisiert wurde, eine Quittungs-Nachricht in Form einer `cBCDBResponse`-Instanz an den Verwaltungs-Thread zurückgeschickt. Dort wird sie dann an den *Speicherpunkt* des entsprechenden Auftages angehängt, so dass der nächste Arbeiter über den Erfolg der Datenbankaktualisierung informiert ist.

4.1.3.3 cBCFSCKCommand und cBCFSCKSweepComplete

Dieses Mitteilungs-Paar kommt für die Kontrolle des Dateisystem-Prüf-Threads zum Einsatz und stellt ebenfalls wieder ein Auftrag-&-Quittung-Muster dar. Mit Hilfe von `cBCFSCKCommand` wird ein Such-Kommando mit dessen zugehörigen Parametern gekapselt. Diese Parameter umfassen dabei ein Quellverzeichnis, in dem Blöcke gesucht werden sollen, die Ziele, wohin gefundene Blöcke gemeldet werden sollen sowie die maximale Anzahl gleichzeitiger Prüfaufträge. Die letzte Angabe dient dabei der Stau- bzw. Überlastkontrolle, da davon auszugehen ist, dass Verzeichnisse schneller durchsucht werden können, als Arbeiter-Threads gefundene Blöcke verifizieren können.
Nachdem das im Kommando übergebene Verzeichnis durchsucht wurde und alle dabei generierten Prüfaufträge abgeschlossen wurden, verschickt der Prüf-Thread eine `cBCFSCKSweepComplete`-Meldung. Sie dient als Quittung für das Prüfkommando und benötigt keinerlei eigene Variablen. Alle Information ist bereits in den Typ kodiert.

4.1.3.4 cBCShutdownMessage

Zuletzt gibt es eine ebenfalls parameterlose Informationsnachricht, die, wie man sich bereits denken kann, den Abbruch der aktuellen Arbeit befiehlt. Man beachte, dass `cBCShutdownMessage` eine zwischenspeicherinterne Nachricht ist, die lediglich vom Verwaltungs-Thread verschickt werden darf. Auch ist es der Konsistenz des Zwischenspeichers nicht dienlich, die ohnehin nicht lange laufenden Arbeiter-Threads abzubrechen, so dass diese Abbruchsnachricht derzeit nur an den lange laufenden Prüf-Thread und an den Datenbank-Thread verschickt wird. Die Datenbank-Komponente wird nach Erhalt dieser Mitteilung keine neuen Verdrängungs-Aufgaben generieren - der Cache wird in einen sicheren und konsistenten Zustand überführt.

4.2 Block-Cache Komponenten

Im nun folgenden Hauptteil dieses Kapitels geht es um die Umsetzung der vier Komponenten, aus denen sich der Block-Zwischenspeicher zusammensetzt: Zunächst

wird in 4.2.1 das Verwaltungs-Modul beschrieben, worauf anschließend 4.2.2 den Arbeiter-Thread und 4.2.3 die Umsetzung der Datenbank vorstellt. Zuletzt wird sich Unterabschnitt 4.2.4 mit der Implementation der Dateisystemprüfung beschäftigen. Aus Zeit- und Arbeitsgründen gilt für alle Komponenten, wo immer es möglich ist, auf die bewährte *C++ Standard Template Library*[6] (kurz STL) zurückzugreifen. Darüber hinaus müssen sich alle Klassen, die durch einen eigenen Thread im Prozess-Netzwerk betrieben werden sollen, von der dafür vorgesehenen Basisklasse `cModule` ableiten.

4.2.1 Kommunikations- und Verwaltungsprozess

Die Aufgabe dieses Prozesses ist die Organisation und Versorgung von Arbeiter-Threads mit ebenfalls zu betreuenden Aufträgen. Dabei dürfen niemals zwei Aufträge, welche die gleiche Block-Kennung betreffen, zeitgleich gestartet werden. Verantwortlich für die Durchsetzung all dieser Bedingungen ist die Klasse `cBlockCache`. Mit deren Instanzierung beginnt der neu entstandene Verwaltungs-Thread automatisch die restlichen Komponenten zu erzeugen. Somit besitzt er natürlich automatisch Kenntnis über alle Zeiger bzw. Transportadressen im Prozess-Netzwerk.

Organisation der Arbeiter

Die Adressen von Arbeiter-Threads befinden sich in zwei verschiedenen Datenstrukturen:

1. `typedef vector<cBCWorker*> tWorkerVector;`
2. `typedef queue<cBCWorker*> tWorkerQueue;`

Bei der Instanzierung der Arbeiter-Komponenten werden alle daraus resultierenden Klassenzeiger in den STL-Vektor[7] `tWorkerVector` aufgenommen. Diese Informationen werden derzeit zwar noch nicht benötigt, in einem zukünftigen[8] Entwicklungsschritt soll jedoch mit Hilfe dieses Vektors die Arbeiter-Menge zur Laufzeit variiert werden können.

`tWorkerQueue` ist eine *first in, first out* Warteschlange in der sich Zeiger auf unbeschäftigte Arbeiter befinden. Somit können eingehende Aufträge an freie Arbeiter verteilt werden. Kehrt ein `cBCJobContainer` zurück, wird durch seine 'lastWorker'-Variable der entsprechende Arbeiter-Thread identifiziert und wieder an die Schlange angehängt. Während sich also alle existierenden Arbeiter stets im Vektor befinden kann die Warteschlange auch leer sein.

Auftragsverwaltung

Die Auftragsverwaltung ist die zentrale Angelegenheit der Klasse `cBlockCache`. Auch hierfür existiert ein Daten-Typ, in den alle bekannten und unfertigen Aufträge eingetragen sind:

[6] Eine Schablonen-Sammlung aus zumeist typunabhängigen Datenstrukturen und Algorithmen.
[7] Ein Feld dynamischer Länge.
[8] Außerhalb dieser Arbeit.

```
typedef map<tRequestId, cBCJobContainer*> tKnownJobs;
```

Diese STL-Map[9] dient lediglich der Entscheidung, ob es noch unfertige Aufträge gibt, denn das Dateisystem darf erst dann endgültig ausgehängt werden, wenn alle Aufträge vollendet sind. Entsprechend wird das in 4.1.2.1 beschriebene *shutdownAndFlush*-Kommando erst beantwortet, nachdem `tKnownJobs` leer ist. Des weiteren kann aus dieser Datenstruktur jederzeit der Zustand der einzelnen Aufgaben abgefragt werden, was für Fehlersuchen hilfreich sein kann.

Schickt nun ein Arbeiter-Thread einen blockierten Auftrag zurück, muss ihm, sofern vorhanden, eine neue Aufgabe zugeteilt werden. Offensichtlich ist es sehr ungeschickt, die gesamte Hash-Tabelle nach Aufträgen im Zustand 'bereit' zu durchsuchen und sie anschließend nach Priorität zu ordnen. Daher existiert, genau wie bei den Arbeitern, auch hier eine zusätzliche Datenstruktur für die unvollendeten und bereiten Aufgaben:

```
typedef priority_queue<cBCJobContainer*,
vector<cBCJobContainer*>, cBCJobContainer> tJobQueue;
```

Genau wie bei den Arbeitern handelt es sich auch hier wieder um eine Warteschlange, jedoch mit dem Unterschied, das diese nach einer Priorität geordnet ist. Priorität bedeutet hierbei, dass zunächst nach der Auftragspriorität geordnet und anschließend innerhalb der entstehenden Prioritätsklassen ein weiteres Mal nach Fortschritt sortiert wird. All dies wird, durch den in `cBCJobContainer` überladenen, *less*-Operator erreicht.

Somit entsteht eine Warteschlange, die mit der aus Betriebssystemen bekannten *ready-queue* für Threads eng verwandt ist. Wann immer ein neues Element in eine der Datenstrukturen `tJobQueue` oder `tWorkerQueue` eingefügt wird, kann jetzt, wenn beide Warteschlangen nicht leer sind, ein Auftrag an einen Arbeiter vergeben werden.

Atomizität[10] auf Block-Ebene.

Wie bereits gesagt, muss dafür gesorgt werden, dass niemals mehrere Aufträge auf den gleichen BlockIds zur selben Zeit gestartet werden bzw. in `tJobQueue` gelangen. Im Prinzip zerfällt also die Menge aller Aufträge in zwei Teile: die zu bearbeitenden Aufträge und die wartenden Aufträge. Offensichtlich ist es auch hier ungeschickt, so selten eine Kollision auch sein mag, jeden neuen Auftrag auf Kollision mit dem Gesamtbestand zu prüfen ($O(n)$). Speziell für diesen Zweck werden daher zwei weitere Datenstrukturen eingeführt:

1. `typedef set<cBCMutexHelper> tActiveBlocks;`
2. `typedef map<cBCMutexHelper, tJobQueue> tWaitingJobs;`

Das STL-Set[11] `tActiveBlocks` speichert mittels der Hilfsklasse `cBCMutexHelper` die BlockIds, die gerade von aktiven Aufträgen angefasst werden. Falls ein neuer Auftag hinzukommt, dessen BlockId bereits in `tActiveBlocks` hinterlegt wurde, nimmt

[9] Eine Hash-Tabelle.

[10] besser bekannt als 'mutual exclusion'.

[11] Vergleichbar mit einem Heap – einem standard Container, der 'einfügen', 'suchen' und 'entfernen' in logarithmischer Zeit beherrscht.

ihn die Hash-Tabelle `tWaitingJobs` auf. Entsprechend muss bei Fertigstellung einer Anfrage ermittelt werden, ob es weitere Anfragen auf dieser BlockId gab, wozu lediglich `tWaitingJobs` befragt werden muss. Innerhalb der Hash-Tabelle dienen Instanzen der priorisierten Warteschlange `tJobQueue` der Kollisionsauflösung. Somit ist sichergestellt, dass hoch priorisierte Aufträge zuerst von der Wartebank geholt werden.

4.2.2 Ein Arbeiter-Prozess

Da die Arbeiter, um austauschbar zu bleiben, keine eigenen Zustände besitzen dürfen, gibt es in diesem Abschnitt natürlich auch keine Datenstrukturen vorzustellen. Tatsächlich bestehen Arbeiter-Threads lediglich aus einem großen endlichen Automaten zur Auftragsverarbeitung und einer ebenfalls zustandslosen Ein- und Ausgabe Klasse `cBCDiskWriter`, präsentiert in 4.2.2.1. Bei allen eingehenden Nachrichten vom Typ `cBCJobContainer` werden zunächst die 'lastWorker' Zeiger überschrieben und anschließend der aktuelle Fortschritt dieses Auftrags als Startzustand des Automaten hergenommen. Mit der Verarbeitung des Auftrages kann nun begonnen oder fortgesetzt werden.

Da es mehrere verschiedene Aufgabentypen gibt, existieren diverse Wege durch den endlichen Automaten. Beispielsweise könnte bei einem einfachen 'load'-Auftrag festgestellt werden, dass der entsprechende Block im unterliegenden Dateisystem korrumpiert wurde. Nun gilt es, eine neue Kopie aus dem Peer-to-Peer-Netz laden zu lassen, um den defekten Block im Zwischenspeicher zu ersetzen. Aus einem ursprünglichen 'load' wurde also ein 'store'-Auftrag.

Zur Übersicht zeigen die Abbildungen 4.2 und 4.3 die möglichen Wege im Automaten. Dabei sind zwei Arten von Zuständen hervorgehoben: Die Startzustände für die verschiedenen Aufgaben sind grün, während für Fehlerbehandlung relevante Zustände orange eingefärbt sind. Auch ein dritter, wenn auch nicht markierter, Typ verdient eine Erwähnung: Zur Herstellung von Statistiken können die mit 'finish' beginnenden Zustände verwendet werden.

4.2.2.1 Hilfsklasse cBCDiskWriter

Wie man bereits aus dem Namen dieser Klasse ableiten kann, handelt es sich bei ihr um eine Schnittstelle zum unterliegenden Dateisystem. Das Ziel ist es, alle Ein- und Ausgabefunktionalität in `cBCDiskWriter` so zu bündeln, dass nach außen hin Operationen auf Block-Ebene bereitgestellt werden können. Zum Beispiel sind für das Lesen und Schreiben von Blöcken folgende Methoden enthalten:

```
int writeBlock(cBlockTXFragment*, bool, iBCMetadata*);

int readBlock(cBlockTXFragment*);
```

Wie anhand dieser Signaturen gut erkennbar ist, kommen auch hier wieder die Klassenfragmente aus 4.1 zum Einsatz: Ein `cBCJobContainer`-Objekt kann also einfach

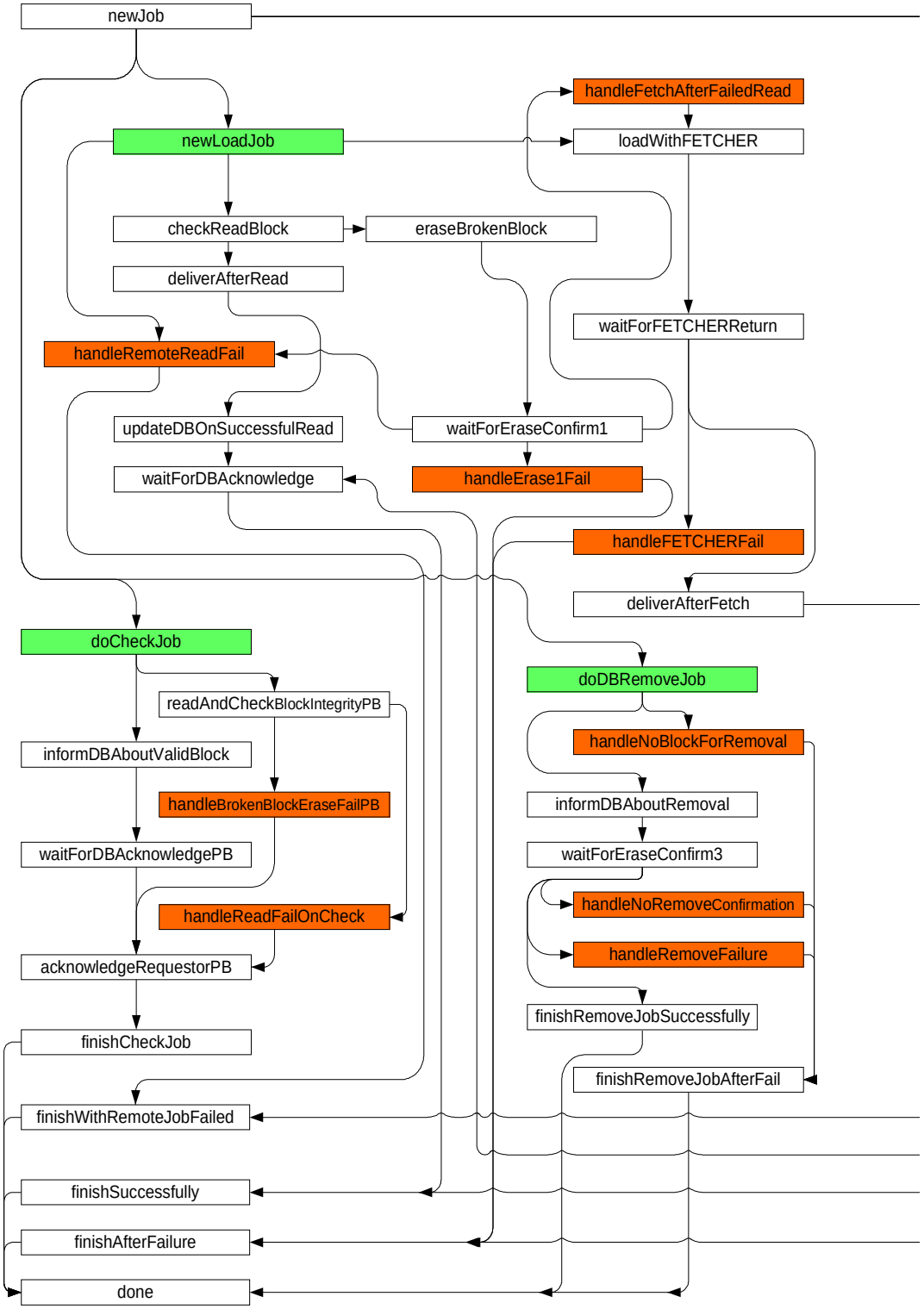


Abbildung 4.2: Zustandsautomat eines Arbeiter-Threads - Teil 1

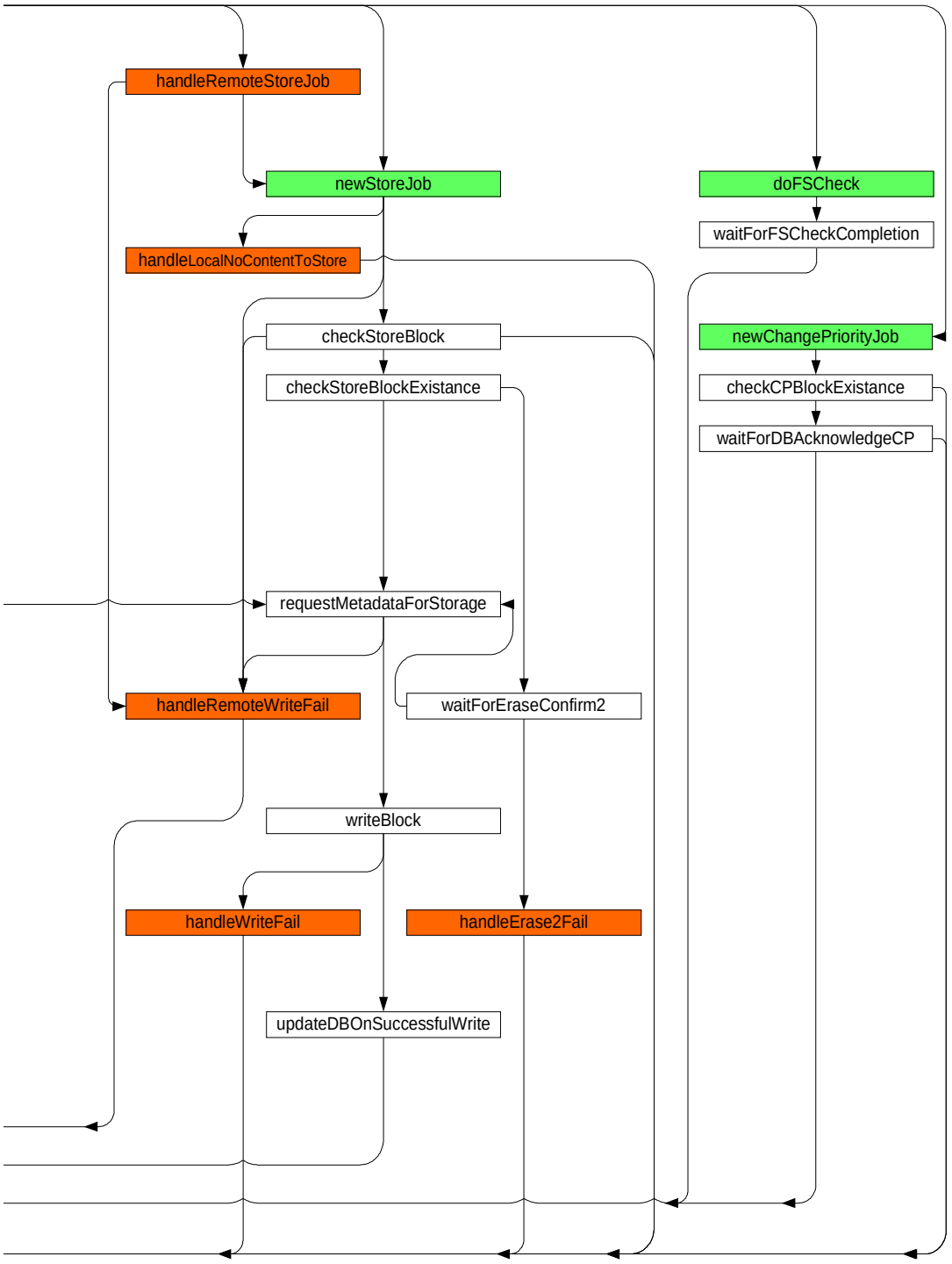


Abbildung 4.3: Zustandsautomat eines Arbeiter-Threads - Teil 2

als erster Parameter übergeben werden. Neben dem 'darf-überschrieben-werden'-Wahrheitswert ist vor allem der dritte Parameter der `writeBlock`-Methode interessant: Die Schnittstelle[12] `iBCMetadata` beschreibt einen generischen Typ für Metadaten, die in die entsprechende Block-Datei eingebettet werden. Beispielsweise greift der Datenbank-Thread auf diese Funktionalität zurück, um die in 3.2.4 beschriebenene Verbindung von einer Block-Datei zu seinem Zeitstempel herzustellen. Darüber hinaus können aber auch beliebige weitere Informationen in Form von Metadaten integriert werden.

Die Bündelung von 'echten' Ein- und Ausgabefunktionen in dieser Klasse bringt eine Reihe von Vorteilen mit sich. Am wichtigsten ist hierbei, dass diese Art von Funktionen im Allgemeinen abhängig vom Betriebssystem sind. Falls also das *IGOR*-Dateisystem auf ein Betriebssystem portiert werden soll, welches nicht auf UNIX basiert, muss für die Block-Cache Komponente lediglich diese Klasse angepasst werden. Auch könnten durch eine entsprechend spezialisierte Klasse evtl. vorhandene, zusätzliche Fähigkeiten des unterliegenden Dateisystems nutzbar gemacht werden. Denkt man beispielsweise an Plugin-basierte Dateisysteme[13], so könnten durch eine entsprechende Erweiterung sowohl Verknüpfungen als auch Blöcke direkter, effizienter und evtl. kostengünstiger auf dem Datenträger abgelegt werden.
Darüber hinaus könnte man dieses System zur Leistungsmessung in einem Simulator ausführen wollen, wofür echtes, langsames Lesen und Schreiben im unterliegenden Dateisystem unpraktisch wäre. Entsprechend könnte auch hier eine für den Simulator angepasste Variante von `cBCDiskWriter` entwickelt werden.

4.2.3 Datenbank-Prozess

Der gesamte Datenbank-Thread ist in der Klasse `cBCLinkDatabase` implementiert, wobei sich diese Komponente genau wie alle anderen Threads im Prozess-Netzwerk natürlich auch von `cModule` ableitet.

Genau wie bei den Arbeitern wurden die einzelnen Schritte in der Datenbank ebenfalls als endlicher Automat modelliert. Auch hier gibt es wieder eine so große Anzahl an Schritten und möglichen Fehlerzuständen, so dass eine Grafik eine deutlich bessere Übersicht bietet. Daher zeigen Abbildungen 4.4 und 4.5 den Automaten, der eingehende Aufträge im Datenbank-Thread verarbeitet. Genau wie beim Arbeiter-Automaten sind die einzelnen Startzustände grün und Fehlerbehandlungs-Zustände orange markiert. Auch existieren wieder einige unmarkierte 'finish'-Zustände, aus denen Daten für Statistiken extrahiert werden könnten. Der einzige Unterschied zum Arbeiter-Automat ist, dass die Zustandsvariable hier zu `cBCLinkDatabase`, der Datenbankklasse, gehört.

Des weiteren ist die Datenbank-Komponente, im Unterschied zu den Arbeitern nicht zustandslos. Hilfsklassen vom Typ `cBCDiskQuota` werden dazu verwendet, sowohl den aktuellen Platzbedarf der einzelnen Prioritätsklassen im unterliegenden Dateisystem, als auch entsprechende Beschränkungen zu speichern. Somit können überlaufende Prioritätsklassen identifiziert werden, was zu entsprechenden Verdrängungsaufträgen führt. Diese Aufträge werden dann genau wie alle anderen an die Verwaltungs-Komponente geschickt, wobei der Datenbank-Thread hierbei der Auftragsteller ist.

[12] Wie Klassen ein 'c', so haben auch alle Schnittstellen ein 'i', von Interface, vorangestellt.
[13] Z. B. ReiserFS4 - siehe http://www.namesys.com, 13.07.2006

Genau wie der Arbeiter-Thread, verwendet auch dieses Modul eine Instanz der Klasse `cBCDiskWriter`. Sie dient dazu die nötigen Verknüpfungen ins unterliegende Dateisystem zu schreiben und die Metadaten einzelner Blöcke zu lesen oder nach einer Re-Referenzierung zu modifizieren. Als einzige Ausnahme verbleibt derzeit noch die `cBCDiskQuota`-Instanz, die den aktuellen Festplatten-Platzverbrauch der einzelnen Prioritätsklassen kennt. Zur Speicherung ihrer Daten in eine Quota-Datei wird gegenwärtig noch eine extra Methode benötigt.

Da der Datenbank-Thread auch für die Umsetzung der Verdrängungsstrategie verantwortlich ist, wird nach jedem eingelagerten Block überprüft, ob die Größen der eingelagerten Elemente die Gesamtkapazität überschreiten. Zu diesem Zweck kommen die `cBCDiskQuota`-Instanzen zum Einsatz. Falls die Kapazität überschritten wurde, wird ein Verdrängungs-Auftrag generiert, der über den Verwaltungs-Thread abgewickelt wird. Es kann also immer nur ein einziger Block gleichzeitig verdrängt werden. Dabei wird stets der älteste Eintrag der untersten Prioritätsklasse, die ihren Grenzwert überschritten hat, ausgewählt. Eine entsprechende Methode zur Selektion wird von `cBCDiskQuota` zur Verfügung gestellt.

4.2.4 Dateisystem-Prüf-Prozess

Eigentlich ist der Begriff Prüfprozess an dieser Stelle nicht ganz richtig. Zwar dient er tatsächlich dazu die Prüfung des Dateisystems voranzutreiben, die einzelnen Blöcke werden jedoch nicht von ihm selbst überprüft. Stattdessen generiert er einfach für jeden gefundenen Block bzw. jede gefundene Verknüpfung einen Prüfauftrag, der an die Verwaltungs-Komponente zur Verarbeitung geschickt wird. Diese wiederum verteilt die Prüfaufgaben auf die Arbeiter, die zwischenzeitlich Rücksprache mit der Datenbank halten müssen.

Die Aufgabe dieses, in der Klasse `cBCFSCheck` implementierten, Threads ist wie folgend: Er muss das unterliegende Dateisystem, beginnend bei einem zu übergebenden Wurzelvereichnis, durchsuchen und gefundene Blöcke und Verknüpfungen melden. Zu diesem Zweck besitzt er zwei Instanzen einer Datenstruktur folgenden Typs:

```
typedef stack<string> tStringStack;
```

Der erste Stack[14] speichert die Dateinamen gefundener Blöcke, um anschließend daraus Prüfaufträge zu generieren. Auf einem zweiten Stack werden die dabei gefundenen Unterverzeichnisse abgelegt. Wann immer der Block-Stack leer ist, wird ein neues Verzeichnis vom Verzeichnisstapel geholt und durchsucht. Sobald auch dieser Stapel leer ist und alle noch offenen Prüfaufträge zurückgekehrt sind, ist die 'Prüfung' abgeschlossen - eine `cBCFSCKSweepComplete`-Nachricht wird an Auftragsteller geschickt. Wie bereits in 3.4 erläutert, ist zusätzlich eine Überlastkontrolle nötig, um die Anzahl der gleichzeitigen Prüfaufträge zu limitieren. Diese kann aber ganz einfach durch Begrenzung der Differenz zwischen der Anzahl verschickter Prüfaufträge und empfangenen Quittungen realisiert werden.

Dieser Ansatz birgt, wie dem aufmerksamen Leser nicht entgangen sein wird, das Risiko, dass eine Dateisystemprüfung eventuell niemals terminiert. Denn in vielen

[14] Ein STL-Stack stapelt Elemente aufeinander und gibt immer nur das jeweils Neueste zurück (LIFO).

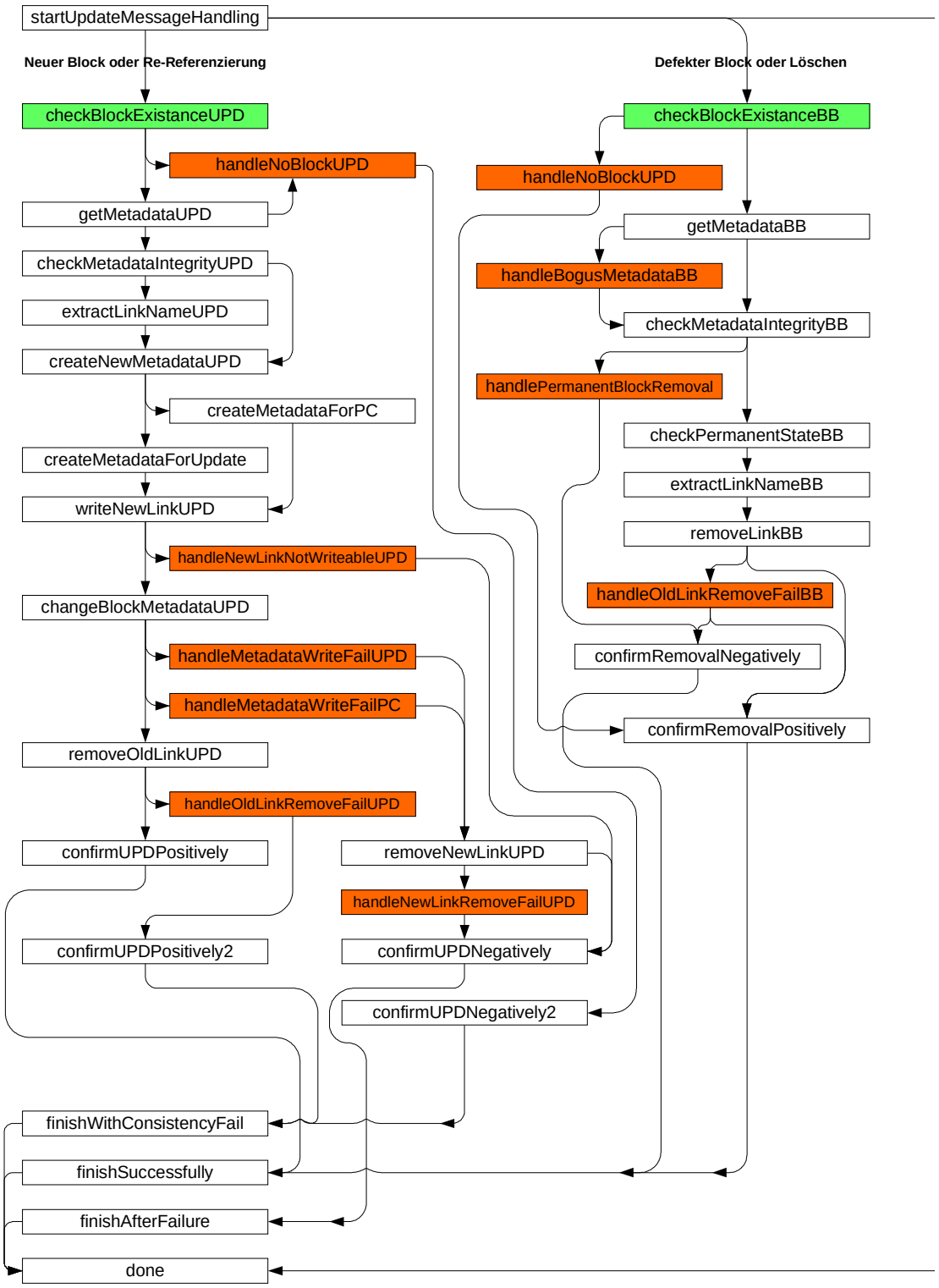


Abbildung 4.4: Zustandsautomat Datenbank-Thread - Teil 1

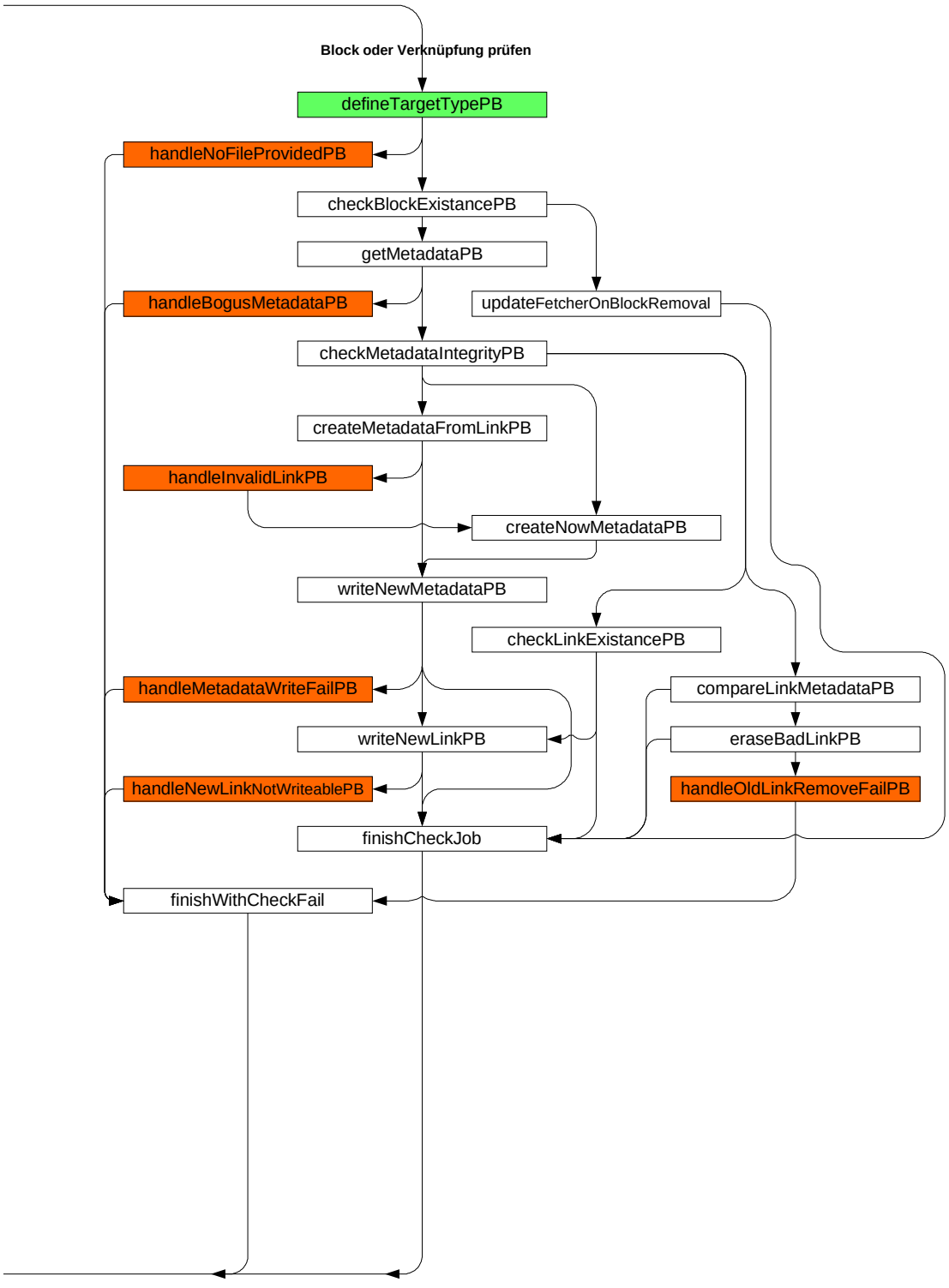


Abbildung 4.5: Zustandsautomat Datenbank-Thread - Teil 2

Dateisystemen ist es möglich Verknüpfungen zu verwenden, egal ob hart oder symbolisch, mit deren Hilfe sich Schleifen im Verzeichnisbaum herstellen lassen. Somit würde der Verzeichnis-Stack immer wieder ein neues Verzeichnis finden, das Durchsuchen also niemals terminieren.

Wie bereits bei `cBCDiskWriter` gezeigt, so ist auch das Durchsuchen von Verzeichnissen nicht in allen Betriebssystemen gleich geregelt. Also müsste in dieser Klasse, im Falle einer Portierung zu einem nicht UNIX-System, die Methode zur Durchsuchung eines gegebenen Verzeichnisses ebenfalls leicht überarbeitet werden. Da aber zur Zeit keinerlei Portierungen der *fuse*-Bibliothek vorliegen, wäre diese Anpassung die kleinste Schwierigkeit der ganzen Angelegenheit.

Eine weitere Besonderheit in der Funktionalität der Klasse `cBCFSCheck` ist, dass ein weiterer Prozess, an den gefundene Blöcke gemeldet werden sollen, definiert werden kann. Gedacht ist hier, dass bei einer Konsistenzprüfung des Dateisystems auch automatisch die Block-Fetcher-Komponente über die gespeicherten Blöcke informiert wird. Dies soll dazu dienen, die Blöcke im Peer-to-Peer-Netzwerk neu publizieren zu können, was für diese Art von Netzwerken typisch ist. Da es keinen zentralen Index gibt, nehmen zumindest die Netzwerkteilnehmer, die das Netzwerk unfreiwillig verlassen, Informationen mit; Informationsverlust entsteht.

5. Evaluation

Dieser Abschnitt befasst sich mit der Bewertung der Block-Cache Komponente. Zunächst stellen einfache Funktionstests den korrekten Ablauf der einzelnen Cache-Operationen sicher, später werden diese Versuche in so genannte *Burst*-Tests erweitert. Hierbei handelt es sich um ein Verfahren, dass mit dem Impulsantwort-Test für Regelungssysteme eng verwandt ist. Anstatt eines einzelnen Impulses werden in diesem Fall schnellstmöglich tausende gleichartige Anfragen an den Zwischenspeicher geschickt um dessen Verhalten zu untersuchen. Sowohl vor dem *Burst*, als auch danach, dürfen keinerlei weitere Anfragen geschickt werden.

Leider sind die Module vor dem Zwischenspeicher bisher noch nicht fertiggestellt. Daher kann auch der Block-Cache bisher nur separat getestet werden, während populäre Dateisystem-Benchmarks[1] wie beispielsweise *bonnie++*[2] noch nicht genutzt werden können. Davon abgesehen ist es auch nicht Ziel dieser Arbeit, das Gesamtsystem zu bewerten.

5.1 Allgemeines zu Tests

Liest man die Bezeichnung 'Zwischenspeicher' kommt man schnell auf die Idee, wie beispielsweise bei Prozessorcaches, Trefferraten zu messen. Dieses Maß ist jedoch für den Block-Cache kaum angemessen. Zum einen könnten entsprechende Arbeitsmengen und *Cachetraces*[3] so synthetisiert werden, dass 100 prozentige Trefferraten entstehen. Zum anderen kann auf dem gleichen Weg auch ein Test, der keinen einzigen Treffer erzielt, generiert werden. Im einfachsten Fall erzeugt man hierfür eine periodische Folge, die beschreibt, in welcher Reihenfolge Elemente im Zwischenspeicher anzufassen sind. Stellt man nun sicher, dass diese Folge keine doppelten Einträge enthält, und die Anzahl der Einträge die Größe des Caches übersteigen, so kann bei einer Re-Referenzierung niemals ein Treffer erfolgen.
Die Alternative wäre den Cache mit einer zufälligen Folge an Anfragen zu testen. Sieht man davon ab, dass real zumeist etwas Lokalität[4] und Zusammenhang einzel-

[1] Tauglichkeits- bzw. Geschwindigkeits-Tests.

[2] siehe http://sourceforge.net/projects/bonnie/, 13.07.2006

[3] Beschreiben welche Elemente wann angefasst werden, so dass Caches simuliert werden können, woraus sich Trefferraten und andere Maßzahlen ermitteln lassen.

[4] Angeforderte Daten kommen oftmals aus nahe zusammenliegenden Speicherbereichen.

ner Anfragen gilt, ist die erwartete Trefferrate bei einem komplett zufällig erzeugten Cachetrace in unseren Größenordnungen praktisch Null. Folgende Formel beschreibt die Wahrscheinlichkeit eines Treffers, wobei nach 2.4.1 $n = 2^{160}$ und $m = 100Mio$ eingesetzt werden könnte:

$$1 - (\frac{n!}{n^m \cdot (n-m)!})$$

Die Aussagekraft von Trefferraten bzw. von allem, was auf irgendeine Art vom verwendeten Cachetrace abhängt, ist also kaum brauchbar. Diese Werte eignen sich lediglich für den direkten Vergleich von Systemen bzw. von Verdrängungsstrategien auf dem gleichen System.

Ein weiteres, im Zusammenhang mit Caches oft genanntes Maß, ist der erzielte Geschwindigkeitsvorteil. Da im *IGOR*-Dateisystem der Zwischenspeicher eine andere Bedeutung hat und es gar nicht möglich ist ohne ihn zu arbeiten, kann auch diese Kenngröße nicht genutzt werden. Stattdessen werden im späteren Abschnitt 5.3 der Burst-Tests die Verarbeitungs-Geschwindigkeiten von Anfragen analysiert.

5.2 Funktionstests

Die in diesem Abschnitt vorgestellten Tests prüfen ein paar sehr simple, aber essentielle Aspekte der Block-Zwischenspeicher-Komponente. Zunächst muss sich das Dateisystem aufgrund des heute üblichen nicht persistenten Hauptspeichers jederzeit in einen 'sicheren' Zustand versetzen lassen, was in 5.2.1 überprüft wird. Anschließend komplettiert der Versuch 5.2.2 die Überdeckung aller Kommunikationswege vom und zum Block-Zwischenspeicher. Nach Absolvierung dieser beiden grundlegenden Tests befassen sich die folgenden Unterabschnitte 5.2.3 bis 5.2.6 mit der Hauptfunktionalität des Caches.

5.2.1 Sicheres Beenden des Dateisystems

Wie bereits festgestellt, muss das Dateisystem jederzeit in einen 'sicheren', das heißt konsistenten Zustand gebracht werden können. Dazu sind alle noch im Zwischenspeicher pendelnden, Aufträge zu komplettieren, wobei nach der Ankunft der *shutdownAndFlush*-Nachricht keinerlei neue Aufträge angenommen werden dürfen. Dies gilt insbesondere auch für die internen Dateisystem-Prüfaufträge, sowie für Verdrängungsaufträge des Datenbank-Threads.

Erwartung

Nachdem eine Nachricht vom Typ `cBCRequest` mit dem *shutdownAndFlush*-Befehl an den Block-Zwischenspeicher geschickt wurde, weist er eine folgende Block-Speicher-Anfrage zurück. Stattdessen verschickt er eine Quittung vom Typ `cBCResponse`, die das Erreichen des konsistenten Zustands bestätigt.

Umgebung

Zur Durchführung wird ein eigener Thread für das Test-Modul, das die besagten `cBCRequest`-Nachrichten an den Zwischenspeicher verschickt, erzeugt. Dieser Thread gibt sich gleichzeitig als Block-Fetcher aus, um somit sicherzustellen, dass keinerlei Anfragen des noch leeren Caches stattfinden. Nach Ankunft beider Antworten des Zwischenspeichers ist der Test abgeschlossen.

Resultat

Wie aus der folgenden Programmausgabe ersichtlich wird, hat der Block-Cache nach 10 Millisekunden den sicheren Zustand erreicht, während der darauf folgende Lesebefehl abgewiesen wurde. Der Test ist also erfolgreich.

```
 0 ms - Sende shutdownAndFlush
 0 ms - Sende readBlock
10 ms - Nachricht vom Typ cBCResponse empfangen:
      E/A Fehler  = 0
      Zustand     = Ok
      Befehl      = shutdownAndFlush
      Blocklaenge = 0 Bytes
      BlockId     = 00000000000000000000000000000000000000000
      Block       = NIL
11 ms - Nachricht vom Typ cBCResponse empfangen:
      E/A Fehler  = 0
      Zustand     = Abgewiesen
      Befehl      = loadBlock
      Blocklaenge = 0 Bytes
      BlockId     = 00000000000000000000000000000000000000000
      Block       = NIL
```

5.2.2 Durchreichen einer Anfrage bis zum Block-Fetcher

Der zweite Test sieht die Überprüfung der Kommunikationswege des Zwischenspeichers vor. Neben dem bereits im vorherigen Versuch getesteten Befehls- und Antwortkanal kommt noch der Nachrichtenaustausch mit dem Block-Fetcher hinzu. Die dafür benötigte Klasse `cMsgFetchBlock` ist bereits aus 4.1.2.3 bekannt.

Erwartung

Eine Leseanforderung für eine beliebige (für den Test feste) BlockId wird, nachdem sie an den Block-Cache gesendet wurde, vom Block-Fetcher empfangen. Natürlich handelt es sich bei der empfangenen Nachricht nicht um das Original, sondern um die entsprechende `cMsgFetchBlock` Repräsentation. Die darauf folgende *shutdownAndFlush*-Nachricht wird niemals (im Test 20 Sekunden) bestätigt, da der Block-Fetcher nicht antwortet.

Umgebung

Genau wie im vorangegangenen Versuch, erledigt auch hier ein eigener Test-Thread, der sich gleichzeitig als Block-Fetcher ausgibt, den Probelauf.

Resultat

Auch hier zeigt die Programmausgabe den Erfolg dieses Versuchs:

```
    0 ms - Sende readBlock
    0 ms - Sende shutdownAndFlush
    2 ms - Nachricht vom Typ cMsgFetchBlock empfangen:
         BlockId   = 0100000000000000000000000000000000000001
         RequestId = 0x8074278
         Priorität = normal
20003 ms - Keine shutdownAndFlush Bestaetigung erhalten!
```

5.2.3 Speichern eines neuen Blockes

Nachdem nun die Grundlagen überprüft wurden, beginnt dieser Test die Cache-Operation 'schreiben' zu prüfen. Zu diesem Zweck wird ein Block mit 64 Kb Größe erzeugt und in Form eines *storeBlock*-Auftrages an den Block-Cache übergeben.

Erwartung

Nach Erhalt der Nachricht wird der Zwischenspeicher den Block auf die Festplatte speichern und einen entsprechenden Eintrag im Datenbank-Thread vornehmen. Diese beiden Aktionen können durch das unterliegende Dateisystem überprüft werden. Es muss eine Verknüpfung und eine Block-Datei erzeugt worden sein.

Umgebung

Wie in 5.2.1

Resultat

Der Test ist erfolgreich, wie aus folgender Programmausgabe (gekürzt), und den neuen Dateien (unter der Linie) erschlossen werden kann:

```
  0 ms - Sende storeBlock
 42 ms - Nachricht vom Typ cBCResponse empfangen:
       requestId   = 0x1
       E/A Fehler  = 0
       Zustand     = Ok
       Befehl      = storeBlock
-------------------------------------------------------------------------
blockcache/01/00/0100000000000000000000000000000000000001
linkDB/000/2006/july/09/17/45/0100000000000000000000000000000000000001
```

5.2.4 Lesen des gespeicherten Blockes

Test 5.2.3 hat erfolgreich einen Block in den Zwischenspeicher eingelagert. Der nächste Schritt ist also, diesen Block wieder auszulesen und mit dem Original zu vergleichen.

Erwartung

Zunächst wird erwartet, dass der angeforderte Block in Form einer entsprechenden `cBCResponse`-Nachricht erfolgreich aus dem Block-Cache gelesen wird. Erfolgreich bedeutet hierbei, dass er mit dem Original-Block übereinstimmt. Darüber hinaus muss nach dieser Re-Referenzierung ein neuer Zeitstempel erzeugt worden sein, während der alte nicht länger existiert. Auch dies lässt sich durch einen Blick ins echte Dateisystem verifizieren.

Umgebung

Wie in 5.2.1

Resultat

Test ist erfolgreich (es sind keine weiteren Dateien gespeichert):

```
   0 ms - Sende readBlock
  11 ms - Nachricht vom Typ cBCResponse empfangen:
      requestId   = 0x1
      E/A Fehler  = 0
      Zustand     = Ok
      Befehl      = loadBlock
      Blocklaenge = 65536 Bytes
      BlockId     = 01000000000000000000000000000000000000001
      Block       = 0x8074648
  40 ms - Vergleich mit dem Original: 0 unterschiedliche Bytes
--------------------------------------------------------------------------
blockcache/01/00/01000000000000000000000000000000000000001
linkDB/000/2006/july/09/18/01/010000000000000000000000000000000000000001
```

5.2.5 Ändern der Priorität des gespeicherten Blockes

Erneut wird das in 5.2.3 erfolgreich gespeicherte Element angefasst – diesmal jedoch zur Änderung der Speicherpriorität. Dieser Vorgang muss vor einer beabsichtigten Löschung eines als ‘permanent’-gekennzeichneten Blockes durchgeführt werden. Nach dem Verlust seines ‘permanent’-Status, wird er nach den Regeln für ‘ganz normale’ Blöcke verdrängt.

Erwartung

Auch hier wird mit einer `cBCResponse` Erfolgsmeldung gerechnet. Erneut kann durch das unterliegende Dateisystem die neue Priorität des Blocks überprüft werden. Auch wenn dabei eine neue Verknüpfung erzeugt werden muss, während die Ehemalige nicht mehr bestehen darf, muss der Zeitstempel selbst gleich bleiben – schließlich soll nur die Priorität geändert werden.

Umgebung

Wie in 5.2.1

Resultat

Test ist erfolgreich (es sind keine weiteren Dateien gespeichert):

```
   0 ms - Sende changePriority
  11 ms - Nachricht vom Typ cBCResponse empfangen:
      requestId   = 0x1
      E/A Fehler  = 0
      Zustand     = Ok
      Befehl      = changePriority
-------------------------------------------------------------------
blockcache/01/00/0100000000000000000000000000000000000001
linkDB/002/2006/july/09/18/01/010000000000000000000000000000000000000001
```

5.2.6 Dateisystem-Prüfen

Im letzten Funktionstest soll das Dateisystem einer Prüfung unterzogen werden. Um, abgesehen von der entsprechenden Quittung nach Abschluss der Prüfung, überhaupt eine Möglichkeit zu haben, den Erfolg oder Misserfolg dieses Tests zu kontrollieren, sind ein paar Vorarbeiten nötig: Durch absichtliches Löschen der Verknüpfung und der Quota-Datei (aus 4.2.3 bekannt) der Datenbank, verliert diese ihren gesamten Zustand. Dieser muss durch einen *startFSCheck*-Befehl wiederherstellbar sein.

Erwartung

Nach Abschluss des Datei-System-Prüfvorgangs, muss sowohl die Originalverknüpfung, als auch die Quota-Datei wieder im unterliegenden Dateisystem existieren. Dabei gilt zu beachten, dass die Verknüpfung wieder ihren originalen Zeitstempel und ihre ursprüngliche Priorität erhalten. Beide Informationen sind immer noch erhalten und lassen sich aus den, in die Block-Datei integrierten, Metadaten ableiten.

Umgebung

Wie in 5.2.1

Resultat

Test ist erfolgreich (sowohl Verknüpfung als auch Quota-Datei wurden wiederhergestellt):

```
   0 ms - Sende startFilesystemCheck
  23 ms - Nachricht vom Typ cBCResponse empfangen:
      requestId   = 0x1
      E/A Fehler  = 0
      Zustand     = Ok
      Befehl      = startFilesystemCheck
-------------------------------------------------------------------
blockcache/01/00/0100000000000000000000000000000000000001
linkDB/002/2006/july/09/18/01/010000000000000000000000000000000000000001
```

5.3 Burst-Tests

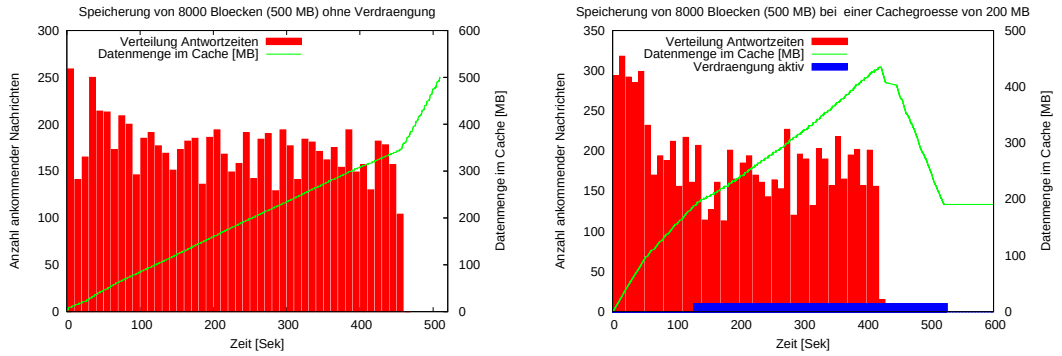
Nachdem die fundamentalen, aber dennoch wichtigen Grundlagentests abgeschlossen sind, kann nun mit den *Burst*-Versuchen begonnen werden. Für jeden Probelauf, 5.3.1 bis 5.3.4, werden zunächst mehrere tausend Anfragen generiert. Erst nachdem alle geplanten Nachrichten im Hauptspeicher bereitstehen, werden sie mittels einer Schleife in das Prozess-Netzwerk geleitet.
Außerdem haben alle Tests noch eine weitere Eigenschaft gemeinsam: Für jeden Probelauf werden genau fünf Arbeiter-Threads erzeugt. Des Weiteren wird die *shutdown*-Nachricht zum Beenden des Block-Caches erst stark verzögert gesendet. Dies dient dazu Cache-interne Operationen, wie zum Beispiel die Verdrängung alter Blöcke in Test 5.3.2, nicht zu blockieren. Unglücklicherweise ist es derzeit noch möglich einem bereits vollen Zwischenspeicher einige weitere Schreibeaufträge zu schicken und den Verdrängungs-Algorithmus mit einer sofort folgenden *shutdownAndFlush*-Mitteilung auszuhebeln. Denn durch eine solche Nachricht werden unter Anderem auch alle neu generierten Verdrängungsaufträge unterbunden – der Zwischenspeicher läuft über.

Während bei den Funktionstests die Geschwindigkeit keinerlei Rolle gespielt hat, ist sie für die Experimente dieses Abschnitts das wichtigste Maß. Allerdings ist Geschwindigkeitsmessung in einer *Multithreading*-Umgebung, in der neben dem eigenen Programm auch noch andere Prozesse und Dienste parallel ausgeführt werden, keine einfache Aufgabe. Zwar könnte man *Jiffies*[5] für die Zeitmessung nutzen, jedoch müsste dafür der Quellcode jedes einzelnen Threads angepasst werden, was zum gegenwärtigen Zeitpunkt zu aufwändig erscheint. Daher begnügt sich diese Arbeit mit der Systemzeit, während das Testsystem selbst mit hoher Priorität läuft. Zusätzlich wird die Anzahl parallel laufender Prozesse deutlich eingeschränkt (Runlevel 2).

Darüber hinaus ist nicht nur die Geschwindigkeit des Prozessors wichtig, sondern auch das Tempo der Festplatte bzw. des unterliegenden Dateisystems. Um die physikalischen Gegebenheiten nicht durch einen Dateisystem-Cache gänzlich zu verdecken, wurde synchrones Lesen und Schreiben auf dem Datenträger angeordnet – Betriebssystem-Cache-Effekte sollten somit weitestgehend ausgeschlossen sein. Allerdings findet sich auf heutigen Festplatten zumeist eigener Hardware-Zwischenspeicher, der im Allgemeinen nicht deaktiviert werden kann. Cache Störeffekte können also nicht gänlich ausgeschlossen werden. Zur Sicherheit beginnt jedes Experiment zunächst damit den gesamten physisch verfügbaren Hauptspeicher anzufordern und mit Nulldaten zu füllen. Gerade für den Lese-Burst-Test 5.3.3 haben sich aus dieser Maßnahme erhebliche Unterschiede ergeben.

Da das Prozess-Netzwerk Betriebssystem-Threads nutzt, führt das Scheduling dieser Threads zu weiteren Ungenauigkeiten, was die exakte Reproduktion der im folgenden gefundenen Ergebnisse praktisch unmöglich macht. Dennoch stellen sie einen relativ guten Anhaltspunkt dar, da sich mehrfach durchgeführte Messungen, wie zu hoffen war, stark ähneln.
Letztendlich könnte an dieser Stelle nur noch ein Simulator noch verlässlichere Ergebnisse herstellen. Um eine eigene Simulationsumgebung für dieses Problem zu entwickeln oder einen bestehenden Simulator dafür anzupassen wäre jedoch noch eine Menge Arbeit zu leisten, welche allerdings den Rahmen dieser Arbeit bei weitem sprengen würde.

[5] Kleinste wahrgenommene Zeiteinheit in UNIX-Systemen, die durch den Timer-Interrupt erzeugt werden.



(a) Geschwindigkeit von 'store'-Aufträgen **ohne** Verdrängung

(b) Geschwindigkeit von 'store'-Aufträgen **mit** Verdrängung

Abbildung 5.1: Einlagerung von Blöcken mit und ohne Verdrängung

5.3.1 Neue Blöcke speichern

In diesem Versuch soll ein halbes Gigabyte in einem einzigen Burst in den Zwischenspeicher geschrieben werden. Dabei wird angenommen, das die Daten in 8000 Blöcke à 64 KB zerschnitten wurden. Die Dauer, bis alle vorbereiteten Befehle in das Prozess-Netzwerk geschrieben wurden, betrug dabei 216 Millisekunden, ist also bei einer Gesamtdauer von 510 Sekunden zu vernachlässigen.

Erwartung

Die Erwartung in diesem Test ist eine konstante Verarbeitungs-Geschwindigkeit. Dies gilt sowohl für die Beantwortung der Anfragen, wie auch für das Tempo, mit der Blöcke auf den unterliegenden Datenträger gesichert werden.

Auswertung

Das Verteilungsbild aus Abbildung 5.1(a) lässt eine Gleichverteilung der Antwortzeiten erahnen. Zwar sind die oberen Spitzen der Säulen etwas wacklig, aber auf einen Berg kommt meist auch ein Tal, so dass sich diese beiden Extreme näherungsweise ausgleichen. Diese Ungleichheiten sind wahrscheinlich auf das Scheduling der Thread zurückzuführen. In jedem Fall kann daraus gefolgert werden, dass 'storeBlock'-Aufträge in einer mehr oder weniger konstanten Zeit verarbeitet werden können.
Weitaus interessanter ist der Verlauf der grünen Kurve. Sie beschreibt die mit ca. 1 MB pro Sekunde stetig anwachsende Datenmenge, die sich aus Sicht der Datenbank aktuell im Cache befindet. Auffällig daran ist, dass sich ihr Verlauf in zwei Bereiche teilt, die beide näherungsweise linear sind bzw. eine konstante Steigung haben. Auch geht die Kurve deutlich über den Zeitpunkt hinaus, an dem der letzte Auftrag bereits bestätigt wurde.
Aus der ersten Beobachtung könnte man schließen, dass die verwendete Festplatte anfangs nicht mit ihrer vollen Geschwindigkeit belastet wird und erst nachdem alle Anfragen quittiert wurden schneller arbeitet. Dieses Szenario ist jedoch äußert unwahrscheinlich! Wie aus dem Automaten der Arbeiter-Threads (siehe Abbildungen 4.2 und 4.3) hervorgeht, wird die Datenbank erst **nach** der tatsächlichen Speicherung eines Blocks informiert. Danach muss der Datenbank-Thread selbst noch eine

Verknüpfung ins Dateisystem schreiben. Dieser Schreibvorgang kann natürlich umso schneller erfolgen, desto weniger die Arbeiter-Threads den Datenträger beschäftigen – diese Last fällt also insbesondere dann weg, wenn alle Blöcke geschrieben wurden. Somit lässt sich der Sprung in der Kurve erklären.
Auch das weitere Wachtum der Kurve, nachdem alle Anfragen quittiert wurden, hat einen einfachen Grund: Die Arbeiter-Threads bestätigen Schreibaufträge unmittelbar nach dem erfolgreichen Schreiben auf die Festplatte. Also noch **vor** dem Eintragen in die Datenbank. Dies ist möglich, da nach dem Speichern eines Blockes nur noch die Konsistenz durch die Datenbank sichergestellt werden muss. Selbst wenn dieser Vorgang fehlschlagen sollte, ist der Block dennoch auf dem persistenten Datenträger gesichert.

Fazit

Zwar haben sich die Erwartungen annähernd erfüllt, sehr beunruhigend ist jedoch, die Dauer, wie lange die grüne Kurve noch weiter ansteigt - ca. 55 Sekunden. Daraus könnte man folgern, dass der einzelne Datenbank-Thread mit seinen eigenen Lese- und Schreibvorgängen tatsächlich einen Flaschenhals im System darstellt.

5.3.2 Neue Blöcke speichern bei gleichzeitiger Verdrängung

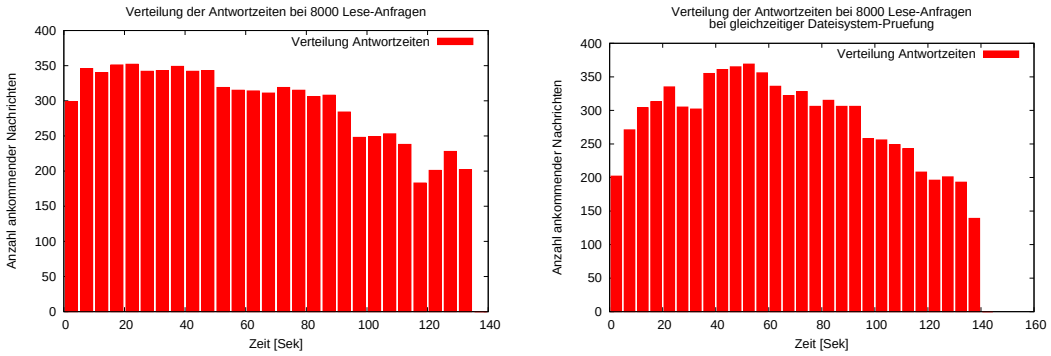
Hierbei handelt es sich um eine Variation des vorherigen Experiments 5.3.1. Es sollen erneut 8000 Blöcke mit einer Gesamtgröße von 500 MB auf die zwischenzeitlich gelöschte Festplatte geschrieben werden. Hinzu kommt, dass die Kapazität des Zwischenspeichers diesmal auf 200MB beschränkt ist. Auch konnten die Aufträge mit 2616 Millisekunden nicht ganz so schnell ins Prozess-Netzwerk geleitet werden wie oben. Die Gesamtlaufzeit betrug 523 Sekunden.

Erwartung

Aufgrund der großen Ähnlichkeit dieses Versuchs mit dem Experiment aus 5.3.1, ist auch mit gleichartigen Ergebnissen zu rechnen. Es wird also eine konstante Verarbeitungs-Geschwindigkeit der einzelnen Aufträge angenommen, die im Verteilungsbild dann als Gleichverteilung zu erkennen sein müsste. Eventuell nimmt das Tempo mit einsetzender Verdrängung leicht ab - in der Verteilung müsste demnach an der entsprechenden Stelle ein Rückgang zu verzeichnen sein.
Die grüne “Datenmenge im Cache”-Kurve sollte wieder konstant ansteigen, bis die 200MB Marke erreicht ist um dann möglichst nahe an dieser Marke zu verweilen.

Auswertung

Die Ergebnisse dieses Versuchs sind in Abbildung 5.1(b) zusammengefasst. Die Verteilung kann grob in vier Abschnitt eingeteilt werden: Anfänglich gibt es einen hohen, ziemlich gleich verteilten Durchsatz. Dieser sinkt kurz darauf auf ein mittleres Niveau. Danach, mit einsetzender Verdrängung, (siehe blaue Markierung), sinkt die Verarbeitungs-Geschwindigkeit wie erwartet kurzzeitig ab, um anschließend bis zum Ende wieder mit dem gleichen Tempo wie zweiten Bereich fortzufahren. Die einigemaßen konstante Schreibgeschwindigkeit in den Abschnitten zwei und vier entspricht vermutlich der Schreibe-Dauerlast der Festplatte. Dagegen scheint es im ersten Bereich noch zu unvermeidbaren Festplatten-Cache Effekten gekommen sein.



(a) Geschwindigkeit von ‘load’-Aufträgen

(b) Geschwindigkeit von ‘load’-Aufträgen bei gleichzeitiger Dateisystem-Prüfung

Abbildung 5.2: Lesen von Blöcken mit und ohne gleichzeitiger Dateisystemprüfung

Leider nimmt die grüne, den Cache-Füllstand beschreibende Kurve, mit einsetzender Verdrängung einen unerwarteten Weg. Anstatt in etwa das 200 MB Niveau zu halten wächst sie, wenn auch langsamer, weiter an. Nach Komplettierung des letzten Schreibaufrags, bei etwa 450 MB, bricht die Kurve abrupt ab und nährt sich zügig der gewünschten 200 MB Marke. Das Ziel, am Ende bei dieser Markierung zu stehen, ist zwar erreicht, der Weg dorthin ist aber nicht der Gewünschte. Die interessante Beobachtung dabei ist, dass die Spitze näherungsweise bei 250 von 300 überlaufenden Megabytes liegt. Demnach ist ca. $\frac{1}{6}$ des Überlaufs noch während der Speicherung abgebaut worden – das könnte auf die fünf Arbeiter-Threads und den sechsten Datenbank-Thread hindeuten.

Fazit

Der Verdrängungsmechanismus funktioniert. Allerdings ist er zu langsam, um bei einer Flut einzulagernder Blöcke noch effektiv die vorgegebene Begrenzung durchsetzen zu können. Mit ca. 100 Sekunden, die der Cache nach Bestätigung des letzten Schreibauftrags, noch mit Verwaltungs- und Verdrängungsarbeit verbringt, verdichten sich die Zeichen, dass der Datenbank-Thread einen Flaschenhals im gesamten Block-Cache-System darstellt. Obwohl sich dieser Wert verglichen mit Experiment 5.3.1 knapp verdoppelt hat, darf hier nicht vergessen werden, dass durch die Verdrängung erheblicher Mehraufwand entsteht - es kann also nicht von einer Verschlimmerung der Problematik gesprochen werden.

5.3.3 Lesen gespeicherter Blöcke

In diesem Experiment sollen Blöcke von der Festplatte gelesen werden. Als Vorbereitung dafür muss entweder Versuch 5.3.1 oder 5.3.2 erfolgreich abgeschlossen worden sein, denn die dort gespeicherten Blöcke werden für das Lesen wiederverwendet. Demnach geht es auch hier wieder um die Weiterverarbeitung eines halben Gigabyte an Daten in Form von 64 KB Stücken. Bei einer insgesamten Laufzeit von 135 Sekunden können die 44 Millisekunden, in denen die Aufträge abgesendet wurden, ignoriert werden.

Erwartung

Da der Dateisystem-Cache des Betriebssystems, durch die initiale Anforderung des gesamten physischen Hauptspeichers, als geleert angenommen werden kann, ist davon auszugehen, dass jeder Auftrag eine konstante Zeit benötigt. Auch ist diese Zeit vor allem durch die Lesezeit der Festplatte definiert, so dass sich durch die Auftragsverwaltung kaum Störeffekte ergeben sollten. Im Burst-Fall handelt es sich also wieder um ein Gleichverteilung der Antwortzeiten.

Auswertung

Aus Abbildung 5.2(a) kann die Verteilung der Antwortzeiten entnommen werden. Wie durch das Scheduling zu erwarten war, sind die Säulen nicht alle exakt von gleicher Höhe, die Schwankungen aber relativ gering. Unerwartet ist die scheinbar konstante Abnahme der Verteilung. Zwar weist das Bild immernoch größte Ähnlichkeit mit dem einer Gleichverteilung auf, die Abnahme scheint aber nicht zufällig entstanden zu sein.
Folgende Regel scheint zu gelten: Je weniger Nachrichten im System sind, desto schneller werden die Verbliebenen verarbeitet. Demnach ließe sich die Zeit der Auftragsverwaltung nicht vernachlässigen!

5.3.4 Lesen gespeicherter Blöcke bei gleichzeitiger Prüfung des Dateisystems

Im Wesentlichen handelt sich bei diesem Experiment um das Gleiche, wie das bereits aus 5.3.3 bekannte. Der einzige Unterschied ist, dass diesmal parallel zum Lesen von Blöcken die Dateisystem-Prüfung gestartet wird. Natürlich müssen auch für diesen Versuch zuvor Blöcke von 5.3.1 und 5.3.2 generiert worden sein. Trotz des Mehraufwandes hat sich die Gesamtlaufzeit nur auf 138 Sekunden erhöht, während alle Aufträge bei 48 Millisekunden nahezu zur gleichen Zeit abgeschickt gewesen sind.

Erwartung

Da lediglich die Dateisystemprüfung hinzukommt, sonst aber alles beim Alten bleibt, stimmen auch die Erwartungen größtenteils überein. Allerdings ist anzunehmen, dass das Verteilungsniveau insgesamt etwas absinkt und dafür breiter wird, um der gleichzeitigen Prüfung Tribut zu zollen. Die Form der Verteilung jedoch bleibt bestehen. Wenn die Nachrichtenmenge, wie im vorherigen Experiment vermutet, tatsächlich Einfluss auf den Verteilungs-Verlauf nimmt, so ist hier zusätzlich mit einer ausgeprägteren Abnahme zu rechnen. Denn diese müsste durch die höhere Anzahl Nachrichten im System beeinflusst werden.

Auswertung

Wie in den bisherigen Versuchen, so informiert auch hier eine Grafik über die Ergebnisse: Abbildung 5.2(b) zeigt, abgesehen vom ersten Drittel, tatsächlich einen ähnlichen und deutlicher ausgeprägten Verlauf der Verteilung an. Diese Erwartung hat sich erfüllt.
Was sich nicht erfüllt hat, ist, dass das Niveau weder gesunken ist, noch dass die

Gesamtzeit signifikant zugenommen hätte. Bei einer Zunahme von gerade mal 3 Sekunden und das, obwohl die doppelte Datenmenge von der Festplatte gelesen werden musste, wird schnell klar, dass ein unerwarteter Effekt eingetreten sein muss. Dabei kann es sich nur um den Lesecache des Betriebssystems handeln. Zwar wird mit Hilfe des `mount`-Befehls die Test-Festplatte zu synchronem[6] Schreiben verpflichtet, der Lesecache lässt sich aber auf diese Art und Weise nicht behindern. Somit werden alle Blöcke nur ein einziges Mal wirklich gelesen, der Betriebssystem-Cache verdeckt den zweiten Zugriff.

Fazit

Betrachtet man die Auswertungen dieses und des letzten Experiments 5.3.3, so kann vermutet werden, dass die gesamte Verarbeitungszeit nicht allein durch die deutlich langsamere Festplatte definiert wird. Auch spielt die Prozessorzeit zwischen den echten Ein- und Ausgabezyklen eine Rolle. Eventuell kann durch eine höhere Parallelverarbeitung bzw. eine Aufstockung der Arbeiter-Threads dieser Effekt gedämpft werden. Insgesamt ist die Geschwindigkeit eines Leseauftrages aber relativ konstant.

[6] Also ohne Betriebssystem-Cache.

6. Zusammenfassung und Ausblick

Zunächst muss festgestellt werden, dass die im Verlauf dieser Arbeit entstandene Implementation einer Zwischenspeicher-Komponente für das geplante *IGOR*-Dateisystem grundsätzlich funktioniert, wie die Funktionstests aus Abschnitt 5.2 nachweisen. Gleichwohl kann die Entwicklung dieser Komponente noch längst nicht als abgeschlossen betrachtet werden. Neben ein paar ungeschickt implementierten Teilproblemen, wie zum Beispiel die Umsetzung der Block-Verdrängung oder das mögliche Verhungern niedrig priorisierter Aufgaben, zeichnet sich, wie aus den *Burst*-Versuchen in 5.3 hervorgeht, der Datenbank-Thread durch eine nur gemächliche Geschwindigkeit aus. Die Problematik darin ist, dass der gesamte Block-Cache nur so schnell sein kann, wie seine langsamste Komponente. Vor allem, wenn eine Vielzahl an Schreibaufträgen zeitgleich eintrifft, stellt dieser Thread einen Flaschenhals im Zwischenspeicher dar. Einerseits kann zur Lösung dieses Problems versucht werden die notwendigen Festplattenzugriffe der Datenbank an die ohnehin schon bereitstehenden Arbeiter-Threads zu delegieren. Alternativ könnte der Datenbank-Thread durch eine andere Implementation, beispielsweise die des vielversprechenden B-Baum-Ansatzes aus 3.2.3, ersetzt werden. Sieht man von dieser noch offenen Schwierigkeit ab, so scheint die im Entwurf konzipierte Gesamtarchitektur zu einem sehr guten Funktionieren des Block-Caches geführt zu haben.

Bevor jedoch eine weitere Entwicklung des Block-Zwischenspeichers vorgenommen wird, sollte zunächst die Anzahl der verfügbaren Tests deutlich erhöht werden! Denn daraus entstehen mehrere Vorteile: Zum Einen kann die einwandfreie Funktion sichergestellt werden und zum Anderen können durch *Burst*-Tests Schwachstellen entdeckt werden. Wie aus Kapitel 5 hervorgeht, wurden dort durch die speziell dafür implementierten Tests, Mehrere entdeckt.
Allerdings ist die Aussagekraft und Qualität der im gleichen Kapitel durchgeführten Tests nicht optimal, da sie noch zu stark vom unterliegenden Betriebssystem, seinem Scheduling und seinen diversen Caches abhängig sind. Ein weiteres wichtiges Werkzeug wäre daher ein Simulator, in dem einzig und allein das *IGOR*-Dateisystem oder eines seiner Module läuft. Somit könnten deutlich präzisiere Messungen vorgenommen werden, die frei von störenden *"Multithreading"*- und Cache-Effekten sind. Der Block-Cache unterstützt diese Idee bereits durch seine I/O-Klasse `cBCDiskWriter`. Diese könnte durch eine entsprechende Simulatorklasse ausgetauscht werden.

Schließlich kann, nachdem alle Komponenten des Dateisystems implementiert sind, ein weiterer interessanter Versuch starten: ein Benchmark von *IGORFS* gegen ein verwandtes Dateisystem. Aufgrund ihrer sehr ähnlichen Zielsetzungen würden sich hierfür die aus 2.3.3 bekannten *DataGrid*-Systeme besonders gut eignen.

Auch entsteht mit der Fertigstellung aller Komponenten ein auf Peer-to-Peer-Technik basierendes, verteiltes Dateisystem der ersten Generation. Von Beginn an wird dieses System an nur einige wenige, noch weit entfernte Grenzen gebunden sein – Benutzer und unterschiedliche Blöcke sind beispielsweise hart auf 2^{160} begrenzt. Realistischerweise würde zwar eine Kollision weit früher stattfinden[1], bei einem solch hohen Bedarf könnte aber jederzeit das kryptische Hash-Verfahren durch ein Anderes, das längere Schlüssel erzeugt, ersetzt werden. Demnach beschränkt hauptsächlich die heute verfügbaren Kapazitäten an persistentem Speicher die maximale Größe des Zwischenspeichers.

Darüber hinaus könnte das entstehende Dateisystem mit einer weiteren, augenblicklich noch nicht vorgesehenen, *"Push"-Funktion* ausgestattet werden. Diese soll dazu dienen, eine in den lokalen Block-Cache eingelagerte Datei bzw. deren Blöcke aktiv an einen anderen Teilnehmer zu senden. Dabei könnte es sich, ähnlich wie im OceanStore-Konzept aus 2.3.2, um einen dedizierten Server oder ein Rechenzentrum handeln, wo die entsprechenden Blöcke persistent abgelegt werden können. Somit wären, auch wenn der publizierende Teilnehmer nicht mehr erreichbar sein sollte, dennoch die entsprechenden Daten abrufbereit und redundant gespeichert.
In jedem Fall darf man gespannt sein, was es für Fortschritte es in den kommenden Jahren auf diesem Gebiet erzielt werden.

[1] Um eine Schlüsselkollision mit 50-prozentiger Wahrscheinlichkeit zu erzeugen sind näherungsweise $\sqrt{2^{160}}$ also 2^{80} Blöcke nötig.

Literaturverzeichnis

[BCEG+00] David Bindel, Yan Chen, Patrick Eaton, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspoon, Westley Weimer, Christopher Wells, Ben Zhao und John Kubiatowicz. OceanStore: An Extremely Wide-Area Storage System. Technischer Bericht UCB/CSD-00-1102, University of California - Computer Science Division (EECS), 2000.

[CaTT94] Rémy Card, Theodore Ts'o und Stephen Tweedie. Design and Implementation of the Second Extended Filesystem. Technischer Bericht, Laboratoire MASI–Institut Blaise Pascal and Massachussets Institute of Technology and University of Edinburgh, 1994.

[CCLW+04] Kenin Coloma, Alok Choudhary, Wei keng Liao, Lee Ward, Eric Roussel und Neil Pundit. Scalable High-level Caching for Parallel I/O. Technischer Bericht, Northwestern University - Center for Parallel and Distributed Computing and Sandia National Laboratories - Scalable Computer Systems Department, 2004.

[ChGM99] Edward Chang und Hector Garcia-Molina. MEDIC: A Memory & Disk Cache for Multimedia Clients. Technischer Bericht, Stanford University - Department of Computer Science, 1999.

[CMHS+02] Ian Clarke, Scott G. Miller, Theodore W. Hing, Oskar Sandberg und Brandon Wiley. Protecting Free Expression Online with Freenet. *IEEE Internet Computing*, 2002.

[CrKF04] Curt Cramer, Kendy Kutzner und Thomas Fuhrmann. Distributed Job Scheduling in a Peer-to-Peer Video Recording System. In *Proceedings of the Workshop on Algorithms and Protocols for Efficient Peer-to-Peer Applications (PEPPA) at Informatik 2004*, Ulm, Germany, 2004. S. 234–238.

[CSWH] Ian Clarke, Oskar Sandberg, Brandon Wiley und Theodore W. Hong. Freenet: A Distributed Anonymous Information Storage and Retrieval System. Technischer Bericht, Uprizer Inc. and Royal Institute of Technology - Department of Numerical Analysis and Computer Science and University of Texas in Austin - College of Communication and Imperial College of Science - Department of Computing, Technology and Medicine.

[dCLS03] Rodrigo S. de Castro, Alair Pereira do Lago und Dilma Da Silva. Adaptive Compressed Caching: Design and Implementation. Technischer Bericht, Universidade de Sao Paulo - Department of Computer Science and IBM T.J. Watson Research Center, 2003.

[GeEi98] Daniela Genius und Jörn Eisenbiegler. Implications of Memory Mappings on Cache Misses. Technischer Bericht, Universität Karlsruhe - Fakultät für Informatik, 1998.

[GrAN92] Knut Stener Grimsrud, James K. Archibald und Brent E. Nelson. Multiple Prefetch Adaptive Disk Caching. Technischer Bericht, Brigham Young University - Department of Electrical and Computer Engineering, 1992.

[Grut99] Matthias Grutzeck. Cache-Optimierungen für geblockte Algorithmen. Diplomarbeit, Universität Karlsruhe (TH), März 1999.

[HaDK99] Ulrich Hahn, Werner Dilling und Dietmar Kaletta. Improved adaptive replacement algorithm for disk caches in HSM systems. Technischer Bericht, Universität Tübingen - Zentrum für Datenverarbeitung, 1999.

[http06] http://www.apgrid.org/. Gfarm file system, 2006.

[JiZh03] Song Jiang und Xiaodong Zhang. Efficient Distributed Disk Caching in Data Grid Management. Technischer Bericht, College of William and Mary - Department of Computer Science, 2003.

[JRAE+] Stephan Jourdan, Lihu Rappoport, Yoav Almog, Mattan Erez, Adi Yoaz und Ronny Ronen. eXtended Block Cache. Technischer Bericht, Intel Corporation.

[JuBB03] Jaeyeon Jung, Arthur W. Berger und Hari Balakrishnan. Modeling TTL-based Internet Caches. Technischer Bericht, MIT Laboratory for Computer Science, 2003.

[Kahn74] Gilles Kahn. The Semantics Of A Simple Language For Parallel Programming. Technischer Bericht, IRIA Laboria - Rocquencourt, France and Commissariat à l'Energie Atomique, France, 1974.

[KaMa77] Gilles Kahn und David B. MacQueen. Coroutines and Networks of Parallel Processes. Technischer Bericht, IRIA Laboria - Rocquencourt, France and University of Edinburgh, Scotland, U.K., 1977.

[KiCh96] Cheolmin Kim und Yookun Cho. An Application-Advised File Caching Scheme. Technischer Bericht, Seoul National University - Department of Computer Engineering, 1996.

[Korn90] Kim Korner. Intelligent Caching for Remote File Service. Technischer Bericht, University of Southern California - Computer Science Department, 1990.

[KuCF05] Kendy Kutzner, Curt Cramer und Thomas Fuhrmann. A Self-Organizing Job Scheduling Algorithm for a Distributed VDR. In *Workshop "Peer-to-Peer-Systeme und -Anwendungen", 14. Fachtagung Kommunikation in Verteilten Systemen (KiVS 2005)*, Kaiserslautern, Germany, 2005.

[Moor65] Gordon E. Moore. Cramming more components onto integrated circuits. Technischer Bericht, Director, Research and Development Laboratories, Fairchild Semiconductor division of Fairchild Camera and Instrument Corp., 1965. ftp://download.intel.com/museum/Moores_Law/Articles-Press_Releases/Gordon_Moore_1965_Article.pdf.

[NgCh01] Wee Teck Ng und Peter M. Chen. The Design and Verification of the Rio File Cache. Technischer Bericht, Bell Laboratories and University of Michigan - Department for Electrical Engineering and Computer Science, 2001.

[Park95] Thomas M. Parks. *Bounded Scheduling Of Process Networks*. Dissertation, University of California at Berkeley, Dezember 1995.

[RaBP00] Ramakrishnan Rajamoni, Ravi Bhagavathula und Ravi Pendse. Timing Analysis of Block Replacement Algorithms on Disk Caches. Technischer Bericht, Wichita State University - Department of Electrical Engineering, 2000.

[REGW+03] Sean Rhea, Patrick Eaton, Dennis Geels, Hakim Weatherspoon, Ben Zhao und John Kubiatowicz. Pond: the OceanStore Prototype. Technischer Bericht, University of California, Berkeley, 2003.

[RoDr01] Antony Rowstron und Peter Druschel. Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems. *Lecture Notes in Computer Science* Band 2218, 2001.

[SMKK+01] Ion Stoica, Robert Morris, David Karger, Frans Kaashoek und Hari Balakrishnan. Chord: A Scalable Peer-To-Peer Lookup Service for Internet Applications. *ACM SIGCOMM Conference*, 2001.

[TMSM+01] Osamu Tatebe, Satoshi Matsuoka, Noriyuki Soda, Youhei Morita, Satoshi Sekiguchi, Hiroyuki Sato, Yoshiyuki Watase, Tomio Kobayashi, Yoshio Tanaka und Masatoshi Imori. Grid Data Farm for Petascale Data Intensive Computing. Technischer Bericht, Electrotechnical Laboratory and High Energy Accelerator Research Organization (KEK) and Tokio Institute of Technology/JST and The University of Tokio and SRA Inc., 2001.

[TMSM+04] Osamu Tatebe, Satoshi Matsuoka, Noriyuki Soda, Youhei Morita und Satoshi Sekiguchi. Grid Datafarm Architecture for Petascale Data Intensive Computing. Technischer Bericht, National Institute of Advanced Industrial Science and Technology (AIST) and Tokio Institute of Technology and Software Research Associates, Inc. and High Energy Accelerator Research Organization (KEK), 2004.

[TMSS+03] Osamu Tateba, Satoshi Matsuoka, Noriyuki Soda, Satoshi Sekiguchi und Youhei Morita. Worldwide Fast File Replication on Grid Datafarm. Technischer Bericht, National Institute of Advanced Industrial Science and Technology(AIST) and High Energy Accelerator Research Organization (KEK) and Tokio Institute of Technology, 2003.

[X9.397] American National Standard X9.30.2-1997. Public Key Cryptography Using Irreversible Algorithms - Part 2: The Secure Hash Algorithm (SHA-1). 1997.

[XuHu03] Zhiyong Xu und Yiming Hu. SBARC: A Supernode Based Peer-to-Peer File Sharing System. Technischer Bericht, University of Cincinnaty - Department of Electrical & Computer Engineering and Computer Science, 2003.