



# SSD-basiertes Caching von Blockgeräten

## Diplomarbeit

vorgelegt von

**Anselm Busse**

Referent: Prof. Dr. Hans-Ulrich Heiß  
Korreferent: Prof. Dr.-Ing. habil. Gero Mühl  
Betreuer: Dr.-Ing. Jan Richling



Die selbständige und eigenhändige Ausfertigung versichert an Eides statt  
Berlin, den 28. Juni 2010

.....  
Unterschrift



# Inhaltsverzeichnis

<b>Inhaltsverzeichnis</b>	<b>v</b>
<b>Abbildungsverzeichnis</b>	<b>vii</b>
<b>Abkürzungsverzeichnis</b>	<b>ix</b>
<b>1 Einführung</b>	<b>1</b>
<b>2 Technisches Umfeld</b>	<b>7</b>
2.1 Festplattenlaufwerk (HDD) . . . . .	7
2.2 Solid State Drive (SSD) . . . . .	10
<b>3 Verwandte Arbeiten</b>	<b>15</b>
3.1 Sun Microsystems' ZFS . . . . .	15
3.2 Microsofts ReadyBoost . . . . .	16
3.3 Intels Turbo Memory . . . . .	17
3.4 Griffin . . . . .	19
3.5 Dynamic Policy Disk Caching for Storage Networking . . . . .	19
3.6 Ersetzungsstrategien für Caches . . . . .	20
<b>4 Problemstellung</b>	<b>23</b>
<b>5 Architektur</b>	<b>27</b>
5.1 High-Level-Design . . . . .	27
5.2 Cache-Algorithmus . . . . .	29
5.2.1 Ersetzungsstrategie . . . . .	30
5.2.2 Schreibstrategie . . . . .	31
<b>6 Implementierung</b>	<b>33</b>
6.1 Grundlage: dm-cache . . . . .	33
6.1.1 Phase 1: Konstruktion . . . . .	33
6.1.2 Phase 2: Lebenszeit . . . . .	34
6.1.3 Phase 3: Destruktion . . . . .	35
6.2 Debuginterface . . . . .	35
6.2.1 message-Funktion . . . . .	36
6.2.2 debugfs-Interface und Visualizer . . . . .	36
6.3 Cache-Algorithmus . . . . .	37
6.4 Optimierung der Cacheblock-Metadaten . . . . .	39
6.5 Anpassung der Metadatensicherung . . . . .	40
6.6 TRIM Unterstützung . . . . .	41
6.7 Bootfähigkeit des Cachemediums . . . . .	41

<b>7</b>	<b>Auswertung</b>	<b>45</b>
7.1	Synthetische Leistungsbewertung . . . . .	45
7.1.1	Testszenario . . . . .	45
7.1.2	Ergebnisse . . . . .	47
7.1.3	Diskussion . . . . .	50
7.2	Reale Leistungsbewertung . . . . .	52
7.2.1	Testszenario . . . . .	52
7.2.2	Ergebnisse . . . . .	53
7.2.3	Diskussion . . . . .	54
7.3	Ressourcenverbrauch . . . . .	55
7.3.1	Ergebnisse . . . . .	55
7.3.2	Diskussion . . . . .	57
<b>8</b>	<b>Zusammenfassung und Ausblick</b>	<b>59</b>
	<b>Anhang</b>	<b>61</b>
	Abbildungen . . . . .	61
	Listings . . . . .	70
	<b>Literaturverzeichnis</b>	<b>73</b>

# Abbildungsverzeichnis

1.1	Entwicklung der Geschwindigkeit von Prozessoren und Speicher . . . . .	2
1.2	Speicherhierarchie . . . . .	2
1.3	Vergleich gebräuchlicher Speichertechnologien . . . . .	3
1.4	Erweiterte Speicherhierarchien . . . . .	4
2.1	Schematische Darstellung einer Festplatte . . . . .	7
2.2	Schematische Darstellung eines Platters . . . . .	8
2.3	Zeitlicher Ablauf des Schreibens auf einen Cacheblock mit ungültigen Seiten	10
2.4	Aufbau eines SSD . . . . .	11
2.5	Schema zur Bereitstellung leerer SSD-Blöcke . . . . .	12
3.1	Zugriffszeit auf den ZFS Pool in Abhängigkeit vom bedienenden Systemteil	16
3.2	Blockdiagramm des Intel Turbo Memory . . . . .	18
3.3	Speicherhierarchie mit Intel Turbo Memory . . . . .	18
5.1	Exemplarische Darstellung der Device-Mapper-Umgebung . . . . .	28
5.2	High-Level-Blockdiagramm des zu implementierenden Caches . . . . .	29
5.3	Verhalten des Caches . . . . .	30
6.1	Verbindungspfad zwischen Cache-Modul und Visualizer . . . . .	37
6.2	Fenster des Visualizers mit aktivem Cache . . . . .	37
6.3	Zustände und Zustandsübergänge der Cacheblöcke . . . . .	38
6.4	Aufbau der Cacheblock-Metadaten für das neue Kernelmodul . . . . .	40
7.1	Benchmarkergebnisse des Caches für $m = 16$ . . . . .	48
7.2	Vergleich zwischen dem implementierten Cache und ZFS . . . . .	49
7.3	Benchmarkergebnisse des Caches für verschiedene Cacheblockgrößen . . . . .	50
7.4	Datendurchsatz verschiedener SSDs . . . . .	51
7.5	Leistungsdaten eines HDD-Bootvorgangs . . . . .	53
7.6	Leistungsdaten eines SSD-Bootvorgangs . . . . .	53
7.7	Leistungsdaten des ersten Cache-Bootvorgangs . . . . .	53
7.8	Leistungsdaten des zweiten Cache-Bootvorgangs . . . . .	53
7.9	Bootzeiten mit HDD, SSD und Cache als Systemdatenträger . . . . .	54
7.10	Prozessorlast in Abhängigkeit von der Assoziativität . . . . .	55
7.11	Speicherverbrauch der Cacheblock-Metadaten . . . . .	57
A.1	Vollständige Zustandsmaschine des Caches . . . . .	61
A.2	Benchmarkergebnisse des Caches für $m = 4$ . . . . .	62
A.3	Benchmarkergebnisse des Caches für $m = 8$ . . . . .	62
A.4	Ausführliches Log eines HDD-Bootvorgangs . . . . .	63
A.5	Ausführliches Log eines SSD-Bootvorgangs mit OCZ SSD . . . . .	64
A.6	Ausführliches Log des ersten Cache-Bootvorgangs mit OCZ SSD . . . . .	65

A.7 Ausführliches Log des zweiten Cache-Bootvorgangs mit OCZ SSD . . . . .	66
A.8 Ausführliches Log eines SSD-Bootvorgangs mit Intel SSD . . . . .	67
A.9 Ausführliches Log des ersten Cache-Bootvorgangs mit Intel SSD . . . . .	68
A.10 Ausführliches Log des zweiten Cache-Bootvorgangs mit Intel SSD . . . . .	69

# Abkürzungsverzeichnis

**BIO** Block-Ein-/Ausgabe

**DM** Device-Mapper

**ECC** Error Correction Code

**ext3** Third Extended Filesystem

**ITM** Intel Turbo Memory

**LFU** Least Frequently Used

**LRU** Least Recently Used

**MLC** Multi-Level-Cell

**RAID** redundante Anordnung kostengünstiger Festplatten

**SAN** Storage Area Network

**SLC** Single-Level-Cell

**SSD** Solid State Drive

**UUID** Universally Unique Identifier



# Kapitel 1

## Einführung

Heutige Rechensysteme haben einen Stand erreicht, in dem pro Sekunde mehrere Gigabyte Daten verarbeitet werden können. Diese Entwicklung bringt jedoch das Problem mit sich, dass die Daten dem Prozessor schnell genug zur Verfügung gestellt werden müssen. Zu Beginn der elektronischen Datenverarbeitung hatten Prozessoren eine Verarbeitungsgeschwindigkeit, die mit den Leseraten der damaligen Speichermedien wie z.B. Lochkarten gleichzusetzen waren. Auf Grund zunehmender Miniaturisierung entwickelten sich jedoch die Prozessoren rasant weiter. Sie folgten hierbei dem MOORESchen Gesetz, welches besagt, dass sich die Komplexität integrierter Schaltkreise alle 18 Monate verdoppelt und dadurch auch die Rechenleistung pro Chipfläche exponentiell zunimmt.

Im Gegensatz zur schnellen Entwicklung auf dem Prozessormarkt stand die Entwicklung im Bereich der Speichertechnik. Die Lochkarten wurden durch schneller lesbare Magnetbänder verdrängt und diese wiederum später durch Magnetplatten bzw. Festplatten ersetzt. Dies führte zum einen zu einer enormen Platzersparnis, zum anderen zu einem Geschwindigkeitsgewinn beim Auslesen der Daten. Zwar konnten die Prozessoren mit ausreichend Daten im Sinne der Menge versorgt werden, aber die Entwicklung im Bereich der Speichertechnik konnte nicht mit dem exponentiellen Wachstum der Prozessorgeschwindigkeit mithalten. Dies führte zu großen Problemen, weil die zu verarbeitenden Daten nicht schnell genug für die Verarbeitung durch den Prozessor zur Verfügung gestellt werden konnten.

Neben der Speichermöglichkeit auf Lochkarten und magnetischen Medien existierte weiterhin die Möglichkeit, Speicher in Form von Halbleitern zu realisieren. In der Praxis hatte dies jedoch die Nachteile, dass selbst kleine Speichermengen extrem teuer waren und Daten sich nur flüchtig speichern ließen. Deshalb kamen Halbleiterspeicher als Massenspeicher nicht in Frage. Allerdings lässt sich bereits mit einer kleinen Menge dieser Art von Speichern die Eigenschaft der Lokalität der meisten Programme ausnutzen. Lokalität bedeutet hierbei, dass Programmteile häufig wieder verwendet werden und sich ggf. nur die zu bearbeitenden Daten ändern. Genau diese Daten lassen sich gut im flüchtigen Halbleiterspeicher halten, der heute allgemein als Haupt- oder Arbeitsspeicher bezeichnet wird. Dies funktionierte zunächst sehr gut, da der Speicher schneller war als die verfügbaren Prozessoren, jedoch entwickelten sich nachfolgend auch hier die Prozessoren schneller weiter als der Speicher, wie Abbildung 1.1 zeigt. Im Diagramm wird die Geschwindigkeit von Prozessoren mit dem hauptsächlich für Arbeitsspeicher verwendeten Halbleiterspeicher DRAM verglichen.

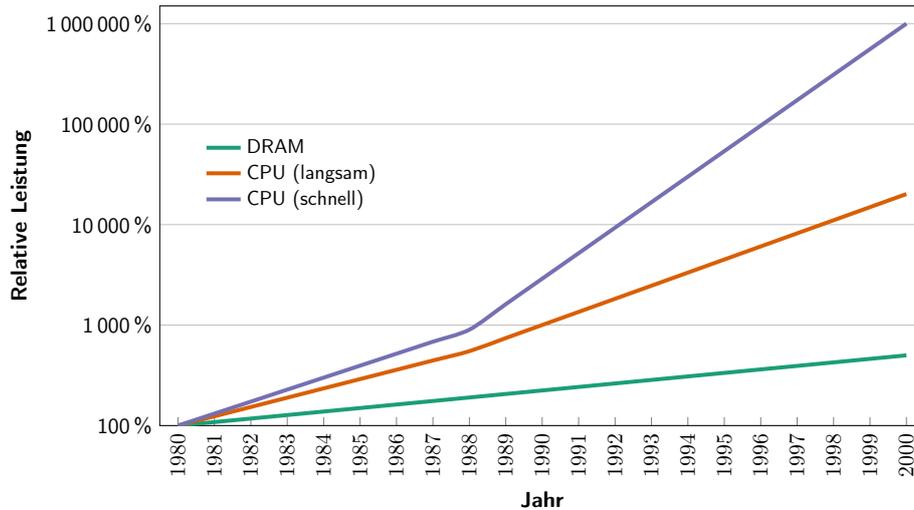


Abbildung 1.1: Entwicklung der Geschwindigkeit von Prozessoren und Speicher (nach Herrmann [15, S. 180])

Das Problem der auseinanderdriftenden Geschwindigkeiten, das im Diagramm deutlich sichtbar wird, ist jedoch nicht primär technologischer sondern ökonomischer Natur. Halbleiterspeicher profitieren ebenso wie Prozessoren von der steigenden Integrationsdichte. Dabei muss beachtet werden, dass die Kosten für die Produktion proportional mit der Geschwindigkeit des Speichers steigen. Es ist zwar denkbar, Arbeitsspeicher mit ausreichender Geschwindigkeit zu produzieren, dies war und ist ökonomisch jedoch nicht sinnvoll. Vielmehr führte es dazu, dass zusätzlich zum Arbeitsspeicher ein weiterer Halbleiterspeicher mit einer geringeren Kapazität aber besserer Leistung und anderer Funktionalität in Rechnersysteme integriert wurde. Dieser als Cache bezeichnete Speicher wird, anders als der Arbeitsspeicher, nicht vom Benutzer verwaltet sondern von der Hardware selbst. Hierbei sollte angemerkt werden, dass es sich meistens bei dem Benutzer nicht direkt um den Endnutzer sondern meist um das Betriebssystem handelt. Folglich ist der Cache dem Benutzer gegenüber vollkommen transparent. Da das Konzept eines einfachen Caches zu einem bestimmten Zeitpunkt auch nicht mehr ausreichte, wurden mehrere Cachestufen mit unterschiedlichen Geschwindigkeiten integriert, so dass es zu einer Speicherhierarchie kam, wie sie in Abbildung 1.2 zu sehen ist. Dabei kann festgestellt werden, dass der Cache

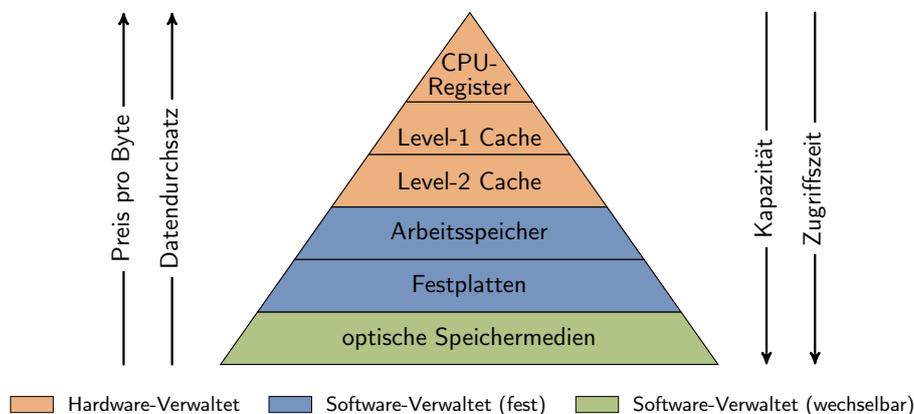


Abbildung 1.2: Speicherhierarchie (nach Stallings [31, S. 45])

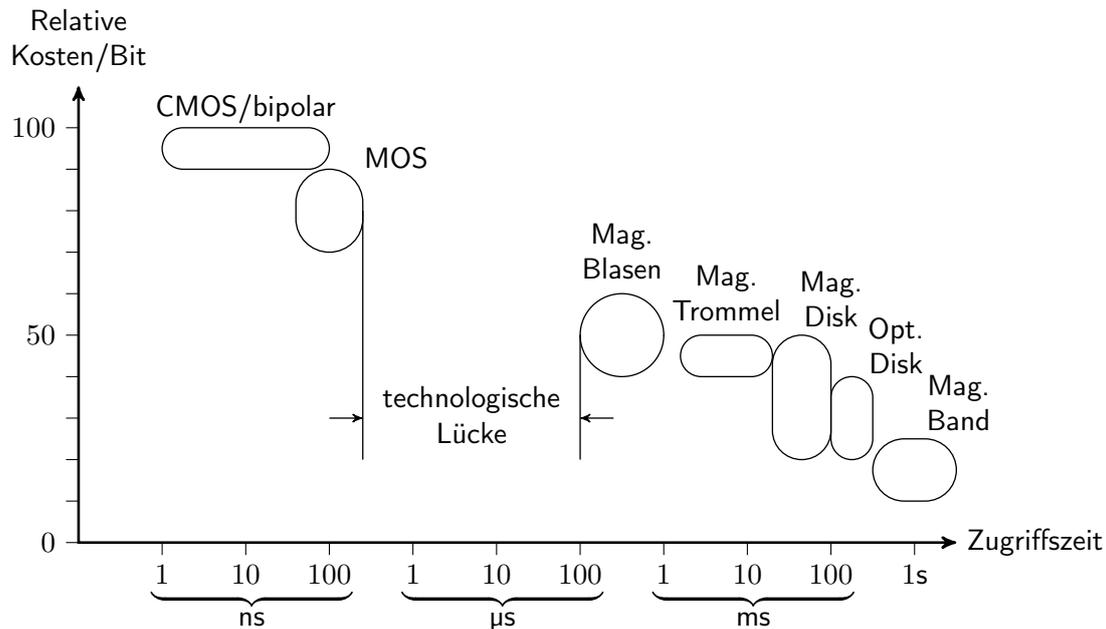


Abbildung 1.3: Vergleich gebräuchlicher Speichertechnologien (nach Schiffmann [26, S. 245])

immer wesentlich kleiner ist als der gecachte Speicher, dafür aber für gewöhnlich um Größenordnungen schneller.

In Abbildung 1.2 werden neben den bereits aufgeführten Speicherarten optische Speichermedien genannt. Diese setzten sich Anfang der 1990er Jahren zum Transport von Daten zwischen Rechensystemen und zum Anfertigen von Sicherheitskopien neben Bandlaufwerken durch. Des Weiteren gilt es zu beachten, dass in der Abbildung ein Trade-off zwischen Speicherplatz und Kosten auf der einen Seite und Datendurchsatz und Zugriffszeit auf der anderen Seite zu erkennen ist, welcher vertikal in der Hierarchie verläuft. Das Prinzip dieser Speicherhierarchie hat sich seit längerem durchgesetzt und stellt die momentan übliche Herangehensweise bei der Architektur eines Rechners dar. Die Hierarchie weist jedoch ein Defizit auf, welches bis in die heutige Zeit zunehmend markanter wurde. Es besteht eine Lücke bei der Zugriffszeit zwischen Halbleiterspeicher und Magnetspeicher. Diese ist mehrere Dekaden groß, wie in Abbildung 1.3 zu erkennen ist.

Neben dem bereits erwähnten flüchtigen Halbleiterspeicher wurden in den 1980er Jahren nichtflüchtige Halbleiterspeicher entwickelt. Da diese jedoch wesentlich langsamer als flüchtige Halbleiterspeicher sind, können sie nicht als Ersatz für die flüchtigen Hauptspeicher dienen. Des Weiteren hatte er ökonomisch ähnliche Nachteile wie der flüchtige Speicher, so dass er ebenfalls nicht als Ersatz für Magnetscheiben basierte Festplatten dienen konnte. Er fand zunächst seine Anwendung in eingebetteten Geräten, setzte sich jedoch Mitte der 1990er Jahre durch die aufkommende Digitalfotografie auch als Wechselspeichermedium durch, da hierbei auf Grund der Größe und physikalischen Belastungen weder optische noch magnetische Speichermedien geeignet waren. Der hierfür verwendete Halbleiterspeicher wird als Flashspeicher bezeichnet. Neben den hauptsächlich für die Digitalfotografie entwickelten Speicherkarten kamen Ende der 1990er Jahre USB-basierte Flashspeichermedien auf den Markt, die bis heute im Begriff sind, optische Speichermedien für den Datentransport zu ersetzen.

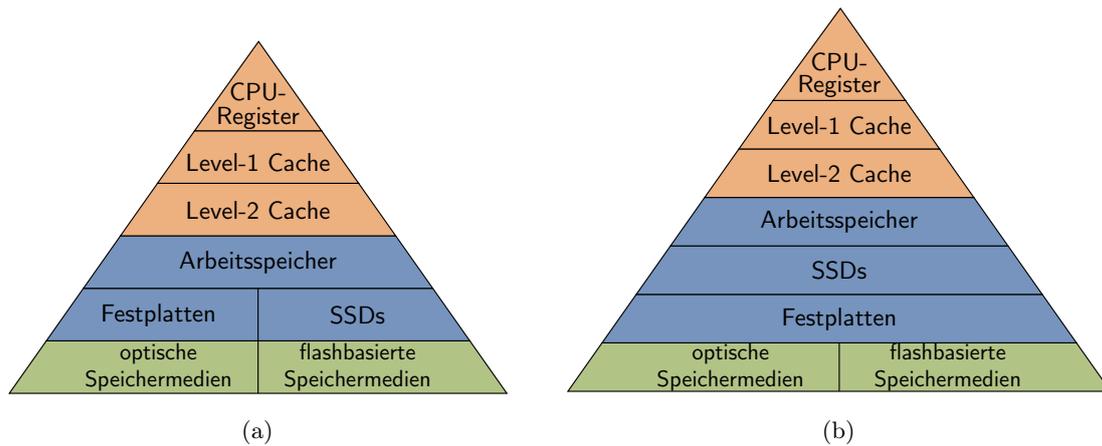


Abbildung 1.4: Erweiterte Speicherhierarchien

In den letzten Jahren gab es allerdings eine neue Entwicklung. Da Flashspeicher auch dem MOORESchen Gesetz unterliegen, sind die Preise pro Byte schnell gesunken und die Leistung gleichzeitig gestiegen. Aus diesem Grund sind seit ein paar Jahren Flashspeicher basierte Festplatten - so genannte Solid State Drives (SSDs) - auf dem Markt. Also lässt sich die Speicherhierarchie zunächst wie in Abbildung 1.4(a) gezeigt erweitern.

SSDs haben jedoch andere Eigenschaften als klassische Festplatten. Sie sind zwar nur moderat schneller in Bezug auf den Datendurchsatz jedoch um Größenordnungen schneller in Bezug auf die Zugriffszeit. Auf Grund dieser Eigenschaft können sie die technologische Lücke, wie sie in Abbildung 1.3 dargestellt ist, schließen. Wenn nun noch vor Augen geführt wird, dass auf bestimmte Arten von Daten wie z.B. Programmdateien hauptsächlich ungeordnet zugegriffen wird, kann leicht erkannt werden, dass SSDs Festplatten in diesem Bereich signifikant überlegen sind. Unglücklicherweise übersteigen die Kosten pro Byte SSD-Speicher die Kosten pro Byte Festplattenspeicher noch immer signifikant. Dies ist deutlich ablesbar in Tabelle 1.1, in der die Preise gängiger Low- bis High-End-Festplatten mit Preisen von Low- bis High-End-SSDs verglichen werden. Aus diesem Grund kann auf Festplatten noch nicht komplett verzichtet werden.

Tabelle 1.1: Preise ausgewählter Festplatten und SSDs (Quelle: Geizhals Preisvergleich, Stand: 13.05.2010)

Produkt	Preis pro TB in Euro
<b>Festplatten</b>	
Samsung EcoGreen F2 1500GB (5400rpm)	48,30
Seagate Barracuda 7200.11 1500GB (7200rpm)	54,32
Western Digital VelociRaptor 600GB (10000rpm)	431,65
Hitachi Ultrastar 15K600 600GB (15000rpm)	665,00
<b>SSDs</b>	
BestMedia Platinum MyDrive 128GB	1373,18
Kingston SSDNow V-Series G2 128GB	1796,10
Intel X25-M G2 Postville 160GB	2295,80
Intel X25-E 64GB	9967,62

Um dennoch die Möglichkeiten und Vorteile der neuen Technologie nutzen zu können, muss der Nutzer die Daten, auf die hauptsächlich sequenziell zugegriffen wird, von den Daten manuell trennen, auf die hauptsächlich ungeordnet zugegriffen wird, und auf die entsprechend passenden Speichermedien verteilen. Da dies sehr umständlich und für unerfahrene Nutzer kaum machbar ist, wäre es sinnvoller SSDs nicht als kompletten Ersatz für herkömmliche Festplatten zu nutzen, sondern als Cache, genau wie es bei Hauptspeicher und (Prozessor)-Cache der Fall ist. Es würde sich dadurch eine veränderte Speicherhierarchie ergeben, wie sie in Abbildung 1.4(b) zu sehen ist.

Diese Arbeit soll sich mit der Möglichkeit der Realisierung dieses Caches auseinandersetzen. Hierfür wird zunächst in Kapitel 2 das technische Umfeld näher untersucht und dabei die Funktionsweise von Festplatten und SSDs erläutert. Das darauf folgende Kapitel 3 setzt sich mit Arbeiten auseinander, die für diese Arbeit relevant sind. Anschließend wird in Kapitel 4 eine Problemstellung mit konkreten Anforderungen formuliert. Sodann wird in Kapitel 5 die Architektur des implementierten Caches vorgestellt und die Implementierung in Kapitel 6 näher erläutert. Anschließend wird in Kapitel 7 die Leistungsfähigkeit des Caches untersucht und ein Ausblick auf zukünftige Arbeitsfelder in Kapitel 8 gegeben.



# Kapitel 2

## Technisches Umfeld

Um einen in der obigen Art beschriebenen Cache realisieren zu können, muss sich zunächst mit der genauen Funktionsweise der beteiligten Geräte auseinandergesetzt werden. Dies sind im konkreten Fall herkömmliche magnetscheibenbasierte Festplatten und Flashspeicher basierte SSDs. Diese haben jeweils spezifische Vor- aber auch Nachteile. Aus diesem Grund wird in diesem Kapitel detailliert auf den Aufbau und die Funktionsweise beider Gerätetypen eingegangen. Festplatten werden hierbei in Abschnitt 2.1 und SSDs in Abschnitt 2.2 betrachtet.

### 2.1 Festplattenlaufwerk (HDD)

Der Aufbau und die Funktionsweise herkömmlicher Festplatten wird unter anderem von Schiffmann [26] und Tanenbaum [33] genauer beschrieben. Eine Festplatte arbeitet nach dem Prinzip, Information in Form der Ausrichtung von magnetischen Teilchen zu speichern.

Sie bestehen physisch aus einer oder mehreren ferromagnetisch beschichteten Scheiben, auch Platter genannt. Diese können sowohl ein- als auch beidseitig beschichtet sein. Sie

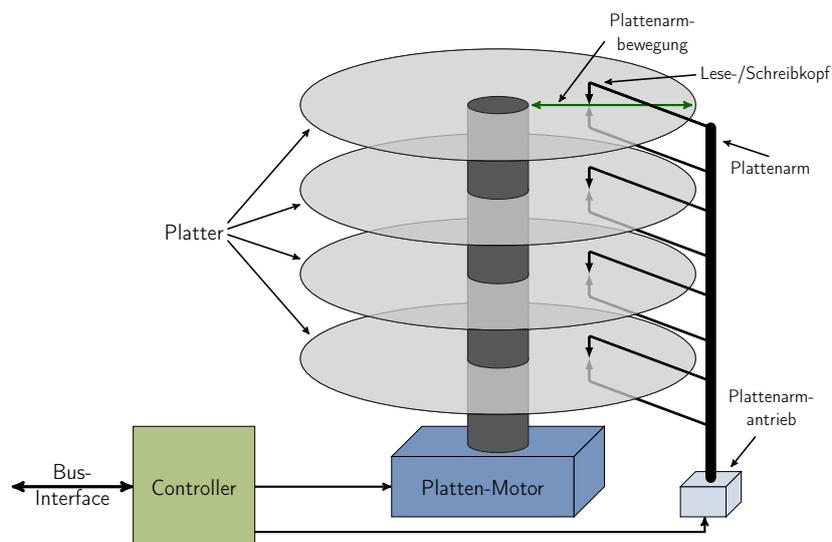


Abbildung 2.1: Schematische Darstellung einer Festplatte (nach Tanenbaum [33, S. 101])

sind an einer Achse angebracht, welche mit Hilfe eines Motors angetrieben wird. Weiterhin gibt es einen Lesearm, der in eine beliebige radiale Position über den Plattern gebracht werden kann. An der Spitze des Armes befindet sich der Lese-/Schreibkopf. Außerdem besitzt eine Festplatte eine Steuerelektronik, mit der sie gesteuert wird. Im Detail übernimmt sie die Steuerung des Platter-Motors sowie die Steuerung und Positionierung des Schreib-/Lesearms, die Verarbeitung der gelesenen und zu schreibenden Daten und die Protokollumsetzung zum Anschluss an ein Rechnerbussystem. Dieser Sachverhalt ist in Abbildung 2.1 skizzenhaft dargestellt.

Die Platter sind in mehrere konzentrische Spuren unterteilt, die als Zylinder bezeichnet werden. Bei den Zylindern handelt es sich nicht, wie möglicherweise vermutet werden kann, um Vertiefungen im Platter sondern der Lese-/Schreibkopf schwebt im Abstand von wenigen Nanometern über ihnen. Das Schweben wird hierbei durch den BERNOULLI-Effekt erreicht. Die Zylinder sind ihrerseits weiter in Blöcke unterteilt, wobei diese nicht nur Nutzdaten enthalten. Vor den Nutzdaten – meist 512kB – befindet sich eine Präambel, welche zur Kalibrierung des Lese-/Schreibkopfes dient. Dahinter befindet sich ein Error Correction Code (ECC), welcher zur Fehlererkennung und Korrektur beim Lesen der Daten dient (Abbildung 2.2). Die Blöcke, die unter dem gleichen radialen Winkel vom Schreib-/Lesekopf erreicht werden können, werden auch Sektor genannt. Dadurch lässt sich ein beliebiger Block durch Angabe des Sektors, Zylinders und Kopfs adressieren. Die Anzahl von Sektoren, Zylindern, Köpfen und Größe der Blöcke wird hierbei auch als Festplattengeometrie bezeichnet, mit deren Hilfe sich die Kapazität der Festplatte wie folgt berechnen lässt:

$$\text{Kapazität} = \text{Köpfe} \cdot \text{Sektoren} \cdot \text{Zylinder} \cdot \text{Blockgröße}$$

Bei modernen Festplatten wird die Geometrie der Festplatte durch den Festplattencontroller jedoch vor dem übrigen System verborgen. Ursprünglich hatte dies als Ursache, dass die Adressierung bestimmten Limitierungen bezüglich der maximalen Anzahl der Köpfe,

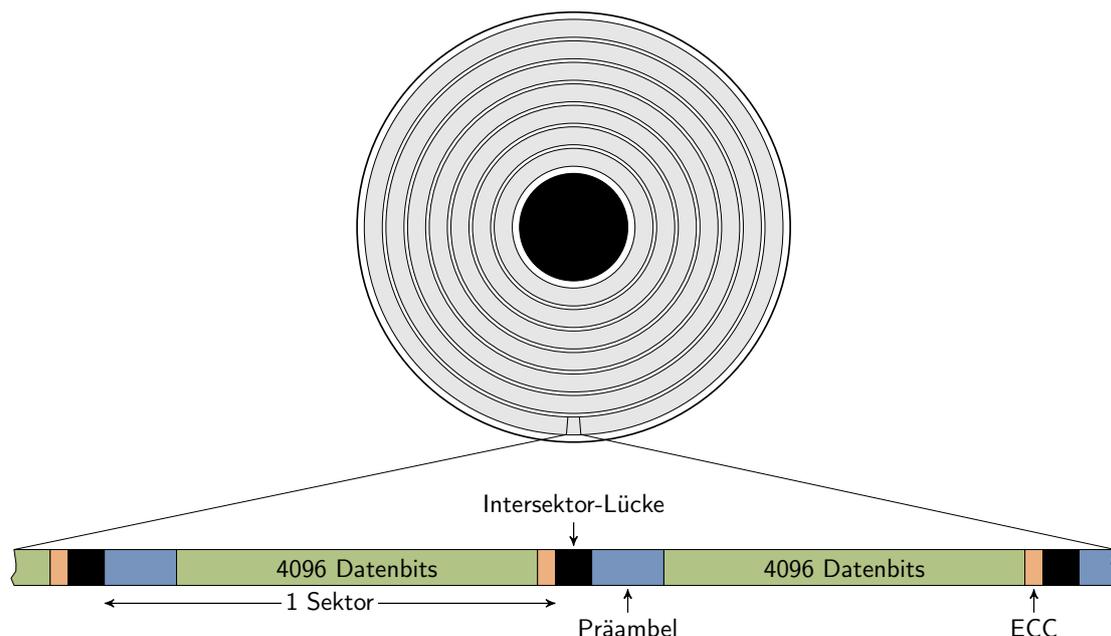


Abbildung 2.2: Schematische Darstellung eines Platters (nach Tanenbaum [33, S. 100])

Sektoren und Zylinder unterworfen war. Die Kapazität der Festplatten wurde dadurch erhöht, dass pro Platter mehr Spuren und mehr Sektoren pro Spur untergebracht wurden. Die Anzahl der Platter veränderte sich jedoch nicht signifikant. Dies führte dazu, dass bald die Anzahl der Sektoren und Zylinder die der maximal adressierbaren Anzahl von Sektoren und Zylinder überstieg. Um dieses Problem zu umgehen, wurde dem restlichen System durch den Festplattencontroller eine größere Anzahl von Köpfen gemeldet, als real vorhanden waren. Dies erweiterte die Anzahl der adressierbaren Sektoren deutlich.

Im weiteren Verlauf wurde die Aufteilung des Platters weiter verändert, so dass bei modernen Festplatten im äußeren Bereich eines Platters mehr Sektoren untergebracht sind als im inneren Bereich. Weiter geht die Tendenz aktuell dahin, dass die Nettoblockgröße von 512 Byte auf vier Kilobyte erhöht wird [vgl. 34], um zusätzlichen Platz einzusparen, der durch Präambel, ECC und Lücke zwischen den Blöcken belegt wird. Aus diesen Gründen wurde die Adressierung durch Kopf, Sektor und Zylinder durch eine fortlaufende Nummer ersetzt. Diese wird vom Festplattencontroller in die festplattenspezifischen Geometrieparameter übersetzt.

Aus den genannten physikalischen Eigenschaften ergeben sich Vorteile aber auch Nachteile. Der größte Vorteil einer Festplatte gegenüber fast allen anderen Speichermedien ist die Möglichkeit, große Datenmengen mit einem wahlfreien Zugriff zu einem sehr niedrigen Preis zu speichern. Weiterhin können die Daten relativ schnell sequenziell gelesen und geschrieben werden, so dass Festplatten ideal sind für eine mittelfristige Datenarchivierung. Festplatten werden jedoch nicht ausschließlich hierfür eingesetzt sondern auch als Systemspeicher, in dem Betriebssystem, Programme und sonstige Nutzerdaten gespeichert werden. Genau hier werden die Nachteile von Festplatten sichtbar. Diese Daten werden nicht sequenziell sondern hauptsächlich ungeordnet gelesen. Dies ist problematisch, da die Leserate von Festplatten beim ungeordneten Lesen dramatisch einbricht. Der Grund hierfür ist im Aufbau zu finden. Um zu einem bestimmten Block zu springen, muss zuerst der Lese-/Schreibarm in die richtige Position gebracht werden. Diese Verzögerung wird *Seek Time* genannt und liegt bei aktuellen Festplatten zwischen ca. drei und zwölf Millisekunden [25, 27]. Außerdem braucht der Platter im Durchschnitt eine halbe Umdrehung, bis der angeforderte Block unter dem Lese-/Schreibkopf liegt. Diese Verzögerung wird *Latency* genannt. Hinzu kommt noch die Verzögerung des Festplattencontrollers (*Controller-Overhead*), die aber im Vergleich zu den anderen beiden Größen recht gering ist. Daher lässt sich die mittlere Zugriffszeit für einen beliebigen Block wie folgt berechnen:

$$\text{Zugriffszeit[ms]} = \text{Spurwechselzeit[ms]} + \underbrace{\frac{60000}{\text{Drehzahl}[\text{min}^{-1}]} \cdot 0,5}_{\text{Latency}} + \text{Controller-Overhead[ms]}$$

Um das eben beschriebene Problem beim unsortierten Lesen zu mindern, besitzen moderne Festplatten einen mehrere Megabyte großen Halbleiterspeicher-basierten Lese-/Schreibcache. Dieser ermöglicht es den Festplatten, häufig angefragte Blöcke vorzuhalten und Lese-/Schreiboperationen ggf. umzuordnen. Das Umordnen kann z.B. sinnvoll sein, wenn ein Block im Anfragepuffer liegt, der sich gerade unter dem Lese-/Schreibkopf befindet. Es ist sinnvoller, zunächst diese Anfrage zu bearbeiten als später wieder den Lese-/Schreibkopf zu dieser Position zu bewegen.

## 2.2 Solid State Drive (SSD)

Der Aufbau und die Funktionsweise von SSDs wird von Shimpi [28, 29] diskutiert. Ein SSD dient zwar genau wie eine herkömmliche magnetscheibenbasierte Festplatte als Massenspeicher, ist jedoch fundamental anders aufgebaut. Bei einem SSD werden die Daten nicht in Form einer magnetischen Ausrichtung gespeichert sondern als elektrische Ladung. Dies wird durch eine Implementierung in Halbleitertechnik ermöglicht.

Die Grundbausteine eines SSD sind Flash-Speicherbausteine. Der Flashspeicher ist ein persistenter Halbleiterspeicher. Es wird eine im Vergleich zum restlichen System relativ hohe Spannung zum Programmieren bzw. Löschen benutzt. Diese Spannung beim Programmieren des Flashspeicher-Transistors macht ihn dauerhaft nicht-leitend. So ist es möglich, beim normalen Schalten des Transistors zwischen programmiert und nicht programmiert bzw. gelöscht zu unterscheiden. Außerdem wird durch diese Technik ein schnelles Auslesen ermöglicht. Das Programmieren mit hoher Spannung hat jedoch den Nachteil, dass es den Transistor physisch schädigt und somit nur eine begrenzte Menge von Programmiervorgängen möglich sind. Weiterhin werden Flashspeicherzellen in Single-Level-Cell (SLC)- und Multi-Level-Cell (MLC)-Speicherzellen unterschieden. Bei SLC-Speicherzellen wird nur ein Bit pro Flashzelle gespeichert, bei MLC-Speicherzellen zwei oder mehr. MLC-Speicherzellen haben einen Kostenvorteil, da sie kleiner sind und somit günstiger bei der Herstellung. Sie haben jedoch den Nachteil, langsamer beim Lesen und Programmieren und Fehleranfälliger als SLC-Speicherzellen zu sein.

Auf Grund des zu hohen Aufwandes bei der Adressierung einzelner Flash-Speicherzellen werden diese heute für gewöhnlich zu Seiten à vier Kilobyte zusammengefasst. Somit ist die kleinstmögliche schreibbare Datenmenge bei einem Flashbaustein vier Kilobyte groß.

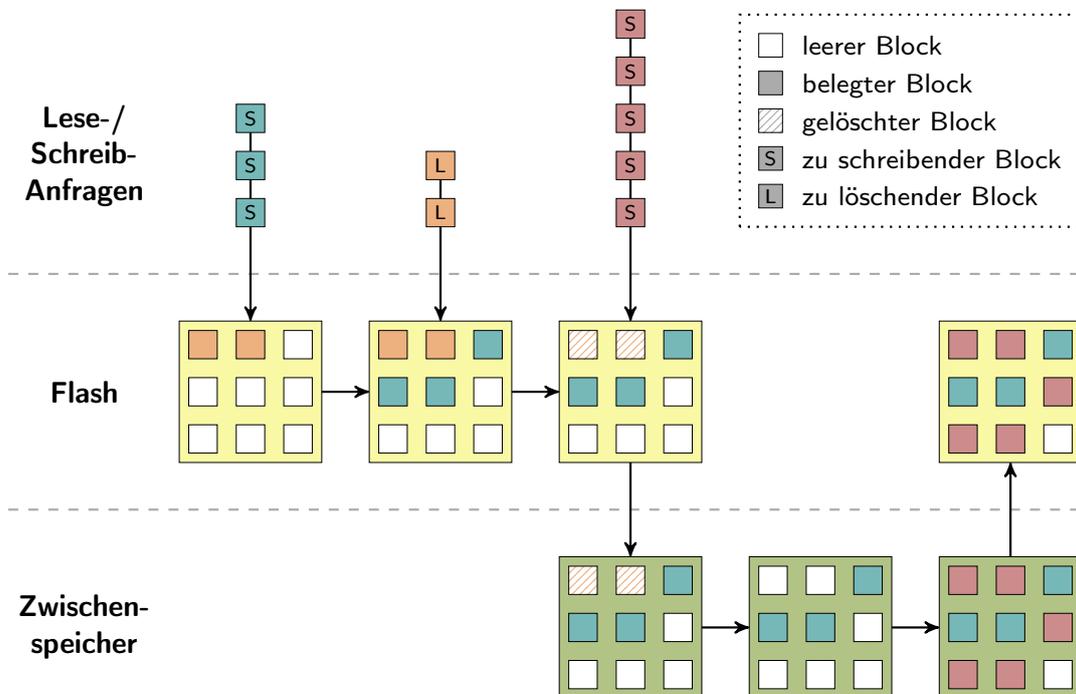


Abbildung 2.3: Zeitlicher Ablauf des Schreibens auf einen Cacheblock mit ungültigen Seiten (nach Shimpi [28])

Da der Aufwand beim Aufbau zum Löschen der Speicherzellen auf Grund der benötigten hohen Spannung noch größer ist, werden für gewöhnlich wiederum 128 der vier Kilobyte Seiten zu einem Löschblock zusammengefasst. Somit ist die kleinste löschbare Einheit 512kB groß. Das bedeutet, dass zum Ändern des Inhalts einer Seite zunächst die anderen 127 Seiten ausgelesen und anschließend zusammen mit der geänderten Seite wieder zurückgeschrieben werden müssen. Das Vorgehen wird im zeitlichen Ablauf in Abbildung 2.3 dargestellt, wobei ein Löschblock der Übersichtlichkeit halber nur aus neun Seiten besteht.

Um die Kompatibilität zu den bisherigen Festplatten zu wahren und den Flash-Speicher an einen S-ATA-Controller, SAS-Controller oder den PCI-Express-Bus anzuschließen, wird ein weiterer Controllerchip benötigt. Neben der Aufgabe der Protokollwandlung übernimmt dieser das so genannte *Wear-Levelling*. *Wear-Levelling* soll die Lebenszeit von Flashspeichermedien verlängern. Dies wird dadurch erreicht, dass Schreibzugriffe möglichst gleichmäßig auf alle Seiten verteilt werden. Ein typisches Szenario hierfür sind die Verwaltungsinformationen des Dateisystems. Sie liegen immer an derselben logischen Speicheradresse und werden regelmäßig verändert. Ohne *Wear-Levelling* würde immer wieder dieselbe Seite beschrieben werden und relativ schnell ausfallen, während die meisten anderen noch ihre nahezu vollständige Lebenszeit hätten. *Wear-Levelling* sorgt nun dafür, dass das Schreiben nicht immer wieder auf derselben Seite stattfindet sondern auf einer,

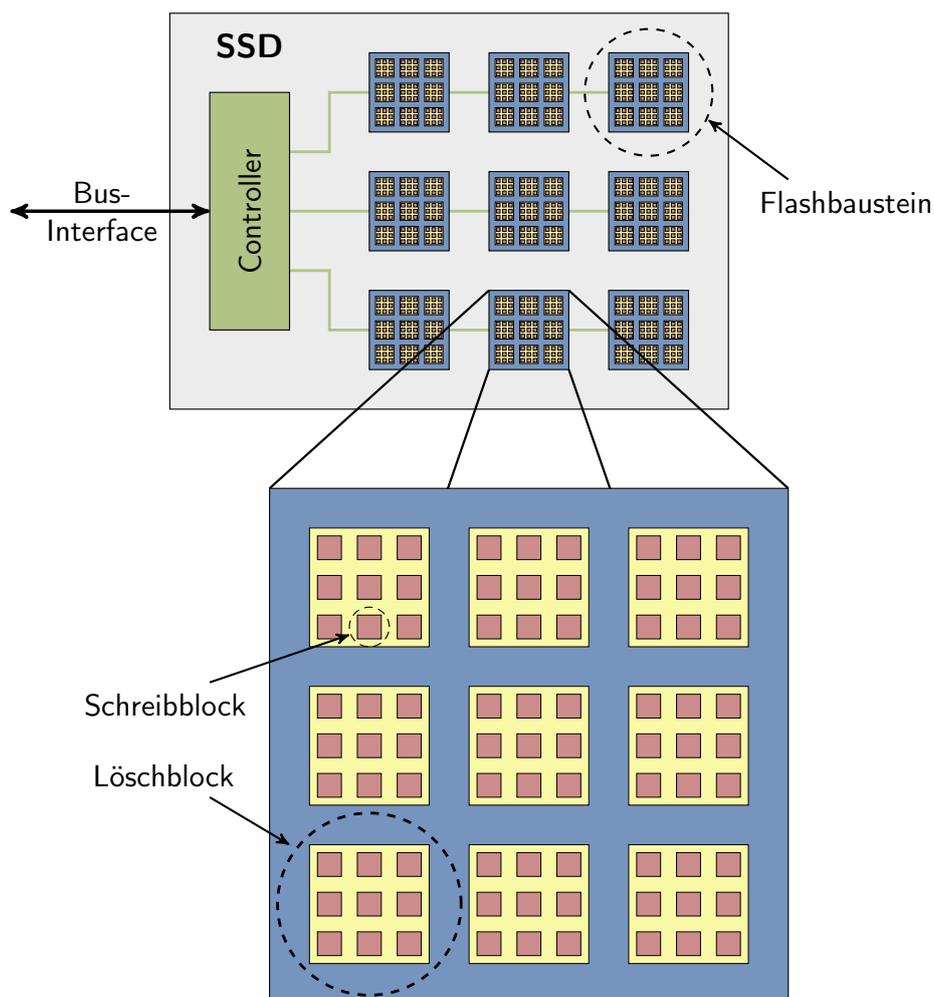


Abbildung 2.4: Aufbau eines SSD (nach Shimpi [28])

auf der in der Vergangenheit weniger Schreibvorgänge stattgefunden haben. So werden sämtliche Schreibvorgänge gleichmäßig auf alle Seiten des SSDs verteilt. Dies bedeutet allerdings auch einen nicht zu unterschätzenden Verwaltungsaufwand. Zum einen muss die Zuordnung der logischen Plattenadresse auf eine konkrete Flashseite vorgenommen und gespeichert werden und zum anderen die Anzahl der Schreibzyklen pro Seite gespeichert werden. Der grundsätzliche Aufbau eines SSDs wird zum besseren Verständnis in Abbildung 2.3 schematisch dargestellt, wobei die Darstellung nur auf Baugruppen eingeht, die für die Programmierung von Relevanz sind.

Ein Problem, das bei der Substitution von Festplatten durch SSDs entsteht, liegt in der unterschiedlichen Funktionsweise beim Löschen von Daten. Daten auf Festplatten können ohne vorheriges Löschen blockweise überschrieben werden. Bei SSDs ist dies nicht der Fall. Hier müssen Blöcke vor dem Neubeschreiben zuerst gelöscht werden. Dies wirft zunächst das bereits oben beschriebene Problem auf, dass sich sowohl gültige als auch ungültige Daten in einem Block befinden können. Wenn die ungültigen Blöcke überschrieben werden sollen, muss erst, wie oben gezeigt, der komplette Block ausgelesen, verändert und dann zurückgeschrieben werden. Da das Betriebssystem kein Wissen über die Löschblockgröße eines SSD hat, ist es der Regelfall, dass Daten mehrerer Dateien in einem Block untergebracht werden, wodurch das genannte Problem sehr häufig auftritt.

Dies ist solange relativ unkritisch, solange leere Blöcke zum Bedienen von Schreibfragen zur Verfügung stehen. Sobald jedoch zwangsweise Daten in die ungültigen Bereiche von teilweise gelöschten Blöcken geschrieben werden müssen, kann ein Leistungseinbruch beobachtet werden. Das Vorgehen eines SSD beim Schreiben ist in Abbildung 2.5 schematisch dargestellt. Solange sich genügend Blöcke in der Liste der leeren Blöcke befinden, kann das SSD mit voller Geschwindigkeit arbeiten. Sobald der Vorrat jedoch erschöpft ist, müssen aus dem oben genannten Grund teilweise gefüllte Blöcke geräumt werden, der Block gelöscht und die geräumten Daten erneut in einen leeren Block geschrieben werden. Dieses Vorgehen ist zeitaufwendig und führt zu dem beobachteten Phänomen des Leistungsverlustes bei der längeren Nutzung eines SSD.

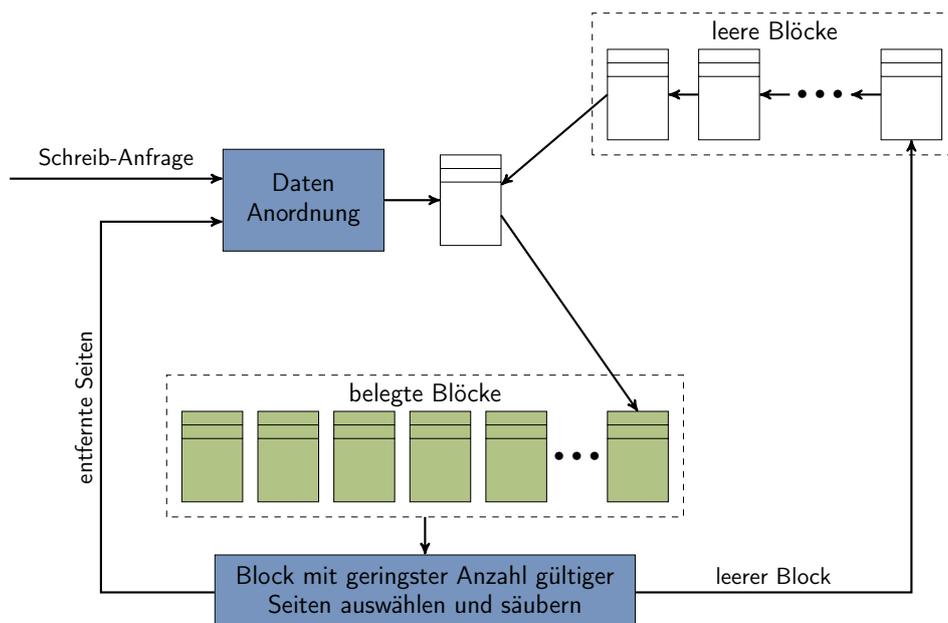


Abbildung 2.5: Schema zur Bereitstellung leerer SSD-Blöcke (nach Shimpi [29])

Das Problem könnte umgangen werden, indem die Daten z.B. während der Leerlaufzeiten des SSD, die im normalen Desktopbetrieb relativ häufig auftreten, umsortiert werden könnten. Somit würden fast immer ausreichend leere Blöcke zur Verfügung stehen, solange das SSD nicht komplett gefüllt ist. Dies ist jedoch nicht ohne weiteres möglich, da die Betriebssysteme bzw. genauer die genutzten Dateisysteme unter der Prämisse entwickelt wurden, dass ungültige Daten ohne Leistungsverlust überschrieben werden können, wie es bei herkömmlichen Festplatten auch der Fall ist. Dateisysteme markieren beim Löschen von Dateien lediglich die zuvor belegten Blöcke als ungültig in ihrem Inhaltsverzeichnis und geben diese Information nicht an das Blockgerät weiter. Bei SSDs ist dies von erheblichem Nachteil, da das Gerät erst beim Überschreiben eines Blockes von seiner Ungültigkeit erfährt und das oben beschriebene Umsortieren erst stattfinden kann, wenn der Block bereits wieder gebraucht wird und der Vorgang somit zeitkritisch ist. Um diesem Problem entgegenzutreten wurde vor kurzem der S-ATA-Standard um den so genannten TRIM-Befehl erweitert [1]. Dieser ermöglicht es, S-ATA-basierte Geräte über die Ungültigkeit von Blöcken zu informieren. Unglücklicherweise müssen jedoch für die Nutzung sowohl das Dateisystem als auch Betriebssystemkomponenten wie Gerätetreiber und Treiberschnittstellen angepasst werden. So kommt es, dass es momentan zwar schon Geräte gibt, die diese Möglichkeit unterstützen, sie jedoch auf Grund der mangelnden Softwareunterstützung nicht genutzt werden kann.



# Kapitel 3

## Verwandte Arbeiten

Vor dem Formulieren einer konkreten Aufgabenstellung gilt es, sich mit Arbeiten auseinander zu setzen, die sich zum einen mit Caching und zum anderen mit dem Zusammenspiel von Flash- und Magnetspeicher beschäftigen. Dazu werden in diesem Kapitel verschiedene Konzepte betrachtet. Abschnitt 3.1 thematisiert das von Sun Microsystems entwickelte ZFS, bei dem eine Cacheimplementierung auf der Dateisebene vorliegt. Unter Punkt 3.2 wird die proprietäre ReadyBoost-Technologie von Microsoft betrachtet, welche die Nutzung eines USB-Flash-Speichermediums als Cache vorsieht. Im Anschluss daran wird in Abschnitt 3.3 auf eine flashbasierte Cacheimplementierung der Firma Intel eingegangen, welche ein dediziertes Hardwaremodul nutzt. In Abschnitt 3.4 wird ein Artikel angesprochen, in dem der gegenteilige Ansatz gewählt wird und ein SSD durch eine Festplatte geacht wird. Die Erwähnung dieses Artikels soll verdeutlichen, dass auch das nahezu komplett gegenteilige Vorgehen erforscht wird. In Abschnitt 3.5 wird auf eine Forschungsarbeit eingegangen, die das Cachen von Netzwerkdaten auf Festplatten betrachtet. Dies ist für diese Arbeit von Relevanz, da im Artikel zum einen von Blockgeräten ausgegangen wird und zum anderen ein ähnliches Problem angesprochen wird, was in einer tieferen Ebenen der Speicherhierarchie liegt. Zum Abschluss dieses Kapitels wird in Abschnitt 3.6 ein Artikel betrachtet, welcher sich grundsätzlich mit Ersetzungsstrategien für Caches beschäftigt. Die Auswertung dieses Artikels ist wichtig, da die Wahl einer geeigneten Strategie von erheblicher Bedeutung für diese Arbeit ist.

### 3.1 Sun Microsystems' ZFS

ZFS ist ein von Sun Microsystems entwickeltes Dateisystem, welches primär für die Verwendung in Server- und Rechenzentren ausgelegt ist. Es ist kein klassisches Dateisystem, welches auf die Verwaltung von Daten auf einem einzelnen logischen oder physischen Gerät beschränkt ist, sondern eines, das die Verwaltung von Verbänden von Blockgeräten ermöglicht. ZFS organisiert die Blockgeräte hierfür in so genannten Pools, wobei diese als ausfallsicherheitssteigernde redundante Anordnung kostengünstiger Festplatten (RAIDs) organisiert werden können. Sollten ausreichend Geräte zur Verfügung stehen, werden zur Steigerung der Geschwindigkeit die Daten auf den Geräten parallel verteilt. Geräte können hierbei jederzeit zu einem Pool hinzugefügt werden, um den Speicherplatz, die Datensicherheit oder die Leistung zu erhöhen.

Die Artikel von System News Inc. [32] und Brendan [5] setzen sich mit einem weiteren Mechanismus von ZFS auseinander. Sie betrachten die Möglichkeit, ein Blockgerät als

Cache zu benutzen. Ein Blockgerät kann hierfür zu einem Pool hinzugefügt werden und als Cache markiert werden, wobei diese Blockgeräte performanter sein sollten als die zu cachenden Geräte. Aus diesem Grund kommen hierfür z.B. SSDs in Frage.

Brendan vergleicht nun die Zugriffszeiten auf das ZFS-Dateisystem in Abhängigkeit davon, ob die Anfrage aus dem Arbeitsspeicher des Systems, des SSDs oder der Festplatte bedient wird. Die Ergebnisse finden sich im Diagramm in Abbildung 3.1. Es ist klar zu erkennen, dass die Zugriffszeiten um Größenordnungen auseinander liegen. Somit wird deutlich, dass die Nutzung von SSDs als zusätzlicher Cache aus Sicht der Leistung sinnvoll ist.

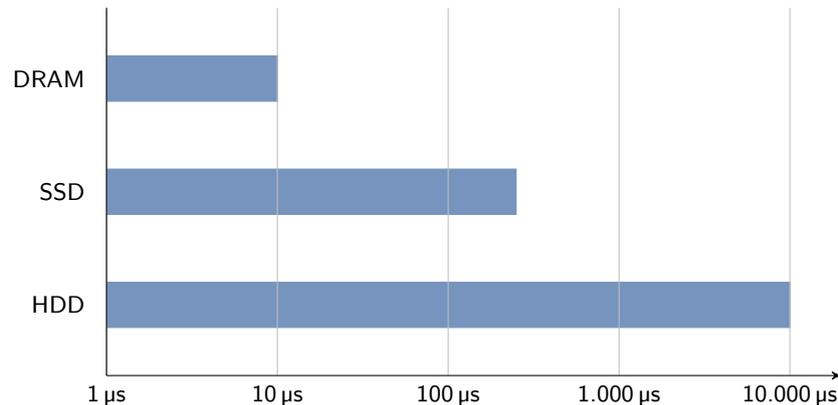


Abbildung 3.1: Zugriffszeit auf den ZFS Pool in Abhängigkeit vom bedienenden Systemteil (nach Brendan [5])

Der Artikel von System News Inc. [32] setzt sich mit dem ökonomischen Aspekt der Nutzung von SSDs als Cache auseinander. Es wird durch eine Beispielrechnung veranschaulicht, wie groß die Ersparnis in Anschaffung und Stromverbrauch und somit indirekt der Betriebskosten durch die Nutzung von SSDs als Cache für ein gegebenes Szenario ist. Der Artikel kommt dabei zu dem Schluss, dass die Anschaffung des SSD gestützten Systems bei gleichbleibender Leistung 64% günstiger und der Stromverbrauch 77% geringer ist. Dabei gilt es jedoch anzumerken, dass für den Vergleich Serverfestplatten genutzt wurden, welche einen Verbrauch von 17,5 Watt hatten. Bei der Nutzung von Desktop- oder Notebookfestplatten mit einem Verbrauch von ca. 3-8 Watt wäre der Unterschied geringer ausgefallen, jedoch hätte es wahrscheinlich einen deutlichen Leistungsunterschied gegeben.

## 3.2 Microsofts ReadyBoost

Im Patent [17] wird eine Technik zum Cachen von Plattensektoren auf einem Wechselmedium beschrieben. Diese Technik wird von Microsoft unter der Bezeichnung *ReadyBoost* vermarktet. Die Technik dient in erster Linie zur Leistungssteigerung älterer Systeme, die eine geringe Menge an Hauptspeicher aufweisen. Bei diesen Systemen ist das Cachen von Daten im Hauptspeicher, wie in der Einleitung beschrieben wurde, kaum möglich, so dass ein intensives Arbeiten auf dem Plattengerät erforderlich ist. Das Patent beschreibt hierbei die Möglichkeit, diesen Umstand zu mildern, indem z.B. ein günstiges USB-Speichermedium zum Zwischenspeichern der Daten eingesetzt wird. Dies spiegelt nochmals die veränderte Speicherhierarchie wieder, welche in der Einleitung gezeigt wurde. Die Ausarbeitung kommt dabei zu teilweise sehr hohen Leistungssteigerungen, wie beispielhaft der Tabelle 3.1 entnommen werden kann.

Tabelle 3.1: Geschwindigkeitsgewinn für Endnutzersysteme bei der Nutzung von Ready-Boost (Quelle: Kirshenbaum u. a. [17])

System	Lesezeit in Sek. der simulierten HDD	Prozentsatz der simulierten Lesezeit mit einem USB2-Speichermedium der Größe:					
		0MB	32MB	64MB	128MB	256MB	512MB
System 1 (128MB)	1259	100%	89%	70%	37%	18%	18%
System 2 (128MB)	1011	100%	90%	70%	38%	22%	22%
System 3 (128MB)	2159	100%	88%	72%	44%	25%	20%
System 4 (128MB)	866	100%	90%	80%	63%	48%	37%
System 5 (256MB)	1747	100%	92%	85%	70%	52%	40%
System 6 (256MB)	2187	100%	94%	87%	76%	66%	57%

Hierbei wurden die Zugriffsmuster typischer Desktopsysteme simuliert. Dabei wurden als Cache USB-Speichermedien mit verschiedenen Größen eingesetzt. Es ist zu erkennen, dass die Lesezeit des Datensatzes sich auf dem mit 512MB Arbeitsspeicher ausgestatteten System 1 auf 18% reduzierte.

### 3.3 Intels Turbo Memory

Matthews u. a. [22] beschreiben die Intel Turbo Memory (ITM) Technologie. Bei ITM handelt es sich um ein Hard-/Software-Codesign. Auf der Hardwareseite finden sich ein NAND-Flash-Speicher, ein Flash-Controller und ein sogenanntes OROM, welches im Fehlerfall aktiv wird. Das Hardwaremodul wird über den PCI-Express-Bus an das System angebunden. Auf der Softwareseite befindet sich der ITM-Treiber, welcher die Ansteuerung des Hardwaremoduls übernimmt, die Schnittstelle für das Betriebssystem zur Verfügung stellt und Funktionen, die die Nutzung des Hardwaremoduls zum einen als SSD zulassen und zum anderen als Cache für Festplatten. Dieser Sachverhalt ist in Abbildung 3.2 zu sehen.

Bei ITM handelt es sich um einen Lese-/Schreibcache, der auch nach einem Neustart erhalten bleibt. Aus diesem Grund sind besondere Maßnahmen für die Ausfallsicherheit zu treffen, da die aktuellen Daten nicht einheitlich auf einem Gerät zu finden sind, sondern auf der Festplatte und dem Cache verteilt sind. Intel hat den bereits von Hensbergen und Zhao [14] angesprochenen Ansatz gewählt, die Metadaten des Caches sowohl im Arbeitsspeicher als auch im Flashspeicher vorzuhalten. Für die Verwaltung der Metadaten werden pro vier Kilobyte Cacheblock 40 Byte Daten im Hauptspeicher benötigt. Es liegt also ein Verhältnis von Speicherverbrauch zu Cachegröße von ca. 1:100 vor. Weiterhin wird auf das Problem hingewiesen, dass durch die Einführung eines persistenten Caches gleichzeitig zwischen zwei persistenten Speicherebenen eine nicht-persistente in Form des Festplattencaches eingefügt wird (Abbildung 3.3). Dadurch erhöht sich die Gefahr von Datenverlust bei einem Systemfehler.

Dies kann zu Problemen im Fehlerfall führen, wenn das Schreiben der Daten vom Cache an das Betriebssystem bereits bestätigt wurde, aber noch nicht sicher ist, ob der Schreibvorgang zwischen Cache und Festplatte korrekt abgeschlossen wurde. Zur Lösung dieses Problems wird im Artikel vorgeschlagen, den Festplattencontroller vor der Aktualisierung der Metadaten des Caches explizit anzuweisen, den Festplattencache zu leeren.

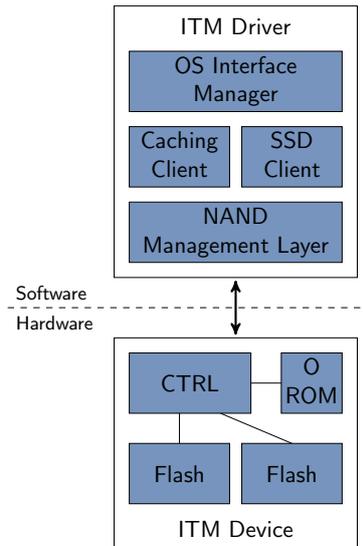


Abbildung 3.2: Blockdiagramm des Intel Turbo Memory (nach Matthews u. a. [22])

Des Weiteren wird auf das Problem der Separierung von Festplatte und Cache hingewiesen, was vor allem bei einem Write-Back-Cache auftritt. Um das Problem zu lösen, werden beim Herunterfahren des Rechners grundsätzlich alle als verändert markierten Blöcke des Caches auf die Festplatte zurückgeschrieben. Sollte dies durch einen Systemfehler nicht möglich sein, werden die Daten beim Starten des Rechners durch das OROM zurück auf die Festplatte geschrieben, ehe der normale Bootvorgang startet. Diese Funktionalität des ITM ist der Grund dafür, dass es sich bei ITM um ein Hard-/Software-Codesign handelt. Die Daten werden unabhängig vom Gerätetreiber selbständig von der Hardware kopiert. Sollten Festplatte und Cache jedoch nach einem Systemfehler separiert werden, sind besondere Maßnahmen zu treffen. Von den Autoren wird hierfür auf Patent [9] verwiesen.

Weiterhin wird in dem Artikel auf Caching-Strategien eingegangen. Es wird ausgeführt, dass es wenig sinnvoll ist, lange Lese-/Schreib-Sequenzen zu cachen. Als effektiver wird angesehen, die Daten zu cachen, die ungeordnet angefragt werden. Um festzustellen, ob Daten zu der einen oder anderen Art gehören, benutzen die Autoren die Größe der jeweiligen Datenanfrage. Außerdem sind sie der Meinung, es sei nur sinnvoll Daten zu cachen, die mehr als einmal wiederbenutzt werden. Von den Autoren wird hierbei darauf hingewiesen, dass dies besonders beim Pagefile<sup>1</sup> des Betriebssystems der Fall ist.

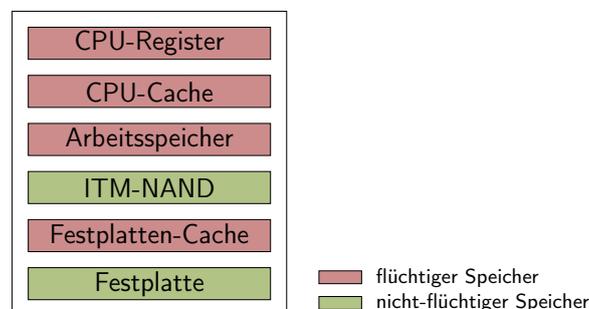


Abbildung 3.3: Speicherhierarchie mit Intel Turbo Memory (nach Matthews u. a. [22])

<sup>1</sup> Bildet die Grundlage des virtuellen Speichers eines Betriebssystems und enthält die ausgelagerten Seiten.

Die Autoren weisen in weiteren Teilen des Artikels außerdem darauf hin, dass die Hit-Rate<sup>2</sup> kein gutes Kriterium ist, um die Qualität und Leistung des Caches zu beurteilen. Sinnvoller sei es, die Länge der aufeinanderfolgenden Anfragen zu messen, welche nicht von der Festplatte sondern vom Cache bedient werden können.

Die Autoren dieses Artikels kommen zu dem Schluss, dass das Nutzen von Flash-Speicher zum Cachen von Festplatten sinnvoll ist. Ein Auszug ihrer Ergebnisse bei der Nutzung von ITM findet sich in Tabelle 3.2.

Tabelle 3.2: Ergebnisse der Nutzung von ITM (Quelle: Matthews u. a. [22])

Art der Systemnutzung	Leistungsgewinn
Arbeit mit Microsoft Office mit einer speicherintensiven Anwendung im Hintergrund	Verdoppelte Reaktionszeit
Bootzeit bis zum Startmenü	22% schneller
PCMark05 disk	20%ige Verbesserung
Ein Vorbeiflug an einem Nationalpark mit Google Earth gefolgt von der Nutzung von Adobe Photoshop Elements 5.0 zur Erstellung einer Slideshow, die Bilder desselben Parks zeigt	2-fache Geschwindigkeit

### 3.4 Griffin

Das von Soundararajan u. a. [30] vorgeschlagene Verfahren nutzt eine Festplatte zum Cachen von SSD-Zugriffen, um die Anzahl der Schreibvorgänge auf das SSD zu verringern. Der Artikel hat auf den ersten Blick wenig mit dem in dieser Arbeit behandelten Thema gemein, ist aber für einige Designentscheidungen trotz alledem von Bedeutung. Von den Autoren wird auf das Problem eingegangen, dass bei SSDs die Anzahl der möglichen Schreibvorgänge begrenzt ist. Aus diesem Grund wird vorgeschlagen, statt Schreibvorgänge direkt auf einem SSD auszuführen, diese auf einer Festplatte zu cachen. Leistungseinbußen werden hierbei durch die Architektur des Caches vermieden. Der Cache wird hierfür in Form eines Logs aufgebaut, so dass neue Schreibvorgänge stets an das Ende des Caches geschrieben werden und somit der ggf. ungeordnete Schreibzugriff auf dem SSD in einen geordneten auf der Festplatte überführt wird. Da ungeordnete Schreibzugriffe auf SSDs meist langsamer sind als geordnete auf einer Festplatte, sollte es laut der Autoren kaum Leistungsunterschiede geben. Des Weiteren wird der Kostenvorteil einer Festplatte gegenüber einem SSD genutzt. Durch den Kostenvorteil ist es möglich, dass der Cache größer ist als das Quellgerät. Abschließend gehen die Autoren näher auf die von ihnen ausgeführten Tests ein, die neben einer Festplatte als Cache auch das Cachen von MLC-SSDs mit Hilfe von SLC-SSDs betrachten.

### 3.5 Dynamic Policy Disk Caching for Storage Networking

Die Arbeit von Hensbergen und Zhao [14] beschäftigt sich mit dem Cachen von Storage Area Networks (SANs) auf lokalen Festplatten. Das Thema des Reports hat auf den ersten Blick nichts mit dem Thema dieser Arbeit gemein, aber es wird auf die Eigenschaft

<sup>2</sup> Ist das Verhältnis zwischen der Gesamtzahl der Anfragen und den aus dem Cache bedienten Anfragen.

eines SANs eingegangen, blockbasierend zu sein. Somit wird im Report das Cachen eines Blockgeräts durch ein anderes beschrieben, was beim Cachen eines SSD durch eine Festplatte ebenfalls der Fall ist. Weiterhin ist der Unterschied bezüglich der Zugriffszeit zwischen Festplatte und SANs vergleichbar mit der zwischen SSDs und Festplatten. Im Report wird eine Linux-Kernelerweiterung beschrieben, die die Aufgabe des Cachens erfüllt. Da die Erweiterung generell auf logische Blockgeräte zugeschnitten ist, ist sie auch für diese Arbeit relevant und soll somit untersucht werden.

Schließlich kommt der Report in Bezug auf SANs zu dem Schluss, dass unter der Verwendung der implementierten Erweiterung ein beachtlicher Leistungsgewinn zu beobachten ist. Die Laufzeiten der gewählten Benchmarks verringern sich auf bis zu einem Drittel. Weiterhin wird die Ausfallsicherheit des Caches diskutiert. Diese Diskussion ist auch für diese Arbeit von Bedeutung. Für die Ausfallsicherheit ist hauptsächlich das Speichern der Metadaten maßgebend. Die gecachten Inhalte liegen ohnehin in einem persistenten Speicher, so dass sie beim Ausfall nicht verloren gehen können. Anders sieht es bei den Metadaten aus, da diese aus Leistungsgründen im nicht persistenten Arbeitsspeicher gehalten werden und bei einem Ausfall verloren gehen.

Hensbergen und Zhao stellen drei Lösungsmöglichkeiten vor, wobei nur die letzten beiden auch für einen Write-Back-Cache geeignet sind. Die erste Möglichkeit besteht im kompletten Neuaufbau des Caches. Die Zweite im zusätzlichen Speichern der Metadaten im persistenten Speicher. Die dritte Möglichkeit ist eine Kombination aus den ersten beiden, wobei nur Daten von Cacheblöcken gespeichert werden, die lediglich im Cache aktuell sind.

### 3.6 Ersetzungsstrategien für Caches

Da ein Cache für gewöhnlich weniger Speicherplatz aufweist als das Quellmedium<sup>3</sup>, ist die Frage, welche Daten im Cache gehalten werden und welche nicht, von zentraler Bedeutung. Genauer gesagt ist es die Frage, welcher Block bei einem vollen Cache als nächster ersetzt wird. Die optimale Vorgehensweise hierfür ist im Verfahren von Belady [2] beschrieben. Es besagt, dass der Block verdrängt wird, auf den in der Zukunft am längsten nicht mehr zugegriffen wird. Da dieses Verfahren jedoch Wissen über das zukünftige Verhalten des Systems voraussetzt, es ist nur bei einer Offline-Planung<sup>4</sup> einsetzbar. Für eine Online-Planung muss versucht werden, das zukünftige Verhalten des Systems vorherzusagen. Die Vorhersage geschieht für gewöhnlich auf Grundlage des vergangenen Verhaltens des Systems. Hierbei liegt die Annahme zu Grunde, dass das zukünftige Verhalten dem vergangenen Verhalten ähnelt. Laut Lee u. a. [20] gibt es hierbei zwei grundsätzliche Herangehensweisen – das *Least Recently Used (LRU)*-Prinzip und das *Least Frequently Used (LFU)*-Prinzip.

Das LRU-Prinzip basiert auf der Annahme, dass auf einen Block, auf den bereits sehr lange nicht mehr zugegriffen wurde, auch in Zukunft in absehbarer Zeit nicht zugegriffen wird. Dieses Verfahren bedeutet einen relativ geringen Aufwand, da das System lediglich den am längsten nicht genutzten Block bestimmen muss. Dies kann z.B. durch eine Liste geschehen, in der bei einem Zugriff auf einen Block dieser immer an den Anfang gesetzt wird. So würde sich der am längsten nicht genutzte Block am Ende befinden.

Das LFU-Prinzip hingegen basiert auf der Zugriffshäufigkeit bzw. Zugriffsfrequenz. Es geht von der Annahme aus, dass Blöcke, die in der Vergangenheit häufig benutzt wurden,

---

<sup>3</sup> Dies ist das gecachte Medium.

<sup>4</sup> Das gesamte Verhalten des Systems ist während der Entwurfszeit bekannt.

auch in der Zukunft häufig benutzt werden. Dieses Verfahren hat im Vergleich zum LRU-Algorithmus einen höheren Aufwand, da zu jedem Cacheblock Zugriffsstatistiken geführt werden müssen und die Blöcke nach ihrer Zugriffsfrequenz sortiert werden müssen. Zudem besteht das Problem, dass Cacheblöcke, auf die in der Vergangenheit oft zugegriffen wurde, lange im Cache verbleiben, auch wenn sie nicht mehr benötigt werden. Ein Beispiel hierfür könnte eine gelöschte Datei sein, die zwar in der Vergangenheit oft genutzt wurde, jetzt jedoch nicht mehr existiert und somit die Daten auch im Cache nicht mehr benötigt werden.

Von Lee u. a. wird nun ein Raum von Caching-Strategien postuliert, der von den beiden genannten Strategien aufgespannt wird und diese als Extremfälle beinhaltet. Es werden dabei Strategien untersucht, die in diesem Raum liegen und mit den Extremfällen und der optimalen ex-post Lösung verglichen werden. Die Autoren kommen zu dem Schluss, dass Strategien, die zwischen LRU und LFU liegen, meist besser geeignet sind. Weiterhin lassen sich diese flexibel implementieren, so dass sich die Strategien durch einen einzigen Parameter Richtung LRU oder LFU steuern lassen und für ein gegebenes Szenario leicht optimieren lassen.



# Kapitel 4

## Problemstellung

Nach der kurzen Einführung im ersten Kapitel in die in dieser Arbeit betrachtete Problematik soll im nun folgenden Kapitel die Problemstellung konkretisiert werden. Dies geschieht auf Grundlage der technischen Gegebenheiten, die in Kapitel 2 erläutert wurden, und den bereits existierenden Arbeiten, welche in Kapitel 3 vorgestellt wurden.

In der Einleitung wurde bereits angedeutet, dass zwischen Festplatten und SSDs ein signifikanter Leistungsunterschied bezüglich der Zugriffszeit besteht. Die Ursachen hierfür wurden in Kapitel 2 ausführlich dargestellt. Somit wäre es zwar wünschenswert auf Grund der Leistungsvorteile der SSDs gegenüber von Festplatten, diese komplett durch SSDs zu ersetzen. Es würde weiterhin den Vorteil mit sich bringen, dass sämtliche Software komplett auf SSDs umgestellt werden könnte. Dies würde sowohl die Leistung der SSDs ausreizen, da auf die spezifischen Eigenschaften wie z.B. das TRIM-Kommando eingegangen werden kann, und Entwicklungskosten gespart werden, da nicht für zwei unterschiedliche Gerätetypen die Software optimiert werden müsste. Aus den preislichen Gegebenheiten, die ebenfalls bereits in Kapitel 1 angesprochen und durch die Tabelle 1.1 verdeutlicht wurden, ist die nahezu vollständige Verdrängung der herkömmlichen Festplatten durch SSDs in naher Zukunft aber nicht zu erwarten.

Aus diesem Grund sollte eine Alternative gefunden werden, die besseren Leistungseigenschaften eines SSDs effektiv zu nutzen. Hierbei sollten zunächst die Kostenunterschiede beachtet werden. Daher sollte eine Lösung auch mit vergleichsweise wenig SSD-Speicher auskommen und trotzdem eine messbare Leistungssteigerung erbringen. Wie einige der Arbeiten aus Abschnitt 3.2 und 3.3 diskutieren, scheint die in der Einleitung in Abbildung 1.4(b) gezeigt Architektur ein Lösungsansatz hierfür zu sein. Dabei wird das SSD als Cache genutzt. Dies hat zum einen den Vorteil, dass der Cache als zusätzliche Schicht in die Speicherhierarchie eingefügt werden kann. Dadurch ist der Anpassungsaufwand an die darüber und darunter liegenden Ebenen sehr gering. Zum anderen ist der Cache sehr flexibel in Bezug auf den zur Verfügung stehenden Speicher. Aus diesem Grund soll die Nutzung von SSDs als Cache für herkömmliche Festplatten in dieser Arbeit detailliert betrachtet werden.

Im Mittelpunkt dieser Arbeit soll der Entwurf, die Implementierung und die Evaluierung eines Linux-Kernelmoduls stehen, welches eine Speicherhierarchie ermöglicht, wie sie in Abbildung 1.4(b) zu sehen ist. Bei der Umsetzung müssen zunächst die momentanen technischen Rahmenbedingungen beachtet werden. Zur Zeit gebräuchliche Festplatten haben eine Kapazität von bis zu zwei Terabyte. Aus diesem Grund sollte der Cache mindestens diese Größe unterstützen, wobei eine Größe um zehn Terabyte wünschenswert ist, um in

den nächsten zwei Jahren nicht an die Grenze der Möglichkeiten des Caches zu stoßen. Ähnliches gilt für die Größe des verwendeten SSD. Ausgehend von der Vermutung, dass das Cachegerät wesentlich kleiner ist als die Festplatte, da es ansonsten als alleiniges Speichergerät verwendet werden könnte, sollte der Cache eine Größe von bis zu einem Terabyte beim Cachegerät unterstützen.

Des Weiteren muss die Entscheidung getroffen werden, ob der Cache block- oder dateisystemorientiert arbeiten soll. Diese Frage ist grundlegender Natur, da auf der Dateisystemebene wesentlich mehr Informationen über Daten zur Verfügung stehen als auf der Blockebene. Dieses beeinflusst nicht nur den Aufbau des Kernelmoduls an sich sondern auch die Möglichkeiten bezüglich der Cachingstrategien. Ein weiterer Punkt, den es zu beachten gilt, ist die Portabilität. Ein blockorientierter Cache könnte mit einem beliebigen Dateisystem genutzt werden. Ein dateisystemorientierter Cache wäre an das zugrunde liegende Dateisystem gebunden und müsste für jedes neue Dateisystem spezifisch implementiert werden. Unter anderem auf Grund dieser Tatsache soll der Cache blockorientiert entwickelt werden. Diese Entscheidung wird dadurch unterstützt, dass mit dem in Abschnitt 3.1 vorgestellten Arbeiten von Brendan [5] und System News Inc. [32] untersuchten ZFS bereits eine effiziente Implementierung eines Caches auf Dateisystemebene existiert.

Der Cache soll weiterhin die Randbedingung erfüllen, möglichst ressourcenschonend zu arbeiten. Die Nutzung eines Caches ist nur sinnvoll, wenn durch die Nutzung andere Teile des Systems nicht massiv verlangsamt werden. Ein besonderes Augenmerk liegt hierbei auf dem Speicherverbrauch und der Prozessorleistung. Der Cache wird auf Grund von Erwägungen bezüglich der Geschwindigkeit große Teile seiner Verwaltungsdaten im Arbeitsspeicher halten müssen. Dadurch steht weniger Speicher für die Nutzung durch Programme bzw. das Cachen von Daten zur Verfügung. Sollte der Cache sehr viel Speicher für die Metadaten benötigen, könnte es zu massiven Leistungseinbußen kommen. Aus diesem Grund soll der zu implementierende Cache möglichst geringen Speicherverbrauch in Abhängigkeit von der Cachegröße aufweisen. Ähnliches gilt für die Nutzung des Prozessors. Auch hier sollte darauf geachtet werden, dass die Prozessorlast durch den Cache möglichst gering ist. Sollte dies nicht der Fall sein, würde es in einem Multitasking-System zu einer Verlangsamung anderer Prozesse kommen. Dies ist nicht erwünscht.

Neben dem ressourcenschonenden Arbeiten sollte der Cache weiterhin einfach zu managen sein. Konkret heißt dies, dass Cachegeräte jederzeit in ein System eingefügt und entfernt werden können ohne größeres Zutun des Nutzers. Somit sollte der Cache zunächst mit einigen wenigen, wenn nicht sogar nur einem Kommando aufgebaut werden können. Weiterhin sollte das Entfernen des Cachegeräts nicht zu einem Ausfall des Systems führen und auch kein Eingreifen des Benutzers erfordern.

Schließlich sollte es auch möglich sein, von dem gecachten Gerät zu booten. Auch hierbei sollte der Cache möglichst einfach in ein bestehendes System eingefügt werden können.

Es lassen sich somit folgende Anforderungen an den Cache feststellen:

- Blockorientiert
- Ressourcenschonend
- Einfaches Management
- Bootfähig

Neben der Umsetzung der technischen Eigenschaften soll ein geeigneter Cache-Algorithmus gefunden werden, der zum einen gute Ergebnisse bezüglich der Leistung verspricht, aber auch durch entsprechende Parameter bezüglich der obigen Kriterien optimiert werden kann. Hierbei sollen unter anderem die Ergebnisse von Belady [2] beachtet werden, welcher in Abschnitt 3.6 vorgestellt wurden.

Nach der erfolgten Beispielimplementierung soll diese auf die Erfüllung der soeben genannten Anforderungen geprüft werden. Dies soll anhand zweier Szenarien stattfinden. Zum einen soll ein reales System betrachtet werden. Dabei sollen die Unterschiede bei Leistung und Gerätenutzung in Abhängigkeit vom genutzten Systemspeicher genauer betrachtet werden. Hierbei soll eine durch ein SSD gecachte Festplatte untersucht werden und mit einer einzelnen Festplatte bzw. einem SSD verglichen werden. Zum anderen soll ein synthetisches Testszenario, welches ein Desktopsystem simuliert, entworfen werden. Dies soll es ermöglichen, in einem vertretbaren Zeitrahmen mehrere Arbeitstage zu simulieren. Des Weiteren soll es den Vergleich verschiedener Massenspeichergeräte unabhängig vom genutzten Betriebs- bzw. Dateisystem ermöglichen. Dies soll ein Vergleich des blockbasierten Caches für Linux mit ZFS ermöglicht, welches in der Referenzimplementierung nicht für Linux zur Verfügung steht. Es soll die Referenzimplementierung, welche z.B. unter OpenSolaris [24] zur Verfügung steht, genutzt werden, um schlechte Leistungswerte auf Grund einer mangelhaften Implementierung vorzubeugen.



# Kapitel 5

## Architektur

Nach der Konkretisierung der Aufgabenstellung im vorangegangenen Kapitel soll nun die zur Lösung der Aufgabe vorgesehene Architektur vorgestellt werden. Dabei setzt sich Abschnitt 5.1 mit dem High-Level-Design auseinander. Zusätzlich wird das Device-Mapper (DM)-Framework vorgestellt, das bei der konkreten Realisierung genutzt wird.

Nach der Beschreibung des High-Level-Designs wird in Abschnitt 5.2 auf den genutzten Cache-Algorithmus eingegangen. Der Algorithmus wird detailliert erklärt und an einem Beispiel verdeutlicht.

### 5.1 High-Level-Design

Bevor eine Architektur für den Cache entwickelt werden kann, muss die Verwaltung von Blockgeräten im Kernel verstanden werden. Sie werden im Kernel durch eine so genannte Block-Ein-/Ausgabe (BIO)-Datenstruktur verwaltet (siehe Anhang Listing A.1). Diese enthält diverse Verwaltungsinformationen, wie z.B. die gewünschten Sektoren, eine Referenz auf das betreffende Blockgerät und Informationen über den Status der Anfrage. BIOs werden hauptsächlich durch Systemaufrufe von Benutzerprogrammen generiert, denen Dateizugriffe zu Grunde liegen. Der Nutzerprozess wird dabei solange blockiert, bis sämtliche von ihm indirekt erzeugte BIOs abgeschlossen sind.

Für die Implementierung des Kernelmoduls bieten sich nun zwei Herangehensweisen an. Die Erste besteht darin, ein Blockgerätetreiber von Grund auf zu entwickeln. Dies würde jedoch einen erheblichen Entwicklungsaufwand bedeuten, da einerseits die komplette Blockgeräteschnittstelle, wie sie vom Linux-Kernel vorgeschrieben wird, implementiert werden müsste und andererseits der gesamte Zugriff auf das Cacheblockgerät und Quellgerät verwaltet werden müsste. Ferner müssten sämtliche BIO-Anfragen verwaltet und ggf. gefiltert werden.

Die Zweite effizientere Möglichkeit die Anforderungen der Aufgabenstellung zu erfüllen, stellt der Linux-DM dar. Der DM stellt ein generisches Framework zur Umlenkung von BIO-Anfragen auf ein oder mehrere logische oder physische Blockgeräte zur Verfügung. Er erscheint im Kernel wiederum selbst als logisches Blockgerät, so dass der Zugriff auf die abgebildeten Geräte vollkommen transparent und ohne Softwareanpassung möglich ist. Der DM stellt somit eine einfache Möglichkeit dar, Kernelerweiterungen zu implementieren, welche auf der Manipulation von BIOs basieren. Als Beispiele hierfür sind

Verschlüsselungsmechanismen, Software RAIDs oder das lineare Aneinanderfügen mehrerer Blockgeräte zu nennen. Da der DM für die Bearbeitung der Aufgabenstellung dieser Arbeit eine zentrale Rolle spielt, soll an dieser Stelle die Funktionsweise und die dadurch ermöglichte Architektur kurz anhand des linearen Aneinanderfügens mehrerer Blockgeräte erläutert werden.

Es wird dafür angenommen, dass im System zwei physische Blockgeräte in Form von Festplatten mit den Bezeichnungen *sda* und *sdb* und einer Kapazität von jeweils ein Terabyte existieren. Ziel ist es, ein logisches Blockgerät zu erzeugen, welches eine Kapazität von zwei Terabyte besitzt und die Daten auf den beiden physischen Festplatten *sda* und *sdb* speichert. Dieses soll im Folgenden *dm* heißen und das hierfür benötigte Kernel-Modul *dm-lincomb*. Der Sachverhalt ist in Abbildung 5.1 schematisch dargestellt.

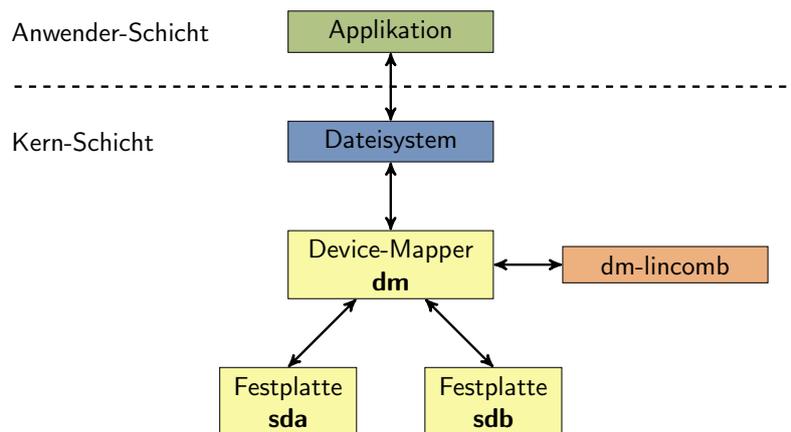


Abbildung 5.1: Exemplarische Darstellung der Device-Mapper-Umgebung

Kernelmodule für das DM-Framework müssen ein bestimmtes Set von Funktionen zur Verfügung stellen. Von zentraler Bedeutung ist hierbei die `map` Funktion. Sie wird vom Framework immer dann aufgerufen, wenn an das logische DM-Gerät eine BIO-Anfrage gestellt wird. Für das gewünschte Blockgerät könnte die `map`-Funktion<sup>5</sup> daher wie folgt aussehen:

---

```

1 void map(bio Request){
2     if(Request.block ≤ 1TB){
3         Request.device = sda;
4     }
5     else{
6         Request.block = Request.block - 1TB;
7         Request.device = sdb;
8     }
9 }
  
```

---

Listing 5.1: `map`-Funktion des `dm-lincomb` Moduls

Die Funktion überprüft, ob der angeforderte Block innerhalb des ersten Terabyte liegt oder im Zweiten. Sollte er im Ersten liegen, wird die Anfrage lediglich auf das Blockgerät *sda* umgelenkt. Liegt er dagegen im Zweiten, wird neben der Umleitung auf das Blockgerät *sdb* die Position des zu lesenden Blockes um ein Terabyte verschoben, damit die Anfrage

<sup>5</sup> Diese Funktion erfüllt nicht die Anforderungen des DM-Frameworks und dient nur zur Veranschaulichung des Sachverhalts.

konsistent mit der Blocknummerierung des Gerätes ist. Nach Verlassen der `map` Funktion wird der BIO-Auftrag vom DM-Framework an die jeweiligen Geräte weitergereicht und dem Benutzer gegenüber transparent ausgeführt.

Auf die genauere Funktionsweise des DM selbst soll in dieser Arbeit nicht eingegangen werden. Hierfür wird auf [10] hingewiesen. Es sind jedoch sind auch komplexere Datenstrommanipulationen möglich, wie sie für den zu implementierenden Cache benötigt werden.

Das Umlenken des Datenstroms über die `map`-Funktion, wie es oben beschrieben ist, kann zum Umlenken von Lese-/Schreibanfragen genutzt werden, je nachdem, ob die Anfrage aus dem Cache oder vom Quellgerät ausgeführt werden soll. Mit dieser Funktion lässt sich jedoch nicht das Ein- und Auslagern von Seiten in bzw. aus dem Cache realisieren. Hierfür muss das prinzipielle Blockdiagramm aus Abbildung 5.2 erweitert werden, so dass das DM-Modul selbständig Cacheblöcke zwischen den Geräten tauschen kann. Daher ergibt sich für den Cache folgendes High-Level-Blockdiagramm:

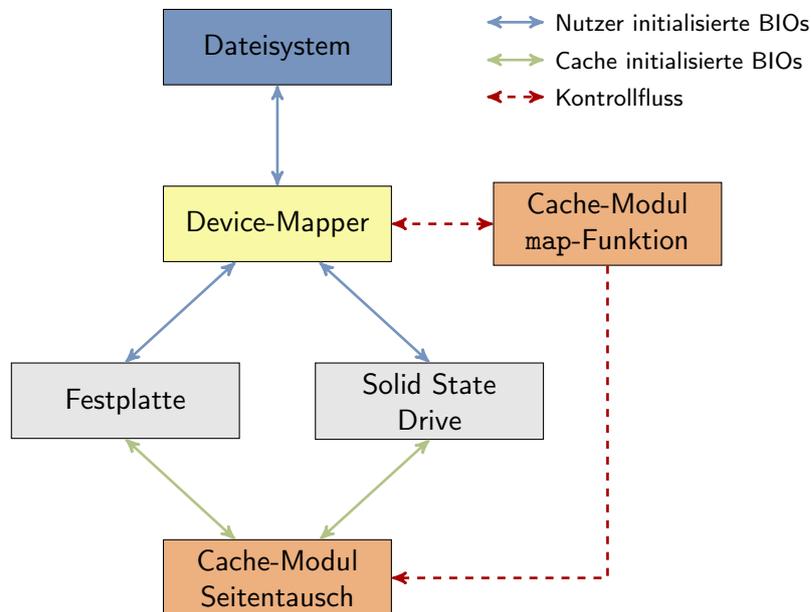


Abbildung 5.2: High-Level-Blockdiagramm des zu implementierenden Caches

Wie zu erkennen ist, lässt sich das Cache-Modul grundsätzlich in zwei Teile aufteilen. Der erste Teil implementiert das DM-Interface und somit unter anderem die `map`-Funktion. Der zweite Teil arbeitet asynchron hierzu und wird für den Tausch von Cacheblöcken zuständig sein. Dabei gilt es einen geeigneten Algorithmus für das Tauschen zu finden, was im nächsten Abschnitt diskutiert werden soll.

## 5.2 Cache-Algorithmus

Der Cache-Algorithmus lässt sich in zwei Bereiche unterteilen. Zum einen in den Bereich der Ersetzungsstrategie und zum anderen in den der Schreibstrategie. Die Ersetzungsstrategie legt fest, wann welche Blöcke durch andere ersetzt werden. Die Schreibstrategie legt fest, ob und wann Daten exklusiv auf das Cache- bzw. Quellgerät geschrieben werden oder auf beide.

### 5.2.1 Ersetzungsstrategie

Der in Abschnitt 3.6 vorgestellte Artikel von Belady [2] kommt zu dem Schluss, dass es durchaus sinnvoll ist, keine reine LRU- oder LFU-Strategie zu verwenden, sondern eine Strategie zu wählen, die zwischen diesen beiden liegt. Aus diesem Grund sollte die Strategie, die für diese Arbeit entwickelt wurde, keine reine LRU- oder LFU-Strategie sein, sondern im Bereich dazwischen liegen. Weiterhin sollte es möglich sein, mit ihr zumindest näherungsweise eine LRU- bzw. LFU-Strategie zu erreichen, um untersuchen zu können, welche Strategien für den implementierten Cache Vor- und Nachteile mit sich bringen. Des Weiteren soll beim Entwurf der Ersetzungsstrategie stark auf den Speicherverbrauch geachtet werden. Als Grundlage soll zunächst ein satzassoziativer Cache dienen. Dies ist ein allgemeiner Ansatz, mit dessen Hilfe sich ebenso ein vollassoziativer als auch ein direkt abgebildeter Cache realisieren lässt, da diese nur Untermengen des satzassoziativen Caches sind. Jedes Set wird neben den Cacheblöcken zusätzlich mit einem Zeiger ausgestattet, der es ermöglicht, über die Cacheeinträge zu iterieren. Die Metainformationen der Cacheblöcke an sich sollen neben einer Statusvariablen und der Information über den aktuell gecacheten Block einen zusätzlichen Zähler beinhalten, um das Zugriffsverhalten auf diesen Block zu reflektieren. Dieser kann theoretisch beliebig lang sein, sollte aber aus Speichersparnis möglichst kurz gehalten werden. Details über die Größe sind im Rahmen der Implementierung zu klären.

Der Zähler der Cacheblöcke wird beim Einlagern mit einem zur Laufzeit definierbaren Wert  $s$  initialisiert. Sollten alle Blöcke eines Sets belegt sein, wird mit Hilfe des Zeigers im Set über die Cacheblöcke iteriert bis einer gefunden wird, dessen Zähler auf Null steht. Dieser Block wird sodann für die Ersetzung ausgewählt. Die Zähler der Blöcke, über die bei der Suche nach dem zu ersetzenden Block iteriert wird, werden um eins vermindert. Beim Zugriff auf einen Cacheblock wird sein Zeiger um einen zur Laufzeit zu definierenden Wert  $i$  erhöht, bis ein ebenfalls zur Laufzeit definierter Maximalwert  $m$  erreicht ist. Die Zähler repräsentieren dadurch die Nutzungsfrequenz der Cacheblöcke. Somit ist eine LFU-Strategie möglich.

Dieses Vorgehen hat allerdings noch den Nachteil, dass beim Suchen eines zu ersetzenden Blockes alle Zähler um den Wert des kleinsten Zählers im Set verringert werden. Sollten in einem Set z.B. nur stark genutzte Blöcke vorliegen, ständen alle Zähler auf dem Maximalwert. Wenn nun für einen neuen Block, der eventuell sogar nur ein einziges Mal benötigt wird, Platz im Set gesucht wird, würden alle Zähler auf Null gesetzt werden, wodurch die

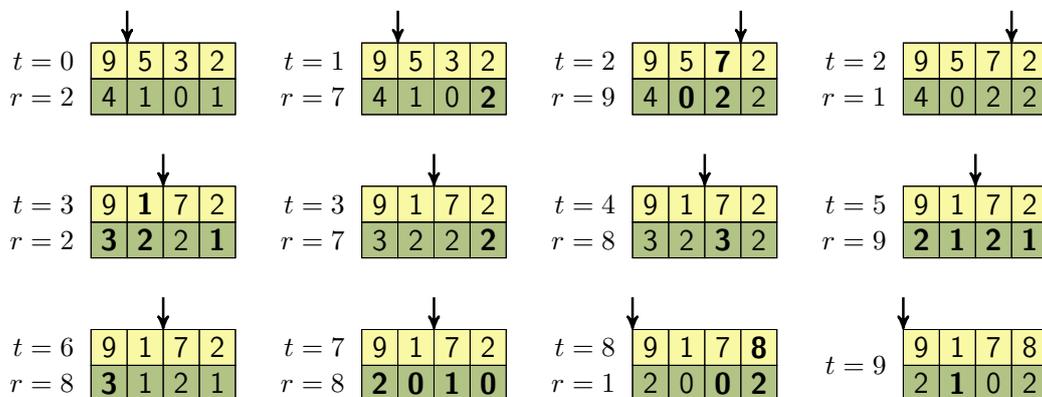


Abbildung 5.3: Verhalten des Caches für den Zugriffsvektor  $\{2, 7, 9, 1, 2, 7, 8, 9, 8, 8, 1\}$  mit  $s = 1$ ,  $m = 4$  und  $i = 1$

Information über die Nutzungsfrequenz der Blöcke verloren ginge. Um dies zu vermeiden, läuft der Zeiger höchstens einmal durch das Set. Dies wirkt dem soeben beschriebenen Problem entgegen, da die Zeiger pro Versuch des Einlagerns um maximal Eins reduziert werden. Blöcke, die neu hinzukommen sollen und eine noch höhere Nutzungsfrequenz als die eingelagerten besitzen, würden nach kurzer Zeit jedoch trotzdem in das Set gelangen, da die Frequenz des Dekrementierens größer wäre als die des Inkrementierens. Der Algorithmus ist in Abbildung 5.3 mit dem Zugriffsvektor  $\{2, 7, 9, 1, 2, 7, 8, 9, 8, 8, 1\}$  für einen Startwert von  $s = 1$ , einem Maximalwert von  $m = 4$  und einem Inkrement von  $i = 1$  dargestellt. Der zum jeweiligen Zeitpunkt im Cache gehaltene Block ist gelb hinterlegt und der momentane Zählerstand grün. Die Iterationslänge von maximal der Setlänge ist zu den Zeitpunkten  $t = 5$  und  $t = 7$  zu erkennen.

Mit dieser Architektur lässt sich durch den Parameter  $i$  die Tendenz Richtung LRU bzw. LFU bestimmen. Je größer dieser Wert ist, desto näher liegt der Algorithmus an der LRU Strategie, da der Zähler quasi als Positionsplatz für die nächste Verdrängung gilt, denn je größer der Zählerwert, desto kleiner ist die Wahrscheinlichkeit in naher Zukunft verdrängt zu werden.

### 5.2.2 Schreibstrategie

Es gibt zwei grundsätzliche Schreibstrategien für Caches. Zum einen existiert die Write-Through-Strategie, die gleichbedeutend mit einem reinen lesenden Cache ist, und zum anderen die Write-Back-Strategie, die gleichbedeutend mit einem lesenden und schreibenden Cache ist. Die beiden Strategien unterscheiden sich im Verhalten bei einem Schreibvorgang. Erstere leitet den Schreibvorgang direkt an das gecachte Gerät weiter und markiert die Cacheseite als ungültig. Die andere schreibt die Daten nur auf das Cachegerät, wodurch die gültigen Daten auf beiden Geräten verteilt werden. Welche der beiden Strategien besser ist, hängt vom jeweiligen Anwendungsfall ab, da aber, wie in Abschnitt 3.4 angesprochen wurde, das intensive Schreiben für SSDs ungünstig ist, soll eine dritte Strategie als Mittelweg eingesetzt werden – *Write-Hybrid* genannt. Sie arbeitet in der Art, dass nur Blöcke, die bereits im Cache liegen, nach der Write-Back-Strategie geschrieben werden. Blöcke, die neu eingelagert werden müssten, werden nach der Write-Through-Strategie geschrieben. Die Idee, die hinter dieser Vorgehensweise steht, ist, dass Daten, die neu erstellt werden, meist sequentiell geschrieben werden. Da Festplatten beim sequentiellen Lesen und Schreiben kaum Geschwindigkeitsunterschiede aufweisen, ist es möglich, Daten ohne größeren Geschwindigkeitsverlust auf die Festplatte zu schreiben. Da außerdem davon ausgegangen werden kann, dass die Blöcke bei einem nicht komplett gefüllten Quellmedium zuvor lange nicht genutzt worden sind, sind diese auch nicht im Cache enthalten.



# Kapitel 6

## Implementierung

In diesem Kapitel soll die konkrete Implementierung der im vorangegangenen Kapitel gezeigten Architektur detailliert erklärt werden. Für die Implementierung wird als Grundlage die von Hensbergen und Zhao [14] entwickelte Implementierung genutzt. Die Funktionsweise und wichtige Implementierungsdetails werden in Abschnitt 6.1 dargestellt. Da die Implementierung jedoch nicht die für diese Arbeit geforderten Eigenschaften erfüllt, wurden mehrere Änderungen und Erweiterungen vorgenommen. Zunächst wurden die Debugmöglichkeiten erweitert. Dies wird in Abschnitt 6.2 genauer beschrieben. Des Weiteren wird die Implementierung des oben beschriebenen Cache-Algorithmus in Abschnitt 6.3 dargestellt. Ferner wies das ursprüngliche Modul starke Defizite in Bezug auf den Speicherverbrauch auf. Aus diesem Grund wurden hier Optimierungen vorgenommen, die in Abschnitt 6.4 genauer betrachtet werden. Auf Grund dieser Änderungen und weiterer Defizite bei der Metadatenicherung, musste auch diese verändert werden. Diese Veränderungen werden in Abschnitt 6.5 beschrieben. Weiterhin wurde im Rahmen dieser Arbeit auch die Nutzung des TRIM-Kommandos, welches in Abschnitt 2.2 vorgestellt wurde, berücksichtigt. Die Änderungen, die hierfür am Kernelmodul vorgenommen wurden, werden in Abschnitt 6.6 beleuchtet. Das ursprüngliche Modul wies außerdem keinerlei Möglichkeit auf von einem gecachten Medium zu booten. Aus diesem Grund musste die Funktion auch von Grund auf neu implementiert werden. Das Vorgehen hierbei wird abschließend in Abschnitt 6.7 erklärt.

### 6.1 Grundlage: dm-cache

Als Grundlage für die Implementierung des Caches dient das von Hensbergen und Zhao entworfene Kernelmodul – dm-cache genannt. Dieses nutzt den DM und entspricht bereits im Wesentlichen der in dieser Arbeit vorgeschlagenen High-Level-Architektur. In diesem Abschnitt soll die Funktionsweise dieses Moduls genauer beleuchtet werden. Die Lebenszeit eines konkreten Caches lässt sich hierbei in drei Phasen teilen:

#### 6.1.1 Phase 1: Konstruktion

Die erste Phase ist das Erstellen des Caches. Hierbei wird die Metadatenstruktur des Caches (vgl. Listing A.2 im Anhang) an sich angelegt. Dasselbe gilt für die Metadaten der einzelnen Cacheblöcke (vgl. Listing A.3 im Anhang). Das Erstellen geschieht durch

einen Aufruf aus der Nutzerebene, für gewöhnlich unter Zuhilfenahme der DM-Tools. Der konkrete Befehl sieht dabei wie folgt aus, wobei beim letzten Parameter 0 für einen Write-Through-Cache und 1 für einen Write-Back-Cache steht:

```
echo [Startblock] [Endblock] cache [Quellgerät] [Cachegerät] 0
[Cacheblockgröße] [Cacheblöcke] [Assoziativität] [0|1] | dmsetup create [Name]
```

Die Konstruktorroutine nimmt dabei die Daten entgegen, fügt die Metainformationen entsprechend in die Datenstruktur ein und erzeugt somit einen neuen leeren Cache. Es ist jedoch auch möglich, einen Cache aus einer vorangegangenen Sitzung wiederherzustellen. Der entsprechende Befehl lautet wie folgt:

```
echo [Startblock] [Endblock] cache [Quellgerät] [Cachegerät] 1 | dmsetup create
[Name]
```

Dieser Befehl veranlasst die Konstruktorroutine, vom Ende des Cache-Blockgerätes eine gespeicherte Metadatenstruktur zu lesen, anstatt Standardwerte zu benutzen. Diese ist durch eine Checksumme geschützt, so dass das Laden ungültiger Informationen verhindert wird. Parallel zu den Vorgängen im dm-cache-Modul wird vom DM-Framework ein neues Blockgerät angelegt, welches das gecachte Blockgerät repräsentiert und vom Nutzer bzw. Betriebssystem wie jedes andere Blockgerät genutzt werden kann.

### 6.1.2 Phase 2: Lebenszeit

Nach der Konstruktion bzw. Rekonstruktion des Caches kann dieser mit der eigentlichen Arbeit beginnen. Dabei ist die Arbeit wie im High-Level-Blockdiagramm vorgesehen, in zwei asynchron zueinander arbeitende Programmteile aufgeteilt.

#### Teil I - Systemaufruf

Der erste Programmteil wird immer aktiv, wenn auf das Blockgerät zugegriffen wird und Daten geschrieben oder gelesen werden. Dies geschieht für gewöhnlich durch eine Dateioperation im Nutzerraum. Er arbeitet also synchron zu Systemaufrufen. Dabei wird vom DM-Framework der `map`-Funktion eine BIO-Anfrage übergeben. Diese überprüft nun, ob die gewünschten Blöcke im Cache vorliegen und sich nicht gerade in der Bearbeitung durch den Kernelthread im zweiten Programmteil befinden. Falls das der Fall ist, werden das Blockgerät und die Blocknummer in der BIO-Anfrage entsprechend angepasst und durch den entsprechenden Rückgabewert der `map`-Funktion dem DM-Framework die Fertigstellung der BIO-Manipulation mitgeteilt. Das DM-Framework leitet sodann die Anfrage an das entsprechende Gerät weiter und schließt die Anfrage ab. Sollte der Cacheblock gerade in Bearbeitung durch den Kernelthread sein, wird die Anfrage in einer Liste in den Metadaten des Blockes vermerkt und dem Framework mitgeteilt, so dass das Modul die weitere Verarbeitung veranlassen wird. Sollte der entsprechende Block nicht im Cache vorrätig sein, wird ein geeigneter Cacheblock nach dem LRU-Algorithmus gesucht, um die Daten einzulagern. Sollte kein geeigneter Block gefunden werden, was immer dann der Fall ist, wenn alle Blöcke des Sets, in dem die entsprechenden Daten eingelagert werden müssten, in Bearbeitung durch den zweiten Programmteil sind, wird die Anfrage analog zum Vorgehen von oben auf das Quellgerät umgeleitet. Beim Auffinden eines passenden

Blockes könnte dieser bei einem Write-Through-Cache verändert worden sein. Sollte dies der Fall sein, wird das Zurückschreiben des Blockes veranlasst und weiter vorgegangen, wie gerade beschrieben. Anderenfalls wird dieser markiert, dass er in Bearbeitung durch den Kernelthread ist und der BIO-Auftrag in die Auftragsliste des zweiten Programmteils eingefügt. Dieser wird dabei in eine entsprechende Datenstruktur A.4 verpackt, die neben der Referenz auf die BIO-Struktur z.B. eine Referenz auf die Metadaten des Caches und den zu nutzenden Cacheblock hat.

## Teil II - Kernelthread

Der zweite Programmteil arbeitet asynchron in einem Kernelthread und ist, wie im High-Level-Blockdiagramm zu sehen, für das Ein- und Auslagern von Cacheblöcken zuständig. Die Aufträge werden der Reihe nach aus der Auftragsliste entnommen. Sollte es sich um eine lesende Anfrage handeln, werden zunächst die Daten vom Quellgerät in den Speicher kopiert. Dies passiert wiederum asynchron. Nach Abschluss des Vorgangs wird durch Aufruf der zuvor übergebenen Callbackmethode das weitere Vorgehen in Gang gesetzt. Zunächst wird dem Kernel mitgeteilt, dass die BIO-Anfrage abgeschlossen ist, so dass der anfragende Programmfluss fortgesetzt werden kann. Anschließend werden die Daten wiederum asynchron auf das Cachegerät geschrieben. Nach dem erneuten Aufrufen der Callbackmethode wird abschließend in dem Metadaten des Cachesblockes vermerkt, dass der betreffende Cacheblock nicht länger in Bearbeitung ist.

### 6.1.3 Phase 3: Destruktion

In der dritten und letzten Phase werden zunächst die als `dirty` markierten Blöcke auf das Quellegehäät zurückgeschrieben. Anschließend werden die Metadaten des Caches an das Ende des Cachegerätes geschrieben. Hierbei werden lediglich die Blocknummern des gecachten Gerätes gespeichert, zu dem der jeweilige Cacheblock gehört, jedoch nicht der Zeitstempel der logischen Uhr. Die Destruktion kann wiederum unter Zuhilfenahme der DM-Tools vom Nutzer durch folgenden Befehl veranlasst werden:

```
dmsetup remove [Name]
```

Parallel hierzu entfernt das DM-Framework das logische Blockgeräät, welches das gecachete Geräät repräsentiert.

## 6.2 Debuginterface

Die gegebene Implementierung des Caches bietet leider nur unzureichende Möglichkeiten der Fehlersuche. Es wurden von den Autoren lediglich Kernellog-Nachrichten als Debugmöglichkeit vorgesehen. Da es aber in einem Cache mehrere Millionen Cacheblöcke und noch mehr Zugriffe während eines Testlaufes geben kann, sind diese nur bedingt geeignet. Zum einen werden die Lognachrichten in sehr kurzen Abständen ausgegeben, so dass ein Log-Daemon gar nicht in der Lage ist, alle Nachrichten zu verarbeiten. Zum anderen kann die Datenmenge der Logmeldungen schnell auf mehrere hundert Megabyte anwachsen, so dass selbst der Puffer für Kernellog-Nachrichten ältere Einträge fallen lassen muss. Aus diesem Grund wurden zwei weitere Debugmöglichkeiten implementiert:

### 6.2.1 message-Funktion

Die erste Möglichkeit eines erweiterten Debuggings liefert die `message`-Funktion des DM-Frameworks, jedoch ist sie nicht darauf limitiert. Die Funktion dient dazu, Nachrichten an eine bestimmte Instanz eines DM-Geräts zu schicken. Diese kann dazu genutzt werden, Einstellungen zur Laufzeit vorzunehmen bzw. zu verändern. Der Zugang aus dem Nutzerraum ist wiederum mit den DM-Tools unter Verwendung folgenden Befehls möglich:

```
dmsetup message [Name] [Blocknummer] [Funktionsnummer] {Argumente}
```

Im Rahmen dieser Arbeit wurde zunächst die `message`-Funktion dahingehend implementiert, dass sie unter anderem weitere Debugmöglichkeiten bietet. Im weiteren Verlauf wurde die `message`-Funktion so erweitert, dass bestimmte Cacheparameter zur Laufzeit angepasst werden können. Die konkret implementierten Funktionen sind in Tabelle 6.1 aufgeführt.

Tabelle 6.1: Funktionen des `message`-Interfaces

Funktionsnummer	Argumente	Funktion
<b>Laufzeitfunktionen</b>		
0	–	Zurückschreiben geänderter Blöcke Ein-/Ausschalten (toggle)
<b>Debugfunktionen</b>		
1	–	alle Cacheblock-Metadaten ausgeben
2	[Set]	Cacheblock-Metadaten des übergebenen Sets ausgeben
3	[Block]	Ausgabe des Sets, in das der Block fällt
4	–	Ausgabe der Anzahl der Cacheblöcke in jedem Zustand
5	–	Zurücksetzen der Statistik
6	–	die Konsistenz des Caches prüfen

### 6.2.2 debugfs-Interface und Visualizer

Das `debugfs`-Dateisystem wurde entworfen, um Informationen zwischen einem Kernelmodul und dem Nutzerraum unter Zuhilfenahme des Dateisystems auszutauschen. Die genaue Funktionsweise wird bei [7] erläutert und soll hier nicht näher diskutiert werden.

Das Cachemodul wurde um eine `debugfs`-Schnittstelle erweitert, um die Metadaten des Caches und der Cacheblöcke auf einfache Weise in den Nutzerraum zu übertragen. Dies ist insbesondere bei den Metadaten der Cacheblöcke nur in binärer Form effizient möglich, da die Datenmenge auf Grund der Anzahl der Blöcke ansonsten zu groß wäre.

Weil die Metadaten zunächst nur in binärer Form vorliegen, musste ein Weg gefunden werden, diese auszuwerten. Auf Grund der Menge der Informationen bietet sich hierfür eine Visualisierung an, die die Daten so aufbereitet, dass sie übersichtlich bleiben. Da jedoch die Auswertung wegen der Datenmenge rechenintensiv ist, wurde es vom Testrechner auf den Entwicklungsrechner verlagert. Hierfür werden die Daten zwischen Test- und Entwicklungsrechner per Ethernet ausgetauscht. Der Sachverhalt ist in Abbildung 6.1 dargestellt:

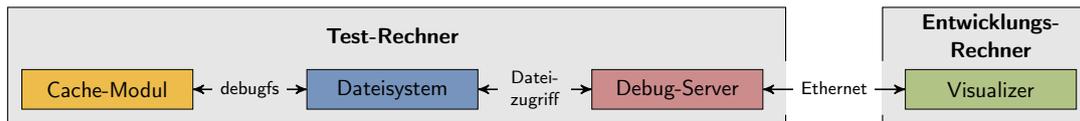


Abbildung 6.1: Verbindungspfad zwischen Cache-Modul und Visualizer

Der Visualizer wertet auf dem Entwicklungsrechner die Daten aus und stellt sie grafisch dar. Dabei werden die Cacheblöcke als farbige Quadrate dargestellt. Die Farbe gibt Auskunft über den Zustand des jeweiligen Cacheblocks, wodurch schnell ein Überblick über die Zustände vieler Cacheblöcke verschafft werden kann. Durch Klicken auf einen Block können schließlich genauere Informationen über diesen erhalten werden. Die Oberfläche des Visualizers ist in Abbildung 6.2 zu sehen.

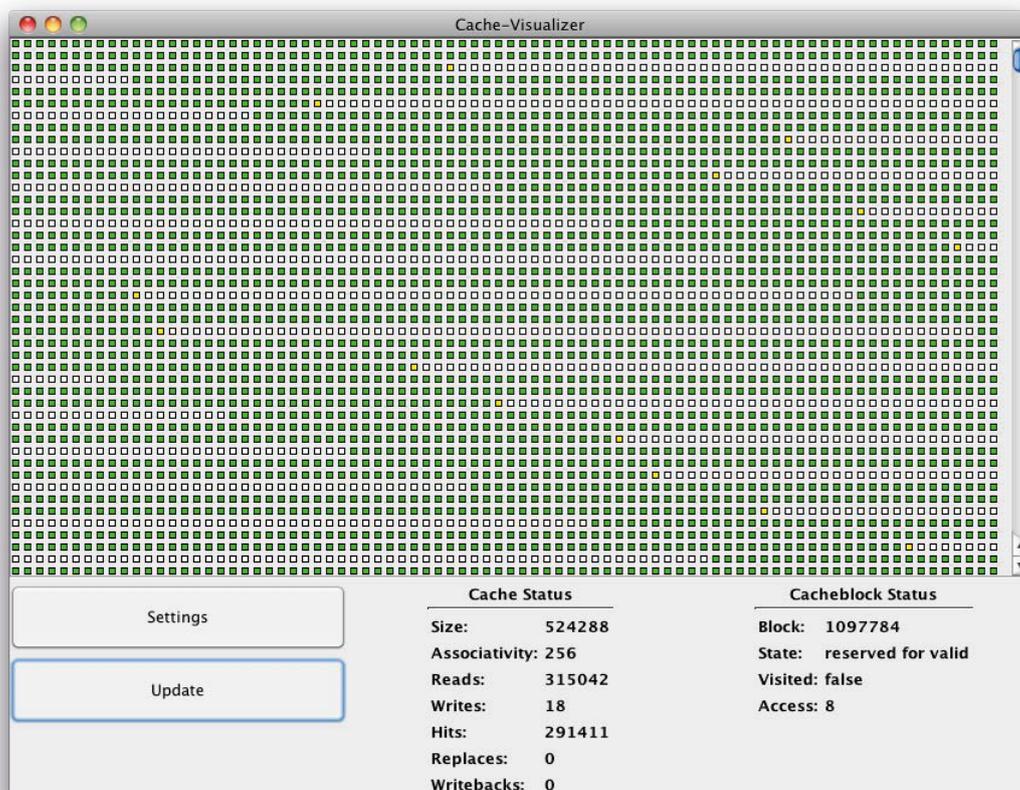


Abbildung 6.2: Fenster des Visualizers mit aktivem Cache

## 6.3 Cache-Algorithmus

Für den in Abschnitt 5.2.1 vorgestellten Verdrängungsalgorithmus mussten die Cacheblock-Metadatenstruktur zunächst um ein Feld für den Zähler erweitert werden und die Cache-metadatenstruktur um einen Zähler je Set. Des Weiteren musste die Konstruktorroutine

angepasst werden, so dass die Informationen über die Werte  $i$ ,  $m$  und  $s$  zur Laufzeit übergeben werden können. Außerdem wurde der Parameter für die Schreibstrategie erweitert, um das in Kapitel 5.2.1 beschriebene Verfahren zu unterstützen. Dies hat zur Folge, dass sich der Aufruf für das Erstellen eines Caches wie folgt erweitert:

```
echo [Startblock] [Endblock] cache [Quellgerät] [Cachegerät] 0
[Cacheblockgröße] [Cacheblöcke] [Assoziativität] [0|1|2] [s] [m] [i] | dmsetup
create [Name]
```

Weiterhin wurde die Art der Speicherung der Cacheblockzustände geändert. In der ursprünglichen Implementierung wurden Statusbits genutzt um einen Zustand zu bestimmen. Dieses führte dazu, dass für das Speichern der Zustände mindestens<sup>6</sup> vier Bit benötigt wurden. Zudem machte es den Quelltext unübersichtlich, da auf den ersten Blick nicht zwingend klar war, in welchem Zustand ein Block beim Setzen eines Bits gebracht wurde bzw. schon war. Dies ist gerade in Anbetracht der Synchronisation der beiden großen Programmeile kritisch. Aus diesem Grund wurden die Statusinformationen auf eindeutige Zustände abgebildet und in eine leichter zu verstehende Zustandsmaschine überführt (Abbildungen 6.3(a) und 6.3(b)). Für die bessere Verständlichkeit werden an dieser Stelle Write-Through- und Write-Back-Cache getrennt dargestellt. Die komplette Zustandsmaschine des Caches findet sich im Anhang unter Abbildung A.1. Die Zustandsmaschine für den Write-Hybride-Cache entspricht der des Write-Back-Caches, wobei der Zustand *Reserved for Dirty* und die Übergänge dorthin bzw. von dort abgehend entfallen.

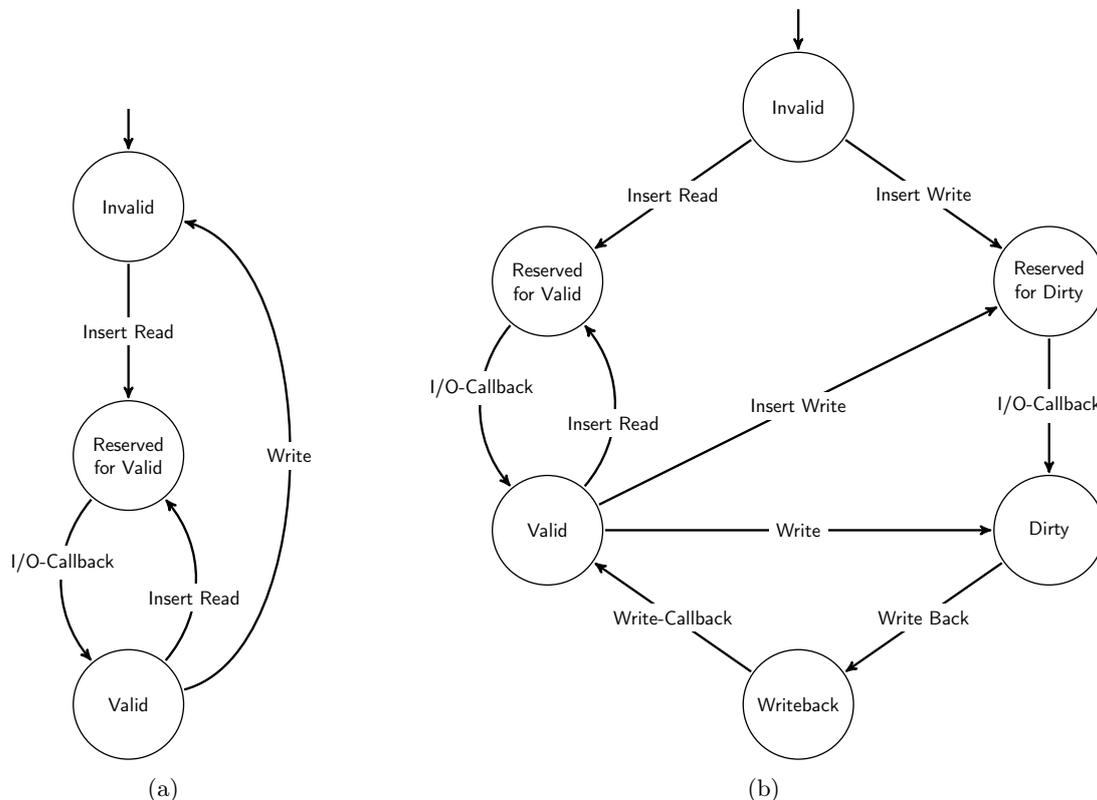


Abbildung 6.3: Zustände und Zustandsübergänge der Cacheblöcke – in (a) für die Write-Through-Konfiguration und in (b) für die Write-Back-Konfiguration

<sup>6</sup> Es wurde in der konkreten Implementierung eine 16 Bit Variable benutzt.

## 6.4 Optimierung der Cacheblock-Metadaten

Um einen möglichst geringen Speicherverbrauch zu erreichen, mussten die Metadaten der Cacheblöcke optimiert werden. Diese enthielten in der ursprünglichen Implementierung jeweils eine Liste, um die Anfragen aufzunehmen, die während der Bearbeitung eines Cacheblockes ankommen und nicht sofort bearbeitet werden können. Da diese Listen von den beiden zueinander asynchron laufenden Programmteilen genutzt wurden, mussten die Zugriffe synchronisiert werden. Aus diesem Grund gab es für jeden Cacheblock eine Schlossvariable. Dies bedeutete einen Speicherverbrauch von 38 Byte pro Cacheblock. Welche Auswirkung diese Größe auf den Gesamtspeicherverbrauch des Caches hat wird in Abschnitt 7.3 ausführlich betrachtet.

Erste Tests ergaben jedoch, dass diese Listen relativ selten genutzt wurden. Aus diesem Grund wurden sie durch eine gemeinsame Liste pro Cacheblock ersetzt, welche in den Gesamtcache-Metadaten untergebracht wurden. Dies benötigte mehrere Anpassungen im Quelltext. Zunächst wurde der Callback-Funktion für das Zurückschreiben von Cacheblöcken auf das Quellgerät lediglich eine Referenz auf den geschriebenen Cacheblock übergeben. Das war ausreichend, da der Cacheblock alle Referenzen auf die zu verändernden Daten enthielt. Durch das Verschieben der Liste für ausstehende BIO-Anfragen musste die übergebene Datenstruktur angepasst werden, da diese Referenz nicht mehr in den Metadaten enthalten war. Hierfür wurden die Cacheblock-Metadaten in eine erweiterte Datenstruktur verpackt, wie sie in Listing 6.1 zu sehen ist.

```

1 /* Cach block with advanced metadata */
2 struct adv_cacheblock {
3     struct cache_c *dmc;
4     struct cacheblock *cacheblock;
5 };

```

Listing 6.1: Metadatenstruktur für die Übergabe an die Write-Back-Callback-Funktion

Da es ineffizient wäre, alle BIO-Aufträge lediglich hintereinander in eine Liste einzufügen und beim Callback des Kopiervorgangs die zu einem Cacheblock gehörenden BIOS herauszusuchen, sind die Einträge in der neuen Auftragsliste nach Cacheblöcken sortiert. Aus dieser Anforderung ergab sich die Listenform in Listing 6.2. Um nun nicht bei jedem Callback die Liste durchsuchen zu müssen, auch wenn keine ausstehenden Anfragen für einen Cacheblock vorliegen, wurde ein zusätzliches Statusbit in die Cacheblock-Metadaten eingefügt. Dieses zeigt an, ob Aufträge für den Cacheblock vorliegen oder nicht. Wenn dies nicht der Fall ist, braucht die Liste nicht durchsucht zu werden.

```

1 /* list for queued BIOSs waiting for running copy */
2 struct queued_bios {
3     struct list_head list; /* list anchor */
4     struct cacheblock *block; /* cacheblock the bios belong to */
5     struct bio_list *bios; /* the bio list */
6 };

```

Listing 6.2: Listenstruktur für ausstehende BIO-Aufträge

Durch die vorherigen Änderungen war es möglich, die Cacheblock-Metadatenstruktur im Hinblick auf den Speicherverbrauch zu optimieren. Mit der bisher verwendeten 64-Bit langen Blocknummervariablen lassen sich bis zu  $2^{64}$  Blöcke bzw. 16PB bei einer Blockgröße von 512 Byte adressieren. Da diese Datenmenge in absehbarer Zeit nicht auf einem Gerät zur Verfügung stehen wird, wurden einige Bits dieser Statusvariablen für die zusätzlichen

Informationen verwendet. Die genaue Aufteilung der 64-Bit Variablen ist in Abbildung 6.4 zu sehen. Die verbleibenden 56-Bit reichen jedoch immer noch für 64EB Daten zu 512 Byte Blöcken. Da heutige Festplatten eine Kapazität von maximal 2TB aufweisen, ist es unwahrscheinlich, dass das Modul an seine Grenzen stößt. Das Modul kann jedoch bei Bedarf auf größere Blocknummern umgestellt werden. So ist es in Zukunft möglich, größere Blockgeräte zu unterstützen. Dabei dürfte auch der wachsende Speicherverbrauch unkritisch sein, da anzunehmen ist, dass die Größe des Arbeitsspeichers parallel zur Festplattenkapazität wächst.



Abbildung 6.4: Aufbau der Cacheblock-Metadaten für das neue Kernelmodul

Die konkrete Implementierung der Datenstruktur (Listing 6.3) zeigt, dass diese leicht erweiterbar ist. Der Zähler für den Ersetzungsalgorithmus ist mit einer Länge von vier Bit an dieser Stelle sehr willkürlich gewählt und könnte auf Kosten der maximal adressierbaren Blöcke problemlos erweitert werden, sollte sich dies zu einem späteren Zeitpunkt als Vorteilhaft erweisen.

```

1 /* Cache block metadata structure */
2 struct cacheblock {
3     sector_t block    : 56; /* Sector number of the cached block */
4     sector_t access  : 4; /* Counter for replacement algorithm */
5     sector_t state   : 3; /* State of a block */
6     sector_t visited : 1; /* Pending bios in bios_list */
7 };

```

Listing 6.3: Optimierte Datenstruktur für Cacheblock-Metainformationen

## 6.5 Anpassung der Metadatensicherung

Auf Grund der Änderung der Cacheblock-Metadatenstruktur musste auch die Sicherung dieser Daten angepasst werden. Die Daten werden prinzipiell am Ende des Cachegeräts gespeichert. Somit muss dieses nicht nur so groß sein wie der Cache, sondern um die Größe der Metadaten größer. Die ursprüngliche Implementierung sorgte dafür, dass vor dem Speichern der Metadaten alle Cacheblöcke, die sich im `dirty` Zustand befanden, auf das Quellgerät zurückgeschrieben wurden. Aus diesem Grund befanden sich beim Sichern alle Cacheblöcke im `valid` oder `invalid` Zustand. Beim Sichern wurde lediglich der Status der Cacheblöcke und die bei der Konstruktion übergebenen Parameter gesichert. Die logischen Zeitstempel wurden verworfen. Des Weiteren wurden die Informationen in einem Puffer kopiert, da nur Datenblöcke mit einer Größe, die ein Vielfaches von 512 Byte sind, auf das Blockgerät geschrieben werden können.

Dieses Vorgehen wurde dahingehend geändert, dass nun sämtliche Metadaten der Cacheblöcke gesichert werden. Zusätzlich mussten im Gegensatz zu oben die Zeiger für das Iterieren über den Sets gesichert werden. Es wird außerdem auf die Nutzung eines Puffers verzichtet, was das Schreiben und Wiederherstellen der Daten ein wenig beschleunigt, da auf Kopiervorgänge im Speicher verzichtet werden kann. Dies bedeutet zwar ggf. einen

Verschnitt, da der Speicherblock für die Metadaten auf ein Vielfaches von 512 Byte aufgerundet werden muss. Da es jedoch nur zwei Speicherblöcke gibt (Cacheblock-Metadaten und Iterierungszeiger), kann der Verschnitt nicht mehr als ein Kilobyte betragen.

Es wurde des Weiteren die Fehlererkennung bei der Wiederherstellung des Caches verbessert. Bei der ursprünglichen Implementierung wurden die gesicherten Metadaten lediglich gelesen und wiederhergestellt. Dies führte dazu, dass bei einem Absturz des Systems und anschließendem Neustart die veralteten Metadaten wiederhergestellt wurden. Dadurch kam es zu einem inkonsistenten Cache und im ungünstigsten Fall zum Datenverlust bzw. Absturz des Systems. Aus diesem Grund wurde die Wiederherstellung verändert in der Weise, dass der ausgelesenen Cachemetadatenblock modifiziert wurde. Es ist ein Flag eingefügt worden, welches nach dem Auslesen geändert und verändert zurück geschrieben wird. Dadurch war es möglich zu erkennen, ob die gelesenen Metadaten zu den auf dem Cachegerät gespeicherten Datenblöcken passen. Sollte dies nicht der Fall gewesen sein, war es weiterhin dennoch möglich den Cache neu aufzubauen, sofern es sich um einen Write-Through-Cache handelte. Bei einem Write-Back- bzw. Write-Hybrid-Cache kann auch durch dieses Verfahren der Cache nicht wiederhergestellt oder neu aufgebaut werden, weil durch den Systemabsturz die Metainformationen über die Cacheblöcke verloren gegangen sind und im Nachhinein nicht mehr festgestellt werden kann, welche Cacheblöcke ausschließlich auf dem Cacheblockgerät aktuell sind und stattdessen auf dem Quellgerät genutzt werden müssten.

## 6.6 TRIM Unterstützung

Das TRIM-Kommando wurde bereits in Kapitel 2.2 erläutert. Es wurde ausgeführt, dass eine explizite Unterstützung in das Dateisystem eingebaut werden muss. Bei der Implementierung des Caches war es jedoch möglich, für einen bestimmten Fall diese Unterstützung auch unabhängig vom Dateisystem anzubieten. Wie in Abbildung 6.3(a) dargestellt, existiert beim Write-Through-Cache ein Übergang in den `invalid` Zustand. An diesem Punkt ist die Nutzung des TRIM-Kommandos sinnvoll, da der Inhalt des Cacheblockes nicht mehr benötigt wird und dies somit auch dem SSD mitgeteilt werden kann. Deshalb wird bei der Änderung des Cacheblock-Zustandes das TRIM-Kommando an das SSD übermittelt. Dieses Verhalten ist zwar per Compiler-Makro ein- und ausschaltbar, jedoch sollte es selbst bei SSDs ohne TRIM-Unterstützung zu keinem Leistungsverlust führen, da der Funktionsaufruf zum Löschen asynchron ist (siehe [6]) und sofort returnieren würde.

Das TRIM-Kommando fand jedoch auch Anwendung bei der Erstellung des Caches. Wie ebenfalls in Kapitel 2.2 angesprochen wurde, nimmt die Leistung eines SSD über einen längeren Nutzungszeitraum ab, da zu wenige freie Blöcke zur Verfügung stehen. Dieses Szenario könnte vielleicht bereits beim Neuaufbau eines Caches gegeben sein. Aus diesem Grund erschien es sinnvoll, vor der Nutzung den für den Cache vorgesehenen Datenbereich mit dem TRIM-Kommando zu löschen. Dies ist nicht nur dahingehend sinnvoll, dass das SSD bereits vor der Verwendung als Cache anderweitig genutzt wurde, sondern auch für das Szenario, dass bei jedem Bootvorgang der Cache neu erstellt wird.

## 6.7 Bootfähigkeit des Cachemediums

Das Booten eines Linuxsystems von einem Blockgerät setzt voraus, dass dieses beim Starten des `init`-Programms zur Verfügung steht. Da das `dm-cache`-Modul in der ursprüngli-

chen Implementierung eine Konfiguration aus dem Nutzerraum heraus erfordert, ist diese Voraussetzung nicht gegeben. Um nun das Booten trotzdem zu ermöglichen gibt es zwei verschiedene Möglichkeiten. Die Erste nutzt das *initramfs*. Da die Erklärung der genauen Funktionsweise des *initramfs* den Rahmen dieser Arbeit sprengen würde, wird auf [18] verwiesen. Durch das *initramfs* ist es möglich zunächst ein gecachetes Blockgerät aus dem Nutzerraum zu erstellen und das Wurzeldateisystem drauf umzulenken.

Die zweite Möglichkeit besteht darin, die nötige Funktionalität für das Erstellen eines DM-Blockgeräts direkt im Kernel fest zu implementieren, so dass beim Übergang zum *init*-Programm das gecachte Blockgerät bereits existiert. Im offiziellen Entwicklungszweig des Linux-Kernels ist die Funktion zum Erstellen von DM-Geräte direkt aus dem Kernel heraus nicht vorgesehen. Es existiert jedoch bei [13] ein Patch, der diese Funktion in den Kernel einfügt. Durch diesen Patch ist es möglich, dem Kernel durch den Bootloader zusätzliche Parameter zu übergeben, um ein DM-Gerät während des Bootens zu erzeugen. Der zusätzliche Parameter sieht dabei wie folgt aus:

```
dm="[name] [uuid] [ro], [parameter]"
```

Das DM-Gerät wird zunächst unter dem Gerätepfad `/dev/dm-*` zur Verfügung gestellt. Hierbei steht das Sternchen für eine Nummerierung, welche bei null beginnt. Dem Kernel muss somit das entsprechende Gerät im `root`-Parameter übergeben werden. Bei der Nutzung von *udev* wird das Gerät beim Booten zu `/dev/mapper/[name]` umbenannt, wobei `name` dem obigen im Kernelparameter entspricht. Weiterhin steht `uuid` für den Universally Unique Identifier (UUID), den das Gerät erhalten soll. Auf die Funktionsweise des UUID soll in dieser Arbeit nicht genauer eingegangen werden, aber es wird hierfür auf [19] verwiesen. Soll dem Gerät kein UUID zugewiesen werden, ist in das Feld `none` einzutragen. Der Parameter `ro` steht für die Einstellung, ob vom DM-Geräte lediglich gelesen oder sowohl gelesen als auch geschrieben werden kann. Die darauf folgenden Parameter entsprechen im wesentlichen den an `dmsetup` übergebenen Parametern zum Erstellen eines DM-Geräts. Somit würde der Kernelparameter für ein gecachetes Blockgerät, welches unter `/dev/mapper/cached` zur Verfügung gestellt wird, wie folgt aussehen:

```
dm="cached none 0, [Startblock] [Endblock] cache [Quellgerät] [Cachegerät] 0
[Cacheblockgröße] [Cacheblöcke] [Assoziativität] [0|1|2] [s] [m] [i]"
```

Dabei wird ein neuer Cache erstellt. Um einen bestehenden Cache wiederherzustellen, muss der Kernel-Parameter wie folgt lauten:

```
dm="cached none 0, [Startblock] [Endblock] cache [Quellgerät] [Cachegerät] 1"
```

Mit dem bestehenden Patch funktionierte das Erstellen des gecachten Mediums und Booten an sich problemlos. Der Patch hatte jedoch das Defizit, dass die Destruktionsroutine des Caches beim Herunterfahren des Systems nicht aufgerufen wurde. Ebenfalls war es nicht möglich die Destruktionsroutine aus dem Nutzerraum während des Herunterfahrens aufzurufen, da das gecachte Gerät als Rootdateisystem eingebunden war. Dies zog einen Verlust der Metadaten nach sich und verhinderte das erneute Booten. Somit war der Cache in diesem Zustand als Rootdateisystem nicht nutzbar. Es musste ein Weg gefunden werden, durch den beim Herunterfahren bzw. Neustart des Systems die Metadaten gesichert werden können. Hierbei gab es wiederum zwei Möglichkeiten. Zum einen eine Erweiterung

des DM-Frameworks und zum anderen die Erweiterung des Cachemoduls. Auf Grund der beschränkten Bearbeitungszeit wurde auf die allgemeinere Implementierung im DM verzichtet und statt dessen das Cachemodul entsprechend erweitert.

Zunächst musste das Cachemodul in der Art erweitert werden, dass es über das bevorstehende Herunterfahren informiert wird. Hierfür wurden Linux *Notifiers* (siehe [8]) genutzt. Diese ermöglichen es, im Kernel einen Funktionsaufruf zu registrieren, welcher beim Herunterfahren bzw. Neustart des Systems ausgeführt wird. Die hierbei übergebene Funktion wurde in der Art gestaltet, dass sie über alle verbliebenen Caches iteriert und jeweils die Destrukturfunktion aufruft. Die Erweiterung ist in Listing 6.4 zu sehen.

Wie in den Zeilen 7ff zu erkennen ist, bedurfte es einer Liste aller verbleibenden Caches. Hierfür musste das Kernelmodul wiederum erweitert werden, da das DM-Framework keine Funktionalität vorsieht, welche das Abfragen nach DM-Geräten eines bestimmten Typs ermöglicht. Aus diesem Grund wurde die Cachemetadatenstruktur zu einer Kernel-Liste erweitert (vgl. Zeile 7 Listing A.5 im Anhang). Vom Modul selbst wird nun eine Liste aller Cachemetadatenstrukturen verwaltet. Hierbei wird beim Erstellen eines Caches die entsprechende Struktur zur Liste hinzugefügt und beim Entfernen des Caches die Struktur ebenfalls aus der Liste entfernt. Mit dieser Liste ist es nun möglich, bei der Benachrichtigung über das Herunterfahren die Metadaten aller Caches zu sichern.

```
1 static int cache_notify_reboot(struct notifier_block *this, unsigned long code, void *x){
2     struct list_head *tmp;
3
4     if ((code == SYS_DOWN) || (code == SYS_HALT) || (code == SYS_POWER_OFF)) {
5         DMINFO("dumping all metadata");
6
7         /* iterate through all remaining caches and destroy them */
8         list_for_each(tmp, &_caches){
9             dump_metadata(list_entry(tmp, struct cache_c, list));
10        }
11    }
12    return NOTIFY_DONE;
13 }
14
15 static struct notifier_block cache_notifier = {
16     .notifier_call = cache_notify_reboot,
17     .next          = NULL,
18     .priority      = INT_MAX,
19 };
```

Listing 6.4: Erweiterung für das automatische Entfernen des Caches beim Herunterfahren



# Kapitel 7

## Auswertung

Nachdem in den beiden vorangegangenen Kapiteln die Architektur und die Implementierung des entwickelten Kernelmoduls thematisiert wurden, sollen nun die damit erzielten Ergebnisse diskutiert werden. Dabei wird zunächst in Abschnitt 7.1 ein synthetisches Test-szenario betrachtet. Im darauf folgenden Abschnitt 7.2 wird ein reales Szenario ausgewertet. Abschließend wird in Abschnitt 7.3 sowohl die Rechenlast als auch der Speicherverbrauch des Moduls untersucht und diskutiert. Für die Tests wurden, sofern nicht anders angegeben, als Festplatte eine Western Digital Raptor mit 73,4GB Kapazität [35] und als SSD eine Intel X25-E mit 32GB Kapazität [16] genutzt.

### 7.1 Synthetische Leistungsbewertung

Die Leistung des implementierten Kernelmoduls wird in diesem Kapitel anhand eines eigens entworfenen Testszenarios ermittelt. Hierfür wird im ersten Unterabschnitt zunächst das Testszenario genauer beschrieben. Anschließend werden in Unterabschnitt 7.1.2 die mit Hilfe des Szenarios gewonnen Testergebnisse vorgestellt und diese abschließend in Unterabschnitt 7.1.3 diskutiert.

#### 7.1.1 Testszenario

Das synthetische Testszenario, das im Rahmen dieser Arbeit entwickelt wurde, musste zunächst die Hauptanforderung der Vergleichbarkeit der Systemkonfigurationen erfüllen. Hierfür mussten die Simulationen wiederholbar sein. Neben dieser Hauptanforderung musste es zusätzlich verschiedene Nebenanforderungen erfüllen. Die Erste besteht darin, die besonderen Eigenschaften eines Caches überhaupt nutzen zu können. Ein gecachtes Medium kann gegenüber einem ungecachten Medium prinzipbedingt nur einen Geschwindigkeitsvorteil realisieren, wenn auf dieselben Daten mehrfach zugegriffen werden kann. Aus diesem Grund sind die meisten gebräuchlichen Festplattenbenchmarks bereits ungeeignet. Des Weiteren stellen diese Benchmarks meist ein synthetisches Testszenario dar, welches ebenfalls für diese Arbeit nicht gewünscht war.

Neben den genannten synthetischen Benchmarks wurde eine andere Möglichkeit in den Arbeiten genutzt, die in Kapitel 3 vorgestellt wurde und vergleichbare Szenarien benötigt. Diese besteht darin, zunächst die Anfragen eines realen Systems an ein Blockgerät aufzuzeichnen. Für den eigentlichen Benchmark können anschließend die Anfragen an die zu

testenden Geräte gestellt und die Verarbeitungszeiten verglichen werden. Dieses Testszenario hat jedoch den Nachteil, sehr zeitintensiv zu sein, denn zunächst müssen die Daten auf einem Produktivsystem gesammelt werden. Um einen repräsentativen Datensatz zu erhalten müsste dies über mehrere Tage oder gar Wochen geschehen. Anschließend wären vergleichbare zeitliche Dimensionen nötig, um einen Testlauf durchzuführen. Da im Rahmen dieser Arbeit mehrere Konfigurationen getestet werden sollten, könnte sich der Zeitbedarf leicht auf Monate summieren. Außerdem würde der Aufbau der nötigen Infrastruktur zur Aufzeichnung von Datensätzen mit einem nicht vernachlässigbaren Aufwand verbunden sein. Aus diesem Grund konnte dieses Testszenario für diese Arbeit ebenfalls nicht verwendet werden.

Aus den genannten Gründen ist als Kompromiss ein Zwischenweg aus den beiden genannten Möglichkeiten gewählt worden. Es wurde der synthetische Benchmark *bonnie++* [3] hierfür angepasst. *Bonnie++* ist ein synthetischer, auf Dateisystemebene arbeitender, nicht destruktiver<sup>7</sup> Benchmark, welcher Daten auf das zu testende Gerät schreibt und geordnet und ungeordnet ausliest. Dabei werden die benötigte Zeit und die Prozessorlast gemessen. Da jedoch bei jedem Testlauf die Dateien neu generiert werden mussten, konnten mit der ursprünglichen Implementierung die Eigenschaften eines Caches nicht genutzt werden. Aus diesem Grund wurde das Programm im Rahmen dieser Arbeit erweitert, so dass es eine Dateiliste einlesen kann und Lese-/Schreibvorgänge entsprechend dieser Liste durchführt. Dabei besteht die Dateiliste aus mehrere Zeilen, die wie folgt aufgebaut sind:

```
[Dateiname], [Dateigröße], [c|s|d]
```

Beim Ausführen von *bonnie++* wird die Liste eingelesen und je nach dem letzten Parameter die Datei mit der gegebenen Größe erstellt (c), gelesen (s) oder gelöscht (d). Mit Hilfe dieser Modifikation ließ sich nun ein realitätsnahes Testszenario aufbauen. Um ein reales Desktopsystem zu simulieren, musste zunächst die Installation und der Bootvorgang simuliert werden. Hierfür wurde als erstes die Datei- und Verzeichnisstruktur eines Ubuntu-Linux-Systems extrahiert und eine Dateiliste mit dem oben gezeigten Format erstellt, die ihrerseits für das Erstellen von Dateien sorgt. So kann die Installation des Systems simuliert werden. Die Liste beinhaltet hierbei knapp 121000 Dateien mit einer Gesamtgröße von ca. 3,6GB. Als zweiter Schritt wurde die Menge der gelesenen Daten beim Booten des realen Linux-Systems gemessen. Diese lag bei ca. 150MB. Anschließend wurden aus der Dateiliste des Linux-Systems zufällige Dateien gewählt, die eine Gesamtgröße von 150MB hatten, und damit eine *bonnie++* Dateiliste erstellt, die das Lesen dieser Dateien durch *bonnie++* veranlasst. Diese Dateiliste wurde einmalig erstellt und während der Tests nicht verändert. Mit der Liste ließ sich das Booten des Systems und das Starten der Programme annähernd genau simulieren.

Abschließend musste eine Möglichkeit gefunden werden, das Erstellen, Lesen und Löschen von Nutzerdaten zu simulieren. Hierfür wurden Datenlisten erstellt, die Angaben zu Dateien mit einer Gesamtgröße von 100MB enthielten. In den Listen überwog dabei die Anzahl von kleinen Dateien stark. Diese Listen wurden ebenfalls einmalig erzeugt und während der Tests nicht mehr verändert. Bei der Simulation wurden nach jedem simulierten Bootvorgang eine gewisse Anzahl  $r$  dieser Sets gelesen,  $c$  Sets geschrieben und  $d$  wieder gelöscht. Dabei wurde die Reihenfolge, welche Sets gelesen, geschrieben oder gelöscht werden, einmalig festgelegt und ebenfalls während aller Tests nicht verändert. Dieses Vorgehen sollte nun zusammen mit dem Bootvorgang einen Arbeitstag simulieren.

<sup>7</sup> Bereits bestehende Daten auf dem Datenträger bleiben erhalten.

Ausnahmen von dem soeben beschriebenen Vorgehen bildeten die ersten Bootvorgänge einer jeden Simulation. Hierbei wurden jeweils doppelt so viele Sets wie gewöhnlich geschrieben, bis eine Maximalanzahl  $M$  von Sets auf der Festplatte vorhanden waren. Beim Lesen wurden alle Sets berücksichtigt, solange die Anzahl von Sets auf den Datenträgern den Wert  $r$  nicht überstiegen. Des Weiteren wurde beim Erstellen der Erstellungs-, Lese- und Löschenreihenfolge der Sets darauf geachtet, dass immer ein Set existierte, welches an jedem Arbeitstag gelesen wurde. Dieses sollte regelmäßig genutzte Daten simulieren. Die Reihenfolge der Arbeitstage wurde ebenfalls im Vorhinein festgelegt und während der verschiedenen Simulationsläufe nicht verändert. Somit soll an dieser Stelle noch einmal betont werden, dass alle zufälligen Festlegungen vor dem Beginn der Evaluierung stattfanden.

Bei den konkreten Tests auf einem System galt es jedoch noch eine weitere Funktion des Linux-Systems zu beachten – nämlich den Datencache im Arbeitsspeicher. Dieser verzerrte die Messergebnisse des Benchmarkprogramms, da das Betriebssystem den Abschluss von Dateioperationen meldet, obwohl diese noch nicht auf dem Blockgerät abgeschlossen sind. Weiterhin konnten durch den Arbeitsspeichercache Daten von mehreren Arbeitstagen gehalten werden, so dass z.B. die Daten des Bootens nicht zwingend vom Blockgerät gelesen wurden sondern aus dem Arbeitsspeichercache. Erste Tests zeigten, dass die Verzerrungen durch diesen Mechanismus enorm waren und ein Vergleich der verschiedenen zu testenden Konfigurationen nicht möglich war. Zwar war es möglich, die Funktion abzuschalten, was jedoch zu erheblichen Leistungsverlusten führte, die die Messungen wiederum unrealistisch machten. Das Problem der Übertragung von Daten von einem Bootvorgang zum Nächsten durch den Arbeitsspeichercache ließ sich durch das Neueinhängen des Blockgerätes in das Dateisystem lösen, da dies das Linux-System dazu veranlasste den Arbeitsspeichercache zu leeren. Jedoch waren die von *bonnie++* gelieferten Messwerte bezüglich der Leserate ungenau. Um dies Problem zu lösen, wurde als Maßstab für die Leistung die Zeit zwischen Ein- und Aushängen des Blockgerätes genutzt.

### 7.1.2 Ergebnisse

Beim Erstellen einer konkreten Testumgebung sind weitere Dinge zu beachten. Zum einen sollte die gelesene Datenmenge größer sein als der Cache. Ist dies nicht der Fall, können sämtliche Daten im Cache gehalten werden und es sind keine Rückschlüsse auf die Effizienz des Verdrängungsalgorithmuses möglich. Zum anderen ist die Laufzeit des Tests zu beachten. Wenn von einem momentan üblichen SSD und somit auch einer Cachegröße von 32GB ausgegangen wird, würde dies bedeuten, dass pro simuliertem Arbeitstag mehr als diese 32GB gelesen werden sollten. Das würde zunächst im optimalen Fall bei einer Leserate von 250MB/s und 50 simulierten Arbeitstagen eine Testlaufzeit von ca. zwei Stunden bedeuten. Weiterhin bietet der implementierte Cache sehr viele Freiheitsgrade. Das sind die Assoziativität, die Cacheblockgröße und die Cacheparameter  $m$ ,  $s$  und  $i$ . Beides in Kombination würde Simulationszeiten von mehreren Monaten bedeuten, sofern sämtliche Konfigurationskombinationen getestet werden sollten. Dies war im Zeitrahmen dieser Arbeit jedoch nicht möglich. Aus diesem Grund mussten sowohl Einschränkungen bei der Testumgebung hingenommen werden als auch bei der Anzahl der durchgeführten Simulationen.

Zunächst wurde die Größe des Caches künstlich auf zwei Gigabyte beschränkt. Weiterhin wurde darauf verzichtet, an jedem simulierten Arbeitstag mehr Daten zu lesen, als der Cache halten kann. Durch eine ausreichend große Anzahl an verschiedenen Datensets auf dem Quellmedium und das Löschen und Schreiben von Sets wurde jedoch sicherge-

stellt, dass die Datenmenge über mehrere simulierte Tage verteilt größer ist als der Cache. Konkret bedeutet dies Werte von  $M = 20$ ,  $r = 9$ ,  $c = 1$  und  $d = 1$ .

Außerdem musste bei den Cacheparametern stark selektiert werden. Dies war notwendig, da die Simulationszeiten starken Schwankungen unterlagen. Beim Erstellen der Vergleichswerte von der genutzten Festplatte und dem SSD kam es zu Abweichungen von bis zu 7,3% bzw. 9,8% vom Mittelwert bei Festplatte bzw. SSD. Dies bedeutete, dass jede Konfiguration mehrfach getestet werden musste, um Ausreißer zu vermeiden. Daher wurde zunächst die Blockgröße auf vier Kilobyte festgelegt. Dies ist die kleinstmögliche sinnvolle Größe, da sie der Blockgröße des verwendeten Third Extended Filesystem (ext3) entspricht. Weiterhin wurde der Cacheparameter  $m$  auf die Werte  $\{4; 8; 16\}$  beschränkt und die Werte  $i$  und  $s$  auf  $\{1; m\}$ . Der Wert von  $i = 1$  bringt hierbei den Cache näher an einen LFU-Cache und der Wert  $i = m$  näher an einen LRU-Cache. In Abbildung 7.1 sind die Ergebnisse für  $m = 16$  dargestellt, wobei für die Assoziativität die Werte  $\{2; 4; 8; 16; 32; 128; 512; 2048; 8192; 32768; 131072\}$  genutzt wurden. Für die Werte  $m = 4$  und  $m = 8$  konnte die Simulation nur einmal durchgeführt werden, wodurch die Ergebnisse sehr mit Ausreißern behaftet sind. Aus diesem Grund finden sie sich im Anhang unter A.2 und A.3. Wie aus den Diagrammen ersichtlich ist, war es lediglich möglich ein spezifisches SSD zu nutzen, da sich ansonsten der Testaufwand wiederum verdoppelt hätte.

Die Tests konzentrierten sich im weiteren Versuchsaufbau auf die Write-Through-Konfiguration. Dies liegt zum einen darin begründet, dass die beiden anderen Konfigurationen nicht im Zeitrahmen dieser Arbeit stabil implementiert werden konnten. So kam es zu sporadischen Synchronisationsproblemen, welche zukünftig noch genauer analysiert werden müssen. Zum anderen konnten bei stichprobenartigen Tests der Write-Hybrid-Strategie nur unwesentliche Leistungsunterschiede zur Write-Through-Strategie festgestellt werden.

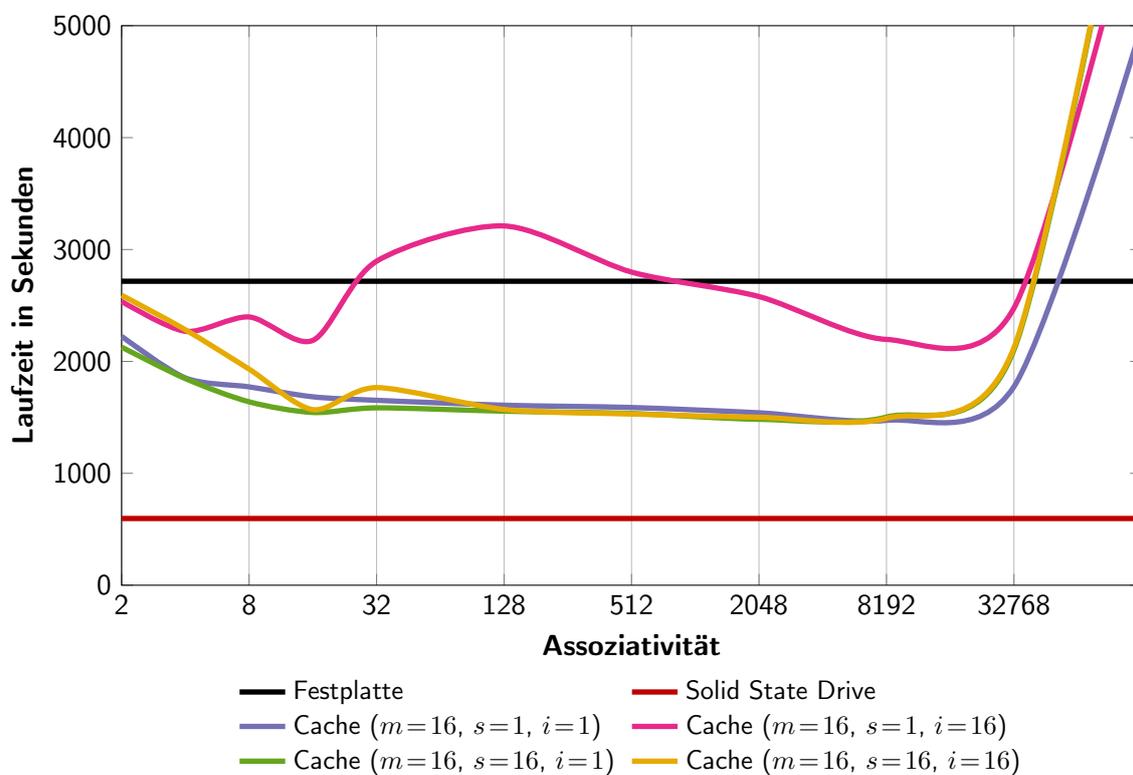


Abbildung 7.1: Benchmarkergebnisse des Caches für  $m = 16$

Die Write-Through Strategie lieferte bei der Simulation sehr schlechte Ergebnisse. Die Laufzeit der Simulationen lag bei allen getesteten Konfigurationen um mehr als das Doppelte über denen der Festplatte. Die genauen Ursachen hierfür werden im Diskussionsteil näher ergründet.

Neben dem Vergleich mit den ungecachten Medien sollte der implementierte Cache zusätzlich mit dem von ZFS verglichen werden. Hierfür wurde die Testumgebung zunächst für OpenSolaris portiert, da für Linux keine ausgereifte ZFS Implementierung existiert. Anschließend wurden erneut Messwerte für Festplatte und SSD ermittelt, da ein anderes Dateisystem neben der Nutzung eines Caches auch hier eine andere Leistung aufweisen könnte. Wie jedoch in Abbildung 7.2 zu sehen ist, war dies nicht der Fall. Anschließend wurde der Test für einen durch das SSD gecachten Pool wiederholt. Dieses Ergebnis findet sich ebenfalls in Abbildung 7.2 mit dem Messwert für eine Assoziativität von 2048 aus der obigen Messung.

Die oben beschriebenen Messungen wurden mit dem sehr performanten Intel-SSD durchgeführt. Bei der Simulation mit einem midrange SSD (OCZ Vertex Series 32GB [23]) kam es zu massiven Leistungseinbrüchen. Die Ursache hierfür war in der Größe der Cacheblockgröße zu finden, welche ungünstig für das spezifische SSD war. Das Diagramm in Abbildung 7.3 zeigt die Leistung des midrange SSD in Abhängigkeit von der genutzten Cacheblockgröße. Die Gründe für die Leistungsunterschiede sollen im nun folgenden Unterabschnitt 7.1.3 diskutiert werden.

Es konnten im Rahmen dieser Arbeit keine fundierten Messwerte ermittelt werden bezüglich des Nutzens des TRIM-Kommandos. Dies lag daran, dass lediglich ein SSD zur Verfügung stand, die diese Funktion unterstützt. Bedauerlicherweise konnten mit ihr jedoch keine verwertbaren Messwerte bezüglich der Cache-Leistung gewonnen werden, so dass auch kein Vergleich zwischen ein- und ausgeschalteter TRIM-Unterstützung ermittelt werden konnte.

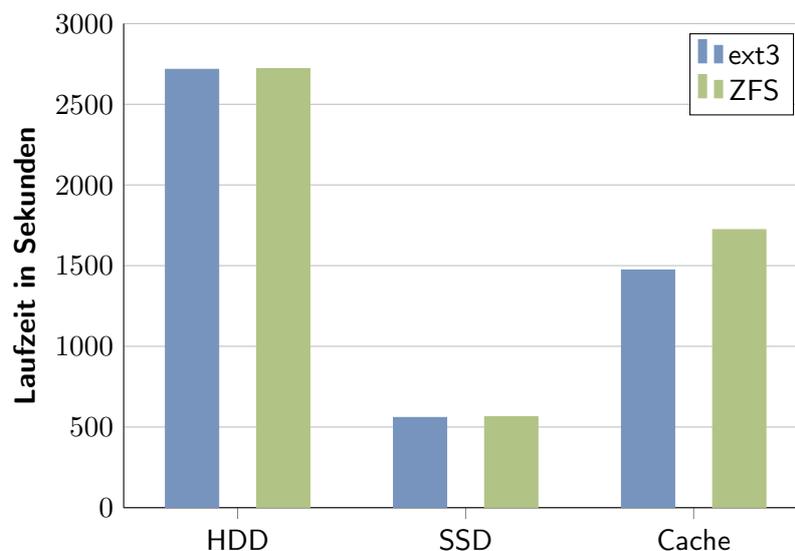


Abbildung 7.2: Vergleich zwischen dem implementierten Cache und ZFS

### 7.1.3 Diskussion

Die Ursachen für die Leistungsunterschiede, die im vorangegangenen Abschnitt aufgezeigt wurden, lassen sich über die verschiedenen internen Blockgrößen und Controller der SSDs erklären. Abbildung 7.4 zeigt die Leistung verschiedener SSDs in Abhängigkeit von der angeforderten Blockgröße. Es ist klar zu erkennen, dass die Leistung des SSDs unter einer individuellen Blockgröße stark sinkt. Dies bedeutet, dass bei der Wahl der Cacheblockgröße dieser Sachverhalt zu beachten ist. Allerdings kann die Cacheblockgröße nicht ohne weiteres an die native Blockgröße des jeweiligen SSDs angepasst werden. Die Vergrößerung der Cacheblockgröße über das der Dateisystemblockgröße hinaus hat den Nachteil, dass ggf. Daten im Cache liegen, die nicht benötigt werden. Sollte die Cacheblockgröße z.B. 256kB betragen und eine Datei von vier Kilobyte oft genutzt werden, würde zunächst für diese Datei ein 256kB Block im Cache eingelagert werden. Im ungünstigsten Fall werden die restlichen 252kB Daten extrem selten bis gar nicht genutzt. Dies könnte z.B. bei Sicherungskopien der Fall sein. Also würden über 98% der Daten in diesem Block unnötig vorgehalten werden. Wenn dieses Verhältnis auf den Rest des Caches übertragen wird, würde im Realfall dieser Wert bei weitem nicht erreicht werden, da es extrem unwahrscheinlich ist, dass die Daten auf dem Quellmedium entsprechend ungünstig verteilt sind. Es ist jedoch möglich, dass ein nicht unerheblicher Teil des Platzes im Cache blockiert wird, was zu Leistungseinbußen führt. Dies ist bei der Leistungskurve des zwei Gigabyte Caches zu beobachten. Darum sollte die individuelle Größe der Cacheblockgröße insbesondere bei kleinen Cachegeräten durch Benchmarks ermittelt werden.

Wie in Abbildung 7.1 zu erkennen ist, hängt die Leistung des Caches bei gleicher Größe in erster Linie von der Assoziativität ab. Es ist mit Ausnahme der Konfiguration  $m = 16$ ,

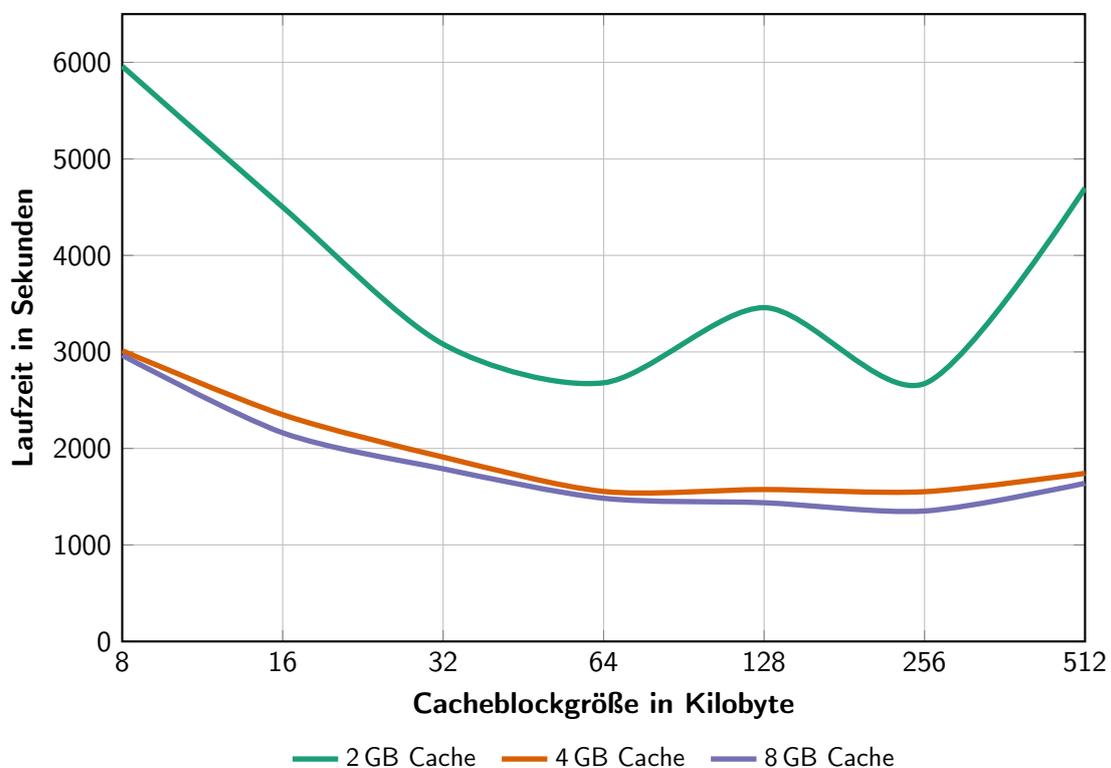


Abbildung 7.3: Benchmarkergebnisse des Caches für verschiedene Cacheblockgrößen ( $m = 16$ ,  $s = 1$ ,  $i = 1$ )

$s = 1$ ,  $i = 16$  sichtbar, dass die Leistung des Systems mit zunehmender Assoziativität steigt. Dieses Verhalten war zu erwarten, da bei einer niedrigen Assoziativität die Wahrscheinlichkeit einer Kollision steigt. Bei der Betrachtung des Extremfalls mit einer Assoziativität von eins würden alle Festplattenblöcke, die voneinander einen Abstand haben, der der Cachegröße entspricht, in den selben Cacheblock fallen. Die Wahrscheinlichkeit, dass dies passiert, ist sehr hoch und führt zu Leistungseinbußen, da ggf. leere Teile des Caches nicht genutzt werden können. Wenn nun die Assoziativität erhöht wird, können kollidierende Blöcke eher in einen leeren Bereich des Caches eingelagert werden. Dies ist bei dem anderen Extrem der Fall, bei dem die Assoziativität genauso groß ist wie der Cache, d.h. der Cache nur aus einem Set besteht. Hierbei kommt es erst zu Verdrängungen, wenn der Cache komplett gefüllt ist. Wie in Abbildung 7.1 zu erkennen ist, sinkt die Leistung des Caches ab einem bestimmten Assoziativitätsgrad wieder. Die Ursache hierfür ist die Größe der Sets. Je größer ein Set ist, umso größer ist der Aufwand, einen Block zu finden, der potentiell in diesem Set liegt. Somit lässt sich ein Tradeoff zwischen ineffizienter Verdrängung und Rechenleistung feststellen.

Das Abweichen der Konfiguration  $s = 1$  und  $i = 16$ , lässt sich nicht ohne weiteres erklären, insbesondere da die Simulationen mit  $m = 4$  und  $m = 8$  diese Anomalie bei vergleichbaren Werten für  $s$  und  $i$  nicht aufweisen. Ganz im Gegenteil, sie zeigen eine ähnliche Anomalie jedoch für andere  $s$ - und  $i$ -Werte. Es müsste zunächst durch längere Simulationen ermittelt werden, ob es sich bei den Messwerten lediglich um statistische Ausreißer handelt oder ob die Ergebnisse repräsentativ sind. Sollte dies der Fall sein, ist es nötig, dass die genauen Ursachen detailliert untersucht werden. Des Weiteren ist festzustellen, dass abgesehen von den Anomalien die Manipulation der Werte  $s$  und  $i$  keine signifikanten Unterschiede

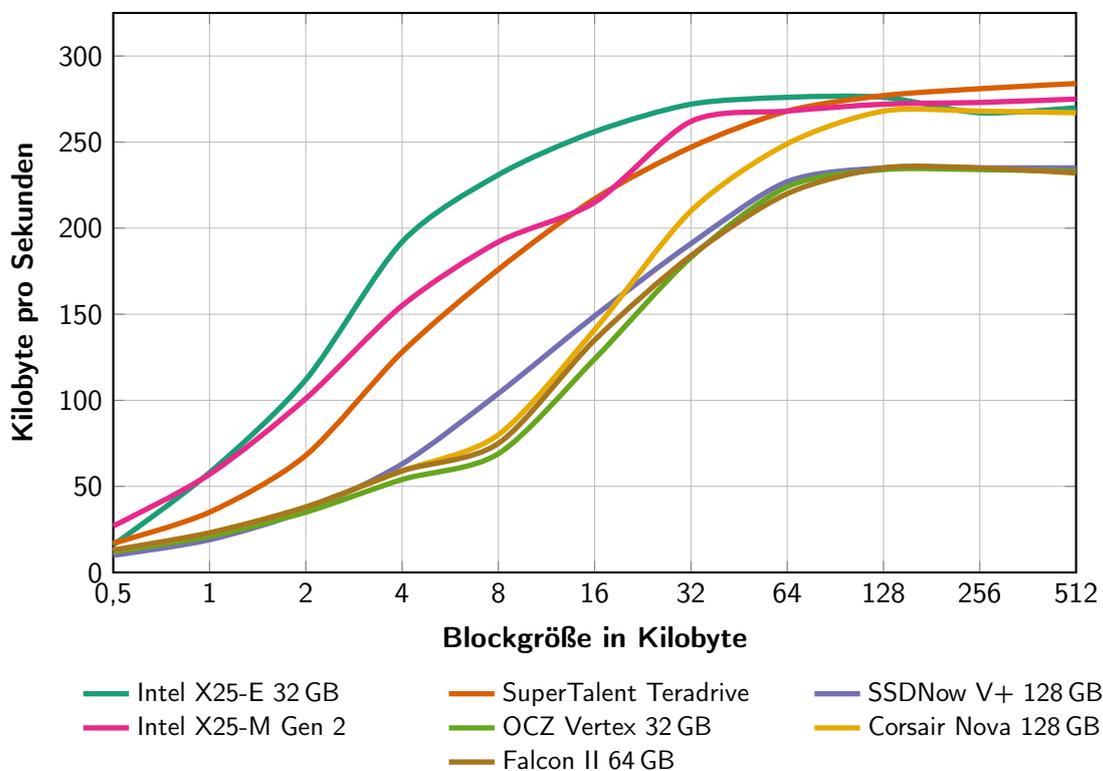


Abbildung 7.4: Datendurchsatz verschiedener SSDs in Kilobyte pro Sekunde in Abhängigkeit der Blockgröße (Quelle: Dittmar [11] und Eigenmessungen)

liefern. Die Wirkung der Änderung des Wertes  $m$  kann nicht genau bestimmt werden, da die Messwerte bei einer zu hohen Varianz zu dicht beieinander liegen.

Ferner wurde oben festgestellt, dass die Write-Back-Konfiguration eine sehr schlechte Leistung liefert. Die Ursache hierfür ist im Aufbau des Tests zu suchen; genauer gesagt beim Schreiben neuer Daten. Diese werden sequenziell auf den Datenträger geschrieben, was sowohl bei der Festplatte als auch bei dem SSD sehr schnell vor sich geht. Durch den Write-Back-Cache werden diese sequenziellen Schreibvorgänge jedoch in ungeordnete transformiert. Dies geschieht dadurch, dass ein Teil des sequenziellen Datenstroms auf das Cachegerät geschrieben wird und der andere Teil direkt auf das Quellgerät. Dies teilt den sequenziellen Datenstrom zunächst in zwei ungeordnete Datenströme auf, was direkt zu massiven Leistungseinbußen führt, da die Festplatten beim ungeordneten Schreiben signifikant langsamer werden als beim geordneten. Auch beim SSD ist ein ähnliches Verhalten zu beobachten, wenn auch bei weitem nicht im selben Ausmaß. Das Problem wird jedoch sogar weiter intensiviert, da die Wahrscheinlichkeit groß ist, dass die auf dem Cachegerät eingelagerten Daten zu einem späteren Zeitpunkt wieder auf das Quellgerät zurückgeschrieben werden müssen. Auch dies geschieht wieder ungeordnet, so dass der ursprünglich gesamte sequenzielle Datenstrom als ungeordneter Datenstrom auf die Festplatte geschrieben wird.

Abschließend gilt es noch, die Leistung des blockbasierten Caches gegenüber dem Cache des ZFS-Dateisystems zu beurteilen. Es ist in Abbildung 7.2 ganz klar und deutlich erkennbar, dass bei guter Wahl der Cacheparameter der blockbasierte Cache dem von ZFS in nichts nachsteht. Ganz im Gegenteil, er kann sogar einen kleinen Leistungsvorsprung für sich verbuchen. So kann an dieser Stelle festgehalten werden, dass ein blockbasierter Cache gegenüber einem Dateisystem basierten Cache konkurrenzfähig ist. In Anbetracht dessen, dass der blockbasierte Cache universeller einsetzbar ist, sollte die Nutzung dessen weiter erforscht werden.

## 7.2 Reale Leistungsbewertung

Nachdem im vorangegangenen Abschnitt die Leistung unter einem synthetischen Test betrachtet wurde, soll in diesem Abschnitt die Leistung unter einem realen Test bewertet werden. Erneut wird im ersten Unterabschnitt das Testscenario beschrieben, im Zweiten die Ergebnisse dargestellt und im Dritten die Ergebnisse diskutiert.

### 7.2.1 Testscenario

Als Testscenario dient der Bootvorgang eines Linux Systems. Dafür wurde die Gentoo-Distribution gewählt, da sich mit ihr der Startvorgang sehr einfach benutzerspezifisch anpassen lässt. Andere Distributionen, wie z.B. die im vorherigen Kapitel simulierte Ubuntu-Distribution, sind auf Benutzerfreundlichkeit und einfache Wartung ausgelegt. Hierbei ist eine Anpassung des Bootvorgangs relativ kompliziert, da von den Entwicklern eine weitreichende Nutzeranpassung des selbigen nicht vorgesehen ist. Bei der Gentoo-Distribution ist dies nicht der Fall, weil sie sich hauptsächlich an Experten wendet und jedes Details des Bootvorgangs einfach kontrollier- und anpassbar ist.

Für die Leistungsmessung wurde zunächst ein Standardsystem mit Gnome-Desktop installiert. Zur detaillierten Bewertung des Bootvorgangs wurde weiterhin das Evaluierungsprogramm *Bootchart* [4] installiert. Dieses misst die Zeit zwischen dem Start des

init-Programms und des Login-Bildschirms des Gnome-Desktops. Weiterhin werden Informationen über die Prozessorauslastung, Ein-/Ausgabewartezeiten, Datendurchsatz der Plattengeräte und deren Auslastung geloggt. Ferner werden die gestarteten Prozesse in Abhängigkeit voneinander dargestellt. Somit ist es möglich den Bootvorgang zu analysieren und ggf. Leistungsengpässe zu bestimmen.

### 7.2.2 Ergebnisse

Die Analyse der Bootvorgänge erfolgte für vier konkrete Szenarien. Es wurde die im letzten Abschnitt erwähnte Western Digital Raptor als Festplatte und als SSDs sowohl die OCZ SSD als auch die Intel SSD genutzt. Wobei die Ergebnisse in diesem Unterabschnitt und die Diskussion im nächsten Unterabschnitt sich auf die Ergebnisse der OCZ SSD beziehen. Die Ergebnisse der Intel SSD finden sich im Anhang in den Abbildungen A.8 bis A.10.

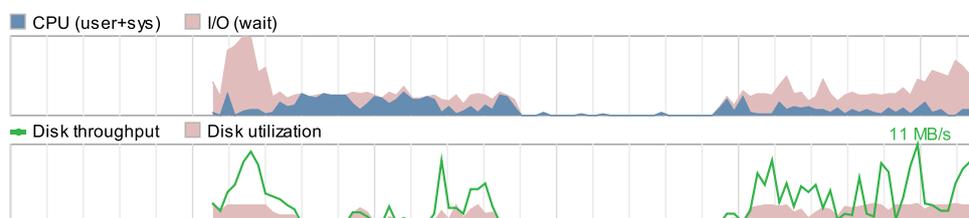


Abbildung 7.5: Leistungsdaten eines HDD-Bootvorgangs

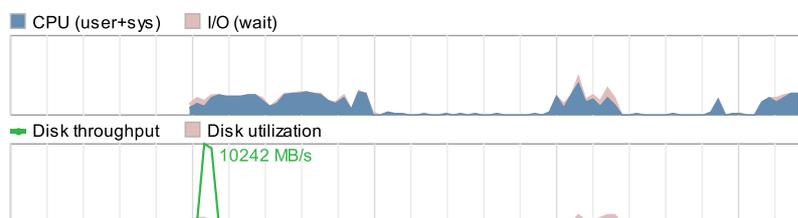


Abbildung 7.6: Leistungsdaten eines SSD-Bootvorgangs

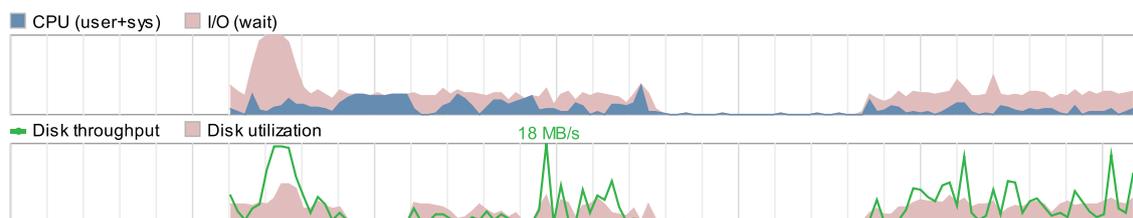


Abbildung 7.7: Leistungsdaten des ersten Cache-Bootvorgangs

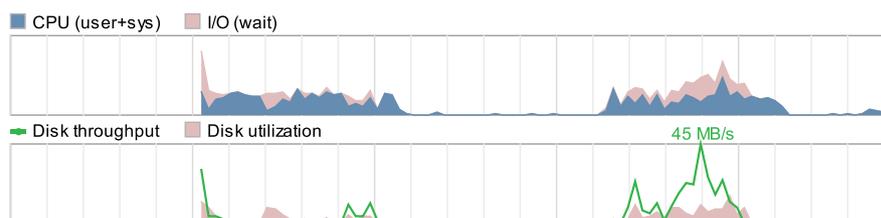


Abbildung 7.8: Leistungsdaten des zweiten Cache-Bootvorgangs

Es wurden zunächst die Daten für den Bootvorgang eines Systems mit einer herkömmlichen Festplatte ermittelt (Abbildung 7.5). Weiterhin wurden die Daten für ein identisches System mit einem SSD als Bootdatenträger ermittelt (Abbildung 7.6). Abschließend wurden die Werte für das selbe System mit der Festplatte als Quellmedium und dem SSD als Cachemedium aufgezeichnet. Dabei wurden zwei Bootvorgänge betrachtet. Zum einen der Bootvorgang beim Erstellen des Caches (Abbildung 7.7) und zum anderen der darauf folgende Bootvorgang (Abbildung 7.8). Zur besseren Übersichtlichkeit sind an dieser Stelle lediglich die Log-Ausschnitte der Prozessorlast und Ein-/Ausgabeaktivität zu sehen, wobei die Breiten der Diagramme proportional zur Bootzeit sind. Die kompletten Datensätze finden sich im Anhang in den Abbildungen A.4 bis A.7.

Des Weiteren wurden die Bootzeiten der verschiedenen Systemkonfigurationen in Abbildung 7.9 zusammengefasst. Hierbei steht „Cache 1“ für den Bootvorgang während des Aufbaus des Caches und „Cache 2“ für den darauf folgenden.

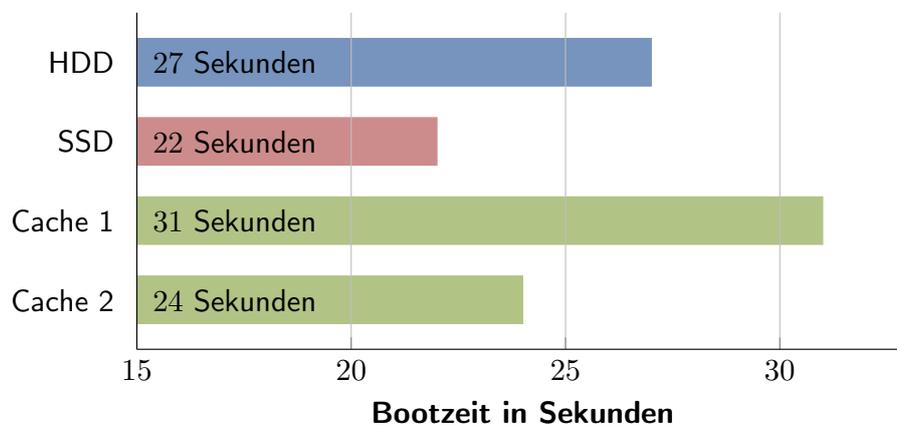


Abbildung 7.9: Bootzeiten mit HDD, SSD und Cache als Systemdatenträger

### 7.2.3 Diskussion

Der Vergleich zwischen den Ein-/Ausgabe-Wartezeiten beim Booten von der Festplatte und dem SSD zeigt, dass die Wartezeiten beim Booten von der Festplatte wesentlich größer sind als die beim SSD. Weiterhin ist ein Unterschied beim Datendurchsatz zu erkennen, jedoch muss es sich beim angegebenen Spitzendurchsatz des SSDs von zehn Gigabyte pro Sekunde um einen Auslesefehler handeln. Aus diesem Grund sind diese Werte zwischen Festplatte und SSD nicht vergleichbar. Beim Vergleich dieser Ein-/Ausgabe-Wartezeiten ist zu sehen, dass die Wartezeiten des ersten Cache-Bootvorgangs sich mit denen der Festplatte ähneln. Sie sind lediglich ein wenig länger, was durch das zusätzliche Schreiben der Bootdaten auf das Cachegerät neben dem Auslesen für den Bootvorgang erklärt werden kann. Die Wartezeiten des zweiten Bootvorgangs mit Cache ähneln eher denen des SSDs, was zu erwarten war, da nach dem ersten Bootvorgang nahezu alle Anfragen aus dem Cache bedient werden konnten.

In den Abbildungen 7.7 und 7.8 ist ebenfalls zu erkennen, dass die Prozessorlast nicht wesentlich größer ist als bei den beiden Bootvorgängen ohne Cache. Dies bestätigt somit die Messungen aus Abschnitt 7.1.

Bei der Betrachtung der Bootzeiten in Abbildung 7.9 ist zunächst eine Verlangsamung des Bootvorgangs beim Erstellen des Caches gegenüber der Festplatte zu beobachten.

Auch dieses Verhalten war zu erwarten, da keine gültigen Daten im Cache vorlagen. Somit konnte der Bootvorgang maximal so schnell sein, wie derjenige mit der Festplatte. Da jedoch neben dem reinen Auslesen der Bootdaten diese gleichzeitig auf das Quellgerät geschrieben werden mussten, fiel ein Overhead an, welcher die längere Bootzeit erklärt. Ab dem zweiten Bootvorgang mit Cache konnten die Anfragen jedoch aus dem Cache bedient werden. Somit verkürzte sich die Bootzeit um 11% gegenüber der Festplatte und liegt 9% über der des SSDs. Auch in diesem Bereich spiegeln sich somit die Messergebnisse aus Abschnitt 7.1 wieder.

## 7.3 Ressourcenverbrauch

Da die Nutzung des Caches wenig sinnvoll ist, wenn dieser viele Systemressourcen belegt und somit das System verlangsamt, soll in diesem Abschnitt der Ressourcenverbrauch näher betrachtet werden. Hierbei wird konkret die Belegung von Rechenkapazität und Speicher genau untersucht.

### 7.3.1 Ergebnisse

#### Prozessorauslastung

Ein Maß für die Auslastung wurde ebenfalls mit Hilfe des modifizierten Programmes *bonnie++* ermittelt. Neben dem Datendurchsatz misst *bonnie++* die Prozessorlast während des Tests. Für die Auswertung der Prozessorlast wurden lediglich die Daten während

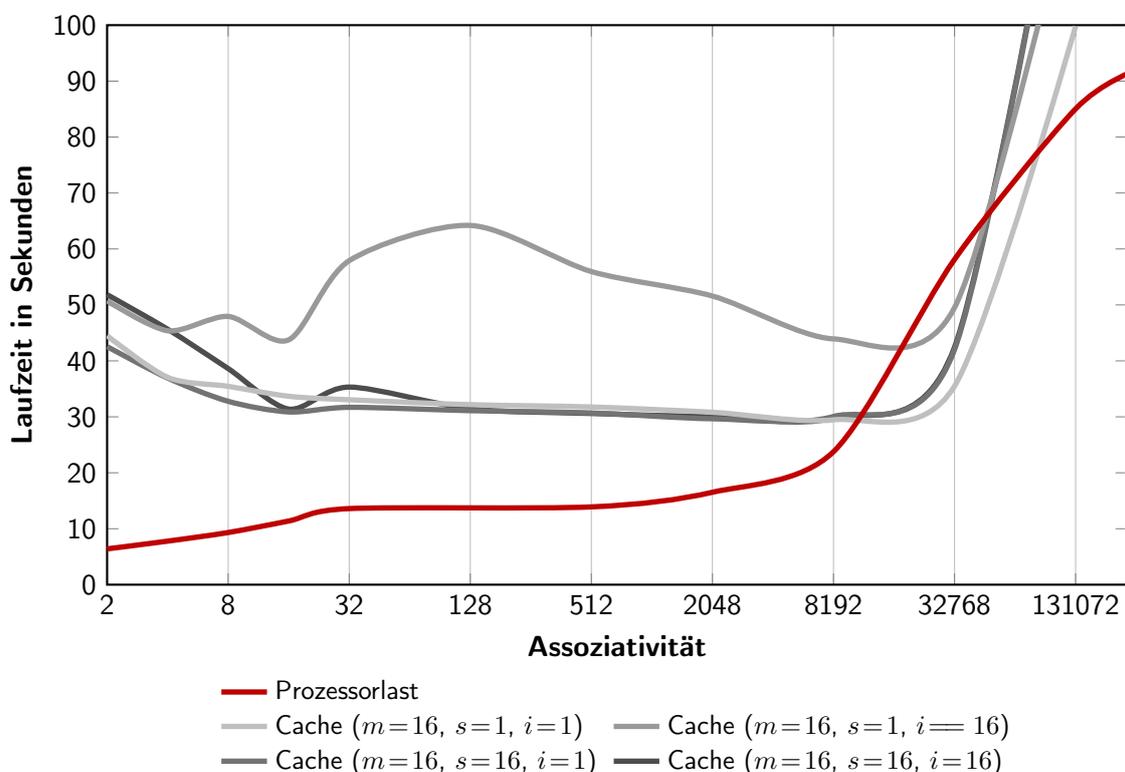


Abbildung 7.10: Prozessorlast in Abhängigkeit von der Assoziativität

des simulierten Bootvorgangs herangezogen, da diese jeweils die stärkste Last aufwiesen. Also sind die Daten der Messung eine Worst-Case-Betrachtung. Die prozentuale Prozessorlast ist im Diagramm in Abbildung 7.10 dargestellt in Abhängigkeit von der Assoziativität. Dabei ist zu beachten, dass eine Last von 100% die Auslastung einer CPU bedeutet. Bei einem 4-Kernsystem würde eine Wert von 400% für eine komplette Auslastung aller Prozessoren stehen. Die Messwerte wurden bei der Simulation mit den Parametern  $m = 16$ ,  $s = 1$  und  $i = 1$  ermittelt. Neben der Prozessorlast sind zur besseren Übersicht die Leistungsdaten des Caches aus der Abbildung 7.1 mit eingetragen. Diese wurden auf einen Wert von 50 Sekunden normiert.

## Speicher

Der Speicherverbrauch der ursprünglichen Cacheimplementierung wurde hauptsächlich von den Cacheblock-Metadaten bestimmt. Bei der neuen Implementierung spielen diese zwar weiterhin eine zentrale Rolle, jedoch wird durch die Iteratorzähler jedes Sets ggf. eine nennenswerte Menge zusätzlichen Speichers belegt je nach Anzahl der Sets. Dies wird jedoch durch die erhebliche Reduktion des Speicherverbrauchs der Cacheblock-Metadaten überkompensiert. Die Strukturen der beiden Metadaten sind in den Listings 7.1 und 7.2 noch einmal gegenübergestellt, wobei die Kommentare zu jedem Eintrag durch Größenangaben ersetzt wurden. Es sollte hierbei angemerkt werden, dass die ursprüngliche Implementierung des Caches nicht nur 38 Byte pro Cacheblock in Anspruch nimmt, sondern 48 Byte. Dies liegt daran, dass auf einem 64-Bit System jedes Element einer Datenstruktur vom GCC-Compiler an eine Adresse, die ein Vielfaches von acht ist, gelegt wird. Dies Problem hätte jedoch durch eine einfache Compileranweisung behoben werden können, weshalb im Weiteren von einer Größe von 38 Byte ausgegangen wird.

```

1 /* Cache block metadata structure */
2 struct cacheblock {
3     spinlock_t lock;          /* 4 Byte */
4     sector_t block;          /* 8 Byte */
5     unsigned short state;    /* 2 Byte */
6     unsigned long counter;   /* 8 Byte */
7     struct bio_list bios;    /* 16 Byte */
8 };

```

Listing 7.1: Gesamtgröße 38 Byte

```

1 /* Cache block metadata structure */
2 struct cacheblock {
3     sector_t block : 56;     /* 56 Bit */
4     sector_t access : 4;    /* 4 Bit */
5     sector_t state : 3;     /* 3 Bit */
6     sector_t visited : 1;   /* 1 Bit */
7 };

```

Listing 7.2: Gesamtgröße 8 Byte

Der exakte Speicherbedarf lässt sich in Abhängigkeit von Cachegröße, Blockgröße und Assoziativität berechnen. Mit der Formel 7.1 lässt sich in einem ersten Schritt die Blockanzahl des Caches berechnen:

$$\text{Blockanzahl} \cdot 2^{20} = \frac{\text{Cachegröße [MB]}}{512\text{B} \cdot \text{Cacheblockgröße}} \quad (7.1)$$

Im Weiteren lässt sich mit der Formel 7.2 der Speicherverbrauch für die Cacheblock-Metadaten der ursprünglichen Implementierung berechnen:

$$\text{Speicherverbrauch [MB]} = \text{Blockanzahl} \cdot 2^{20} \cdot 38\text{B} \quad (7.2)$$

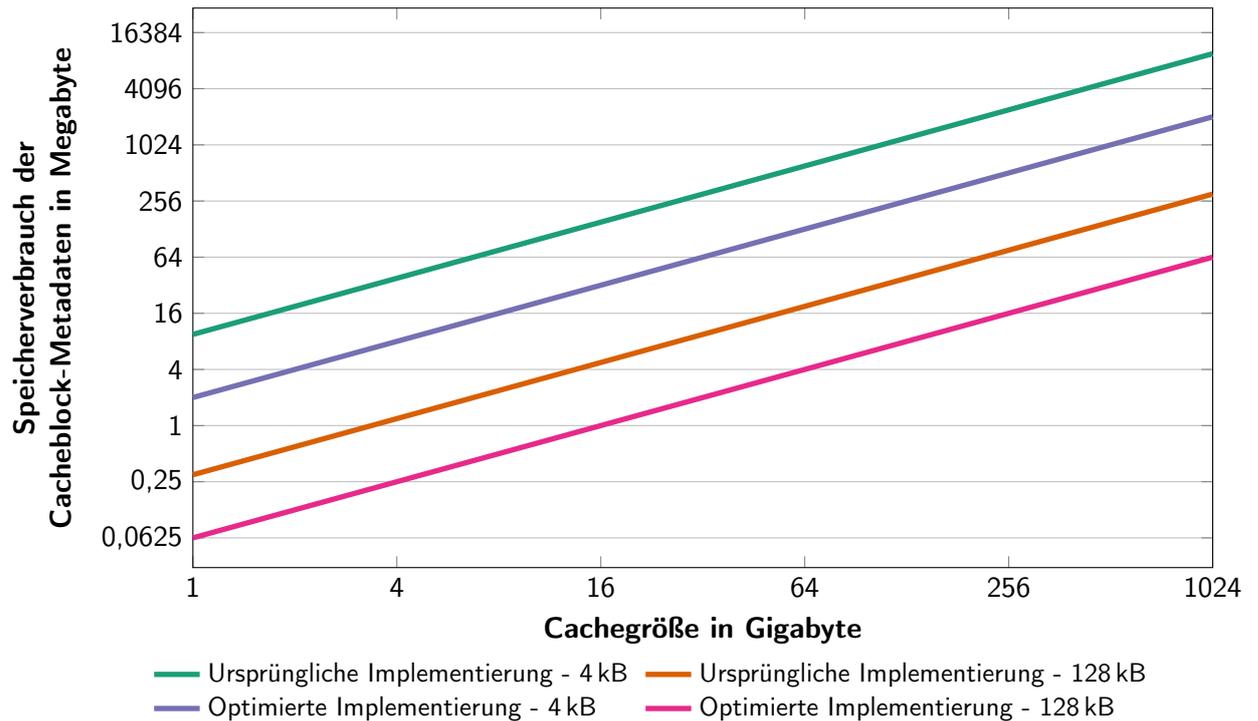


Abbildung 7.11: Speicherverbrauch der Cacheblock-Metadaten in Abhängigkeit von der Cachegröße

Für den Speicherverbrauch der neuen Implementierung müssen zusätzlich die Iterationszähler berücksichtigt werden. Aus diesem Grund erweitert sich die Formel wie in der Gleichung 7.3 zu sehen ist:

$$\text{Speicherverbrauch [MB]} = \underbrace{\text{Blockanzahl} \cdot 2^{20} \cdot 8\text{B}}_{\text{Cacheblock-Metadaten}} + \underbrace{\frac{\text{Blockanzahl} \cdot 2^{20}}{\text{Assoziativität}} \cdot 4\text{B}}_{\text{Iterationszähler}} \quad (7.3)$$

Zur besseren Veranschaulichung ist in Abbildung 7.11 der Speicherverbrauch der Cacheblock-Metadaten in Abhängigkeit von der Cachegröße dargestellt. Für die optimierte Implementierung wurde eine Assoziativität von 2048 angenommen, da sich diese, wie oben diskutiert, als effizient herausgestellt hat. Das Diagramm stellt dabei den Bedarf für die beiden Cacheblockgrößen von 4kB und 128kB dar.

### 7.3.2 Diskussion

Das Diagramm der Prozessorlast in Abbildung 7.10 verdeutlicht das in Abschnitt 7.1 bereits angesprochene Problem der steigenden Prozessorlast bei größer werdender Assoziativität. Demzufolge kann festgestellt werden, dass es sinnvoll ist für jedes System das optimale Assoziativitätsmaß zu finden. Weiterhin ist ersichtlich, dass sich die Prozessorlast bezogen auf einen Kern bei optimaler Assoziativität zwischen 10% und 20% bewegt, bei nahezu vollständiger Auslastung des Cachegeräts. Dieser Wert ist gut akzeptierbar, insbesondere da dies bei einem modernen 4-Kern System eine Gesamtsystemlast von weniger als 5% bedeutet. Folglich steht der Nutzung des Caches aus Sicht der Prozessorauslastung nichts im Wege.

Weiterhin gilt es den Speicherverbrauch des Caches zu beurteilen. Hierbei kann zunächst festgestellt werden, dass durch die Optimierungen, die an der *dm-cache* Implementierung im Rahmen dieser Arbeit durchgeführt wurden, selbst im ungünstigsten Szenario einer Assoziativität von lediglich eins, der Speicherverbrauch um knapp 70% gesenkt wurde. Dies zeigt die folgende Rechnung:

$$\begin{aligned} \text{Blockanzahl} \cdot 2^{20} \cdot 38B &\stackrel{?}{>} \text{Blockanzahl} \cdot 2^{20} \cdot 8B + \max_{\text{Assoziativität}} \left\{ \frac{\text{Blockanzahl} \cdot 2^{20}}{\text{Assoziativität}} \cdot 4B \right\} \\ \text{Blockanzahl} \cdot 2^{20} \cdot 38B &\stackrel{?}{>} \text{Blockanzahl} \cdot 2^{20} \cdot 8B + \frac{\text{Blockanzahl} \cdot 2^{20}}{1} \cdot 4B \\ 38B &> 12B \end{aligned}$$

Wie Abbildung 7.11 zu entnehmen ist, ist der Speicherverbrauch des Caches bei einem ein Terabyte großen Cache mit einer Blockgröße von vier Kilobyte ca. zwei Gigabyte groß. Zunächst ist anzumerken, dass es momentan nicht sinnvoll ist, Caches mit mehr als einem Terabyte zu betrachten, da in nächster Zeit kein solches SSDs auf dem Consumer-Markt verfügbar sein wird. Der Speicherverbrauch von zwei Gigabyte ist sehr kritisch zu sehen, da dies einen Großteil des bei aktuellen Systemen verfügbaren Speichers in Anspruch nehmen würde. Jedoch verdeutlichen Abbildungen 7.3 und 7.4, dass eine Blockgröße im Bereich von 128kb-256kb besser geeignet ist, als eine von vier Kilobyte. Hierbei sinkt der Speicherverbrauch auf ca. 128MB bzw. 64MB. Dies sollte für aktuelle Systeme kein Problem darstellen. Daher kann auch in Bezug auf den Speicherverbrauch festgestellt werden, dass ein blockbasierter Cache vorteilhaft ist.

## Kapitel 8

# Zusammenfassung und Ausblick

Zusammenfassend kann konstatiert werden, dass die Erweiterung der Speicherhierarchie, wie sie in der Einleitung dargestellt wurde, realisierbar und sinnvoll ist. Es wurde gezeigt, dass eine Implementierung auf Blockebene mindestens dieselbe Leistung erbringt wie eine Implementierung auf Dateisystemebene. Auf Grund der universelleren Einsetzbarkeit sollte die blockbasierte Implementierung vorgezogen werden und die Entwicklung weiter vorangetrieben werden.

Auf systemspezifischer Ebene des Linux Betriebssystems hat sich die Nutzung des DM-Framework als solide und effizient herausgestellt. Des Weiteren wurde gezeigt, dass es prinzipiell möglich ist, von einem in dieser Arbeit entworfenen Cache zu booten. Da diese Funktionalität jedoch eine Kernelerweiterung nutzt, von der momentan nicht klar ist, ob sie in den offiziellen Kernel einfließt, ist zur Zeit auch unklar, ob der beschrittene Weg der richtige ist. Sollte die Erweiterung in den Kernel aufgenommen werden, ist diese Frage ganz klar mit ja zu beantworten, sollte sie jedoch nicht aufgenommen werden, sollte auch der Weg über eine moduleigene Lösung betrachtet werden. Ähnliches gilt für das in Abschnitt 6.7 angesprochene Problem beim Neustart. Auch hier sollte zunächst versucht werden, eine einheitliche Lösung durch Erweiterung des DM-Frameworks zu finden. Sollte dies scheitern, muss auch hier der Ansatz der modulinternen Lösung weiterverfolgt werden.

Bei der weiteren Erforschung des Themenbereichs ist es notwendig eine Reihe von Problemen auf der Seite der Implementierung zu lösen. Zunächst gilt es die Synchronisationsprobleme der Write-Back- und Write-Hybrid-Strategie zu betrachten. Hierfür bedarf es einer detaillierten Analyse des bestehenden Codes in Hinblick auf Konkurrenzsituationen und mögliche Deadlock-Gefahren. Nachdem dieses Problem gelöst ist, müssen die Leistungsdefizite des Write-Back-Caches beseitigt werden. Dies könnte in der Art geschehen, dass in die `map`-Funktion ein Filter eingefügt wird, der sequenzielle Schreibzugriffe erkennt und entsprechende Anfragen direkt an das Quellgerät weiterleitet. Die Implementierung des Filters wird dabei vor die Herausforderung gestellt, dass die `map`-Funktion nur Anfragen in Größe der Cacheblockgröße erhält. Darum ist es problematisch die Gesamtgröße einer Anfrage zu bestimmen.

Nach der erfolgreichen Implementierung des Write-Back-Caches muss unbedingt die Sicherung der Cachemetadaten verbessert werden. Bei einem Write-Through-Cache kann beim Verlust der Metadaten der Cache problemlos neu aufgebaut werden. Beim Write-Back- und Write-Hybrid-Cache ist dies nicht möglich, da eine Menge gültiger Daten exklusiv auf dem Cachegerät vorliegen. Demzufolge ist es schwierig die Metainformationen über diese

Daten nicht zu verlieren. Die momentane Implementierung nimmt auf diesen Sachverhalt keinerlei Rücksicht, so dass hierbei eine weitere Analyse notwendig ist.

Nach der Behebung der Implementierungsdefizite sollte die Leistungsmessung verbessert werden. Es wäre wünschenswert ähnlich wie in den Arbeiten aus Kapitel 3 die Leistungsbeurteilung anhand realer Zugriffe durchzuführen. Hierfür müssten zunächst Blockgerätzugriffe auf Produktivsystemen aufgezeichnet werden. Anschließend sollten diese auf den implementierten Cache angewendet werden. Da gerade Letzteres sehr zeitaufwendig ist, kann die in dieser Arbeit verwendete Messungsmethode als Vorauswahl für vielversprechende Konfigurationen genutzt werden. Diese können dann in einer länger andauernden Testreihe mit den aufgezeichneten Realdaten evaluiert werden.

# Anhang

## Abbildungen

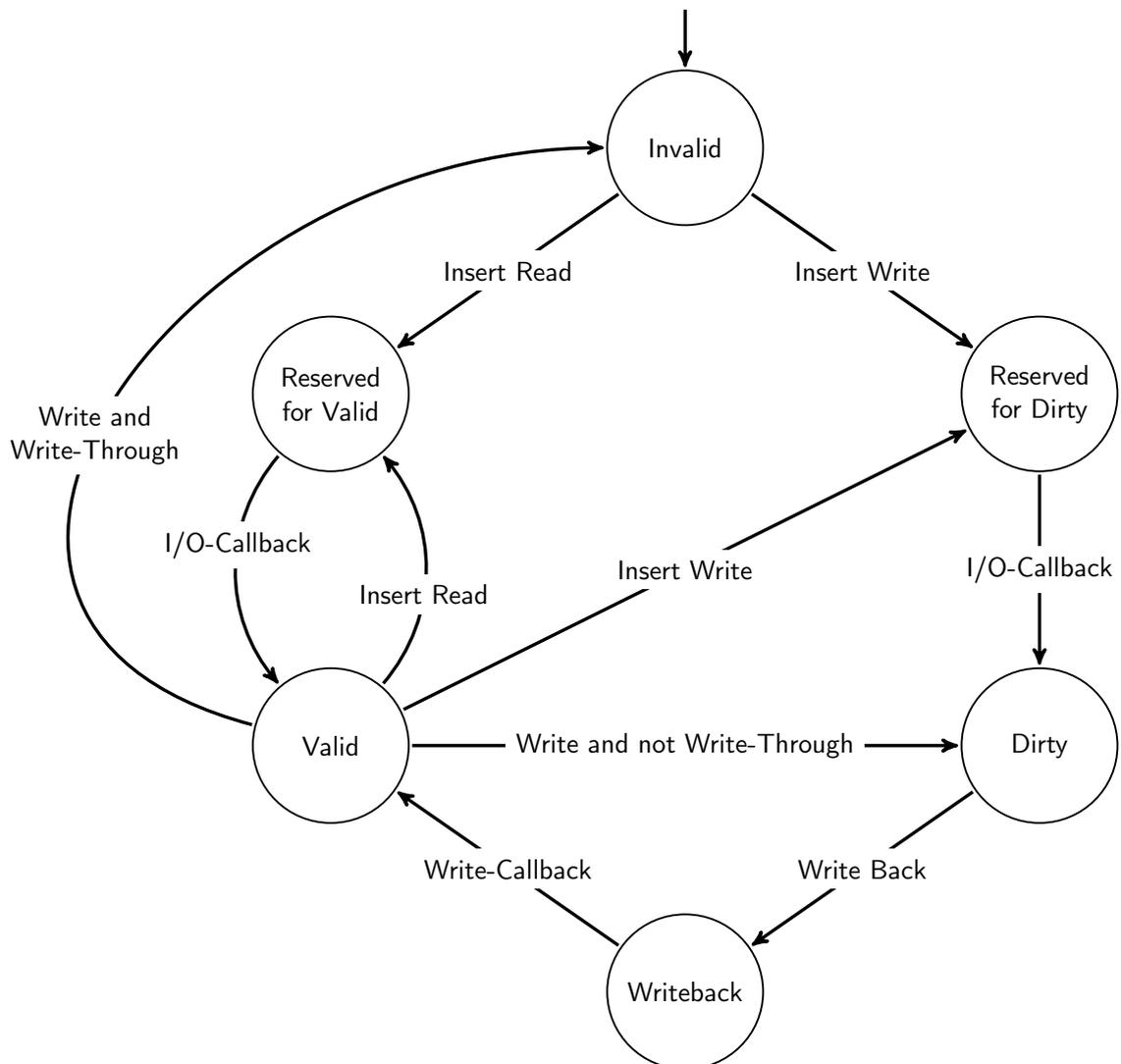
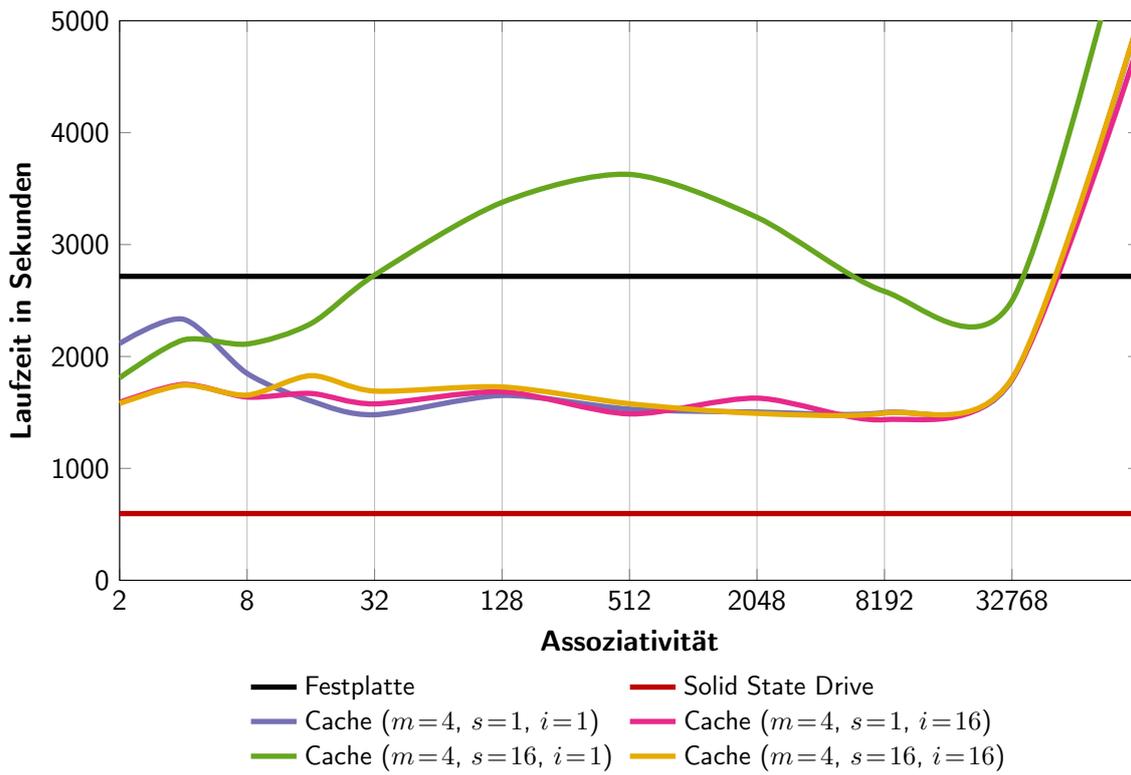
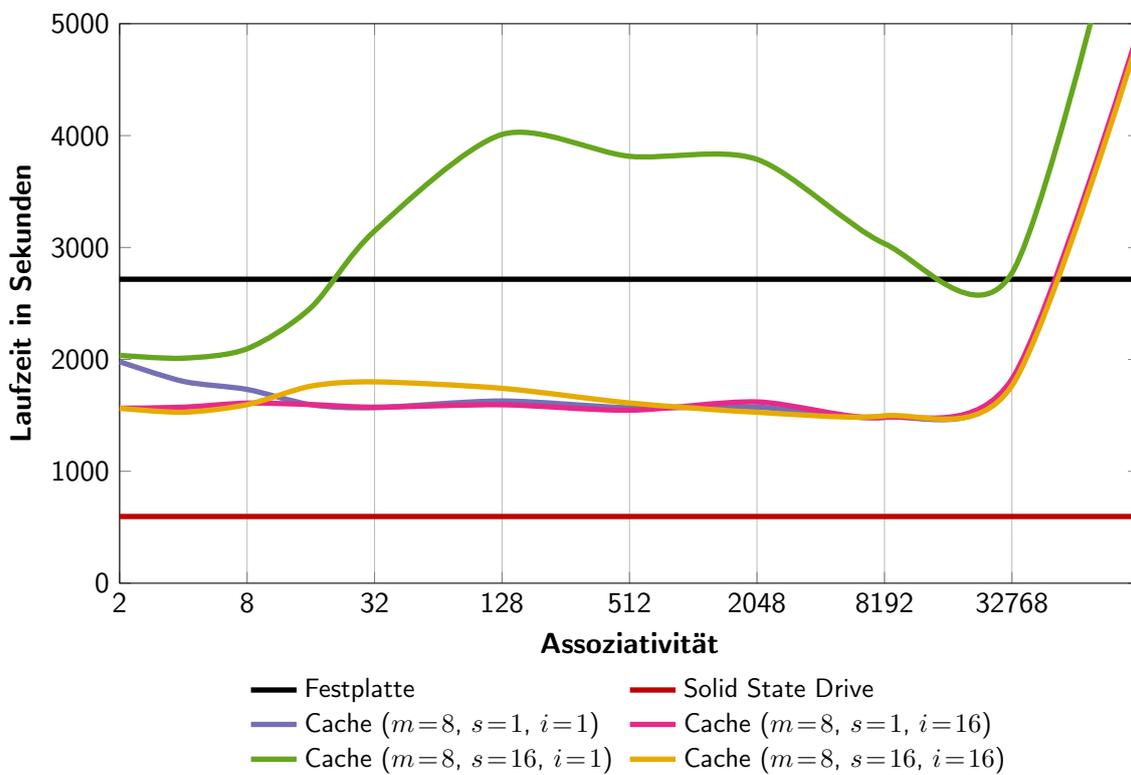


Abbildung A.1: Vollständige Zustandsmaschine des Caches

Abbildung A.2: Benchmarkergebnisse des Caches für  $m = 4$ Abbildung A.3: Benchmarkergebnisse des Caches für  $m = 8$

**Boot chart for brutzler (Tue Jun 15 13:57:17 CEST 2010)**

uname: Linux 2.6.34-gentoo #23 SMP Mon Jun 14 15:59:44 CEST 2010 x86\_64  
 release: Gentoo Base System release 1.12.13  
 CPU: AMD Phenom(tm) 9750 Quad-Core Processor (4)  
 kernel options: video=vesafb:mtr:3,ywrap vga=842 root=/dev/sda3  
 time: 0:27

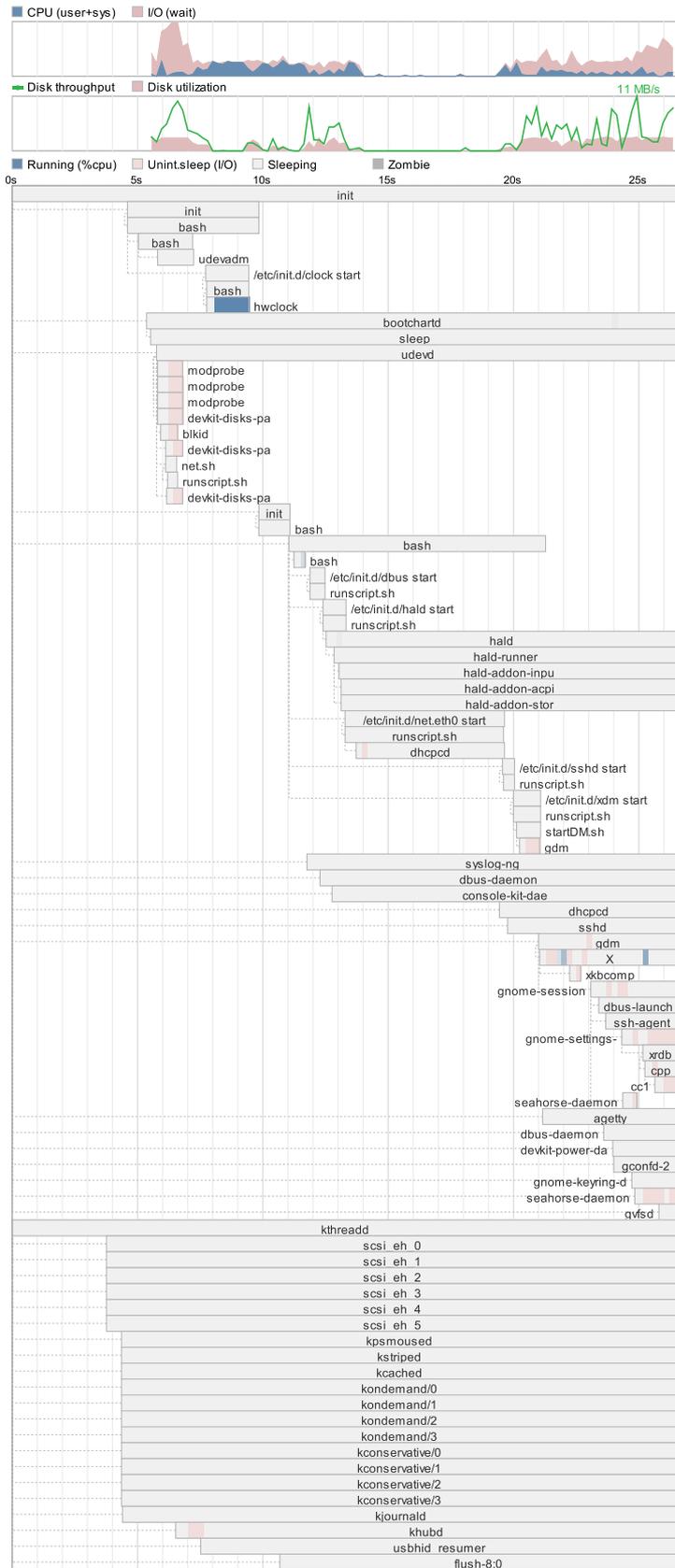


Abbildung A.4: Ausführliches Log eines HDD-Bootvorgangs

### Boot chart for brutzler (Tue Jun 15 16:25:37 CEST 2010)

uname: Linux 2.6.34-gentoo #23 SMP Mon Jun 14 15:59:44 CEST 2010 x86\_64  
 release: Gentoo Base System release 1.12.13  
 CPU: AMD Phenom(tm) 9750 Quad-Core Processor (4)  
 kernel options: video=vesafb:mtr:3,ywrap vga=842 root=/dev/sda3  
 time: 0:22

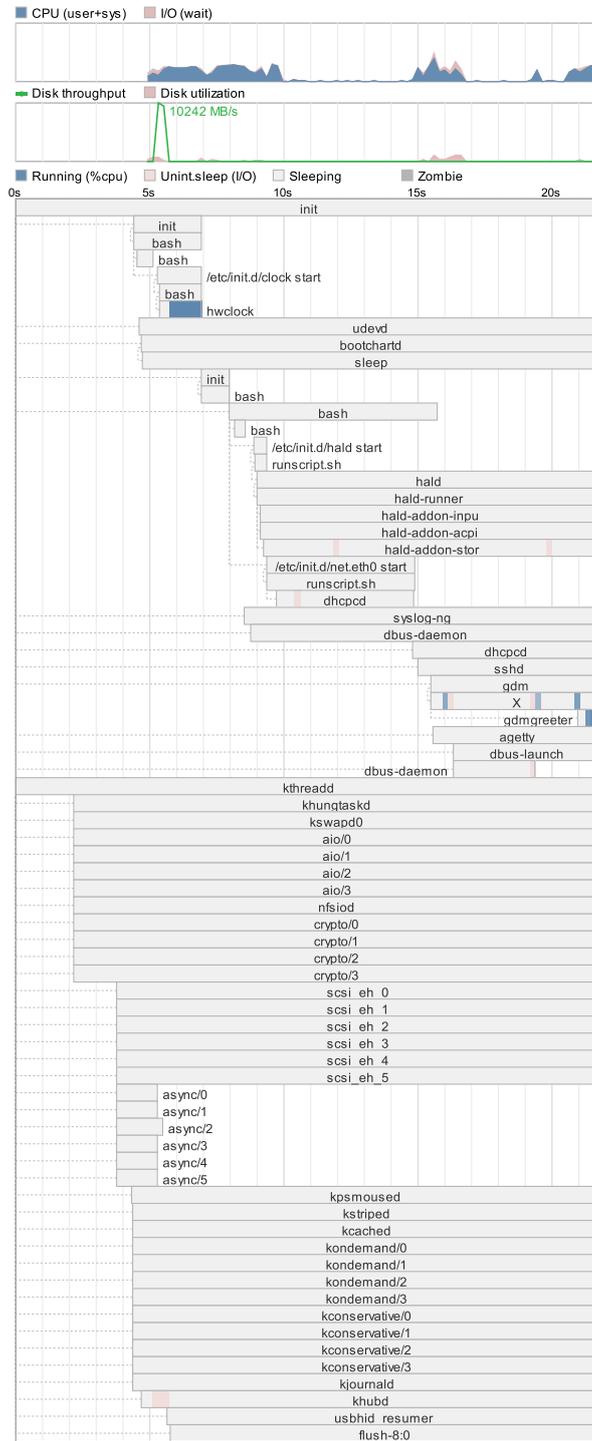


Abbildung A.5: Ausführliches Log eines SSD-Bootvorgangs mit OCZ SSD

**Boot chart for brutzler (Tue Jun 15 14:23:35 CEST 2010)**

uname: Linux 2.6.34-gentoo #23 SMP Mon Jun 14 15:59:44 CEST 2010 x86\_64  
 release: Gentoo Base System release 1.12.13  
 CPU: AMD Phenom(tm) 9750 Quad-Core Processor (4)  
 kernel options: video=vesafb:mtrr:3,ywrap vga=842 root=/dev/dm-0 dm="cached none 0, 0 136729215 cache /dev/sda3 /dev/sdb1 0 8 524288 2048 0 16 1 1"  
 time: 0:31

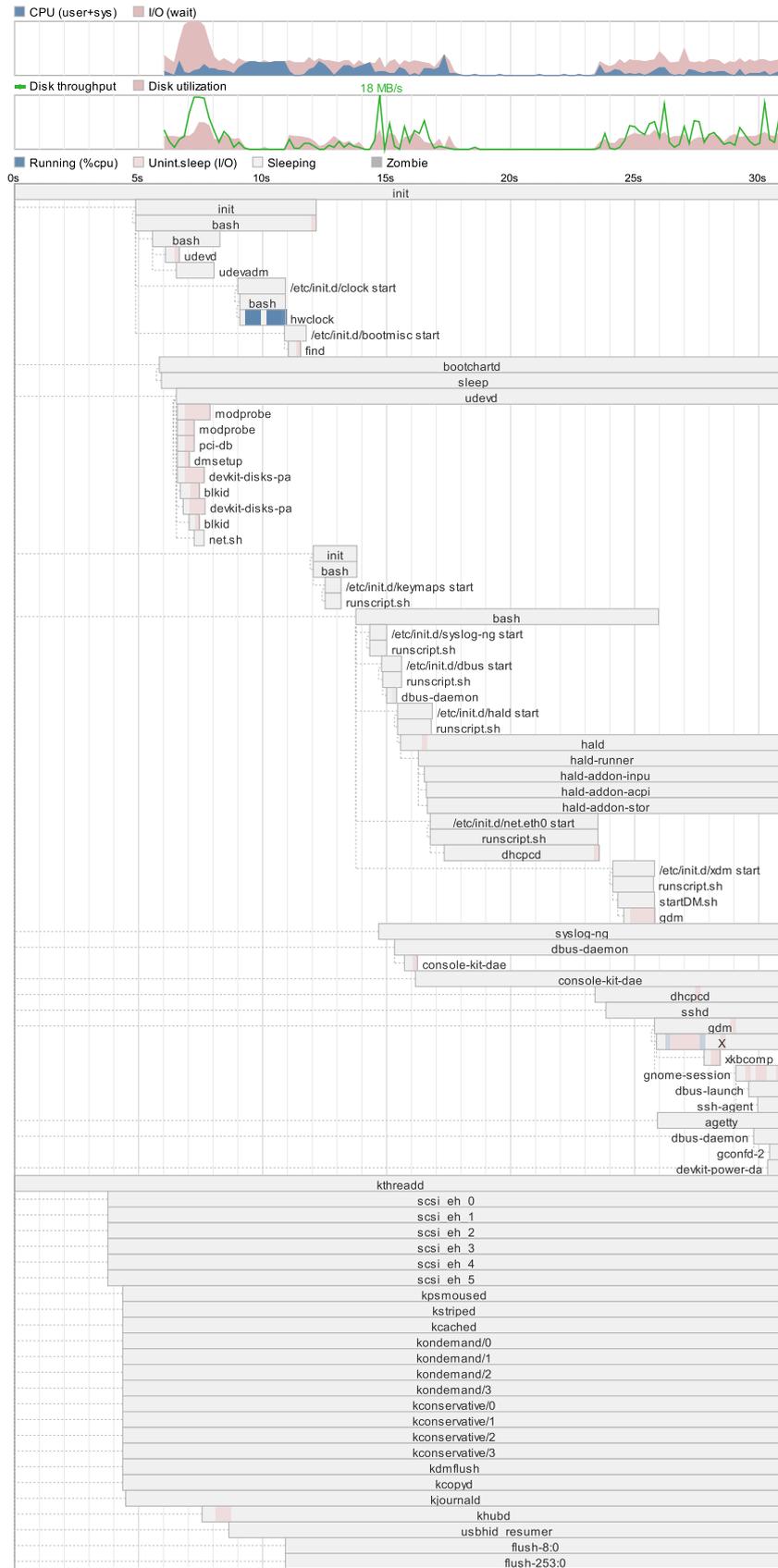


Abbildung A.6: Ausführliches Log des ersten Cache-Bootvorgangs mit OCZ SSD

**Boot chart for brutzler (Tue Jun 15 14:27:43 CEST 2010)**

uname: Linux 2.6.34-gentoo #23 SMP Mon Jun 14 15:59:44 CEST 2010 x86\_64  
 release: Gentoo Base System release 1.12.13  
 CPU: AMD Phenom(tm) 9750 Quad-Core Processor (4)  
 kernel options: video=vesafb:mtr:3,ywrap vga=842 root=/dev/dm-0 dm=""cached none 0,0 136729215 cache /dev/sda3 /dev/sdb1 1\*  
 time: 0:24

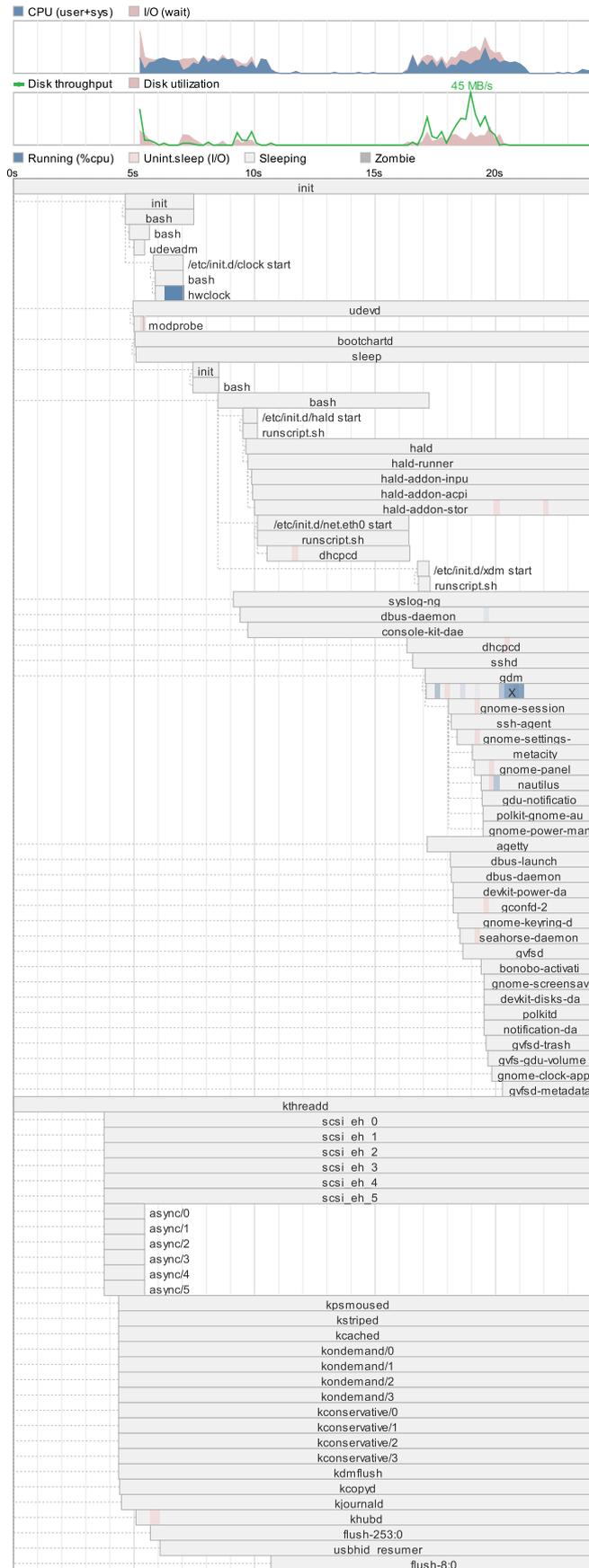


Abbildung A.7: Ausführliches Log des zweiten Cache-Bootvorgangs mit OCZ SSD

**Boot chart for brutzler (Thu Jun 24 16:05:45 CEST 2010)**

uname: Linux 2.6.34-gentoo #23 SMP Mon Jun 14 15:59:44 CEST 2010 x86\_64  
 release: Gentoo Base System release 1.12.13  
 CPU: AMD Phenom(tm) 9750 Quad-Core Processor (4)  
 kernel options: video=vesafb:mtrr:3,ywrap vga=842 root=/dev/sda3  
 time: 0:22

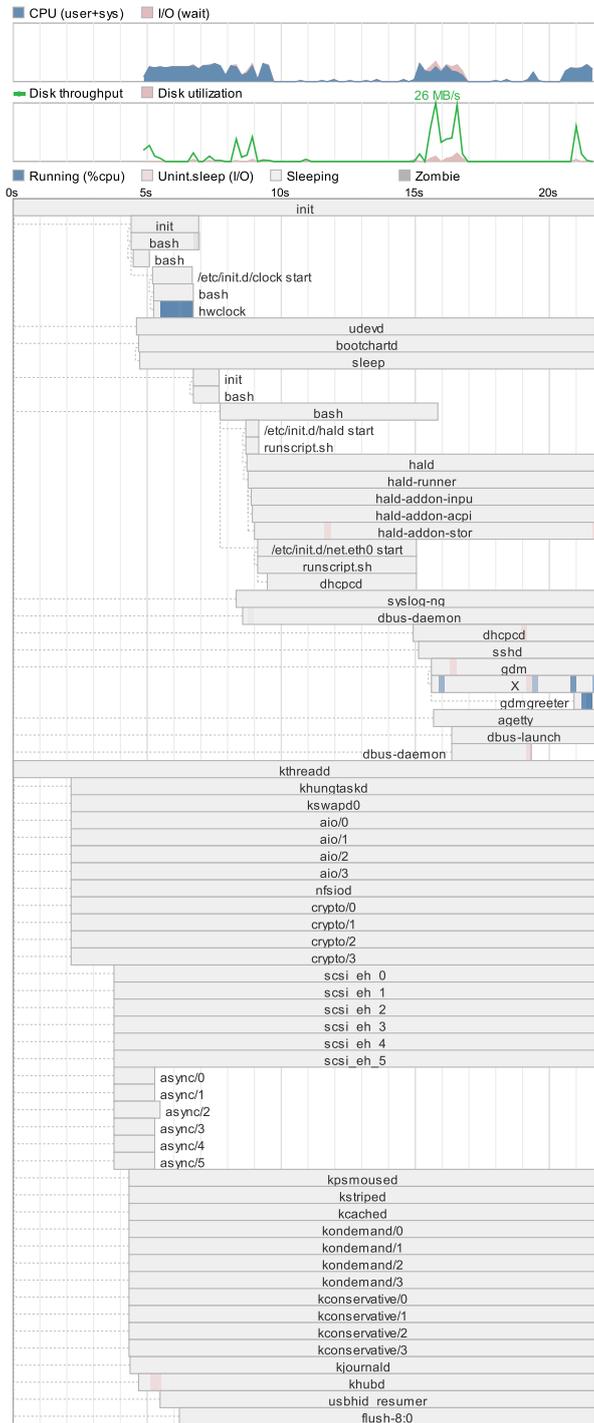


Abbildung A.8: Ausführliches Log eines SSD-Bootvorgangs mit Intel SSD

**Boot chart for brutzler (Thu Jun 24 16:16:21 CEST 2010)**

uname: Linux 2.6.34-gentoo #23 SMP Mon Jun 14 15:59:44 CEST 2010 x86\_64  
 release: Gentoo Base System release 1.12.13  
 CPU: AMD Phenom(tm) 9750 Quad-Core Processor (4)  
 kernel options: video=vesafb.mtrr:3,ywrap yga=842 root=/dev/dm-0 dm="cached none 0,0 136729215 cache /dev/sda3 /dev/sdb1 0 8 524288 2048 0 16 1 1"  
 time: 0:31

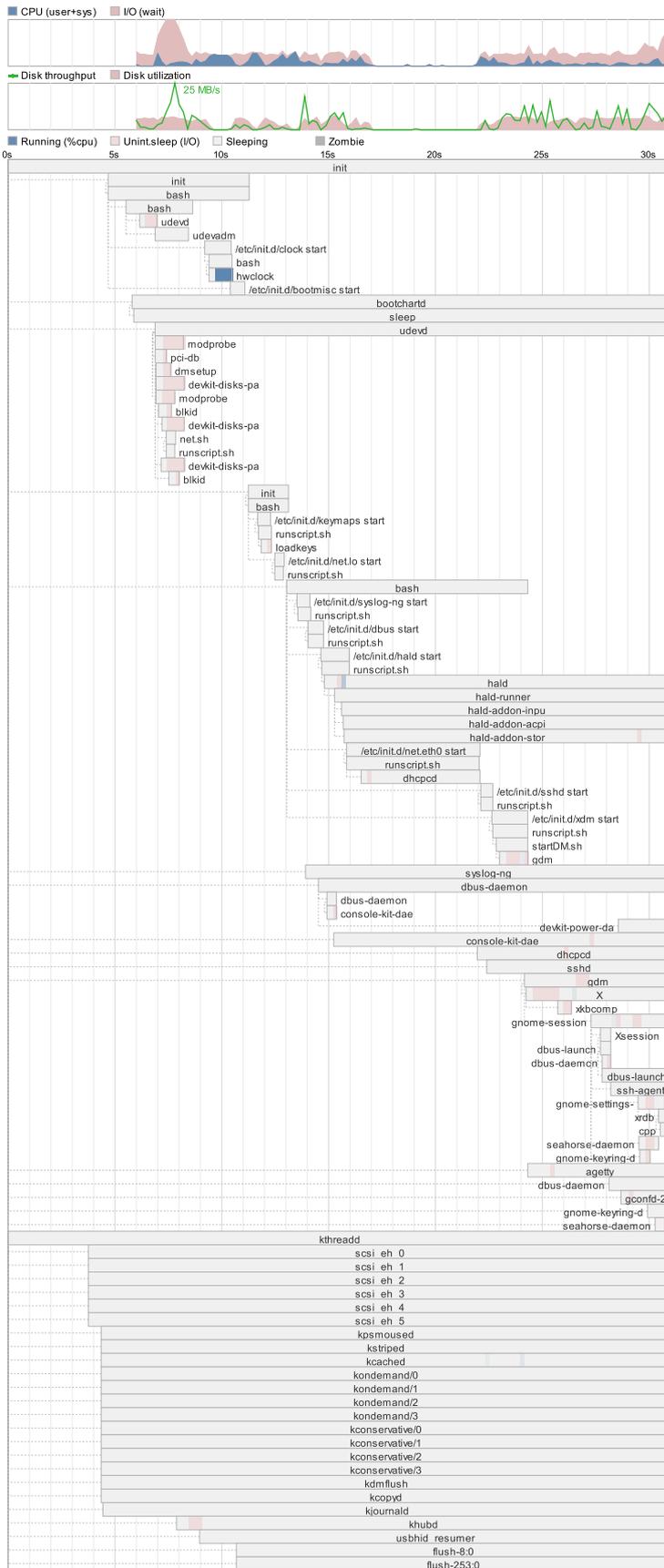


Abbildung A.9: Ausführliches Log des ersten Cache-Bootvorgangs mit Intel SSD

**Boot chart for brutzler (Thu Jun 24 16:18:18 CEST 2010)**

uname: Linux 2.6.34-gentoo #23 SMP Mon Jun 14 15:59:44 CEST 2010 x86\_64  
 release: Gentoo Base System release 1.12.13  
 CPU: AMD Phenom(tm) 9750 Quad-Core Processor (4)  
 kernel options: video=vesafb.mtrr:3.wrap vga=842 root=/dev/dm-0 dm=""cached none 0,0 136729215 cache /dev/sda3 /dev/sdb1 \*  
 time: 0:22

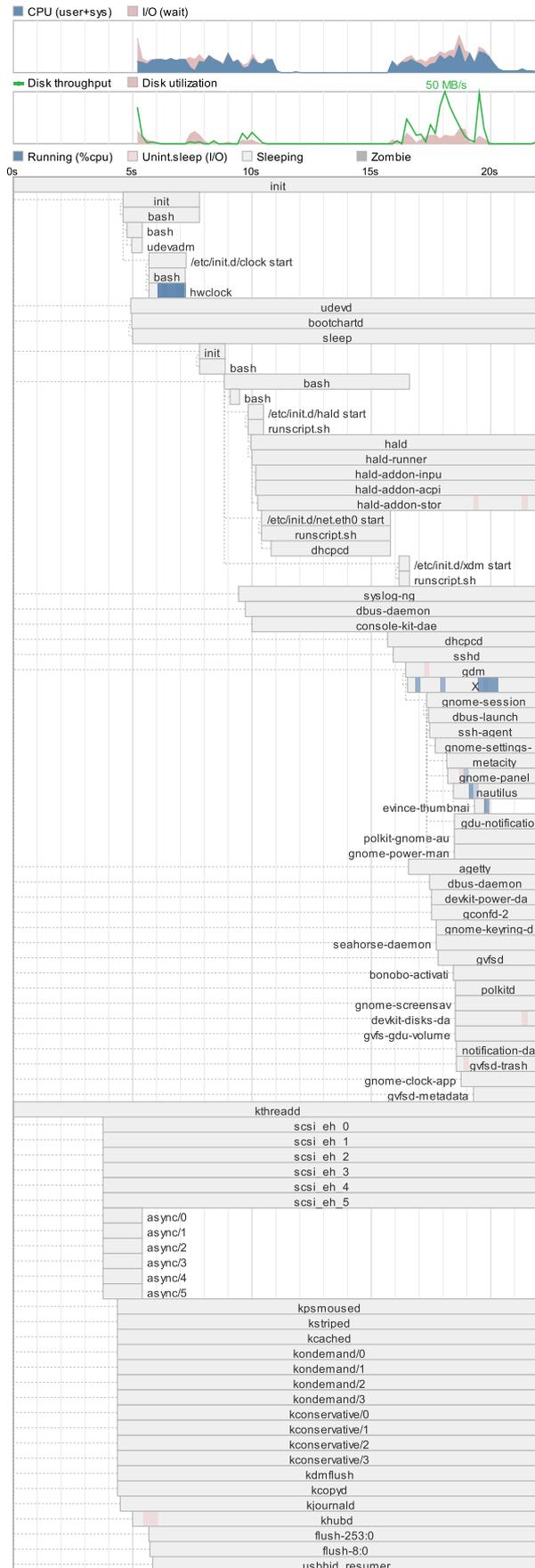


Abbildung A.10: Ausführliches Log des zweiten Cache-Bootvorgangs mit Intel SSD

## Listings

### Linux Kernel

```

1  /*
2  * main unit of I/O for the block layer and lower layers (ie drivers and stacking drivers)
3  */
4  struct bio {
5      sector_t          bi_sector; /* device address in 512 byte sectors */
6      struct bio        *bi_next; /* request queue link */
7      struct block_device *bi_bdev;
8      unsigned long     bi_flags; /* status, command, etc */
9      unsigned long     bi_rw; /* bottom bits READ/WRITE, top bits priority */
10
11     unsigned short     bi_vcvt; /* how many bio_vec's */
12     unsigned short     bi_idx; /* current index into bvl_vec */
13
14     /* Number of segments in this BIO after physical address coalescing is performed. */
15     unsigned int       bi_phys_segments;
16
17     unsigned int       bi_size; /* residual I/O count */
18
19     /*
20     * To keep track of the max segment size, we account for the
21     * sizes of the first and last mergeable segments in this bio.
22     */
23     unsigned int       bi_seg_front_size;
24     unsigned int       bi_seg_back_size;
25
26     unsigned int       bi_max_vecs; /* max bvl_vecs we can hold */
27
28     unsigned int       bi_comp_cpu; /* completion CPU */
29
30     atomic_t          bi_cnt; /* pin count */
31
32     struct bio_vec     *bi_io_vec; /* the actual vec list */
33
34     bio_end_io_t       *bi_end_io;
35
36     void               *bi_private;
37 #if defined(CONFIG_BLK_DEV_INTEGRITY)
38     struct bio_integrity_payload *bi_integrity; /* data integrity */
39 #endif
40
41     bio_destructor_t   *bi_destructor; /* destructor */
42
43     /*
44     * We can inline a number of vecs at the end of the bio, to avoid
45     * double allocations for a small number of bio_vecs. This member
46     * MUST obviously be kept at the very end of the bio.
47     */
48     struct bio_vec     bi_inline_vecs[0];
49 };

```

Listing A.1: Kernelinterne Repräsentation eines BIO (Quelle: [21])

## dm-cache Modul

```

1  /* Cache context */
2  struct cache_c {
3      struct dm_dev *src_dev;      /* Source device */
4      struct dm_dev *cache_dev;   /* Cache device */
5      struct dm_kcopyd_client *kcp_client; /* Kcopyd client for writing back data */
6
7      struct cacheblock *cache;   /* Hash table for cache blocks */
8      sector_t size;             /* Cache size */
9      unsigned int bits;         /* Cache size in bits */
10     unsigned int assoc;        /* Cache associativity */
11     unsigned int block_size;    /* Cache block size */
12     unsigned int block_shift;   /* Cache block size in bits */
13     unsigned int block_mask;   /* Cache block mask */
14     unsigned int consecutive_shift; /* Consecutive blocks size in bits */
15     unsigned long counter;      /* Logical timestamp of last access */
16     unsigned int write_policy;  /* Cache write policy */
17     sector_t dirty_blocks;     /* Number of dirty blocks */
18
19     spinlock_t lock;           /* Lock to protect page allocation/deallocation */
20     struct page_list *pages;   /* Pages for I/O */
21     unsigned int nr_pages;     /* Number of pages */
22     unsigned int nr_free_pages; /* Number of free pages */
23     wait_queue_head_t destroyq; /* Wait queue for I/O completion */
24     atomic_t nr_jobs;         /* Number of I/O jobs */
25     struct dm_io_client *io_client; /* Client memory pool*/
26
27     /* Stats */
28     unsigned long reads;       /* Number of reads */
29     unsigned long writes;     /* Number of writes */
30     unsigned long cache_hits; /* Number of cache hits */
31     unsigned long replace;    /* Number of cache replacements */
32     unsigned long writeback;  /* Number of replaced dirty blocks */
33     unsigned long dirty;      /* Number of submitted dirty blocks */
34 };

```

Listing A.2: Datenstruktur für Cachemetainformationen (Quelle: [12])

```

1  /* Cache block metadata structure */
2  struct cacheblock {
3      spinlock_t lock;          /* Lock to protect operations on the bio list */
4      sector_t block;          /* Sector number of the cached block */
5      unsigned short state;    /* State of a block */
6      unsigned long counter;   /* Logical timestamp of the block's last access */
7      struct bio_list bios;    /* List of pending bios */
8 };

```

Listing A.3: Datenstruktur für Cacheblock-Metainformationen (Quelle: [12])

```

1  /* Structure for a kcached job */
2  struct kcached_job {
3      struct list_head list;
4      struct cache_c *dmc;
5      struct bio *bio;        /* Original bio */
6      struct dm_io_region src;
7      struct dm_io_region dest;
8      struct cacheblock *cacheblock;
9      int rw;
10     /*
11     * When the original bio is not aligned with cache blocks,
12     * we need extra bvecs and pages for padding.
13     */
14     struct bio_vec *bvec;
15     unsigned int nr_pages;
16     struct page_list *pages;
17 };

```

Listing A.4: Datenstruktur für Auftragsliste des dm-cache (Quelle: [12])

## Erweitertes und optimiertes dm-cache Modul

```

1  /* Cache metadata */
2  struct cache_c {
3      struct dm_dev *src_dev;          /* Source device */
4      struct dm_dev *cache_dev;       /* Cache device */
5      struct dm_kcopyd_client *kcp_client; /* Kcopyd client for writing back data */
6
7      struct list_head list;          /* List hook for the other dm-cach instances in the system */
8
9      struct cacheblock *cache;       /* Hash table for cache blocks */
10     unsigned int *marker;
11     sector_t size;                  /* Cache size */
12     unsigned int bits;              /* Cache size in bits */
13     unsigned int assoc;             /* Cache associativity */
14     unsigned int assoc_bits;        /* Cache associativity in bits */
15     unsigned int assoc_mask;        /* Cache associativity mask */
16     unsigned long hash_pattern;     /* pattern for block hashing */
17     unsigned int sets;              /* Number of sets */
18     unsigned int block_size;        /* Cache block size */
19     unsigned int block_shift;       /* Cache block size in bits */
20     unsigned int block_mask;        /* Cache block mask */
21     unsigned int consecutive_shift; /* Consecutive blocks size in bits */
22     unsigned int write_policy;      /* Cache write policy */
23     unsigned int wb_on_dtr;         /* Write back dirty pages on cache destruction */
24     unsigned short age_max;         /* Maximum age for a cacheblock */
25     unsigned short age_default;     /* Default age for a cacheblock */
26     unsigned short increment;       /* Increment at block access */
27
28     spinlock_t lock;                /* Lock to protect page allocation/deallocation */
29     struct page_list *pages;        /* Pages for I/O */
30     unsigned int nr_pages;          /* Number of pages */
31     unsigned int nr_free_pages;     /* Number of free pages */
32     wait_queue_head_t destroyq;     /* Wait queue for I/O completion */
33     atomic_t nr_jobs;              /* Number of I/O jobs */
34     struct dm_io_client *io_client; /* Client memory pool */
35
36     /* queue and lock for delayed BIOS */
37     spinlock_t queue_lock;          /* Lock to protect operations on the bio list */
38     struct queued_bios queue;       /* List of pending bios */
39
40     /* Stats */
41     unsigned long reads;            /* Number of reads */
42     unsigned long writes;           /* Number of writes */
43     unsigned long cache_hits;       /* Number of cache hits */
44     unsigned long replace;          /* Number of cache replacements */
45     unsigned long writeback;        /* Number of replaced dirty blocks */
46     unsigned long dirty;            /* Number of submitted dirty blocks */
47
48     #if defined(CONFIG_DEBUG_FS)
49     struct dentry *debugfs;         /* Root node for debugfs */
50     struct debugfs_blob_wrapper blob_cache; /* Blob for cachblock metadata export */
51     #endif
52 };

```

Listing A.5: Erweiterte Datenstruktur für Cache-Metainformationen

# Literatur

- [1] American National Standard for Information Technology. Hrsg. von C. E. Stevens. *Information technology - ATA/ATAPI Command Set - 2 (ACS-2)*. T13/2015-D Revision 2. Aug. 2009. Working Draft Project.
- [2] L. A. Belady. „A Study of Replacement Algorithms for a Virtual-Storage Computer“. In: *IBM Systems Journal* 5.2 (1966), S. 78–101. DOI: 10.1147/sj.52.0078.
- [3] *Bonnie++*. URL: <http://www.coker.com.au/bonnie++> (besucht am 07.06.2010).
- [4] *Bootchart*. URL: <http://www.bootchart.org> (besucht am 15.06.2010).
- [5] G. Brendan. *Hybrid Storage Pool: Top Speeds*. Okt. 2009. URL: [http://blogs.sun.com/brendan/entry/hybrid\\_storage\\_pool\\_top\\_speeds](http://blogs.sun.com/brendan/entry/hybrid_storage_pool_top_speeds) (besucht am 15.05.2010).
- [6] J. Corbet. *Block Layer Discard Requests*. Aug. 2008. URL: <http://lwn.net/Articles/293658/> (besucht am 10.06.2010).
- [7] J. Corbet. *Debugfs*. Dez. 2004. URL: <http://lwn.net/Articles/115405/> (besucht am 25.05.2010).
- [8] J. Corbet. *Notifiers, 2.6.17 Style*. Mai 2006. URL: <http://lwn.net/Articles/185500/> (besucht am 15.06.2010).
- [9] R. Coulson, S. Trika, J. Matthews und R. Faber. „Maintaining Write Ordering in a System“. US20070233947 A1. Eingereicht 30. März 2006. Okt. 2007.
- [10] *Device-Mapper Resource Page*. URL: <http://sources.redhat.com/dm/> (besucht am 09.05.2010).
- [11] N. Dittmar. *Test: SuperTalent Teradrive FT SSD*. Mai 2010. URL: [http://www.computerbase.de/artikel/laufwerke/2010/test\\_supertalent\\_teradrive\\_ft\\_ssd/](http://www.computerbase.de/artikel/laufwerke/2010/test_supertalent_teradrive_ft_ssd/) (besucht am 10.06.2010).
- [12] *dm-cache Sourcecode*. URL: <https://github.com/mingzhao/dm-cache/> (besucht am 25.05.2010).
- [13] W. Drewry. *init: Boot to Device-Mapper Targets Without an initr\**. Mai 2010. URL: <http://permalink.gmane.org/gmane.linux.kernel/988034> (besucht am 03.06.2010).
- [14] E. van Hensbergen und M. Zhao. *Dynamic Policy Disk Caching for Storage Networking*. Techn. Ber. RC24123. IBM Research, Nov. 2006.
- [15] P. Herrmann. *Rechnerarchitektur. Aufbau, Organisation und Implementierung, inklusive 64-Bit-Technologie und Parallelrechner*. 3. Aufl. Vieweg+Teubner Verlag, 2002. ISBN: 978-3-528255985.

- [16] Intel. *Intel X25-E SATA Solid State Drive*. Mai 2009. URL: <http://download.intel.com/design/flash/nand/extreme/319984.pdf> (besucht am 22.06.2010).
- [17] A. Kirshenbaum, C. Ergan, M. R. Fortin und R. L. Reinauer. „Using External Memory Devices to Improve System Performance“. US20060090031 A1. Eingereicht 21. Oktober 2004. Apr. 2006.
- [18] R. Landley. *ramfs, rootfs and initramfs*. Okt. 2005. URL: <http://git.kernel.org/?p=linux/kernel/git/torvalds/linux-2.6.git;a=blob;hb=HEAD;f=Documentation/filesystems/ramfs-rootfs-initramfs.txt> (besucht am 03.06.2010).
- [19] P. J. Leach, M. Mealling und R. Salz. *A Universally Unique Identifier (UUID) URN Namespace*. RFC 4122. RFC Editor, Juli 2005. URL: <http://www.rfc-editor.org/rfc/rfc4122.txt> (besucht am 03.06.2010).
- [20] D. Lee, J. Choi, J.-H. Kim, S. H. Noh, S. L. Min, Y. Cho und C. S. Kim. „LRFU: A Spectrum of Policies that Subsumes the Least Recently Used and Least Frequently Used Policies“. In: *IEEE Transactions on Computers* 50.12 (Dez. 2001), S. 1352–1361. DOI: 10.1109/TC.2001.970573.
- [21] *Linux-Kernel Sourcecode Version 2.6.34*. URL: <http://www.kernel.org/pub/linux/kernel/v2.6/linux-2.6.34.tar.bz2> (besucht am 25.05.2010).
- [22] J. Matthews, S. Trika, D. Hensgen, R. Coulson und K. Grimsrud. „Intel Turbo Memory: Nonvolatile Disk Caches in the Storage Hierarchy of Mainstream Computer Systems“. In: *ACM Transactions on Storage* 4.2 (Mai 2008), 4:1–4:24. DOI: 10.1145/1367829.1367830.
- [23] OCZ. *OCZ Vertex Series SATA II 2.5"SSD OCZ Technology*. URL: [http://www.ocztechnology.com/products/solid\\_state\\_drives/ocz\\_vertex\\_series\\_sata\\_ii\\_2\\_5-ssd](http://www.ocztechnology.com/products/solid_state_drives/ocz_vertex_series_sata_ii_2_5-ssd) (besucht am 22.06.2010).
- [24] *OpenSolaris*. URL: <http://opensolaris.org> (besucht am 21.06.2010).
- [25] Samsung. *Spinpoint M7*. URL: [http://www.samsung.com/global/system/business/hdd/prdmodel/2009/1/13/163182m7\\_sheet\\_0.5.pdf](http://www.samsung.com/global/system/business/hdd/prdmodel/2009/1/13/163182m7_sheet_0.5.pdf) (besucht am 25.05.2010).
- [26] W. Schiffmann. *Technische Informatik 2. Grundlagen der Computertechnik*. 5. Aufl. Springer-Lehrbuch. Springer-Verlag, 2005. ISBN: 978-3-540-22271-2.
- [27] Seagate. *Cheetah 15K.7*. URL: [http://www.seagate.com/docs/pdf/datasheet/disc/ds\\_cheetah\\_15k\\_7.pdf](http://www.seagate.com/docs/pdf/datasheet/disc/ds_cheetah_15k_7.pdf) (besucht am 25.05.2010).
- [28] A. L. Shimpi. *The SSD Anthology: Understanding SSDs and New Drives from OCZ*. März 2009. URL: <http://www.anandtech.com/show/2738> (besucht am 25.05.2010).
- [29] A. L. Shimpi. *The SSD Relapse: Understanding and Choosing the Best SSD*. Aug. 2009. URL: <http://www.anandtech.com/show/2829> (besucht am 25.05.2010).
- [30] G. Soundararajan, V. Prabhakaran, M. Balakrishnan und T. Wobber. „Extending SSD Lifetimes with Disk-based Write Caches“. In: *Proceedings of the 8th USENIX Conference on File and Storage Technologies*. FAST'10. San Jose, California: USENIX Association, 2010.
- [31] W. Stallings. *Operating Systems. Internals and Design Principles*. 6. Aufl. Prentice Hall Press, 2009. ISBN: 978-0-13-600632-9.
- [32] System News Inc. „Sun ZFS Plus Intel SSDs Equal Hybrid Storage Pools“. In: *System News* 142.5 (Dez. 2009). URL: <http://sun.systemnews.com/articles/142/5/Solaris/22462> (besucht am 25.05.2010).

- [33] A. S. Tanenbaum. *Computerarchitektur. Strukturen, Konzepte, Grundlagen*. 5. Aufl. Pearson Studium, 2006. ISBN: 978-3-8273-7151-5.
- [34] Western Digital. *Advanced Format Technology*. White Paper. Western Digital, Apr. 2010. URL: <http://products.wdc.com/library/WhitePapers/ENG/2579-771430.pdf> (besucht am 25.05.2010).
- [35] Western Digital. *Specifications for the 74 GB Serial ATA Raptor drive (model WD740ADFD)*. URL: [http://wdc.custhelp.com/cgi-bin/wdc.cfg/php/enduser/std\\_adp.php?p\\_faqid=1441&p\\_created=&p\\_cats=185&p\\_cv=1.185&p\\_pv=2.281&p\\_prods=227%2C281](http://wdc.custhelp.com/cgi-bin/wdc.cfg/php/enduser/std_adp.php?p_faqid=1441&p_created=&p_cats=185&p_cv=1.185&p_pv=2.281&p_prods=227%2C281) (besucht am 22.06.2010).





Anselm Busse

## **SSD-basiertes Caching von Blockgeräten**

Diese Arbeit beschäftigt sich mit der Frage, ob es möglich und sinnvoll ist ein Solid State Drive (SSD) als Cache für eine herkömmliche Magnetscheiben-basierte Festplatte zu nutzen.

In aktuellen Rechnersystemen stellt häufig der Massenspeicher in Form einer Festplatte den Flaschenhals des Systems dar. Dies ist darauf zurückzuführen, dass Festplatten auf Grund ihres mechanischen Aufbaus eine Zugriffszeit haben, die um mehrere Größenordnungen schlechter ist als die der nächsten Stufe in der Speicherhierarchie – dem Arbeitsspeicher. Um dieses Problem zu beseitigen, wurde in den letzten Jahren halbleiterbasierter Massenspeicher als Ersatz für Festplatten eingeführt, der dieses Defizit nicht besitzt. Diese Laufwerke werden unter der Bezeichnung SSD vermarktet. SSDs haben jedoch den Nachteil, dass die Kosten pro Byte wesentlich über denen von Festplatten liegen. Darum ist mit einer vollständigen Substitution von Festplatten durch SSDs in den kommenden Jahren kaum zu rechnen.

Die momentane Situation, die daraus resultiert, ist die, dass Anwender häufig genutzte Daten auf einer meist kleinen SSD speichern und die restlichen Daten auf einer langsameren Festplatte. Dieses Vorgehen ist für den Nutzer jedoch sehr umständlich. Deshalb wird in dieser Arbeit die Nutzung von SSDs als transparenter Cache für Festplatten vorgeschlagen. Dadurch würde nur ein geringes Eingreifen des Nutzers erforderlich sein und es ihm trotzdem ermöglicht, die Vorteile von SSDs zu nutzen.

Es werden im Verlauf dieser Arbeit dafür zunächst die technischen Grundlagen von Festplatten und SSDs dargestellt und andere Arbeiten betrachtet, die ein ähnliches Konzept verfolgen bzw. für die praktische Realisierung des Cache von Bedeutung sind. Auf Grundlage dieser technischen und theoretischen Rahmenbedingungen wird eine konkrete Problem- und Aufgabenstellung formuliert. Basierend auf dieser wird eine Architektur für einen block-basierten Cache vorgeschlagen, deren konkrete Implementierung ebenfalls in dieser Arbeit beispielhaft dargestellt wird. Mit Hilfe dieser Beispielimplementierung wurden für diese Arbeit Simulationen und Messungen durchgeführt. Sie ermöglichen es die Frage zu beantworten, ob es sinnvoll ist, eine SSD als Cache zu nutzen. Somit wird abschließend diese Fragestellung anhand der gewonnenen Messergebnisse unter den Gesichtspunkten der Leistungssteigerung und des zusätzlichen Ressourcenverbrauchs durch den Cache diskutiert.