



**UNIVERSITÄT PADERBORN**  
*Die Universität der Informationsgesellschaft*

# **Verfahren zur redundanten Datenplatzierung in skalierbaren Speichernetzen**

Dissertation

von

**Sascha Effert**

Heinz Nixdorf Institut und Institut für Informatik  
Universität Paderborn  
Juni 2011

**Gutachter:**

- Prof. Dr. math. Friedhelm Meyer auf der Heide, Universität Paderborn
- Jun.-Prof. Dr.-Ing. André Brinkmann, Universität Paderborn

*Für mein größtes Vorbild im Leben:  
mein Opa*



# Überblick

Moderne Datenzentren sind mit einer rasant wachsenden Menge an Daten konfrontiert, welche sie mit immer höherer Geschwindigkeit hochverfügbar speichern müssen. Daher brauchen sie Speichersysteme, welche mit ihren Anforderungen wachsen. Zunehmend werden dazu Speichernetze eingesetzt, in welchen Datenserver einen virtuellen Speicher über Festplatten erzeugen. Dabei ist die Last des virtuellen Speichers so zu verteilen, dass die physikalischen Festplatten optimal genutzt werden. Um Ausfälle kompensieren zu können ist es nötig, Daten redundant zu speichern. Die Verfahren zur Datenverteilung müssen diesen Anforderungen gerecht werden. Einen wichtigen Beitrag liefern hier pseudorandomisierte Hashfunktionen.

Innerhalb dieser Arbeit gehe ich auf verschiedene Speichersysteme ein. Speziell untersuche ich Speichernetze, welche aus Datenservern mit lokalen Festplatten bestehen. Für diese zeige ich, wie sie bei verschiedenen Arten der Datenverteilung skalieren. Leider wird keines der betrachteten Speichersysteme allen Anforderungen gerecht.

Als Lösung stelle ich das Verfahren *Redundant Share* vor, welches alle Anforderungen erfüllt. Mittels *Redundant Share* kann eine beliebige Anzahl an Kopien der Daten des virtuellen Speichers wie gefordert verteilt werden. Gleichzeitig erfordert das Hinzufügen neuer Festplatten einen begrenzten Aufwand. Abschließend vermesse ich eine Implementierung von *Redundant Share* und vergleiche die Ergebnisse mit anderen Verteilern.



# Abstract

Modern data centers are faced with a rapidly growing amount of data which they have to keep highly available with increasing performance. Therefore they need storage systems, which grow with the demands. Storage networks are more and more used in this area. In such systems data servers create a virtual storage on top of a number of physical hard disks. Therefore the load of the virtual storage has to be distributed in a way that the physical disks are used in an optimal way. It is also important to store all data with redundancies to be able to compensate broken hardware. The algorithms used for data distribution have to solve all these demands. Pseudo randomized hash functions have a big impact in this area.

In this dissertation I describe different storage systems. I take a closer look at storage networks with data servers using directly attached hard disks. For these I show how they scale using different kinds of data distribution. Unfortunately none of the described storage systems solves all demands.

As a solution I introduce *Redundant Share* which solves all demands. Using *Redundant Share* it is possible to distribute an arbitrary number of copies of each piece of data using each hard disk in an optimal way according its capacity. Moreover adding a new physical hard disk ends up in a bounded effort. I perorate with this dissertation by measuring an implementation of *Redundant Share* and by comparing the results with other distribution algorithms.





# Danksagungen

Diese Arbeit konnte nur durch die Hilfe vieler Personen entstehen, welche mir Rat und Unterstützung lieferten. Diesen Personen möchte ich hiermit danken. Einige davon möchte ich besonders erwähnen:

Zunächst ist hier mein Doktorvater Prof. Dr. math. Friedhelm Meyer auf der Heide zu nennen. Du hattest immer ein offenes Ohr für mich. Unsere Diskussionen haben mich in vielen Bereichen sehr inspiriert.

Ich danke weiterhin den (teilweise ehemaligen) Kollegen im Projekt *V:Drive*: Jun.-Prof. Dr. Ing. André Brinkmann, Michael Platt, Michael Heidebuer und Hubert Dömer. Ohne Eure Zusammenarbeit wäre dieses Speichersystem nie entstanden und ich hätte mich nie tiefergehend mit dieser Materie beschäftigt. Die Zusammenarbeit mit Euch hat mir immer Spaß gemacht.

Für das Korrektur lesen dieser Arbeit bedanke ich mich bei Dr. Marcel Rudolf Ackermann, Tanja Heinrichs und Tim Süß. Ohne Euch wären viele Fehler unentdeckt geblieben.

Ich hätte diese Arbeit auch nie ohne Unterstützung im privaten Bereich fertig stellen können. Ich bedanke mich besonders bei meinen Eltern Heidelore und Siegmund Zibulski sowie Gabriele und Wilfried Effert. Eure Unterstützung war mir immer sehr wichtig und hat mir bereits im Studium einen wichtigen Halt gegeben.

Besonders bedanken möchte ich mich bei meiner Frau Lysann Effert. Du bist der eigentliche Grund, warum ich diese Arbeit jemals fertig stellen konnte. Ich liebe Dich, Lysann.

Um diese Dissertation zu erstellen habe ich verschiedene freier Softwareprodukte verwendet. Die Arbeit wurde in  $\LaTeX$  gesetzt, zum Schreiben habe ich *Kile* unter *Linux* und *TeXShop* unter *Mac OS* verwendet. Zur Verwaltung meiner *Bibtex* Datenbank habe ich *Jabref* benutzt. Die Diagramme habe ich mittels der *matplotlib* erstellt, die restlichen Grafiken mit *Inkscape*. Dabei habe ich auf verschiedene Cliparts aus der *Open Clip Art Library* zurück gegriffen. Ich danke allen Personen, die an der Entwicklung dieser hochwertigen Produkte beteiligt waren und sie frei zur Verfügung stellen.



# Inhaltsverzeichnis

<b>1</b>	<b>Einleitung</b>	<b>1</b>
1.1	Ziel der Arbeit . . . . .	4
1.2	Aufbau dieser Arbeit . . . . .	5
1.3	Verwendete Einheiten . . . . .	7
1.4	Veröffentlichungen . . . . .	7
<b>2</b>	<b>Virtualisierung von Blockgeräten</b>	<b>9</b>
2.1	Stand der Technik . . . . .	10
2.1.1	Erasure Codes . . . . .	11
2.1.2	Cluster Blockvirtualisierungen . . . . .	17
2.1.3	Objektbasierte Speicher . . . . .	19
2.2	V:Drive . . . . .	20
2.2.1	Physikalische Komponenten . . . . .	20
2.2.2	Virtuelle Komponenten . . . . .	23
2.3	Einfluß der Kommunikation beim Einsatz lokaler Festplatten . . . . .	26
2.3.1	Architektur . . . . .	27
2.3.2	Kommunikation zwischen den Datenservern . . . . .	28
2.3.3	Messungen . . . . .	33
2.4	Redundanz innerhalb einer Speichergruppe . . . . .	41
<b>3</b>	<b>Strategien zur redundanten Datenverteilung</b>	<b>43</b>
3.1	Modell . . . . .	45
3.2	Stand der Technik . . . . .	48
3.3	Probleme der redundanten Datenverteilung . . . . .	53
3.3.1	Kapazitätseffizienz . . . . .	53
3.3.2	Trivialer Ansatz . . . . .	56
3.4	Redundant Share: Ein Algorithmus zur redundanten Datenverteilung . . . . .	62
3.4.1	Platzierung ohne Redundanz . . . . .	63
3.4.2	2-redundante Platzierung . . . . .	69
3.4.3	$k$ -redundante Platzierung . . . . .	77
3.4.4	$k$ -redundante Platzierung in Laufzeit $O(k)$ . . . . .	85

## Inhaltsverzeichnis

3.5	Peer Replikation: Lesen ohne komplette Sicht . . . . .	89
3.5.1	Modell . . . . .	90
3.5.2	Redundant Share mit unvollständiger Sicht . . . . .	90
3.5.3	Peer Replikation . . . . .	93
<b>4</b>	<b>Evaluation der Strategien</b>	<b>95</b>
4.1	Hauptspeicherverbrauch und Laufzeit . . . . .	96
4.1.1	Homogene Konfigurationen . . . . .	98
4.1.2	Heterogene Konfigurationen . . . . .	107
4.2	Fairness . . . . .	112
4.2.1	Einfluss der Anzahl der platzierten Blöcke . . . . .	113
4.2.2	Einfluss der Anzahl homogener Festplatten . . . . .	116
4.2.3	Einfluss der Anzahl heterogener Festplatten . . . . .	123
4.3	Adaptivität . . . . .	129
4.3.1	Adaptivität beim Einfügen homogener Festplatten . . . . .	130
4.3.2	Adaptivität beim Einfügen heterogener Festplatten . . . . .	139
<b>5</b>	<b>Zusammenfassung</b>	<b>143</b>
5.1	Ausblick . . . . .	146
	<b>Verzeichnisse</b>	<b>149</b>
	Abbildungsverzeichnis . . . . .	149
	Tabellenverzeichnis . . . . .	153
	Algorithmenverzeichnis . . . . .	155
	Literaturverzeichnis . . . . .	157

# 1 Einleitung

Datenspeicherung ist eins der großen und zentralen Themen unserer Zeit. Dies macht sich in vielen Bereichen deutlich. Regierungen sowie staatliche und private Institutionen unternehmen immer größere Anstrengungen, um historisch oder gesellschaftlich bedeutende Werke für die Zukunft zu sichern. Gleichzeitig versuchen Forschungseinrichtungen neues Wissen aus der Auswertung immer größerer Datenmengen zu gewinnen, die mit teilweise immenser Geschwindigkeit gesichert werden müssen.

Ein Beispiel bildet das Speichersystem *JUST* des Forschungszentrums Jülich für ihren Parallelrechner *Jugene*<sup>1</sup>. Im Jahresabschluss 2009 berichteten Mextorf und Schmidt, dass ihre Anwendungen einen Durchsatz von 64 GByte/s beim Lesen und 46 GByte/s beim Schreiben benötigen, siehe hierzu [Mextorf und Schmidt, 2009a]. Im aktuellen Ausbau *JUST3* erreicht das Speichersystem laut [Mextorf und Schmidt, 2009b] einen maximalen Durchsatz von 66 GByte/s. Die dazu verwendeten 6144 Festplatten haben eine Gesamtkapazität von 5,3 PByte. Um nicht bei Ausfall einzelner Komponenten Datenverluste erleiden zu müssen, werden verschiedene Stufen zur Wahrung von Redundanz eingesetzt.

Allerdings spielt Datenspeicherung nicht nur in solch großen Systemen eine wichtige Rolle, auch in kleinen Bereichen ist der Erhalt und die Zugänglichkeit von Daten von wachsender Bedeutung. Mittelständige und kleine Unternehmen sind immer stärker auf ihre elektronischen Daten angewiesen. Ausfälle der Speichersysteme führen teils zu tief einschneidenden Verlusten, die schnelle Verfügbarkeit stellt häufig einen wichtigen Wettbewerbsvorteil dar. Auch im privaten Bereich gewinnt der Erhalt von Daten einen zunehmenden Wert. Das liegt hauptsächlich an der wachsenden Menge digitaler Dokumente, beispielsweise Fotos oder Videos.

Allen Bereichen gemein ist, dass die letztendlich verwendete Menge an Speicherplatz ebenso wenig abzusehen ist wie die benötigte Geschwindigkeit. Will man seine Speichersysteme nicht regelmäßig ersetzen und Daten aus alten Systemen in neue migrieren, ist es wichtig ein System einzusetzen, welches mit den Anforderungen des Nutzers wachsen kann. Idealerweise sollte dies bei einigen Festplatten beginnen und bis zur Verwendung tausender Festplatten skalieren.

Diese Probleme versucht die blockbasierte Speichervirtualisierung zu lösen. Blockbasierte Geräte organisieren ihren Speicher in Blöcken, welche die kleinste Instanz für Zugriffe auf solche Geräte bilden. Fast alle Systeme zur Massendatenspeicherung, wie Festplatten, CD-Laufwerke

---

<sup>1</sup><http://www.fz-juelich.de/jsc/jugene>, am 4.11.2010

## 1 Einleitung

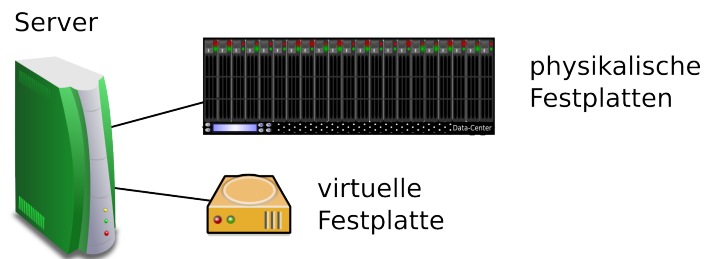


Abbildung 1.1: Virtualisierung innerhalb eines Servers



Abbildung 1.2: Mehrere Server mit gemeinsamem Speicher

und Bandlaufwerke, werden als Blockgeräte angesprochen. Im Bereich der Speichervirtualisierung werden im allgemeinen physikalische Festplatten verwendet, über welche virtuelle Festplatten abgebildet werden. Dadurch können solche virtuellen Laufwerke eine Geschwindigkeit, Größe und Verfügbarkeit erhalten, welche sich mittels einer einzelnen Festplatte nicht erreichen ließe.

Die einfachste Art solch einer Virtualisierung findet innerhalb eines einzelnen Rechners statt. Hier wird über einer Menge lokaler Festplatten eine virtuelle Festplatte erstellt, welche nur innerhalb des Rechners verfügbar ist. Abbildung 1.1 zeigt ein Beispiel einer solchen Konfiguration.

Wird ein gemeinsamer Speicher von mehreren Servern benötigt, so kann dieser durch ein *Storage Area Network (SAN)* [Vacca, 2001] bereit gestellt werden. Mittels solch eines Netzwerks können mehrere Server parallel auf die physikalischen Festplatten zugreifen, wie in Abbildung 1.2 dargestellt. Die Geschwindigkeit und die Größe des Speichers ist dabei zunächst auf eine Festplatte beschränkt.

Um diese Beschränkung zu umgehen, kann über das SAN ein virtuelles Laufwerk bereit gestellt werden. Hierzu erstellt ein Virtualisierungsserver zunächst ein virtuelles Laufwerk. Da lediglich der Virtualisierungsserver direkt auf die Festplatten zugreift, können lokale Festplatten verwendet werden. Das virtuelle Laufwerk wird dann über das SAN verfügbar gemacht, wie in Abbildung 1.3 dargestellt. Bei dieser Art der Virtualisierung ist die Geschwindigkeit des virtuellen Laufwerks auf die Leistung und den Netzwerkdurchsatz des Virtualisierungsservers beschränkt.

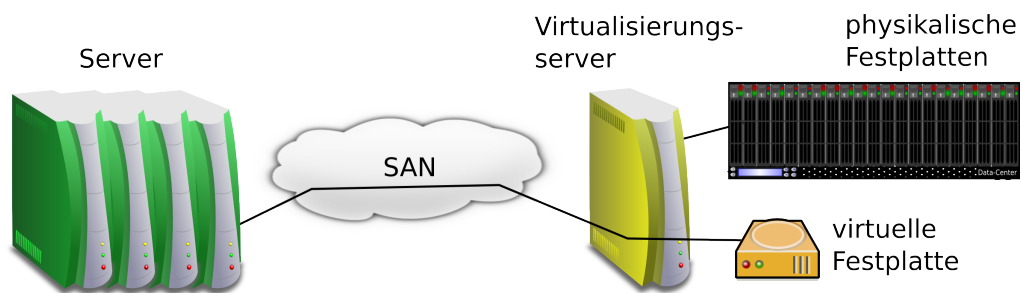


Abbildung 1.3: Mehrere Server mit gemeinsamem, zentral virtualisiertem Speicher

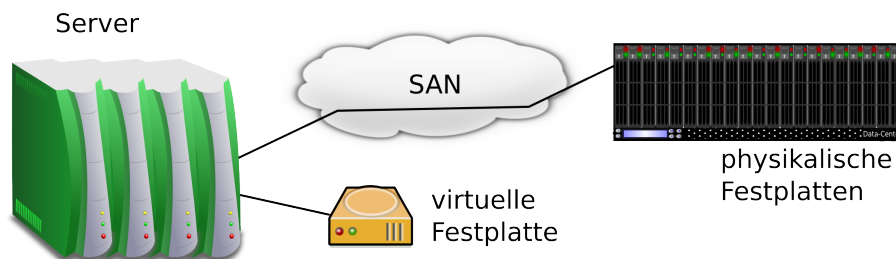


Abbildung 1.4: Mehrere Server mit gemeinsamem, verteilt virtualisiertem Speicher

Um auch diese Beschränkung aufzuheben ist es nötig, über einer Menge von gemeinsamen Festplatten ein virtuelles Laufwerk zu erstellen, welches gleichzeitig von einer beliebigen Menge von Servern zur Verfügung gestellt werden kann. Bei solch einem Speichernetzwerk müssen die einzelnen Server in der Lage sein, die Daten des virtuellen Laufwerks über die physikalischen Laufwerke zu verteilen, daher müssen auch die physikalischen Festplatten für alle Server erreichbar sein. Abbildung 1.4 zeigt ein Beispiel eines solchen Speichersystems.

Im folgenden bezeichne ich virtuelle Festplatten, welche diesen Anforderungen genügen, als *Cluster Blockgeräte* und ein Speichersystem, welches solche Laufwerke unterstützt, als *Cluster Blockgerätevirtualisierung*. Eine *Cluster Blockgerätevirtualisierung (CBV)* kann also virtuelle *Cluster Blockgeräte (CBG)* auf mehreren Servern über einem gemeinsamen Speicher abbilden. Jeder Server innerhalb einer CBV verteilt die Daten der CBG selbstständig über den unterliegenden Speicher. Für Anwendungen auf den Servern verhält sich ein CBG wie ein gemeinsames Blockgerät, welches über ein *Storage Area Network (SAN)* eingebunden ist.

Eine *CBV* stellt besondere Anforderungen an die Art, wie die Daten eines *CBGs* über die physikalischen Festplatten verteilt werden. Zunächst muss schnell entschieden werden können, auf welcher physikalischen Festplatte ein Block einer virtuellen Festplatte abgelegt werden muss. Diese Anforderung bezeichne ich als *Zeiteffizienz*. Dabei sollte die Verteilung auch möglichst wenig Hauptspeicher verwenden. Diese Anforderung bezeichne ich als *Speichereffizienz*.

Durch die Verteilung der Daten über mehrere Festplatten ist der Ausfall eines Teils des Speichers wesentlich wahrscheinlicher als beim Einsatz einer einzelnen Festplatte. Daher ist es nö-

## 1 Einleitung

tig, die Daten so zu speichern, dass auch beim Ausfall einiger Festplatten das virtuelle Laufwerk noch vollständig verfügbar ist. Die Anzahl der Festplatten, die ausfallen dürfen, ist ein Parameter des Speichersystems, welcher bei der Initialisierung festgelegt wird. Um diese Ausfalltoleranz zu gewährleisten werden zusätzliche Informationen auf den Festplatten gespeichert. Diese Anforderung bezeichne ich als *Redundanz*.

Um die physikalischen Festplatten möglichst optimal zu nutzen, sollten die Daten des virtuellen Laufwerks gleichmäßig über die physikalischen Festplatten verteilt werden. Auf diese Art sollte jede physikalische Festplatte im erwarteten Fall den gleichen Anteil der Last der virtuellen Festplatte erhalten. Bei Verwendung von Festplatten mit unterschiedlicher Kapazität, sollte jede Festplatte einen proportionalen Anteil zu ihrer Kapazität am Gesamtspeicher erhalten. So können auch unterschiedliche Festplatten optimal genutzt werden. Diese Anforderung bezeichne ich als *Fairness*. Unterstützt ein Verfahren die Verteilung über Festplatten mit unterschiedlicher Kapazität, so bezeichne ich es als *heterogen*, andernfalls als *homogen*.

Sehr häufig ist es schwer abzuschätzen, welche Menge an Speicher und welche Geschwindigkeit über die Einsatzzeit eines Speichersystems hinweg benötigt wird. Daher ist es wichtig, dass ein Speichersystem im Betrieb um Festplatten erweitert werden kann. Die Möglichkeit Festplatten zu entfernen ist wichtig, um alte oder defekte Festplatten aussortieren zu können. Um nach einer solchen Anpassung des Speichersystems alle Daten wieder finden zu können, ist es notwendig einige Daten von ihrer ursprünglichen Festplatte auf eine andere zu verschieben. Die Anzahl dieser Verschiebungen soll möglichst gering sein. Diese Anforderung bezeichne ich als *Adaptivität*.

### 1.1 Ziel der Arbeit

Ziel dieser Arbeit ist die Konzeption einer *CBV*. Diese Virtualisierung soll allen zuvor genannten Merkmalen genügen. Speziell die Einhaltung der Fairness über Festplatten mit unterschiedlicher Kapazität ist dabei ein bisher noch offenes Problem, wenn gleichzeitig die Adaptivität und die Redundanz gewährleistet werden sollen. Dies ist ein dringend zu lösendes Problem, um Speichergeräte zu entwickeln, welche auch über längere Zeit den Bedürfnissen der Endanwender genügen.

Um die Problemstellung aufarbeiten zu können stelle ich im Laufe dieser Arbeit zunächst verschiedene Virtualisierungen für Speichergeräte vor. Eine wichtige Lösung in diesem Bereich ist *V:Drive*, eine an der Universität Paderborn entwickelte Lösung zur Speichervirtualisierung. Dieses Speichersystem ist die Grundlage meines Konzepts.

Um nun eine *CBV* zu entwickeln, welche allen Anforderungen genügt, wird ein passendes Verteilungsverfahren benötigt. Es gibt Verfahren, welche über Festplatten unterschiedlicher Kapazität Daten fair verteilen können. Einige davon erlauben auch das Hinzufügen oder Entfernen



von Festplatten bei minimalen Aufwand für die Verschiebung von Daten. Allerdings ist es mit keinem dieser Verfahren möglich, die Daten redundant zu platzieren. Ich werde zeigen, dass diese Anforderung zu erfüllen auch nicht trivial ist. Wird beispielsweise eine Verfahren ohne Redundanz mehrfach über Festplatten unterschiedlicher Kapazität angewandt, so können die einzelnen Festplatten nicht optimal genutzt werden.

Mittels des Verteilungsverfahrens *Redundant Share* kann *V:Drive* oder eine andere Speichervirtualisierung so erweitert werden, dass sie alle von mir beschriebenen Anforderungen erfüllt. Innerhalb dieser Arbeit werde ich *Redundant Share* vorstellen und eine Implementierung dieses Verfahrens vermessen. Die Ergebnisse vergleiche ich mit den Ergebnissen der Vermessung von Implementierungen anderer Verfahren.

## 1.2 Aufbau dieser Arbeit

In dieser Arbeit beschäftige ich mich in Kapitel 2 mit verschiedenen Realisierungen zur Virtualisierung von blockbasierten Speichergeräten. Dazu gebe ich zunächst in Abschnitt 2.1 einen Überblick über den aktuellen Stand der Technik indem ich verschiedene Systeme zur Speichervirtualisierung vorstelle. Dabei gehe ich neben der blockbasierten auch auf die objektbasierte Virtualisierung ein. Bei einem objektorientiertem Speichergerät werden die Daten nicht in Blöcken, sondern in Objekten unterschiedlicher Größe verwaltet.

Tiefgehend behandle ich in Abschnitt 2.2 die an der Universität Paderborn entwickelte CBV *V:Drive* [Brinkmann u. a., 2004]. *V:Drive* kann CBG über einem gemeinsamen Speicher erzeugen. Dieser Speicher kann über ein *SAN* bereit gestellt werden, welches es ermöglicht, dass jeder Server direkt auf jede Festplatte zugreifen kann. Es können aber auch lokale Festplatten der Server verwendet werden. In diesem Fall wird ein eigenes *SAN* erstellt, indem die lokalen Festplatten über das Netzwerk allen Servern verfügbar gemacht werden. Die Kapazität der Festplatten kann variieren, jede Festplatte erhält gemäß ihrer Kapazität mit hoher Wahrscheinlichkeit den gleichen Anteil der gespeicherten Daten.

Ich habe Speichersysteme untersucht, welche auf Speicherknoten mit lokalen Festplatten aufbauen. In Abschnitt 2.3 gebe ich die Ergebnisse dieser Untersuchungen wieder. Dabei zeige ich, dass ein solches Speichersystem nur begrenzt gut skalieren kann. Ich zeige verschiedene Schranken für die Skalierung, abhängig von den eingesetzten Verfahren zur redundanten Datenspeicherung über mehrere Speicherknoten. Diese theoretischen Ergebnisse belege ich mit der experimentellen Untersuchung einer auf *V:Drive* aufbauenden Virtualisierung über lokalem Speicher.

Ich schließe Kapitel 2 ab, indem ich in Abschnitt 2.4 erläutere, warum sich *V:Drive* bisher nur unzureichend zur redundanten Speicherung von Daten eignet.

Dieses Problem greife ich in Kapitel 3 auf. Dazu gebe ich zunächst einen kurzen Überblick über

## 1 Einleitung

die Problemstellung und stelle die prinzipielle Arbeitsweise von pseudorandomisierten Verteilern vor. In Abschnitt 3.1 modelliere ich das Problem der redundanten Datenverteilung und definiere verschiedene Kriterien, anhand derer sich die Qualität dieser Verfahren unterscheiden lässt. Dazu spezifiziere ich die zuvor skizzierten Anforderungen an die Verteilung von Daten genauer, also die Zeiteffizienz, Speichereffizienz, Fairness, Adaptivität und Heterogenität.

In Abschnitt 3.2 gehe ich auf verschiedene pseudorandomisierte Verteiler ein. Keiner dieser Verteiler schafft es, alle in Abschnitt 2.4 gezeigten Probleme zu lösen. Speziell kann keines der Verfahren Daten redundant so platzieren, dass alle Festplatten einen Anteil der verteilten Daten gemäß ihrer Kapazität zu erwarten haben, wenn sich die Kapazitäten der Festplatten beliebig unterscheiden können.

In Abschnitt 3.3 beschreibe ich, welche Probleme bei der redundanten Platzierung von Daten bestehen. Ich betrachte dazu die Heterogenität von Festplatten, also die Frage wie weit die Kapazitäten einzelner Festplatten abweichen dürfen. Ich zeige eine Schranke für die maximale Heterogenität von Festplatten auf, wenn sie zur redundanten Datenspeicherung verwendet werden sollen. Für jedes Speichersystem innerhalb dieser Schranke ist es möglich, Daten redundant so zu platzieren, dass jede Festplatte einen Anteil der Daten gemäß ihrer Kapazität erhält. Ein Ansatz dies zu lösen könnte in der mehrmaligen Verwendung nicht redundanter Verteiler liegen. Ich werde zeigen, dass auf diesem Weg die Anforderung der Fairness nicht eingehalten werden kann.

In Abschnitt 3.4 stelle ich *Redundant Share* vor. *Redundant Share* war der erste Verteiler, der Daten pseudorandomisiert, redundant und fair über mehrere Festplatten verteilen konnte. Ich gehe auf den Algorithmus in verschiedenen Schritten ein, um die Funktionsweise des Verfahrens zu erklären. Ich zeige als erstes wie *Redundant Share* ohne Redundanz arbeitet, danach gehe ich auf die Verteilung von zwei Kopien jedes Blocks und schließlich auf die Verteilung einer beliebigen Menge an Kopien ein.

In Abschnitt 3.5 betrachte ich, wie effizient sich Daten mit *Redundant Share* wiederfinden lassen, wenn nicht die gesamte Konfiguration eines Speichersystems bekannt ist. In diesem Szenario nehme ich an, dass nur noch ein Teil der Festplatten bekannt ist, über die zuvor Daten verteilt wurden. Weiterhin nehme ich an, dass nicht bekannt ist, welche Festplatten fehlen. Ich stelle mit *Peer Replikation* eine Abwandlung von *Redundant Share* vor, welche unter diesen Bedingungen effizient alle vorhandenen Daten wieder findet.

In Kapitel 4 belege ich mittels einer Implementierung von *Redundant Share*, dass sich die Ergebnisse aus der Theorie auf die Praxis übertragen lassen. Dazu vergleiche ich die Implementierung von *Redundant Share* mit Implementierungen anderer Verfahren. Zunächst gehe ich in Abschnitt 4.1 auf den Hauptspeicherverbrauch und die Laufzeit der unterschiedlichen Implementierungen ein. Danach vergleiche ich in Abschnitt 4.2 die Güte der Verteilung der Algorithmen. Ich schließe die Vermessungen ab, indem ich in Abschnitt 4.3 betrachte, wie viele Daten verschoben werden, wenn ein Speichersystem um neue Festplatten erweitert wird.

Tabelle 1.1: Verwendete Einheiten

Name	Abkürzung	Wert
Kilo	K	$2^{10}$
Mega	M	$2^{20}$
Giga	G	$2^{30}$
Terra	T	$2^{40}$
Peta	P	$2^{50}$
Exa	E	$2^{60}$

Abschließend gebe ich in Kapitel 5 eine Zusammenfassung meiner Ergebnisse und einen Ausblick auf mögliche Arbeiten, welche auf den hier vorgestellten Ergebnissen aufbauen könnten.

## 1.3 Verwendete Einheiten

Ich spreche in dieser Arbeit sehr häufig über Kapazitäten und Durchsatz von Festplatten oder auch Netzwerken. Dabei verwende ich Einheiten wie GByte oder MBit. In diesen Fällen spreche ich immer von Einheiten zur Basis 2, zu erkennen an dem Großbuchstaben. In der Literatur werden diese Werte auch manchmal als GiByte oder MiBit bezeichnet. Tabelle 1.1 gibt einen Überblick über die gängigen Größen.

## 1.4 Veröffentlichungen

Teile dieser Arbeit wurde bereits in früheren Veröffentlichungen behandelt, in einigen Bereichen auch tiefergehend. Diese Arbeiten sind im einzelnen:

**Brinkmann u. a. 2005** BRINKMANN, A. ; EFFERT, S. ; HEIDEBUER, M. ; VODISEK, M.: Distributed MD. In: *Proceedings of the 3rd international workshop on Storage Network Architecture and Parallel I/Os (SNAPI)*. Washington, DC, USA : IEEE Computer Society, 2005, S. 81 – 88

**Brinkmann u. a. 2006a** BRINKMANN, A. ; EFFERT, S. ; HEIDEBUER, M. ; VODISEK, M.: Influence of Adaptive Data Layouts on Performance in dynamically changing Storage Environments. In: *Proceedings of the 14th Euromicro conference on Parallel, Distributed and network based Processing (PDP)*. Washington, DC, USA : IEEE Computer Society, 2006, S. 155–162

## 1 Einleitung

- Brinkmann u. a. 2006b** BRINKMANN, A. ; EFFERT, S. ; HEIDEBUER, M. ; VODISEK, M.: Realizing Multilevel Snapshots in Dynamically Changing Virtualized Storage Environments. In: *Proceedings of the 5th IEEE International Conference on Networking (ICN)*. Berlin, Germany : Springer Verlag, 2006
- Brinkmann u. a. 2007** BRINKMANN, A. ; EFFERT, S. ; MEYER AUF DER HEIDE, F. ; SCHEIDLER, C.: Dynamic and Redundant Data Placement. In: *Proceedings of the 27th IEEE International Conference on Distributed Computing Systems (ICDCS)*. Washington, DC, USA : IEEE Computer Society, 2007
- Brinkmann und Effert 2007a** BRINKMANN, A. ; EFFERT, S.: Inter-node Communication in Peer-to-Peer Storage Clusters. In: *Proceedings of the 24th IEEE IEEE conference on Mass Storage Systems and Technologies (MSST)*. Washington, DC, USA : IEEE Computer Society, 2007, S. 257–262
- Brinkmann und Effert 2007b** BRINKMANN, A. ; EFFERT, S.: Snapshots and Continuous Data Replication in Cluster Storage Environments. In: *Proceedings of the 4th international workshop on Storage Network Architecture and Parallel I/Os (SNAPI)*. Washington, DC, USA : IEEE Computer Society, 2007, S. 3–10
- Brinkmann und Effert 2008a** BRINKMANN, A. ; EFFERT, S.: Data Replication in P2P Environments. In: *Proceedings of the 20th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. New York, NY, USA : ACM, 2008, S. 191–193
- Brinkmann und Effert 2008b** BRINKMANN, A. ; EFFERT, S.: Redundant Data Placement Strategies for Cluster Storage Environments. In: *Proceedings of the 12th international conference On Principles Of DIstributed Systems (OPODIS)*. Berlin, Germany : Springer-Verlag, 2008, S. 551–554

## 2 Virtualisierung von Blockgeräten

Ziel dieses Kapitels ist die Betrachtung von Blockgerätevirtualisierungen. Bei dieser Form der Virtualisierung werden virtuelle Blockgeräte zumeist als virtuelle Festplatten über einem vorhandenen Speicher erzeugt. Die virtuellen Blockgeräte werden in der Regel auf andere Blockgeräte, z.B. physikalische Festplatten, abgebildet. Es gibt daneben aber auch noch verschiedene andere Möglichkeiten, Speicher zu virtualisieren. Ein prominentes Beispiel stellen Dateisysteme dar, welche Dateien und Verzeichnisse auf ein Blockgerät abbilden.

Innerhalb meiner Forschung habe ich mich mit der Konzeption virtueller Blockgeräte beschäftigt. Diese Geräte bilden heute die am weitesten verbreitete Form persistenten Speichers<sup>1</sup>. Die älteste noch verwendete Variante dieser Art von Virtualisierung ist die Partitionierung von Festplatten, wodurch es möglich ist eine große Festplatte als mehrere kleine Festplatten erscheinen zu lassen.

Heutzutage betrachtet man Blockvirtualisierung eher auf umgekehrte Weise: Über einer Menge physikalischer Festplatten werden eine oder mehrere virtuelle Festplatten abgebildet. Prominentester Vertreter dieser Form der Virtualisierung sind *RAID*-Systeme [Patterson u. a., 1988] und andere auf *Erasure Codes* aufbauende Verfahren, wie ich sie in Abschnitt 2.1 beschreibe.

Anwendungen können auf solche virtualisierten Festplatten direkt zugreifen, sie folgen den gleichen Schnittstellen wie physikalische Festplatten. Im Falle einer CBV stehen die virtualisierten Festplatten auf verschiedenen Servern parallel zur Verfügung, ähnlich einem *Storage Area Network*. Aufsetzend auf diesem Speicher können Cluster Dateisysteme wie GPFS [Schmuck und Haskin, 2002], GFS2 [Preslan u. a., 2000; Whitehouse, 2007] oder OCFS2 [Fasheh, 2006] zum Einsatz kommen. Ohne solch einen gemeinsamen Speicher müssten sich die Dateisysteme selbst um den Austausch der Daten kümmern. Vertreter dieser Art von Dateisystemen sind beispielsweise Lustre [Schwan, 2003], Ceph [Weil u. a., 2006a], Farsite [Adya u. a., 2002] und OceanStore [Kubiatowicz u. a., 2000; Rhea u. a., 2003].

In Abschnitt 2.1 beschreibe ich verschiedene Blockgerätevirtualisierungen. Dazu gehe zunächst auf *Erasure Codes* ein. Das sind Verfahren, um Prüfsummen über Daten zu erstellen, mittels derer ausgefallene Festplatten wiederhergestellt werden können. Darauf aufbauend betrachte ich verschiedene Systeme zur Blockvirtualisierung. Ich beschränke mich dabei auf Systeme, welche

---

<sup>1</sup>Als persistent bezeichnet man Speichergeräte, deren Daten auch über einen Stromverlust hinweg erhalten bleiben.

## 2 Virtualisierung von Blockgeräten

im wissenschaftlichen Umfeld evaluiert wurden und verzichte auf kommerzielle Systeme, über deren Verteilungsverfahren keine frei verfügbaren Daten existieren. Besonderen Wert lege ich auf Cluster Blockgerätevirtualisierungen, welche virtuelle Blockgeräte auf einer skalierbaren Anzahl von Servern gleichzeitig zur Verfügung stellen. Über die blockbasierte Speichervirtualisierung hinausgehend betrachte ich auch objektbasierte Speichervirtualisierungen.

An der Universität Paderborn wurde innerhalb des Projekts *V:Drive* eine CBV entwickelt. *V:Drive* erlaubt es, Cluster Blockgeräte über einem gemeinsamem Speicher auf unterschiedlichen Rechnern zu nutzen. In Abschnitt 2.2 stelle ich diese CBV und ihre wichtigsten Merkmale vor.

In Abschnitt 2.3 gehe ich auf CBV über *Storage Bricks* ein. *Storage Bricks* sind handelsübliche Server mit lokalen Festplatten. Ich zeige, wie ein Speichersystem über solche *Storage Bricks* skalieren kann, wenn verschiedene Verfahren zur redundanten Speicherung der Daten verwendet werden. Die theoretischen Ergebnisse belege ich mit der Vermessung einer Implementierung eines solchen Speichersystems auch in der Praxis.

Dieses Kapitel schließe ich in Abschnitt 2.4 mit der Betrachtung der speziellen Anforderungen der redundanten Datenplatzierung innerhalb einer CBV. Diese Anforderungen greife ich dann im folgenden Kapitel 3 auf.

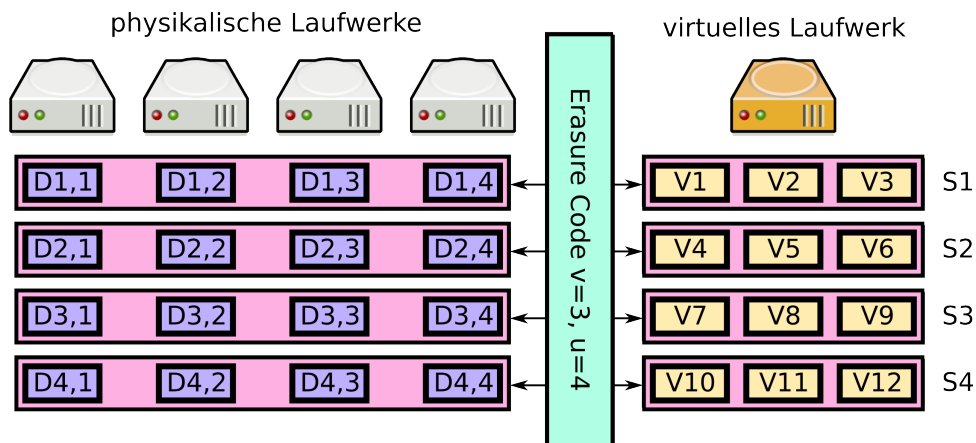
### 2.1 Stand der Technik

Es gibt verschiedene Blockvirtualisierungen, welche über einer Menge physikalischer Festplatten virtuelle Festplatten abbilden. In diesem Abschnitt stelle ich verschiedene solcher Virtualisierungen vor.

Zunächst werde ich in Abschnitt 2.1.1 auf *Erasur Codes* eingehen. Diese Verfahren können verwendet werden, um Daten redundant über Festplatten zu verteilen. Dazu werden auf verschiedenen Wegen Prüfsummen gebildet, welche zusammen mit den Daten auf unterschiedlichen Festplatten abgelegt werden.

Darauf aufbauend stelle ich in Abschnitt 2.1.2 verschiedene Ansätze zur Blockgerätevirtualisierung in Speichernetzen vor. Diese Systeme erlauben es, virtuelle Laufwerke zu erstellen, welche über mehrere Server zugreifbar sind.

In Abschnitt 2.1.3 werde ich objektbasierte Speichergeräte einführen und verschiedene Lösungen für ihre Virtualisierung in Speichernetzen vorstellen. Ich betrachte diese Art der Geräte, weil sie eine zunehmende Bedeutung im Bereich der Datenspeicherung finden. Da ich mich innerhalb dieser Arbeit allerdings mit der Blockgerätevirtualisierung beschäftige, werde ich auf diese Art des Speicher nicht weiter eingehen.

Abbildung 2.1: Erasure Codierung eines virtuellen Laufwerks mit  $v = 3$  und  $u = 4$ 

### 2.1.1 Erasure Codes

Mittels *Erasure Codes* [Mense, 2008] lassen sich Daten redundant über Festplatten verteilen. Bei einem *Erasure Code* wird eine Menge von  $v$  Blöcken derart auf  $u \geq v$  Blöcke erweitert, so dass später die ursprüngliche Nachricht aus beliebigen  $r \leq u$  Blöcken wieder hergestellt werden kann. Im Idealfall gilt  $r = v$ , die ursprünglichen Daten können also aus beliebigen  $v$  Blöcken wieder hergestellt werden. Codes, für die  $r = v$  gilt, heißen auch *Maximum Distance Separable Codes (MDS Codes)*.

Um *Erasure Codes* zur Blockgerätevirtualisierung zu verwenden, wird das virtuelle Laufwerk zunächst in *Chunks* aufgeteilt. Dabei hat jedes *Chunk* die gleiche Größe. Jeweils  $v$  aufeinander folgende *Chunks* bilden einen *Stripe*. Mittels des jeweiligen *Erasure Codes* werden dann  $u$  kodierte *Chunks* berechnet, welche den kodierte *Stripe* bilden. Diese kodierte *Chunks* werden auf unterschiedlichen Festplatten gespeichert. Ich bezeichne  $v$  auch als Breite des *Stripes* und  $u$  als Breite des kodierte *Stripes*.

Abbildung 2.1 zeigt ein Beispiel eines solchen virtuellen Laufwerks für  $v = 3$  und  $u = 4$ . Das virtuelle Laufwerk besteht aus zwölf *Chunks* gruppiert in vier *Stripes* mit jeweils drei *Chunks*. Mittels eines *Erasure Codes* wird zu jedem *Stripe* ein kodierte *Stripe* mit vier kodierte *Chunks* berechnet. Diese kodierte *Chunks* werden auf vier unterschiedlichen Festplatten abgelegt.

Die verbreitetsten *Erasure Codes* beruhen auf den 1988 von Patterson, Gibson und Katz vorgestellten *RAID*-Codes [Patterson u. a., 1988]. *RAID* stand ursprünglich für *Redundant Array of Inexpensive Disks*. Mit wachsender Verbreitung und der Tatsache, dass *RAID* vermehrt auch im Bereich des High-End-Speichers eingesetzt wurde, passte der Begriff *Inexpensive* (auf deutsch: billig) nicht mehr. Daher wurde *RAID* neu definiert als *Redundant Array of Independent Disks*.

*RAID*-Verfahren arbeiten über homogenen Festplatten, also haben alle Festplatten die gleiche Kapazität. Den Zusammenschluss mehrerer Festplatten zu einem *RAID*-Laufwerk bezeichnet

## 2 Virtualisierung von Blockgeräten

man auch als *RAID-Verbund*. Die Anzahl der Festplatten innerhalb eines *RAID-Verbundes* lässt sich nur mit sehr viel Aufwand anpassen, in der Regel ist dazu die Umplatzierung fast jedes kodierte *Chunks* nötig [Gonzalez und Cortes, 2004].

Paterson u. a. haben in [Patterson u. a., 1988] verschiedene Verfahren vorgestellt. Diese Verfahren werden auch als *RAID-Level* bezeichnet. Die Autoren haben sechs Verfahren vorgestellt, welche am besten unter den Namen *RAID 0* bis *RAID 5* bekannt sind. Praktisch relevant sind davon heute nur noch *RAID 0*, *1*, *4* und *5*. *RAID 2* und *3* arbeiten nicht auf Blockebene. Dadurch sind speziell bei kleinen Anfragen große Einbußen in der Performanz der virtuellen Laufwerke zu verzeichnen.

### RAID 0

Bei einer *RAID 0*-Kodierung (auch *striping* genannt) werden mehrere Laufwerke miteinander verbunden, um ein Maximum an Performance und Kapazität aus den unterliegenden Festplatten zu erhalten. Die *Chunks* werden reihum über die physikalischen Festplatten verteilt. Der *Chunk* des virtuellen Laufwerks  $j$  wird also auf der Festplatte  $j \bmod u$  gespeichert, wenn  $u$  Festplatten verwendet werden. Fällt eine Festplatte aus, so sind mit hoher Wahrscheinlichkeit auch Daten des virtuellen Laufwerks verloren.

*RAID 0* ist ein besonderer Fall eines *Erasure Codes*. Hier gilt  $u = v$ , daher dürfen keine Festplatten ausfallen. Laut meiner oben angegebenen Definition eines *Erasure Codes* ist dies gültig. Andere Definition, wie z.B. in [Mense, 2008], verlangen dass ein Erasure den Ausfall mindestens einer Festplatte ausgleichen können muss.

### RAID 1

*RAID 1* (auch *mirroring* oder auf Deutsch *Spiegelung* genannt) legt jeden *Chunk* der virtuellen Festplatte auf jeder physikalischen Festplatte ab. Bei  $u$  verwendeten Festplatten ergibt dies einen *Erasure Code* mit  $v = 1$ . So wird ein Höchstmaß an Redundanz erreicht. Dies geht aber zu Lasten des verfügbaren Speichers und der Geschwindigkeit des virtuellen Laufwerks.

### RAID 4

Mittels *RAID 4* werden jeweils  $v$  *Chunks* auf  $u = v + 1$  kodierte *Chunks* erweitert. Die ersten  $v$  *Chunks* sind identisch der unkodierten *Chunks*, wie auch bei einem *RAID 0*. Der zusätzliche kodierte *Chunk* enthält eine Prüfsumme, welche sich durch eine `xor`-Verknüpfung der unkodierten *Chunks* des *Stripes* berechnet wird.

Beim Ausfall einer Festplatte können ihre Daten durch die verbleibenden Festplatten und die Redundanzfestplatte wieder hergestellt werden. Ist keine Festplatte ausgefallen, so liefert das



entstandene virtuelle Laufwerke eine hohe Leseperformance sowie eine hohe Performance beim Schreiben großer sequentieller Datenmengen. Beim Speichern von kleinen Datenmengen kann die Performance auf die Geschwindigkeit einer einzelnen Festplatte absinken. Dies wird auch als *Small-Write-Problem* [Stodolsky u. a., 1993] bezeichnet.

### RAID 5

*RAID 5* arbeitet sehr ähnlich zu *RAID 4*, allerdings gibt es hier keine explizite Redundanzfestplatte. Statt dessen werden die *Chunks* mit den Prüfsummen reihum auf einer der  $u$  Festplatten gespeichert. Der *Chunk*  $i$  der Redundanzfestplatte aus *RAID 4* wird dabei auf der Festplatte  $i \bmod n$  abgelegt, die übrigen  $u - 1$  Festplatten speichern die Daten. Auf diese Weise wird die erhöhte Schreiblast, welche bei *RAID 4* der Redundanzfestplatte zukommt, über alle Festplatten verteilt.

### RAID 6

Neben den von Paterson u. a. in [Patterson u. a., 1988] definierten Verfahren hat sich mittlerweile *RAID 6* als Begriff durchgesetzt. Im Gegensatz zu *RAID 0* bis *RAID 5* beschreibt *RAID 6* jedoch keine Implementierung. Die *Storage Networking Industry Association (SNIA)* definiert *RAID 6* in [Storage Networking Industry Association, 2010]<sup>2</sup> wie folgt:

[Storage System] Any form of RAID that can continue to execute read and write requests to all of a RAID array's virtual disks in the presence of any two concurrent disk failures.

Several methods, including dual check data computations (parity and Reed Solomon), orthogonal dual parity check data and diagonal parity have been used to implement RAID Level 6.

Ein virtuelles Laufwerk, welches *RAID 6* konform ist, muss bei Ausfall von zwei Festplatten weiter arbeiten können. Ich stelle einige Verfahren vor, welche dies erlauben. Innerhalb meiner Beschreibungen gehe ich davon aus, dass dedizierte Festplatten zur Speicherung von Prüfsummen und Daten verwendet werden, wie dies bei *RAID 4* der Fall ist. Alle Verfahren können die Prüfsummen über alle Festplatten verteilen, wie für *RAID 5* beschrieben. Dies würde die Erläuterung der Verfahren jedoch unnötig komplizierter machen.

### Evenodd Codes und Row Diagonal Parity

*Evenodd-Codes* [Blaum u. a., 1994] sind eine Erweiterung des *RAID 4* Codes. Neben den un-kodierten  $v$  *Chunks* enthält der kodierte *Stripe* zunächst einen weiteren kodierten *Chunk* mit

<sup>2</sup><http://www.snia.org/education/dictionary/>, am 4.11.2010

## 2 Virtualisierung von Blockgeräten

einer Prüfsumme über die  $v$  unkodierten *Chunks*. Mittels dieser Prüfsummen kann eine defekte Festplatte rekonstruiert werden. Der Ausfall einer zweiten Festplatte wird kompensiert indem ein weiteres kodierte *Chunk* dem kodierten *Stripe* hinzugefügt wird. Hier wird eine Prüfsumme abgeleitet, welche diagonal über  $v + 1$  *Stripes* erzeugt und mit einer Justierungskomponente verknüpft wird. Damit die Berechnung funktioniert muss  $v$  eine Primzahl sein. Diese Einschränkung lässt sich aber aufheben, indem imaginäre Festplatten in die Berechnungen aufgenommen werden. Solch imaginäre Festplatten enthalten für jeden *Chunk* einen festen Wert, beispielsweise 0. Für den so entstehenden *Erasure Code* gilt  $u = v + 2$  und er kann zwei Festplattenausfälle ausgleichen.

*Row Diagonal Parity* [Corbett u. a., 2004] arbeitet ähnlich zu *Evenodd Codes*. Allerdings wird hier die erste Prüfsumme in die Berechnung der zweiten Prüfsumme mit aufgenommen, wodurch  $v + 1$  eine Primzahl sein muss. Darüber hinaus benötigt dieser Code keine Justierungskomponente.

### Liberation Codes

*Liberation Codes* [Plank, 2007, 2008] wurden 2007 von James S. Plank als frei verfügbare *RAID 6* konforme Datenverteilung entwickelt. Neben  $v$  Festplatten zur Speicherung der Daten gibt es wieder eine Festplatte, welche eine Prüfsumme nach *RAID 4* aufnimmt. Eine weitere Festplatte speichert die zweite Prüfsumme. Um dieses zu berechnen werden ähnlich zu den *Evenodd Codes* und dem *Row Diagonal Parity* die Diagonalen verwendet, hier aber nur noch mit einem weiteren Feld per xor verknüpft. Auf diese Art braucht der Algorithmus weniger xor-Operation zum kodieren und dekodieren. Auch *Liberation Codes* können zwei Festplattenausfälle bei  $u = v + 2$  ausgleichen.

### STAR

*STAR* [Huang und Xu, 2008] arbeitet sehr ähnlich zu *Row Diagonal Parity*, erlaubt allerdings Ausfälle von bis zu drei Festplatten. Hierzu wird dem kodierten *Stripe* ein weiterer *Chunk* hinzugefügt. Die Prüfsummen der ersten beiden Redundanzfestplatten berechnen sich wie zuvor. Für die Prüfsumme der dritten Redundanzfestplatte werden die Diagonalen in umgekehrter Richtung über die *Stripes* gebildet. Für den so entstehenden *Erasure Code* gilt also  $u = v + 3$ .

### Reed Solomon Codes

*Reed Solomon Codes* [Reed und Solomon, 1960] erlauben den Ausfall von  $k < n$  Festplatten und kodieren in  $u$  *Chunks*  $v = u - k$  unkodierte *Chunks*. Um dies zu realisieren verwendet der *Reed Solomon Code* die Arithmetik eines *Galois-Körper*, z.B.  $\mathbb{F}_{256}$ . Dies würde *Chunks* mit einer Größe von einem Byte entsprechen, es kann aber auch ein entsprechend größerer Körper

verwendet werden. Zunächst wird ein fester Vektor  $U = (u_0, \dots, u_{n-1})$  mit  $u_i \in \mathbb{F}_{256}$  festgelegt. Zum Speichern der Daten werden nun die  $v$  *Chunks* eines *Stripes* als Eingabe verwendet, ich bezeichne diese Blöcke als  $s_0, \dots, s_{m-1}$ . Der Inhalt, den  $d_i$  speichern soll, berechnet sich nun aus  $\sum_{j=0}^{m-1} s_j u_i$ , wobei die Multiplikation und die Addition auch wieder Operationen auf  $\mathbb{F}_{256}$  sind. Mittels einer *Lagrange-Interpolation* [von zur Gathen und Gerhard, 1999] kann dann die ursprüngliche Nachricht aus beliebigen  $v$  kodierten *Chunks* wieder hergestellt werden.

Der Nachteil von *Reed Solomon Codes* liegt in dem Rechenaufwand zum kodieren und dekodieren von *Stripes*. Blömer u. a. konnten in [Blömer u. a., 1995] zeigen, dass sich die Berechnung mittels Cauchy-Matrixen auf reine xor-Operationen reduzieren lassen.

### Lincoln Erasure Code

Cooley u. a. beschreiben in [Cooley u. a., 2003] mit *Lincoln Erasure Code (LEC)* ein Verfahren, bei welchem ein bipartiter Graph eingesetzt wird, um die Prüfsummen zu erzeugen. Dieses Verfahren baut auf den von Luby in [Luby, 2002] vorgestellten Verfahren auf. Ein auf *LEC* basierendes Verfahren stellt die *Datenchunks* und die *Prüfchunks* gegenüber. Die  $u$  *Datenchunks* halten dabei die ursprüngliche Nachricht, die restlichen  $v - u$  *Chunks* des kodierten *Stripes* werden für Prüfsummen verwendet. Zufällig verteilt werden Kanten zwischen *Datenchunks* und *Prüfchunks* hinzugefügt, bis der Graph einen minimalen Grad erreicht hat. Die Prüfsumme, die in einem *Prüfchunk* gespeichert werden soll, wird berechnet, indem ein xor über alle mit ihm verbundenen *Datenchunks* gebildet wird.

Wie einfach zu sehen ist bildet *LEC* keinen *MDS Code*. Ferner ist der Grad der möglichen Ausfälle schwer zu bestimmen, da die Prüfsummen hier über ein Zufallsexperiment gebildet werden.

### WEAVER Codes

Die 2005 von Hafner in [Hafner, 2005] vorgestellten *WEAVER Codes* erlauben eine redundante Speicherung der Daten, so dass bis zu zwölf Festplattenausfälle toleriert werden können. Diese Verfahren sind darauf ausgelegt, schnell mittels einfacher Hardware berechnet werden zu können. Im Gegenzug verzichten diese Verfahren auf die *MDS*-Eigenschaft. Um bei einer Größe der *Stripes* von  $v$  bis zu  $k$  Ausfälle tolerieren zu können, werden also mehr als  $v + k$  Festplatten benötigt. Dementsprechend muss  $u$  bei diesen Verfahren groß genug sein, um die Daten und Prüfsummen zu verteilen. Die genaue Größe unterscheidet sich bei den einzelnen Codes.

Ähnlich zu den *Lincoln Erasure Codes* berechnet auch Hafner die Prüfsummen immer nur über einen Teil der Daten innerhalb eines *Stripes*. Allerdings wählt er die Kombination so, dass mittels einfacher Verschiebungen der Daten innerhalb eines *Stripes* die Prüfsummen berechnet werden können. Laut Hafner können die so entstehenden Prüfsummen sehr einfach in einem *RAID*-Controller berechnet werden.

### Heterogenes RAID

Cortes und Labarta haben in [Cortes und Labarta, 2000] ein Verfahren vorgestellt, um ein *RAID 0* über Festplatten unterschiedlicher Kapazität zu erzeugen. Die Autoren haben innerhalb des virtuellen Laufwerks verschiedene Breiten der *Stripes* gewählt, um den unterschiedlich großen Festplatten gerecht zu werden. Um eine gleichmäßigere Geschwindigkeit über das gesamte virtuelle Laufwerk zu erhalten, haben sie die *Stripes* unterschiedlicher Breite über das virtuelle Laufwerk verteilt.

Cortes und Labarta haben in [Cortes und Labarta, 2001] auch einen Algorithmus für *RAID 5* Laufwerke über heterogenen Festplatten vorgestellt. Wieder verwenden sie eine variable Breite der *Stripes*. Allerdings haben alle *Stripes* eine Breite der Form  $2^i + 1$  für ganzzahlige  $i$ . Mit  $j$  als größtes verwendetes  $i$  (also  $1 \leq i \leq j$ ) melden sie das Laufwerk mit  $2^j + 1$  als Breite der *Stripes* an. So können weiterhin die Optimierungen des Betriebssystems zum Umgehen des *Small-Write-Problems* genutzt werden. Es können also immer ganze *Stripes* auf einmal geschrieben werden.

Aufbauend auf diesen Arbeiten haben Cortes und Gonzales in [Gonzalez und Cortes, 2008] weitere Verfahren zur Datenplatzierung über heterogenen Festplatten vorgestellt. Diese Verfahren erlauben auch das Nachträgliche Hinzufügen von Festplatten. Diese Verfahren arbeiten dabei abhängig der Entwicklung des Speichersystems. Je nachdem wie häufig das Speichersystem angepasst wurde, wird eine steigende Anzahl an Verteilungsfunktionen benötigt.

### Hierarchische Verfahren

Es ist möglich, die zuvor vorgestellten Verfahren zu kombinieren. So sind beispielsweise *RAID 10* Laufwerke sehr verbreitet. Hier wird über mehreren *RAID 1* Laufwerke ein *RAID 0* Laufwerk erstellt, um die Vorteile der Geschwindigkeit und der erhöhten Datensicherheit zu kombinieren.

Hierarchische Speichersysteme verwenden verschiedene Laufwerke, um Daten z.B. gemäß ihres Werts oder ihrer Popularität (Häufigkeit der Nutzung) zu speichern. Wilkes u. a. beschreiben in [Wilkes u. a., 1996] *HP AutoRAID*. In diesem hierarchischen Verfahren werden Daten abhängig von der Häufigkeit der Nutzung auf einem *RAID 1* oder *RAID 5* Speicher abgelegt.

### Declustered RAID

Holland und Gibson haben in [Holland und Gibson, 1992] untersucht, wie sich ein *RAID*-System verhält, bei dem die Anzahl der Festplatten und die Breite der kodierten *Stripes* unabhängig voneinander sind. Um die Redundanz der verschiedenen *RAID*-Level weiter garantieren zu können, müssen bei einer Breite der kodierten *Stripes* von  $u$  eine Menge von  $n \geq u$  Festplatten verwendet werden. Danach werden die einzelnen *Stripes* hintereinander platziert, falls

der kodierte *Stripe*  $i$  also auf der Festplatte  $d_j$  endet, so beginnt *Stripe*  $i + 1$  auf der Festplatte  $d_{j+1 \bmod n}$ .

## 2.1.2 Cluster Blockvirtualisierungen

Die zuvor vorgestellten *RAID*-Verfahren liefern eine Verteilung von Daten über Festplatten und in den meisten Fällen eine Berechnungsvorschrift für Prüfsummen. Dabei sind diese Verfahren darauf ausgelegt, innerhalb genau eines *RAID*-Controllers oder Software-*RAID*<sup>3</sup> berechnet zu werden. Wie in der Einleitung dieser Arbeit festgelegt arbeitet eine CBV serverübergreifend. Die CBG existieren auf mehreren Servern, welche ihre Daten über den unterliegenden Speicher verteilen.

### Logical Volume Manager

Ein *Logical Volume Manager (LVM)* (z.B. [Kofler, 2009], Seite 681ff) bietet die Möglichkeit physikalische Festplatten in Speichergruppen zu gruppieren und daraus virtuelle Festplatten zu erstellen. Die Art der Verteilung der Daten ist dabei zunächst nicht festgelegt. Ziel der virtuellen Laufwerke ist es, die Größe im Betrieb erhöhen zu können, indem weitere physikalische Festplatten hinzugenommen werden können.

Ein weit verbreiteter *LVM* ist beispielsweise der ursprünglich von *Sistina* entwickelte und mittlerweile in den Linux-Kernel übernommene *LVM2*. Dieser unterstützt zur Verteilung der Daten neben der einfachen Konkatenation der physikalischen Festplatten auch eine Verteilung gemäß *RAID 0*. Weiterhin lassen sich von einem virtuellen Laufwerk zur Laufzeit Snapshots erzeugen. Snapshots sind eine Technik, um den Stand eines Laufwerks zu einem bestimmten Zeitpunkt zu sichern.

*Logical Volume Manager* arbeiten nur innerhalb eines Rechners. Damit sind die virtuellen Laufwerke nicht serverübergreifend als CBG verfügbar.

### Petal

Lee und Thekkath stellten mit *Petal* [Lee und Thekkath, 1996] die erste Blockvirtualisierung vor, welche über mehrere Server verteilt funktionierte. Dazu verwenden die Autoren eine Menge von Servern mit lokalen Festplatten. Werden Festplatten hinzugefügt oder entfernt, so werden nur innerhalb der Server Daten umverteilt.

Clients, welche auf die Speicherknoten zugreifen, verwenden spezielle Treiber. Diese Treiber sorgen dafür, dass virtuelle Laufwerke, welche über die Speicherknoten abgebildet werden,

<sup>3</sup>Als Software-*RAID* bezeichnet man ein *RAID*-System, welches ohne spezielle Hardware durch Software innerhalb eines Server realisiert wird.

## 2 Virtualisierung von Blockgeräten

auf den Clients als normale Blockgeräte verwendet werden können. Die Daten eines virtuellen Laufwerks werden immer auf zwei unterschiedlichen Servern abgelegt, so dass *Petal* den Ausfall eines Servers ausgleichen kann.

### FAB

Frølund u. a. haben die Speichervirtualisierung *Federated Array of Bricks (FAB)* [Frølund u. a., 2003; Saito u. a., 2004] vorgestellt. Ein Speichersystem setzt sich hier aus *Storage Bricks* zusammen, das sind Server mit eigener CPU, Hauptspeicher und lokalen Festplatten. Jeder der Server ist gleichberechtigt und stellt jedes virtuelle Laufwerk bereit. Die Laufwerke werden als Blockgeräte den Clients zu Verfügung gestellt, normalerweise über das *iSCSI*-Protokoll (siehe [Satran u. a., 2004]).

Virtuelle Laufwerke zerlegt *FAB* in mehrere Segmente gleicher Größe. Die Daten jedes Segments werden redundant über die verschiedenen *Storage Bricks* verteilt. Dazu werden entweder Kopien oder *Erasur*-kodierte Daten abgelegt. Die Auswahl der *Storage Bricks*, die als physikalischer Speicher eines Segments verwendet werden, ist ein Zufallsprozess, wobei geringer belastete Server bevorzugt werden. Die Platzierung wird dann in einer Tabelle gespeichert. Um diese Tabelle klein zu halten, verwendet *FAB* relativ große Segmente. In der Veröffentlichung von 2003 [Frølund u. a., 2003] arbeiten die Autoren mit einer Segmentgröße von 8 GByte.

### Slice

Einen anderen Ansatz haben Anderson u. a. in [Anderson u. a., 2002] vorgestellt. Ähnlich zu den *Storage Bricks* bei *FAB* verwenden sie für ihre Speicherarchitektur *Slice* lokale Festplatten in verschiedenen Servern. Zugriffe auf ein virtuelles Laufwerk werden an einen Proxy gerichtet, welcher jede Anfrage an den Server weiterreicht, an den der entsprechenden Speicher angeschlossen ist. Dieser Proxy kann beispielsweise auf dem Netzwerkschicht des Speichernetzes implementiert werden, um einen effizienten Zugriff zu gewähren.

### Parallax

Meyer u. a. haben in [Meyer u. a., 2008] *Parallax*, eine verteilte Verwaltung von virtuellen Laufwerken für virtuelle Maschinen, vorgestellt. Mehrere Server teilen sich dabei Zugriff auf ein gemeinsames Blockgerät, beispielsweise eine Festplatte, welche von allen Servern über ein *SAN* erreichbar ist.

Auf jedem Server läuft eine virtuelle Maschine, welche für die Bereitstellung virtueller Laufwerke für andere virtuelle Maschinen auf dem gleichen Server verantwortlich ist. Ein virtuelles Laufwerk kann dabei immer nur von einer virtuellen Maschine gleichzeitig benutzt werden.

Der Vorteil von *Parallax* gegenüber einem gemeinsamen *Network Attached Storage (NAS)* [Gibson und Van Meter, 2000; Troppens u. a., 2007] mit Protokollen wie CIFS [Hertel, 2003] oder NFS [Callaghan, 1999; Shepler u. a., 2000] liegt in der Anpassung an die speziellen Bedürfnisse virtueller Maschinen. Ein sehr wichtiges Merkmal von *Parallax* liegt beispielsweise in ihrer Technik zur Erstellung von Snapshots der virtuellen Festplatten. Laut [Meyer u. a., 2008] kann mittels *Parallax* eine sehr feingranulare Sicherheit gegen den Datenverlust durch Fehlverhalten einer virtuellen Maschine erreicht werden, indem beispielsweise alle 10 ms ein Snapshot der virtuellen Festplatten angelegt wird.

### 2.1.3 Objektbasierte Speicher

Neben blockbasiertem Speicher gewinnt objektbasierter Speicher immer mehr an Bedeutung. Der Gedanke hinter objektbasierten Speichergeräten (*OSD* für *Object-based Storage Device* [Surhone u. a., 2010]) geht zurück auf die 1998 von Gibson u. a. in [Gibson u. a., 1998] vorgestellten *NASDs* (für *Network-Attached Secure Disks*). Wie der Name bereits sagt, wird der Speicher in einem *OSD* nicht auf Blockebene, sondern auf Objektebene verwaltet. Jedes Objekt hat einen auf dem *OSD* eindeutigen Identifizierer, über welchen es referenziert wird. Im Gegensatz zu Blöcken kann die Größe eines Objektes variieren. Das *OSD* ist selbst dafür verantwortlich, wie die Objekte auf dem physikalischen Speicher abgelegt werden.

Objektbasierte Speichergeräte werden grundsätzlich über ein Netzwerk angesprochen. Daher können sie parallel durch mehrere Server verwendet werden. Teil des Protokolls zum Zugriff auf objektbasierte Speichersysteme sind deshalb auch Mechanismen zur Authentifizierung und Autorisierung von Nutzern sowie zur verschlüsselten Datenübertragung.

Die erste Version des Protokolls zur Kommunikation mit einem *OSD* wurde im Jahr 2004 standardisiert (siehe [T10 committee, 2004]). Die Arbeiten an *OSD-2*, der zweiten Version dieses Protokolls, wurden 2008 abgeschlossen und befinden sich seit dem in der Standardisierung. Zu den wichtigsten Neuerungen in *OSD-2* gehört die Unterstützung von Snapshots eines *OSD*. Aktuell wird an *OSD-3* gearbeitet.

#### **Antara**

Mit *Antara* [Azagury u. a., 2003] präsentierten Azagury u. a. eine Implementierung von objektbasierten Speichergeräten. Zur Kommunikation verwenden sie ein eigenes Protokoll, welches ähnliche Kernfunktionalitäten besitzt wie das *OSD*-Protokoll der *T10* [T10 committee, 2004], dessen erste Version 2003 noch in der Standardisierung war.

## 2 Virtualisierung von Blockgeräten

### Ursa Minor

*Ursa Minor* [Ganger u. a., 2003; Abd-El-Malek u. a., 2005; Sinnamohideen u. a., 2010] ist eine objektbasierte Virtualisierung für Speichernetze. Das Ziel von *Ursa Minor* ist es, einen feingranular anpassbaren Speicher zu erzeugen. Dazu ermöglicht es *Ursa Minor*, für jedes Objekt einzeln anzugeben, wie es abgelegt werden soll. Einstellbar ist die Kodierung (z.B. *RAID 1* oder *RAID 5*), die Größe der *Chunks* und die verwendeten Speicherknotten.

### Kinesis

MacCormick u. a. haben in [MacCormick u. a., 2009] einen Ansatz für die replizierte Speicherung von Objekten über mehrere Speicherknotten vorgestellt. Hierbei werden die Server zunächst in  $k$  disjunkte Segmente aufgeteilt, so dass jedes Segment in etwa die gleiche Größe hat.

Von jedem Objekt legt *Kinesis*  $r \leq k$  Kopien in unterschiedlichen Segmenten ab. Innerhalb eines Segments werden Objekte mittels einer uniformen Hashfunktion platziert. Neben der Platzierung durch linearen Hashfunktionen betrachten MacCormick u. a. auch die Möglichkeit, Daten gemäß der aktuellen CPU- oder Netzwerklast der Server zu platzieren.

## 2.2 V:Drive

*V:Drive* ist eine Cluster Blockgerätevirtualisierung, welche an der Universität Paderborn entwickelt wurde. Ursprünglich konzipiert als Testumgebung verschiedener Verteilungsverfahren ist daraus mittlerweile eine stabile Umgebung zum Aufbau von hochwertigem Speicher entstanden. Dieser Abschnitt führt *V:Drive* ein und beschreibt verschiedene Schlüsselfunktionalitäten, welche innerhalb einer modernen Blockgerätevirtualisierung wichtig sind.

Um ein gemeinsames Vokabular zu schaffen, stelle ich die wichtigsten Komponenten in *V:Drive* vor. Ich unterteile die Komponenten dabei in zwei Bereiche: In Abschnitt 2.2.1 betrachte ich die physikalischen Komponenten, welche in *V:Drive* eine Rolle spielen. In Abschnitt 2.2.2 behandle ich dann die virtuellen Komponenten und stelle damit grundlegende Fähigkeiten von *V:Drive* vor.

### 2.2.1 Physikalische Komponenten

Die physikalischen Kernkomponenten in *V:Drive* setzen sich hauptsächlich aus einem Speichernetz mit angeschlossenen Datenservern (DS) zusammen. Die Konfiguration der virtuellen Laufwerke wird durch den Metadatenserver (MDS) verwaltet. In Abbildung 2.2 ist der Aufbau



vereinfacht dargestellt. Die Datenserver nutzen zur Kommunikation untereinander und mit dem Metadatenserver ein internes Netzwerk, während ein externes Netzwerk zur Kommunikation mit den Clients verwendet wird. Über das externe Netzwerk können die Clients auf die virtuellen Laufwerke beispielsweise mittels *iSCSI* zugreifen. Natürlich kann auch die gesamte Kommunikation über ein geteiltes Netzwerk erfolgen, dies ist aber aus Sicherheitsgründen oft nicht wünschenswert. Anbindungen des virtuellen Speichers auf anderen Wegen als der Blockebene können mittels zusätzlicher Dienste auf den Datenservern ermöglicht werden.

### Metadatenserver

Der Metadatenserver ist die zentrale Komponente für die Konfiguration des Speichersystems. Neben sämtlichen Datenservern kennt der Metadatenserver auch sämtliche Festplatten und weiß welcher Server auf welche Festplatten zugreifen kann. Diese Informationen erhält der Metadatenserver von den Datenservern.

Intern hält der Datenserver sämtliche Information über die Konfiguration des Speichersystems und die Metadaten der virtuellen Laufwerke in einer Datenbank vor. Über dieser Datenbank liegt eine Java-Applikation, welche für Änderungen an der Konfiguration und die Kommunikation mit den Servern verantwortlich ist.

### Datenserver

Die Datenserver sind die eigentlichen Speicherknotten in *V:Drive*. Diese Rechner haben Zugriff auf eine Menge von Festplatten. Der Zugriff wird in der Regel über ein *SAN* realisiert, um auch den parallelen Zugriff zu ermöglichen. Dies schafft neben einer besseren Performance auch eine höhere Ausfallsicherheit. Werden lokale Festplatten verwendet, so müssen sie allen Servern verfügbar gemacht werden (beispielsweise mittels *iSCSI*).

Die virtuellen Laufwerke werden im Datenserver durch einen Treiber erstellt und verwaltet. Der Treiber setzt sich aus einem Modul im Linux-Kernel und einem Userspace Service zusammen. Der Service öffnet einen Socket, über welchen der Metadatenserver Pakete an den Datenserver senden kann. Solche Pakete werden z.B. beim Anlegen eines neuen virtuellen Laufwerks versendet. Der Service verarbeitet die Pakete selbst oder reicht sie mittels eines *IOCTL*-Aufrufs an das Kernelmodul weiter. Das Kernelmodul erstellt für jedes virtuelle Laufwerk ein Blockgerät, welches durch Anwendungen wie eine normale Festplatte verwendet werden kann. Daten, welche auf einer virtuellen Festplatte gespeichert werden sollen, werden von dem Modul auf eine physikalische Festplatte umgeleitet. Die Information, auf welcher physikalischen Festplatte die Daten abgelegt werden sollen, erhält das Modul von dem Metadatenserver.

## 2 Virtualisierung von Blockgeräten

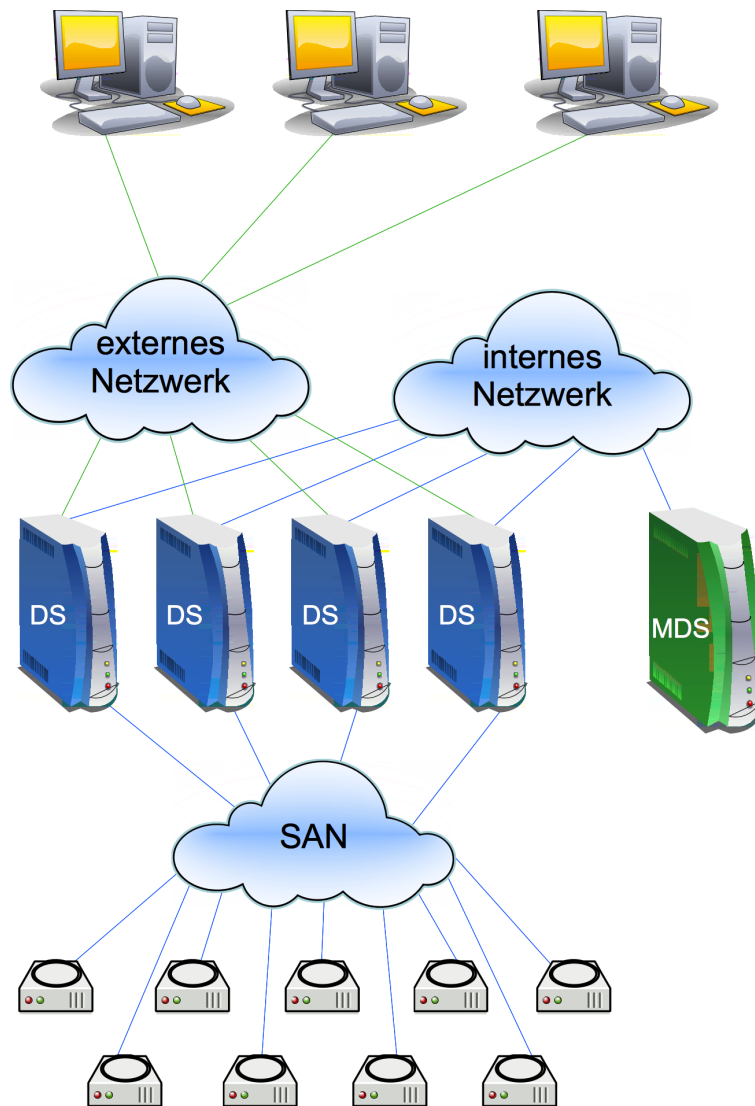


Abbildung 2.2: Aufbau eines Speichersystems mit *V:Drive*

## Internes Netzwerk

Die Konfiguration in Abbildung 2.2 verwendet zwei Netzwerke. Dies hat den Vorteil, dass der interne Datenverkehr einfacher vor externen Zugriffen geschützt werden kann.

Das interne Netzwerk ist zum Austausch von Verwaltungsinformationen zwischen den Datenservern sowie zur Kommunikation des Metadaten-servers mit den Datenservern zuständig. Die Verwaltungsinformationen, welche hier ausgetauscht werden, sind sehr klein, so dass ein schmalbandiges Netzwerk ausreicht. Da über dieses Netzwerk unter anderem Informationen zur Platzierung einzelner Daten versendet werden, ist eine geringe Latenz hier wichtiger.

Verwendet das Speichersystem lokale Festplatten, so können diese mittels eines Netzwerkprotokolls (z.B. *iSCSI*) über das interne Netzwerk den anderen Servern zur Verfügung gestellt werden. In diesem Fall ist es wichtig, ein breitbandiges Netzwerk zu verwenden, da es zusätzlich die Übermittlungsfunktionen des *SAN* übernimmt.

## Externes Netzwerk

Über das externe Netzwerk werden die virtuellen Laufwerke unterschiedlichen Clients zur Verfügung gestellt. Dies kann auf unterster Ebene direkt auf dem Blockgerät mittels *iSCSI* oder eines anderen Protokolls geschehen, oder auf Dateiebene über Netzwerkdateisysteme wie CIFS [Hertel, 2003] oder NFS [Shepler u. a., 2000]. Da es sich bei den Datenservern um Server mit einem Linux Betriebssystem handelt, können hier auch andere Anwendungen installiert werden, welche direkt auf den virtuellen Festplatten arbeiten. Naheliegend wären hier Anwendungen, welche auch wieder Informationen im Netzwerk bereit stellen, wie ein Web- oder Kalenderserver.

### 2.2.2 Virtuelle Komponenten

Als Blockspeichervirtualisierung erstellt *V:Drive* unter Verwendung des physikalischen Speichers einen virtuellen Speicher. Abbildung 2.3 gibt mittels eines Beispiels einen abstrakten Überblick über die verschiedenen virtuellen Komponenten.

Zunächst fasst *V:Drive* eine beliebige Menge von physikalischen Festplatten in einer Speichergruppe zusammen, in Abbildung 2.3 die Komponenten SG1 und SG2. Eine Festplatte kann zu maximal einer Speichergruppe gehören.

Über den Festplatten einer Speichergruppe kann dann eine beliebige Menge an virtuellen Laufwerken erstellt werden. Solch ein virtuelles Laufwerk erfüllt alle Anforderungen an ein CBG. In Abbildung 2.3 bestehen oberhalb von SG1 die virtuellen Laufwerke VV1 und VV2, oberhalb von SG2 das virtuelle Laufwerk VV3.

## 2 Virtualisierung von Blockgeräten

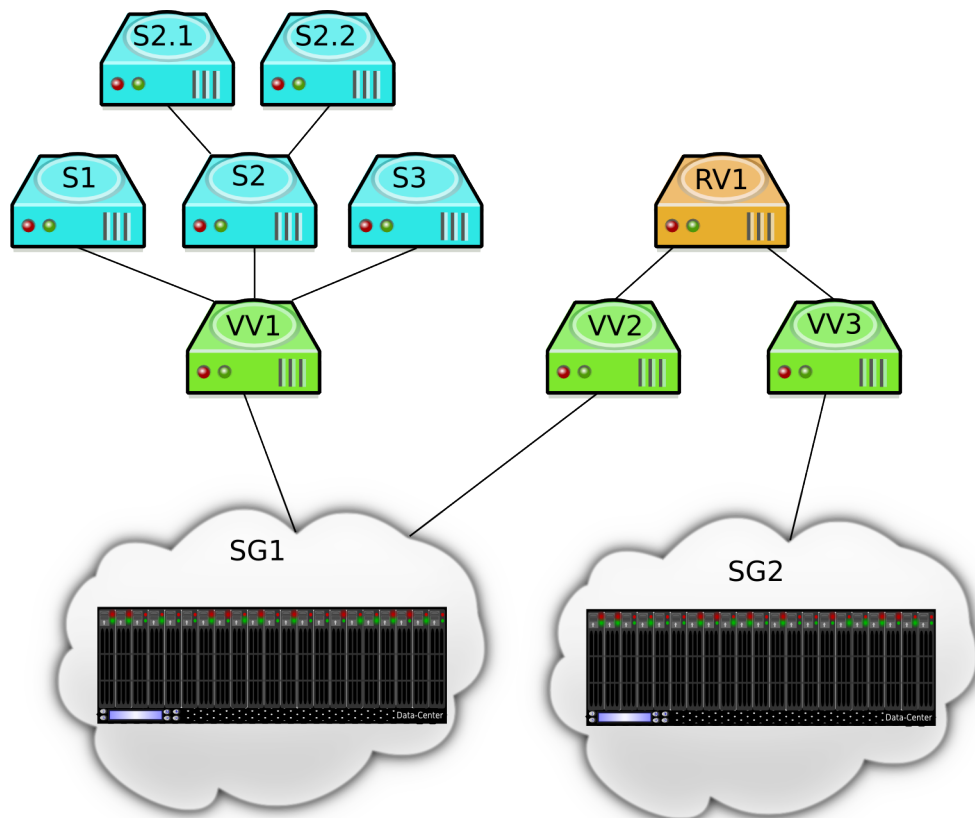


Abbildung 2.3: Repräsentation der Speicherkonfigurationen im Metadatenserver

Von jedem virtuellen Laufwerk können Snapshots, also Abbildungen des Stands eines Laufwerks zum Zeitpunkt der Erstellung des Snapshots, erstellt werden. *V:Drive* unterstützt Snapshots, welche nur für lesende Zugriffe oder auch für schreibende Zugriffe freigegeben sind. *V:Drive* unterstützt auch kaskadierende Snapshots, es können also Snapshots von Snapshots erzeugt werden. In Abbildung 2.3 sind S1, S2 und S3 Snapshots des Laufwerks VV1. S2 hat in dem Beispiel die Snapshots S2.1 und S2.2.

*V:Drive* ermöglicht auch die Abbildung von *RAID*-Laufwerken über verschiedene virtuelle Laufwerken. Auf diese Weise kann beispielsweise eine Spiegelung von Daten in verschiedene Speichergruppen erreicht werden. In dem Beispiel stellt RV1 ein solches *RAID*-Laufwerk dar.

### Speichergruppen

Speichergruppen sind die unterste Instanz virtuellen Speichers in *V:Drive*. Sie bilden einen unstrukturierten Speicher, welcher eine Menge von physikalischen Festplatten zusammen fasst. Eine Speichergruppe kann auf jedem Datenserver verwendet werden, welcher Zugriff auf die enthaltenen physikalischen Festplatten hat.

Die Festplatten innerhalb einer Speichergruppe können in ihrer Kapazität variieren. Mittels des Verfahrens *Share*, welches in Abschnitt 3.2 erläutert wird, stellt *V:Drive* sicher, dass die Festplatten gemäß ihrer Kapazität einen gleichen Anteil der gespeicherten Daten halten.

Der Speicher der physikalischen Festplatten innerhalb einer Speichergruppe wird in gleich große Einheiten zerlegt, diese werden in *V:Drive* als Extents bezeichnet. Die Größe der Extents kann zwischen Speichergruppen variieren, muss allerdings immer eine ganzzahlige Potenz von zwei sein. Vorgesehene Werte liegen zwischen einem MByte und einem GByte. Durch die Wahl kleinerer Extents kann eine bessere Verteilung der Daten und höherer Durchsatz bei sequentiellen Zugriffen erreicht werden, die Wahl großer Extents verringert dagegen den Verwaltungsaufwand für die Platzierung der Daten.

### Virtuelle Laufwerke

Innerhalb der Datenserver bilden die virtuellen Laufwerke den ansprechbaren Speicher. Sie können als normale Festplatten angesprochen werden, deren Daten durch die Server über den physikalischen Speicher verteilt werden.

Ein virtuelles Laufwerk ist immer genau einer Speichergruppe zugehörig. Auch der Speicher der virtuellen Laufwerke ist in Extents aufgeteilt. Die virtuellen Extents haben die gleiche Größe wie die physikalischen Extents der Speichergruppe.

Sollen Daten auf ein virtuelles Laufwerk geschrieben werden, so prüft der Server zunächst in welchen Extent des virtuellen Laufwerks die Daten fallen. Überschneidet der Zugriff eine Extentgrenze, so wird er in mehrere einzelne Anfragen aufgeteilt. Der Server versucht dann mittels eines internen Caches von Platzierungsdaten zu entscheiden, auf welches physikalische Extent das virtuelle Extent abgebildet wurde. Hat der Server diese Daten nicht vorrätig, so ermittelt er sie durch Abfrage des MetadatenServers. Danach schreibt der Server die Daten in das entsprechende physikalische Extent. In [Brinkmann u. a., 2006a] haben Brinkmann u. a. die Performance von *V:Drive* vermessen und mit dem *Logical Volume Manager* unter Linux verglichen. Sie konnten zeigen, dass *V:Drive* über einer festen Konfiguration ähnliche Geschwindigkeiten erreicht wie *LVM*. Im Gegensatz zu *LVM* kann *V:Drive* die Geschwindigkeit eines virtuellen Laufwerks recht schnell erhöhen, wenn neue Festplatten in die zugehörige Speichergruppe aufgenommen werden.

### Snapshots

Snapshots sind ein Mechanismus, um den Stand eines Blockgerätes zu einem bestimmten Zeitpunkt zu sichern. Dabei wird ein zweites virtuelles Laufwerk erstellt, welches auf die Datenbestände des ersten Laufwerks verweist. Werden die Daten auf einem der beiden Laufwerke

## 2 Virtualisierung von Blockgeräten

verändert, so wird zunächst eine Kopie des entsprechenden Extents erstellt, um eine Unabhängigkeit der Daten zu gewährleisten (*copy on write*). Die näheren Details der Implementierung der Snapshots in *V:Drive* haben Brinkmann u. a. in [Brinkmann u. a., 2006b] beschrieben. In [Brinkmann und Effert, 2007b] haben Brinkmann und Effert gezeigt, wie Snapshots beschaffen sein müssen, um unabhängig von der Anzahl der Snapshots eines Laufwerks den Geschwindigkeitsverlust konstant zu halten.

### RAID Laufwerke

*V:Drive* unterstützt zur Erhöhung der Datensicherheit *RAID*-Laufwerke über mehrere virtuelle Laufwerke unterschiedlicher Speichergruppen. Ein Problem an dieser Stelle ist, dass die virtuellen Laufwerke und die darüber liegenden *RAID*-Laufwerke auf mehreren Servern parallel verfügbar sein sollen. Bei klassischen *RAID* Implementierungen kann dies zu Problemen durch gegenseitiges Überschreiben von Prüfsummen führen. In [Brinkmann u. a., 2005] haben Brinkmann u. a. ein Sperrprotokoll vorgestellt, welches ein paralleles *RAID* auf mehreren Servern ermöglicht.

## 2.3 Einfluß der Kommunikation beim Einsatz lokaler Festplatten

Sollen Datenserver in *V:Drive* ein CBG bereit stellen, so brauchen sie Zugriff auf den gesamten physikalischen Speicher der zugehörigen Speichergruppe. Im Idealfall werden dafür Festplatten verwendet, welche über ein eigenes *Storage Area Network* erreichbar sind. Wird hier ein entsprechend schnelles Netzwerk verwendet (z.B. *FibreChannel*), so hängt die Geschwindigkeit der virtuellen Festplatten hauptsächlich von den physikalischen Festplatten ab.

Leider ist die Anschaffung eines expliziten Speichernetzes mit nicht zu vernachlässigenden Kosten verbunden. Daher verwenden Speichersysteme, welche möglichst kosteneffizient arbeiten sollen, oftmals *Storage Bricks*, über welche sie den virtuellen Speicher erzeugen. Ein *Storage Brick* ist dabei ein Server mit eigener CPU, RAM, Netzwerkanschluss und lokalen Festplatten. Ein Beispiel eines solchen Speichersystems ist das in Abschnitt 2.1.2 vorgestellte *FAB* [Frølund u. a., 2003].

Auch mittels *V:Drive* ist es möglich, ein solches Speichersystem zu erstellen. Hierzu werden die lokalen Festplatten der Datenserver über das interne Netzwerk zur Verfügung gestellt. In diesem Abschnitte werde ich untersuchen, wie sich ein so aufgebautes Speichersystem verhält. Die Ergebnisse beruhen auf in [Brinkmann und Effert, 2007a] vorgestellten Arbeiten.

Ich stelle zunächst in Abschnitt 2.3.1 die Architektur vor, auf der meine Beschreibungen beruhen. Danach betrachte ich in Abschnitt 2.3.2 die theoretische Skalierbarkeit eines solchen

Speichersystems unter Verwendung verschiedener Replikationsmuster. Ich schließe diesen Bereich mit der Evaluierung eines so aufgebauten Speichersystems in Abschnitt 2.3.3.

### 2.3.1 Architektur

Teilweise abweichend von der in Abschnitt 2.2.1 vorgestellten Architektur treffe ich folgende Annahmen:

- Als physikalischer Speicher werden Festplatten verwendet, welche lokal mit genau einem Server verbunden sind (*Direct Attached Storage, DAS*).
- Die einzelnen Server stellen die lokalen Festplatten über das interne Netzwerk den anderen Servern über *iSCSI* zur Verfügung. Auf diesem Weg wird das von *V:Drive* benötigte *SAN* hergestellt.
- Die lokalen Festplatten jedes Servers sind schnell genug, um den Netzwerkanschluss des Servers zu saturieren. Dies ist z.B. mittels vieler Festplatten in einem *RAID* zu erreichen.
- Das interne und das externe Netzwerk werden über dem gleichen physikalischen Netzwerk simuliert.
- Die Clients greifen per *iSCSI* auf die virtuellen Laufwerke zu.
- Die Datenserver sind homogen. Sie haben insbesondere
  - die gleiche Menge an eingebrachtem physikalischen Speicher und
  - die gleiche Bandbreite, mittels derer sie an das verwendete Netzwerk angeschlossen sind.
- Der Kommunikationsaufwand zur Metadatenverwaltung ist zu vernachlässigen.

Um eine einfachere Beschreibung zu ermöglichen definiere ich  $n$  als die Anzahl der Datenserver in dem Speichersystem. Für  $i \in \{1, \dots, n\}$  ist  $S_i$  ein Datenserver und  $L_i$  der Anteil seines physikalischen Speichers am gesamten physikalischen Speicher. Da die Server homogen sind, gilt  $L_i = \frac{1}{n}$ .

Die Daten der virtuellen Laufwerke sind über den gesamten Speicher verteilt. Jeder Server hat die gleiche Netzwerkanbindung mit der Geschwindigkeit  $b$ .  $x_n$  beschreibt den Erwartungswert der Geschwindigkeit, mit der ein Server Daten eines virtuellen Laufwerks speichern kann. Dieser Wert unterscheidet sich von  $b$ , da zusätzliche Kommunikation mit anderen Servern notwendig sein kann. Ist  $f(n)$  ein Funktion, welche den erwarteten zusätzlichen Kommunikationsaufwand zum Schreiben eines Blocks berechnet, so gilt  $x_n = f(n) \cdot b$ .

$B$  beschreibt die erwartete akkumulierte Geschwindigkeit, mit welcher alle Server zusammen ein virtuelles Laufwerk zur Verfügung stellen können. Theoretisch ist die maximale akkumulierte Geschwindigkeit  $B = n \cdot x_n = n \cdot f(n) \cdot b$ . Mit  $g(n) = n \cdot f(n)$  bezeichnet  $g(n)$  die Skalierungsfunktion des Speichernetzes. Ich sage auch, das Netzwerk skaliert mit  $g(n)$ .

## 2 Virtualisierung von Blockgeräten

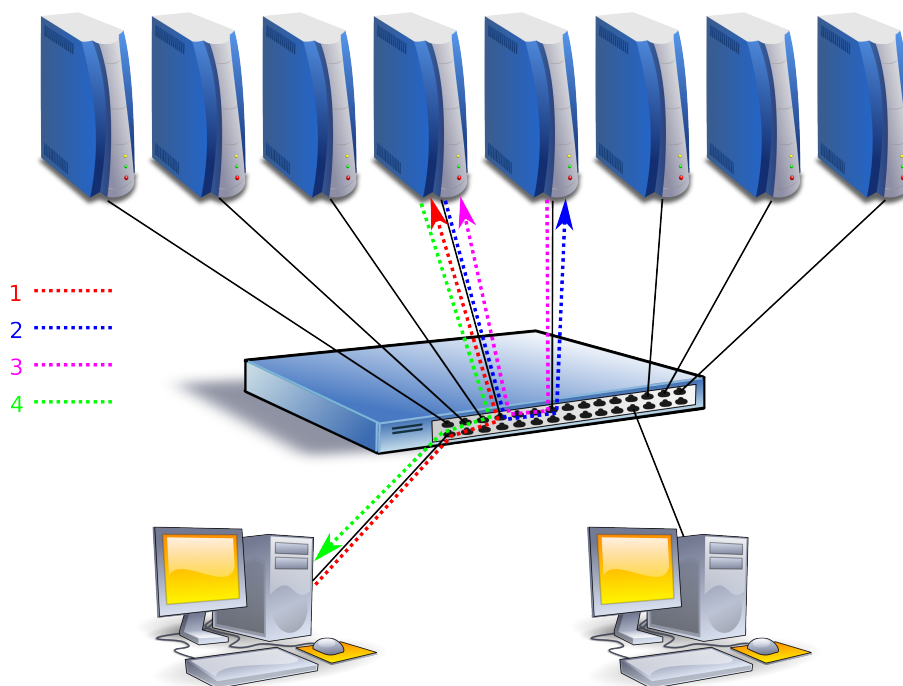


Abbildung 2.4: Typische inter-Server Kommunikation bei Verwendung lokaler Festplatten

In einer realistischen Umgebung nehme ich an, dass  $B$  sich ergibt aus

$$B = n \cdot \alpha \cdot x_n$$

$\alpha$  ist eine Konstante, welche den von  $n$  unabhängigen Parallelisierungsaufwand beschreibt. Ich bestimme  $\alpha$  wie folgt: Ein Test besteht aus  $p$  unabhängigen Testläufen mit einer unterschiedlichen Zahl an Datenservern.  $\delta_i$  ist die gemessene Bandbreite für den Test in der Runde  $1 \leq i \leq p$  und  $x_n$  die erwartete Bandbreite. Dann wähle ich  $\alpha$  so, dass folgende Formel ein minimales Ergebnis liefert:

$$\sum_{i=1}^p (\delta_i - \alpha \cdot x_n)^2$$

### 2.3.2 Kommunikation zwischen den Datenservern

In dem angegebenen Szenario kommunizieren zwei Datenserver immer dann miteinander, wenn einer von beiden auf Daten zugreifen muss, welche auf einer Festplatte des anderen Servers platziert wurden. Abbildung 2.4 zeigt dies beispielhaft für die Leseanfrage eines Clients.

Zunächst fragt der Client in Schritt 1 einen Datenserver nach den gewünschten Daten. Sind die Daten nicht auf dem angefragten Server verfügbar so sendet er in Schritt 2 eine Anfrage an



den Server, welcher die Daten hält. Dieser sendet die Daten in Schritt 3 zurück an den zuerst angefragten Server, welcher sie dann in Schritt 4 an den Client übermitteln kann.

Dieses Vorgehen erscheint unnötig kompliziert, da der Server, der die Daten gespeichert hat, diese auch direkt an den Client senden könnte. Dies ist bei Verwendung von *iSCSI* als Kommunikationsprotokoll allerdings nicht ohne großen Aufwand möglich. Da *iSCSI* auf TCP basiert, baut ein Client zur Abfrage von Daten zunächst einen Socket zu einem Datenserver auf. Die Antwort auf die Anfrage erwartet der Client dann über den bestehenden Socket. Damit innerhalb des Clients keine Anpassungen nötig sind, müsste also der bestehende Socket jeweils an den Datenserver weitergereicht werden, auf welchem die angefragten Daten platziert wurden. Wie Olaru und Tichy in [Olaru und Tichy, 2005] gezeigt haben, ist dies nicht effizient möglich.

#### Skalierbarkeit ohne Redundanz

Zunächst betrachte ich, wie die erwartete Skalierbarkeit des in Abschnitt 2.3.1 beschriebenen Speichersystems ist, falls die Daten nicht redundant gespeichert werden. Dazu unterscheide ich zwischen einem halb-duplexen (hd) und einem voll-duplexen (vd) Netzwerk. Ein halb-duplexes Netzwerk kann nicht gleichzeitig senden und empfangen. Daher kann die akkumulierte Bandbreite für das Lesen und das Schreiben von Daten  $b$  nicht überschreiten. Im Gegensatz dazu kann ein voll-duplexes Netzwerk Daten gleichzeitig senden und empfangen, jeweils mit der vollen Bandbreite  $b$ .

Die Clients wissen nicht, auf welchem Server die von ihnen angeforderten Daten physikalisch abgelegt sind. Versucht ein Client Daten von einem Server  $S_i$  zu lesen, so hat  $S_i$  diese mit einer Wahrscheinlichkeit von  $L_i = \frac{1}{n}$  auf seinem lokalen Speicher vorliegen. Daher muss er mit einer Wahrscheinlichkeit von  $1 - \frac{1}{n} = \frac{n-1}{n}$  die Anfrage an einen anderen Datenserver weiterleiten.

Somit wird folgende Bandbreite  $b$  eines halb-duplexen Netzwerks benötigt, damit im erwarteten Fall jeder Server eine ausgehende Bandbreite  $x_n$  zu den Clients haben kann:

$$b = \frac{1}{n} \cdot x_n + 2 \cdot \frac{n-1}{n} \cdot x_n + \frac{n-1}{n} \cdot x_n$$

Der erste Term  $\frac{1}{n} \cdot x_n$  ist der Anteil an Anfragen, die ein Server lokal beantworten kann. Der zweite Term  $2 \cdot \frac{n-1}{n} \cdot x_n$  ist der Kommunikationsaufwand zur Beantwortung von Anfragen, die nicht lokal vorliegen. Der letzte Term  $\frac{n-1}{n} \cdot x_n$  ist der Anteil an Anfragen an andere Datenserver, welche durch diesen Server beantwortet werden müssen.

Durch Umformung dieser Formel kann ich nun  $x_n$  in Abhängigkeit von  $b$  bestimmen:

$$x_n = \frac{n}{3n-2} \cdot b$$

Für die  $n$  Datenserver ergibt sich eine erwartete nutzbare Bandbreite des Netzwerks von:

## 2 Virtualisierung von Blockgeräten

$$\begin{aligned} B_{hd}^{noRep} &= n \cdot \frac{n}{3n-2} \cdot b \cdot \alpha \\ &\approx \frac{n}{3} \cdot \alpha \cdot b \end{aligned}$$

Damit skaliert ein halb-duplexes Netzwerk im gegebenen Szenario lediglich mit  $\frac{1}{3} \cdot n$ .

Im Falle eines voll-duplexen Netzwerks können Daten mit der vollen Geschwindigkeit gleichzeitig empfangen und gesendet werden. Die benötigte Bandbreite des Netzwerks um eine verfügbare Bandbreite  $x_n$  zu den Clients zu bestimmen lässt sich jetzt wie folgt ausdrücken:

$$b = \max \left( x_n + \frac{n-1}{n} \cdot x_n, \frac{n-1}{n} \cdot x_n \right)$$

Der linke Teil der Maximum-Funktion ist die benötigte Bandbreite um Daten an die Clients und an die anderen Datenserver zu senden. Der rechte Teil ist die benötigte Bandbreite zum Empfangen der Daten von anderen Datenservern. Der linke Teil der Funktion ist der dominierende. Damit kann ich  $x_n$  mittels folgender Formel berechnen:

$$x_n = \frac{n}{2n-1} \cdot b$$

Wie zuvor bestimmt sich nun die zu erwartende Bandbreite wie folgt:

$$\begin{aligned} B_{vd}^{noRep} &= n \cdot \frac{n}{2n-1} \cdot b \cdot \alpha \\ &\approx \frac{n}{2} \cdot \alpha \cdot b \end{aligned}$$

Während ein halb-duplexes Netzwerk also noch mit  $\frac{1}{3} \cdot n$  skaliert verbessert sich dies für ein voll-duplexes Netzwerk auf  $\frac{1}{2} \cdot n$ . Ähnliche Gleichungen entstehen für das Schreiben von Daten, sowohl für ein halb-duplexes als auch voll-duplexes Netzwerk.

### Skalierbarkeit bei $k$ -facher Spiegelung

Ein wichtiger Unterschied zwischen klassischen *RAID*-Systemen und Cluster Blockvirtualisierungen liegt in der Verteilung von Daten über verschiedenen Servern. Bei der Verwendung von lokalen Festplatten wird die Wahrscheinlichkeit des Ausfalls von physikalischen Festplatten wesentlich erhöht, da hier nicht nur bei einem Defekt einer Festplatte Speicher ausfällt, sondern auch bei einem Defekt eines Servers oder seines Netzwerkanschlusses.

### 2.3 Einfluß der Kommunikation beim Einsatz lokaler Festplatten

Um dies zu kompensieren werden häufig mehrstufige Verfahren eingesetzt. Zunächst ist es möglich den Speicher innerhalb eines Servers mit einer eigenen Redundanz zu versehen, um den Ausfall einzelner Festplatten kompensieren zu können, ohne das Netzwerk zu belasten. Zum Schutz vor dem Ausfall ganzer Datenserver ist eine zusätzliche Redundanz über die Server notwendig.

In diesem Abschnitt betrachte ich zunächst die Skalierungseigenschaften eines Netzwerks, in welchem die Daten über  $k \leq n$  Datenserver gespiegelt werden. Wie zuvor betrachte ich dabei halb-duplexe und voll-duplexe Netzwerke. Im nächsten Abschnitt gehe ich dann auf die Verwendung von *MDS*-Codes ein.

Beim Schreiben von Daten werden  $k$  Kopien über die Datenserver verteilt. Dies bedeutet eine erhöhte Kommunikation. Analog zum vorherigen Abschnitt kann die benötigte Netzwerkbandbreite um im erwarteten Fall eine Bandbreite von  $x_n$  von jedem Server zu den Clients zu gewährleisten, wie folgt berechnet werden:

$$b = x_n + k \cdot \frac{n-1}{n} \cdot x_n + k \cdot \frac{n-1}{n} \cdot x_n$$

Hier beschreibt wieder der erste Term die benötigte Leistung, um die  $x_n$  Daten an die Clients zu liefern. Der zweite Term  $k \cdot \frac{n-1}{n} \cdot x_n$  ist die benötigte Bandbreite, um die Anfragen an die anderen Server zu senden. Der letzte Term  $k \cdot \frac{n-1}{n} \cdot x_n$  ist die benötigte Bandbreite um Daten aus Anfragen an andere Datenserver zu schreiben. Dies lässt sich wieder nach  $x_n$  umformen, um für eine gegebene Bandbreite des Netzwerks die ausgehende Bandbreite zu den Clients zu berechnen:

$$x_n = \frac{n}{n + 2 \cdot k \cdot (n-1)} \cdot b$$

Die erwartete ausgehende Bandbreite über alle Server ist somit:

$$\begin{aligned} B_{hd}^{k-Mirror} &= n \cdot \frac{n}{n + 2 \cdot k \cdot (n-1)} \cdot b \cdot \alpha \\ &\approx \frac{n}{1 + 2 \cdot k} \cdot \alpha \cdot b \end{aligned}$$

Das gesamte Netzwerk skaliert also lediglich mit  $\frac{1}{1+2 \cdot k} \cdot n$ . Mittels eines voll-duplex Netzwerks kann die Skalierung wieder verbessert werden. Die benötigte Bandbreite  $b$  um von jedem Server eine Bandbreite von  $x_n$  zu den Clients zu ermöglichen ist dann:

$$b = \max \left( x_n + k \cdot \frac{n-1}{n} \cdot x_n, k \cdot \frac{n-1}{n} \cdot x_n \right)$$

## 2 Virtualisierung von Blockgeräten

Der erste Parameter der Maximum-Funktion ist die erwartete Bandbreite, die zum Empfangen von Schreibanfragen benötigt wird, während der zweite Parameter die Sendeleistung beschreibt. Die zum Empfangen von Daten benötigte Bandbreite ist der größere Wert. Damit kann  $x_n$  wie folgt bestimmt werden:

$$x_n = \frac{n}{(k+1) \cdot n - k} \cdot b$$

Daraus ergibt sich eine Netzwerkbandbreite über alle Datenserver zu den Clients von:

$$\begin{aligned} B_{vd}^{k-Mirror} &= n \cdot \frac{n}{(k+1) \cdot n - k} \cdot b \cdot \alpha \\ &\approx \frac{n}{k+1} \cdot \alpha \cdot b \end{aligned}$$

Werden also jeweils zwei Kopien platziert, so skaliert das Netzwerk mit  $\frac{1}{3} \cdot n$ , bei mehr Kopien entsprechend schlechter. Diese Skalierungseigenschaft kann mittels MDS-Codes verbessert werden.

### Skalierbarkeit bei Verwendung von MDS-Codes

In Abschnitt 2.1.1 habe ich verschiedene *Erasure-Codes* vorgestellt, mittels welcher sich Daten redundant über Festplatten verteilen lassen. Generell zeichnen sich *Erasure-Codes* dadurch aus, dass  $v$  Datenblöcke so auf  $u \geq v$  Datenblöcke erweitert werden, dass die ursprüngliche Nachricht aus  $r \leq u$  Blöcken wieder hergestellt werden kann. Speziell gilt für *MDS Codes*  $r = v$ , die ursprünglichen Daten können also aus beliebigen  $v$  Datenblöcken rekonstruiert werden.

Soll ein Speichersystem auch bei Ausfall eines Server weiter verfügbar sein, so kann dies wie im vorherigen Abschnitt beschrieben durch Platzierung von zwei Kopien jedes Blocks erreicht werden. Das gleiche Ziel lässt sich aber auch erreichen, indem ein *RAID 5* Code über einer Menge unterschiedlicher Datenserver erzeugt wird. Wird beispielsweise ein *RAID 5* über jeweils vier Datenblöcke gebildet, so müssen nur fünf Datenblöcke gespeichert werden. Somit werden lediglich 25% mehr Speicher gebraucht, statt 100% bei einer Spiegelung der Daten.

Ich zeige nun, dass sich neben einem geringeren Speicherbedarf auch eine bessere Skalierung des Netzwerks ergibt. Dabei gehe ich davon aus, dass immer volle *Stripes* geschrieben werden.

Ich unterscheide wie zuvor zwischen halb-duplexen und voll-duplexen Netzwerken. Für halb-duplexe Netzwerke wird folgende Bandbreite  $b$  benötigt, damit jeder Datenserver eine erwartete Bandbreite  $x_n$  zu den Servern erreichen kann:

$$b = \frac{u}{v} \cdot \frac{1}{n} \cdot x_n + 2 \cdot \frac{u}{v} \cdot \frac{n-1}{n} \cdot x_n + \frac{u}{v} \cdot \frac{n-1}{n} \cdot x_n$$

## 2.3 Einfluß der Kommunikation beim Einsatz lokaler Festplatten

Wie in den vorherigen Formeln für halb-duplexe Netzwerke bestimmt der erste Term den Anteil der Daten, welche direkt durch den jeweiligen Datenserver beantwortet wird. Der zweite Term ist wieder der Anteil der Kommunikation, die ein Datenserver an andere Datenserver weiter reichen muss. Der letzte Term ist die Bandbreite, die der Datenserver für Anfragen an andere Datenserver bereitstellen muss. Dies kann wieder nach  $x_n$  aufgelöst werden:

$$x_n = \frac{v \cdot n}{u(3 \cdot n - 2)} \cdot b$$

Damit lässt sich folgende akkumulierte Bandbreite erreichen:

$$\begin{aligned} B_{hd}^{u,v} &= n \cdot \frac{v \cdot n}{u \cdot (3 \cdot n - 2)} \cdot b \cdot \alpha \\ &\approx \frac{v \cdot n}{3 \cdot u} \cdot \alpha \cdot b \end{aligned}$$

Für voll-duplexe Netzwerke lässt sich die benötigte Bandbreite des Netzwerks, um von jedem Server eine erwartete Bandbreite  $x_n$  zu den Clients zu erhalten, wie folgt ausdrücken:

$$b = \max \left( x_n + \frac{u}{v} \cdot \frac{n-1}{n} \cdot x_n, \frac{u}{v} \cdot \frac{n-1}{n} \cdot x_n \right)$$

Der erste Wert der Maximumfunktion beschreibt die erforderliche Bandbreite zum Empfangen, der zweite die zum Senden. Wieder dominiert der Wert zum Empfangen von Daten. Damit kann die Formel nach  $x_n$  aufgelöst werden:

$$x_n = \frac{v \cdot n}{(v+u) \cdot n - u} \cdot b$$

Somit ist die zu erwartende akkumulierte Bandbreite:

$$\begin{aligned} B_{vd}^{u,v} &= n \cdot \frac{v \cdot n}{(v+u) \cdot n - u} \cdot b \cdot \alpha \\ &\approx \frac{v \cdot n}{v+u} \cdot \alpha \cdot b \end{aligned}$$

### 2.3.3 Messungen

Um zu zeigen, wie gut die oben berechneten Werte die Realität abdecken, haben Brinkmann und Effert in [Brinkmann und Effert, 2007a] ein Speichersystem aufgebaut und vermessen, welches dem beschriebenen Model folgt.

### Testumgebung

Als Testumgebung wurden bis zu 48 identische Rechner verwendet, jeder ausgestattet mit einer 1 GHz *Pentium 3* CPU und 512 MByte RAM. Als Betriebssystem kam *RedHat AS 4* zum Einsatz. Die Knoten waren mittels eines voll-duplexen 100 MBit/s Netzwerks über einen Ethernet Switch *Cisco Catalyst 5509* verbunden. Der Switch war ausgestattet mit sechs *WS-X5234-RJ45* Erweiterungsmodulen, welche Anschlüsse für jeweils 24 Knoten boten. Die Backplane des Switches besaß drei Busse, welche jeweils eine theoretische Bandbreite von 1,2 GBit/s hatten. Die maximal jemals gemessene Bandbreite über einen Bus lag allerdings nur bei 900 MBit/s.

Für die Tests wurden die Knoten in maximal 24 Datenserver und gleich viele Clients aufgeteilt. Jeder Datenserver hat eine 1,8 GByte große Partition seiner lokalen Festplatte und eine ein TByte große RAM-Disk exportiert. Von dieser RAM-Disk wurden lediglich die ersten 512 KByte im Speicher gehalten, für den restlichen Bereich lieferte sie zufällige Werte. Auf diese Weise konnte eine beliebig große RAM-Disk simuliert werden, welche einen funktionierenden Boot-Sektor besaß.

Um Caching-Effekte zu minimieren wurde ein Modul entwickelt, welches einen Teil des RAM-Speichers so reserviert, dass er dem Betriebssystem nicht mehr zur Verfügung gestellt werden kann. Je nach Größe der verwendeten virtuellen Festplatten wurden mittels des Moduls bis zu 420 MB RAM reserviert.

Für die Skalierungstests wurde pro Datenserver ein Client eingesetzt. Der Speicher der Server wurde in einer gemeinsamen Speichergruppe zusammengefasst. Jeder Server stellte seinem Client zwei virtuelle Laufwerke per *iSCSI* zur Verfügung. Die Daten der virtuellen Laufwerke wurden über alle Server verteilt.

Sowohl für die Verteilung der lokalen Laufwerke zwischen den Servern als auch der virtuellen Laufwerke an die Clients wurde auf den Datenservern der *iSCSI Enterprise Target* in der Version 0.4.12 eingesetzt. Als *iSCSI*-Initiator wurde das von *RedHat* mitgelieferte *iSCSI*-Initiator Modul verwendet.

Für die Benchmarks wurde *IOMeter*<sup>4</sup> verwendet. *IOMeter* startete auf jedem Client einem Prozess (*Dynamo*), welcher die Last generiert und lokale Messungen vornahm. Gesteuert wurden diese Prozesse durch einen Manager auf einem Windows XP PC. Falls nichts anderes angegeben wird, so wurden für die Tests maximal 16 ausstehende IO-Operationen erlaubt. Die Zugriffsgröße lag bei 32 KByte und die Tests liefen über einen Zeitraum von fünf Minuten nach einer Einschwingzeit von 30 Sekunden.

---

<sup>4</sup><http://www.iometer.org/>, am 4.11.2010

Tabelle 2.1: Vermessung IO/s eines einzelnen Servers

	RAM-Disk	virtuelle Festplatte	RAM-Disk über <i>iSCSI</i>	virtuelle Festplatte über <i>iSCSI</i>
Seq. schreiben	4340	3862	321	178
Rand. lesen	11429	7116	355	152

### Vermessung eines einzelnen Servers

Um besser einschätzen zu können, welche Verluste ein Speichersystem bereits durch die Virtualisierungsschicht und das Netzwerk verliert, wurde zunächst ein einzelner Server vermessen. Dies geschah in vier Stufen.

Zunächst wurden zwei RAM-Disks auf einem der Datenserver erzeugt. Mittels *IOMeter* wurden diese RAM-Disks im ersten Test direkt auf dem Datenserver vermessen. Dazu wurde, wie in allen weiteren Tests zur Vermessung des Servers, ein Prozess pro RAM-Disk eingesetzt. Die erste Spalte in Tabelle 2.1 zeigt die Ergebnisse dieses Tests. Pro Sekunde konnte die RAM-Disk beim sequentiellen Schreiben 4340 IO/s erreichen. Bei einer Größe von 32 KByte pro IO bedeutet dies einen Durchsatz von 136 MByte/s. Bei zufällig verteilten Leseoperation konnte die RAM-Disk 11429 IO/s bei 357 MByte/s erreichen. Der Speicher kann also schneller gelesen als geschrieben werden. Messungen für randomisiert verteilte Schreiboperation lagen ähnlich der sequentiellen (4466 IO/s bei 140 MByte/s), sequentielle Leseoperation ähnlich der randomisiert verteilten (11514 IO/s bei 360 MByte/s).

Für den zweiten Test wurde mittels *V:Drive* eine Speichergruppe erstellt, welche als einzige physikalische Laufwerke die RAM-Disks des Servers beinhaltet. Aus dieser Speichergruppe wurden zwei virtuelle Laufwerke erstellt und wieder lokal auf dem Datenserver vermessen. Vor dem eigentlichen Test wurde das gesamte Laufwerk einmal beschrieben, so dass die Metadateninformationen von *V:Drive* lokal vorlagen und während der Testläufe keine Kommunikation mit dem Metadatenserver notwendig war. Die zweite Spalte der Tabelle 2.1 zeigt die Ergebnisse dieses Tests.

Die Anzahl der IO-Operationen für das sequentielle Schreiben ist gegenüber der Vermessung der RAM-Disk auf 3862 IO/s gesunken. Somit hat die Virtualisierung des Speichers einen Verlust von 11% der IO-Leistung bewirkt. Für zufällig verteilte Zugriffe sank die IO-Leistung sogar um 38% auf 7116 IO/s.

Für den dritten Test wurden die RAM-Disks aus dem ersten Test per *iSCSI* im Netzwerk bereit gestellt. Von einem Client wurden diese Laufwerke eingebunden und vermessen. Die Ergeb-

## 2 Virtualisierung von Blockgeräten

nisse finden sich in der dritten Spalte von Tabelle 2.1. Beim sequentiellen Schreiben konnte der Client 321 IO/s erreichen. Die damit verbunden 10 MByte/s an Daten haben das 100 MBit/s Netzwerk annähernd saturiert. Beim randomisiert verteilten Lesen konnte das Netzwerk mit 355 IO/s und 11 MByte/s noch näher an seine Grenzen gebracht werden.

Für den vierten Test wurden die virtuellen Laufwerke aus dem zweiten Test per *iSCSI* im Netzwerk bereit gestellt. Wie im dritten Test wurden diese Laufwerke auf einem Client eingebunden und dort vermessen. Die Ergebnisse sind in Spalte vier der Tabelle 2.1 zu finden. In dieser Konfiguration konnten nur noch 178 IO/s bei 6 MByte/s beim sequentiellen Schreiben erreicht werden. Das randomisiert verteilte Lesen war in dieser Konfiguration langsamer als das sequentielle Schreiben, es konnten nur noch 152 IO/s bei 5 MByte/s erreicht werden. Dies liegt an dem geringen Durchsatz, dadurch konnte der Cache im Client beim Schreiben Wirkung zeigen.

### Vermessung ohne Redundanz mit RAM-Disks

Durch die Verwendung vieler oder besonders schneller Festplatten (wie *Solid State Disks*) ist es möglich Datenserver aufzubauen, deren Speicherleistung die Netzwerkleistung zumindest für sequentielle Zugriffe saturiert. Begünstigt werden kann dies zusätzlich durch die Verwendung großer Caches auf den Datenservern. Um so ein System zu simulieren verwenden Brinkmann und Effert die zuvor beschriebenen RAM-Disks um schnellen und großen Speicher zu erhalten.

Im ersten Skalierungstest wurde auf einer wachsenden Anzahl von Knoten jeweils eine virtuelle RAM-Disk mit einer Kapazität von einem TByte erstellt und per *iSCSI* im Netzwerk zur Verfügung gestellt. Danach wurde eine Speichergruppe erstellt, welche alle virtuellen RAM-Disks vereinigte. Für jeden Server wurden zwei virtuelle Laufwerke mit einer Kapazität von 40 GByte erstellt, welche ebenfalls per *iSCSI* im Netzwerk zur Verfügung gestellt wurden. Zu jedem Server wurde ein Client aufgesetzt, welcher die virtuellen Laufwerke des zugehörigen Servers per *iSCSI* einband. Über die Clients wurden diese virtuellen Laufwerke mittels *IOMeter* hinsichtlich sequentieller Schreibzugriffe vermessen.

Abbildung 2.5 zeigt die Ergebnisse dieser Tests. Gruppiert nach der Anzahl der verwendeten Datenserver (bzw. Clients, die Anzahl ist identisch) findet sich dort im jeweils linken Balken der optimale Durchsatz, normiert auf die Ergebnisse für einen Server. Hierfür wurde die in Abschnitt 2.3.2 gezeigte Formel verwendet:

$$B_{vd}^{noRep} = n \cdot \frac{n}{2n-1} \cdot b \cdot \alpha$$

Hierzu wurde  $\alpha = 1$  gesetzt. Der zweite Balken zeigt die gleichen Ergebnisse für  $\alpha = 0,74$ . Die Berechnung dieses Werts ist in Abschnitt 2.3.1 beschrieben. Mittels dieses Wertes konnte der Abstand zwischen der optimalen und der realen Skalierbarkeit des Speichernetzes minimiert werden. Hierzu wurden die Messungen für  $n = 2$  bis  $n = 16$  herangezogen.



### 2.3 Einfluß der Kommunikation beim Einsatz lokaler Festplatten

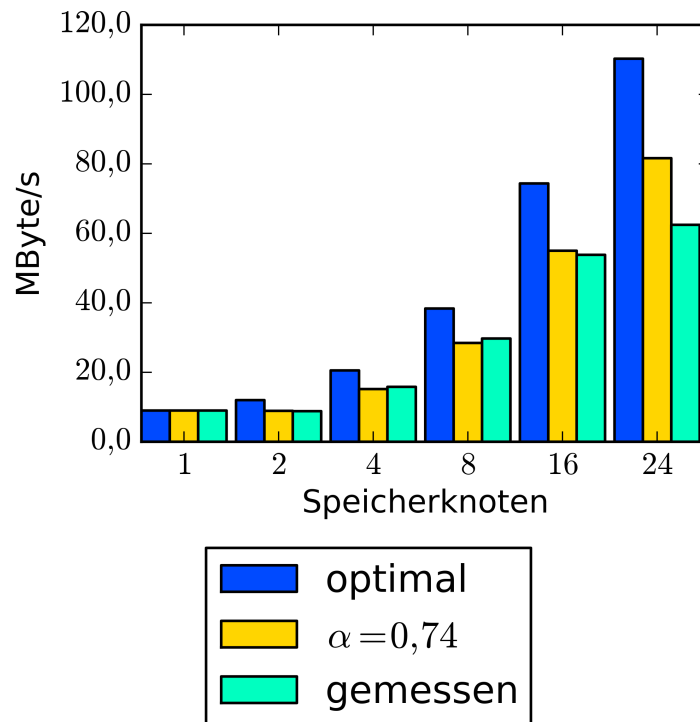


Abbildung 2.5: Skalierung des Speichersystems ohne Redundanz

Der letzte Wert ist der gemessene Durchsatz über alle Clients. Wie Abbildung 2.5 zeigt, entwickelt sich dieser Wert für  $\alpha = 0,74$  für bis zu 16 Datenserver sehr nah der erwarteten Skalierung. Bei der Verwendung von 24 Datenservern kann das reale System aber nicht mehr der berechneten Skalierung gerecht werden. Der Grund dafür liegt im verwendeten Switch. Dieser war durch die aufgekommene Netzwerklast saturiert.

#### Vermessung ohne Redundanz mit physikalischen Festplatten

In realen Speichernetzen kommen in den meisten Fällen magnetische Festplatten zum Einsatz. In manchen Konfigurationen ist es dabei nicht möglich, die vorhandene Netzwerkbandbreite zu saturieren. In diesem Fall wird der maximale Durchsatz nicht mehr durch das Netzwerk, sondern durch die verwendeten physikalischen Festplatten bestimmt.

Um so eine Konfiguration zu simulieren wurde auf jedem Datenserver eine Partition mit einer Kapazität von 1,8 GByte per *iSCSI* allen Datenservern verfügbar gemacht. Lokal kann diese Partition mit ca. 10 MByte/s sequentiell beschrieben werden. Über diesen Partitionen wurde wieder eine gemeinsame Speichergruppe erstellt. Jeder Server erhielt zwei virtuelle Laufwerke mit einer Kapazität von je 500 MByte aus diesem Speicher, die er per *iSCSI* an einen Client exportierte.

## 2 Virtualisierung von Blockgeräten

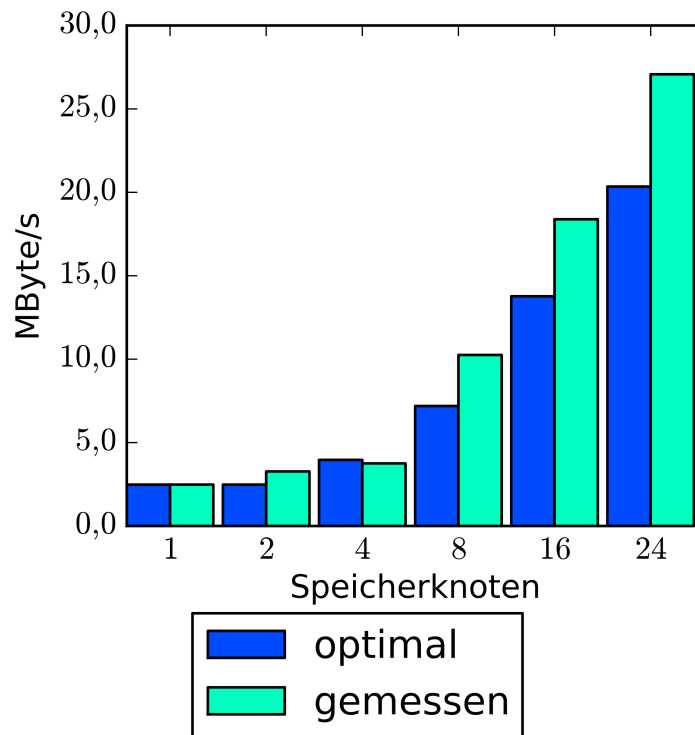


Abbildung 2.6: Skalierung des Speichersystems mit langsamen Festplatten ohne Redundanz

Auf den Clients wurden die virtuellen Laufwerke eingebunden und mittels *IOMeter* vermessen. Dazu wurden pro Laufwerk zwei Prozesse mit sequentiellen Schreibzugriffen gestartet. Durch die Verwendung von zwei Prozessen wird auf den physikalischen Festplatten eine höhere Verteilung der Zugriffe erwirkt. Ohne diese Maßnahme hätte die Gefahr bestanden, dass die physikalischen Festplatten den Netzwerkanschluss saturieren.

Um Caching-Effekte zu vermeiden wurden auf allen Clients und allen Servern 256 MByte RAM durch ein Kernelmodul reserviert, so dass diese dem Betriebssystem nicht mehr zur Verfügung standen.

Abbildung 2.6 zeigt die Skalierung des Speichernetzes für sequentielle Schreibzugriffe. Wie direkt zu sehen ist skaliert das Speichernetz besser, als in den Berechnungen gezeigt. Dies liegt daran, dass die lokalen Partitionen nicht genug Durchsatz erreichen, um die Netzwerkanbindung der Datenserver zu saturieren. Ein einzelner Server schafft in dieser Konfiguration ca. 2,8 MByte/s an Durchsatz. Da in dieser Konfiguration durchschnittlich vier Prozesse auf unterschiedlichen Stellen der Festplatte arbeiten, ist die Festplatte durch häufige Bewegungen des Schreib-Lese-Kopfes ausgelastet und schafft es nicht, einen höheren Durchsatz zu erzielen.

### 2.3 Einfluß der Kommunikation beim Einsatz lokaler Festplatten

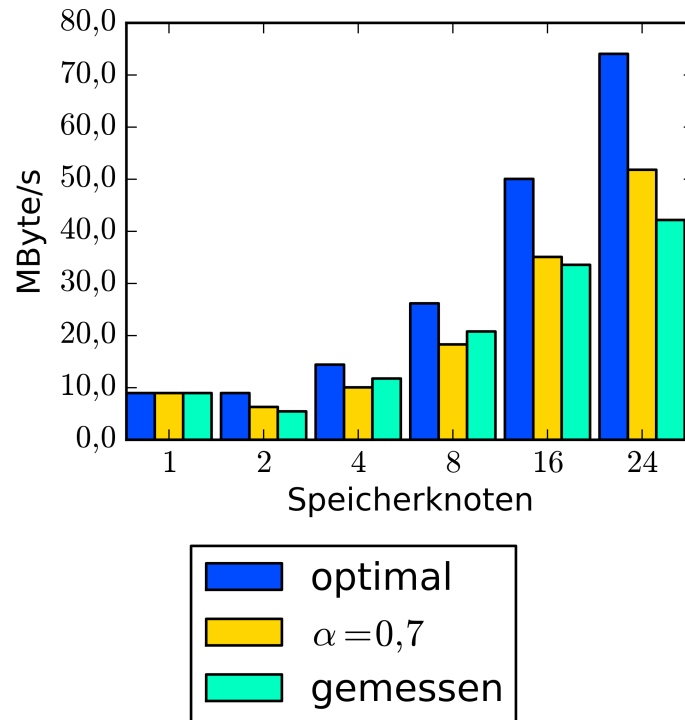


Abbildung 2.7: Skalierung des Speichersystems mit *RAID 1*

#### Vermessung mit *RAID 1*

Die Auswirkungen einer einfachen Spiegelung haben Brinkmann und Effert wieder über simulierte RAM-Disks gezeigt, so dass die Netzwerkanschlüsse der Datenserver saturiert werden können. Wie für die Tests ohne Redundanz wurde auf jedem Server eine ein TByte große RAM Disk erstellt und per *iSCSI* an die anderen Datenserver exportiert. Aus der darüber gebildeten Speichergruppe wurden für jeden Datenserver vier virtuelle Laufwerke erstellt. Jeweils zwei dieser virtuellen Laufwerke wurden zu einem *RAID 1* Laufwerk zusammengefasst. Diese wurden wieder per *iSCSI* an jeweils einen Client exportiert und dort vermessen.

Abbildung 2.7 zeigt wie sich der Durchsatz des Speichernetzes entwickelt. Als erwartete Skalierung wurde die zuvor gezeigt Formel verwendet:

$$B_{vd}^{k-Mirror} = n \cdot \frac{n}{(k+1) \cdot n - k} \cdot b \cdot \alpha$$

Für die als optimal bezeichneten Werte wurde  $\alpha = 1$  gesetzt. Aus den gemessenen Ergebnissen für  $2 \leq n \leq 16$  wurde der Wert  $\alpha = 0,7$  als Parallelisierungskosten berechnet. Wie in Abbildung 2.7 zu sehen, skaliert das Netzwerk bis dahin gemäß der berechneten Formel. Für  $n = 24$  wurde wie zuvor der Switch saturiert.

## 2 Virtualisierung von Blockgeräten

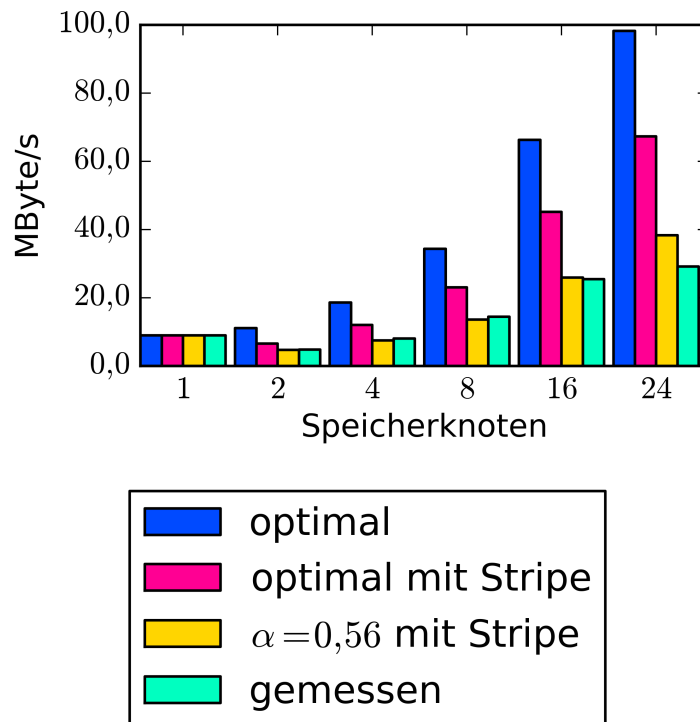


Abbildung 2.8: Skalierung des Speichersystems mit *RAID 5*

### Vermessung mit *RAID 5*

Als Beispiel für *Erasure Codes* haben Brinkmann und Effert ein *RAID 5* über den Speicherknotten vermessen. Wie zuvor wurden virtuelle RAM-Disks verwendet, per *iSCSI* exportiert und in einer Speichergruppe zusammengefasst. Danach wurden für jeden Server zehn virtuelle Laufwerke mit einer Kapazität von jeweils 40 GByte erstellt. Über jeweils fünf dieser Laufwerke wurde auf den Datenservern ein *RAID 5*-Laufwerk erstellt und an einen Client per *iSCSI* exportiert. Auf den Clients wurden die Laufwerke wieder eingebunden und mit *IOMeter* vermessen.

In Abbildung 2.8 ist als optimal die zuvor gezeigt Skalierung für  $\alpha = 1$ ,  $v = 4$  und  $u = 5$  eingezeichnet:

$$B_{vd}^{u,v} = n \cdot \frac{v \cdot n}{(v+u) \cdot n - u} \cdot b \cdot \alpha$$

$$\approx \frac{v \cdot n}{v+u} \cdot \alpha \cdot b$$

Vergleiche ich diese Werte mit den gemessenen Durchsätzen, so ergibt sich ein unerwartetes Ergebnis. Bei gleicher Anzahl an Servern sollte der Durchsatz bei Verwendung von *RAID 5* höher sein, als bei den vorherigen Testläufen mit gespiegelten Laufwerken. Für mehr als einen

Server ist der Durchsatz des Netzwerks allerdings geringer. Auch folgt die Skalierung nicht der gezeigten Formel.

Der Grund dafür liegt in der Behandlung der Anfragen im verwendeten *iSCSI* Target. Jeder 32 KByte Schreibzugriff wird hier in 4 KByte Zugriffe zerlegt, welche einzeln durch den darunter liegenden *page cache* des Kernels verarbeitet werden. Dadurch wird der erste Block jedes *Stripes* als neuer *Stripe* angesehen. Um diesen zu initialisieren müssen die restlichen Datenblöcke des *Stripes* zunächst gelesen werden, um dann die Prüfsummen bilden und speichern zu können. Dadurch sind zusätzliche 32 KByte an Lesezugriffen nötig. Weiterhin müssen 40 KByte auf den Knoten in dem *Stripe* geschrieben werden. Dadurch entsteht folgende notwendige Bandbreite um über das voll-duplexe Netzwerk im erwarteten Fall eine Bandbreite von  $x_n$  von jedem Datenserver an die Clients liefern zu können:

$$b = \max \left( \frac{13 \cdot n - 9}{4 \cdot n} \cdot x_n, \frac{9 \cdot n - 9}{4 \cdot n} \cdot x_n \right)$$

Der erste Wert der Maximum-Funktion beschreibt die notwendige Empfangsleistung, der zweite Wert die Sendeleistung. Die Empfangsleistung überwiegt, und so kann die Formel nach  $x_n$  umgeformt werden:

$$x_n = \frac{4 \cdot n}{13 \cdot n - 9} \cdot b$$

Als erwartete maximale Bandbreite des Speichernetzes ergibt sich somit:

$$\begin{aligned} B_{vd}^{4,5-rs} &= n \cdot \frac{4 \cdot n}{13 \cdot n - 9} \cdot b \cdot \alpha \\ &\approx \frac{4 \cdot n}{13} \cdot \alpha \cdot b \end{aligned}$$

Abbildung 2.8 zeigt als optimal mit *Stripe* die erwartete Entwicklung für  $\alpha = 1$ . Mittels dieser Daten wurde wieder für die Messungen mit bis zu 16 Datenservern der Parallelisierungsaufwand berechnet. Mit  $\alpha = 0,56$  skaliert das Netzwerk bis 16 Knoten gemäß der berechneten Formel, danach ist der Switch durch die zusätzliche Kommunikation saturiert.

## 2.4 Redundanz innerhalb einer Speichergruppe

Im vorangegangenen Abschnitt 2.3 habe ich *V:Drive* verwendet, um die Skalierungseigenschaften eines Netzwerks bei Verwendung verschiedener Methoden zur redundanten Speicherung von Daten zu zeigen. Dabei wurden mehrere virtuelle Laufwerke aus der gleichen Speichergruppe verwendet, um darüber ein *RAID* Laufwerk zu erstellen.

## 2 Virtualisierung von Blockgeräten

Dieser Ansatz bietet jedoch keinen wirkliche Schutz vor Ausfällen. An keiner Stelle wird sicher gestellt, dass nicht zwei virtuelle Laufwerke die *Chunks* eines *Stripes* auf dem selben physikalischen Laufwerk ablegen. *V:Drive* verteilt die Daten jedes virtuellen Laufwerks zufällig uniform über die Festplatten einer Speichergruppe und beachtet dabei nicht, wo die Daten anderer virtueller Laufwerke platziert wurden.

Die zuvor in *V:Drive* erwähnte Unterstützung für *RAID*-Laufwerke umgeht dieses Problem, indem sie virtuelle Laufwerke aus verschiedenen Speichergruppen verwendet. Da keine physikalische Festplatte gleichzeitig zu zwei Speichergruppen gehören kann, ist die redundante Datenhaltung gesichert. Allerdings verlangt dieses Vorgehen eine Partitionierung des vorhandenen physikalischen Speichers in verschiedene Speichergruppen.

Um die redundante Speicherung innerhalb einer Speichergruppe zu ermöglichen, werden also Verteilungsalgorithmen benötigt, welche es erlauben  $k$  Elemente so über  $n \geq k$  Festplatten zu verteilen, dass keine zwei Elemente auf der gleichen Festplatte platziert werden. Im folgenden Abschnitt zeige ich, dass diese Forderung zu erfüllen nicht trivial ist, falls weiterhin über heterogenen Festplatten ein Verteilung beibehalten werden soll, die jede Festplatte fair gemäß ihrer Kapazität belastet. Danach stelle ich den Algorithmus *Redundant Share* vor, der dieses Problem löst.

## 3 Strategien zur redundanten Datenverteilung

Im vorherigen Abschnitt habe ich die *CBV V:Drive* vorgestellt. Diese bildet über einer Menge physikalischer Festplatten eine Speichergruppe. Dabei gehört ein physikalisches Speichergerät zu maximal einer Gruppe. Aus dem Speicher einer solchen Gruppe werden dann die virtuellen Laufwerke erstellt, die durch andere Anwendungen verwendet werden können. Abbildung 3.1 zeigt eine Menge physikalischer Festplatten, aus denen zwei virtuelle Laufwerke exportiert werden. Teil meiner Definition einer *CBV* ist, dass ein virtuelles Laufwerk auf mehreren Rechner existieren kann. In dem Beispiel in der Abbildung steht das Laufwerk A auf beiden Datenservern zur Verfügung. Dabei ist davon auszugehen, dass entweder beide Server auf den gesamten physikalischen Speicher über ein *SAN* zugreifen können oder dass die Anfragen an den Server weitergereicht werden, welcher den physikalischen Speicher hält, wie in Abschnitt 2.3 beschrieben.

Im vorherigen Kapitel habe ich in Abschnitt 2.1 verschiedene Verfahren vorgestellt, um Daten redundant über einer Menge von Festplatten zu speichern. Neben der Platzierung der Daten enthielten diese Algorithmen auch Vorschriften zur Berechnung der Redundanzdaten. Innerhalb dieses Kapitels betrachte ich nur die Platzierung von Daten.

Die in Abschnitt 2.1.1 vorgestellten Verfahren arbeiten auf einer festen Menge von Festplatten. Die zu Anfang gewählte Konfiguration kann nur durch Replatzierung fast aller Daten um eine Festplatte erweitert oder verkleinert werden. Beispielsweise beschreiben Gonzales und Cortes dieses Problem in [Gonzalez und Cortes, 2004]. Diese Beschränkung kann durch regel- oder tabellenorientierte Verfahren aufgehoben werden. Der Nachteil von tabellenorientierten Verfahren ist allerdings, dass sie sich nicht zur Speicherung einer beliebig großen Menge von Blöcken eignen, da sie für jeden platzierten Block speichern müssen, auf welchen Festplatten seine Kopien platziert wurden. Die Menge dieser Metadaten ist bei großen Speichersystemen nur mit großem Aufwand zu bewältigen. Regelbasierte Verfahren haben unter schlechten Bedingungen Fragmentierungsprobleme, sodass eine gelegentliche Defragmentierung notwendig ist.

Innerhalb dieses Kapitels löse ich das Probleme der redundanten Datenverteilung durch die pseudorandomisierte Verteilung mittels Hashfunktionen. Dazu stelle ich zunächst in Abschnitt 3.1 das Modell vor, welches meinen folgenden Betrachtungen zu Grunde liegt.

Im anschließenden Abschnitt 3.2 werden einige auf Hashfunktionen basierende Verfahren zur

### 3 Strategien zur redundanten Datenverteilung

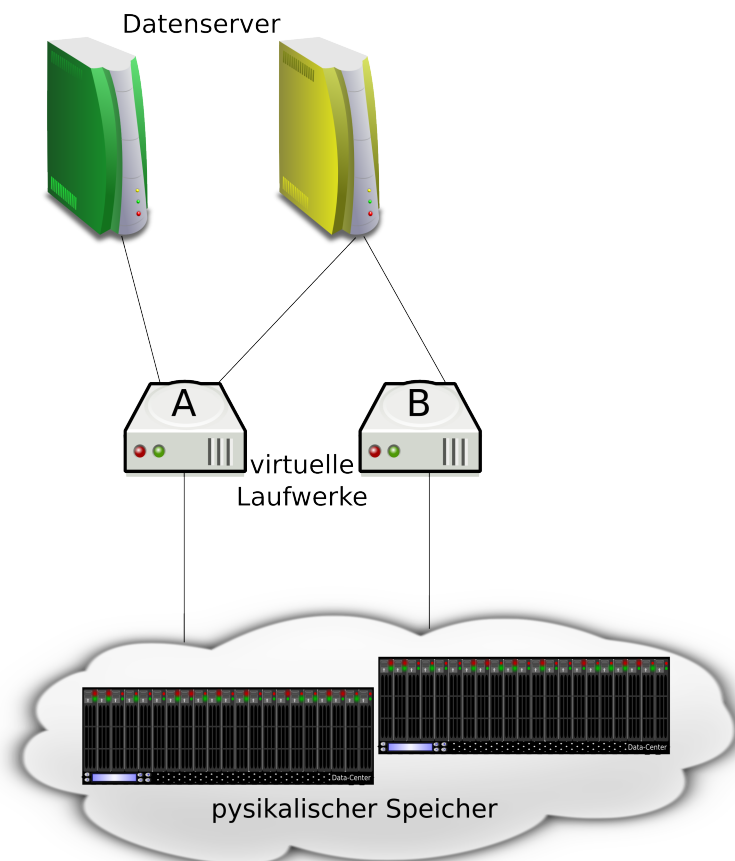


Abbildung 3.1: Aufbau der virtuellen Laufwerke in *V:Drive*

Platzierung von Daten vorgestellt. Dabei gehe ich auf zwei Verfahren näher ein. Das erste Verfahren dient der Platzierung über homogenen Festplatten, das zweite Verfahren kann auch heterogene Festplatten verwalten. Keines der beiden Verfahren unterstützt Redundanz. Diese Verfahren werden auch in der in Abschnitt 2.2 beschriebenen Speichervirtualisierung *V:Drive* eingesetzt.

In Abschnitt 3.3 gehe ich auf die grundsätzlichen Probleme der redundanten Datenverteilung ein. Ich zeige eine Schranke für die Heterogenität von Festplatten bei der Platzierung von mehr als zwei Kopien auf. Ausserdem erkläre ich, warum das mehrfache Verwenden nicht redundanter Verfahren zur redundanten Platzierung nicht immer jede Festplatte gemäß ihrer Kapazität auslasten kann.

Das in Abschnitt 3.4 vorgestellte Verfahren *Redundant Share* erfüllt die Anforderung an die Redundanz, indem Kopien jedes Blocks auf unterschiedlichen physikalischen Festplatten gespeichert werden. Statt Kopien können mittels *Redundant Share* auch kodierte *Stripes* verteilt werden, wie sie mittels der in Abschnitt 2.1.1 vorgestellten *Erasure-Codes* erzeugt werden. In diesem Fall erfordert das Hinzufügen und Entfernen von Festplatten zusätzlichen Aufwand.



Ich stelle *Redundant Share* in mehreren Schritten vor. Das in Abschnitt 3.4.1 vorgestellte *LinPlace* platziert Daten ohne Redundanz. In Abschnitt 3.4.2 zeige ich darauf aufbauend *LinMirror*, welches zwei Kopien jedes Blocks platziert. In Abschnitt 3.4.3 erweitere ich *LinMirror*, um eine beliebige Anzahl an Kopien platzieren zu können. Dieses Verfahren bezeichne ich als *Redundant Share*. In Abschnitt 3.4.4 verbessere ich die Laufzeit von *Redundant Share* mittels einiger in 3.2 vorgestellter Verfahren. Dieses beschleunigte Verfahren nenne ich *Fast Redundant Share*.

Abschließend betrachte ich in Abschnitt 3.5, wie sich *Redundant Share* verhält, falls zuvor gespeicherte Elemente gefunden werden sollen, ohne dass alle beteiligten physikalischen Festplatten einer Konfiguration bekannt sind. Dazu stelle ich eine Modifikation von *Redundant Share* vor, welche für dieses Szenario angepasst ist.

## 3.1 Modell

In diesem Abschnitt stelle ich das Modell vor, innerhalb dessen ich die verschiedenen Verteilungsalgorithmen theoretisch betrachte. Weiterhin definiere ich die Kriterien, nach denen ich die Qualität eines Verteilungsalgorithmuses beurteile.

Die Daten der virtuellen Festplatte sollen über eine Menge von  $n$  physikalischen Festplatten verteilt werden. Solch eine Menge von Festplatten bezeichne ich als Konfiguration.  $d_i$  bezeichnet die Festplatte  $i \in \{1, \dots, n\}$  und ihren eindeutigen Identifizierer.  $c_i$  ist die Kapazität von  $d_i$  und  $L_i$  ihre aktuelle Last, also die Anzahl der auf ihr platzierten Daten.  $C = \sum_{i=1}^n c_i$  ist die Gesamtkapazität des Speichersystems.

Über die Festplatten sollen  $m$  Blöcke gleicher Größe verteilt werden, wobei  $b_j$  den Block an der Position  $j \in \{1, \dots, m\}$  auf der virtuellen Festplatte bezeichnet. Bei der  $k$ -redundanten Datenverteilung werden  $k$  Kopien jedes Blocks platziert. Insgesamt werden  $m \cdot k$  Daten über die Festplatten verteilt. Zur besseren theoretischen Betrachtung definiere ich  $N$  als die maximale Anzahl an Festplatten und  $M$  als die maximale Anzahl an Blöcken, die zu einem beliebigen Zeitpunkt im System sein können. Diese Werte werden nur in Abschnitt 3.2 verwendet.

Sowohl bei der Benennung der Festplatten als auch bei der Benennung der Blöcke gehe ich davon aus, dass sie aufsteigend durchnummeriert sind. Dies geschieht nur zur Vereinfachung der Beschreibung, die Blöcke und die Festplatten brauchen lediglich eindeutige Identifizierer, damit die Aussagen in diesem Kapitel Bestand haben.

**Definition 3.1.1.** Ein  $[n, k]$ -Verteiler, oder kurz Verteiler, ist definiert als ein Algorithmus, welcher aus  $n$  gegebenen Festplatten  $k \leq n$  Festplatten auswählt. Ein  $[n]$ -Verteiler ist definiert als  $[n, 1]$ -Verteiler.

Ich setze für die theoretischen Analysen der Laufzeiten und des Hauptspeicherverbrauchs in dieser Arbeit eine Registermaschine (RAM) voraus, welche beliebige rationale Zahlen in einem

### 3 Strategien zur redundanten Datenverteilung

Speicherregister vorhalten kann. Ferner setze ich voraus, dass arithmetische Operationen zweier Zahlen miteinander, sowie Operationen zum Lesen, Schreiben und Kopieren einer reellen Zahl in konstanter Zeit möglich sind.

Die im Folgenden vorgestellten Verteiler arbeiten an verschiedenen Stellen mit einer Hashfunktion zur Pseudorandomisierung von Identifizierern. Idealerweise modelliert man diese Hashfunktion als *Zufallsorakel*, welches die Elemente aus einer Wertemenge  $W$  auf eine Zielmenge  $Z$  abbildet. Hierzu definiere ich die Funktion  $\text{rand}(W) \rightarrow Z$ , welche zu einem Wert  $w \in W$  zufällig uniform verteilt einen Wert  $z \in Z$  bestimmt. Für ein festes  $w$  liefert  $\text{rand}(w)$  immer den gleichen Zielwert. Für eine Sequenz  $w_1, \dots, w_x$  mit  $w_i \in W$  für  $i \in [1, \dots, x]$  liefert  $\text{rand}(w_i)$  paarweise unabhängige Werte. Innerhalb meiner Betrachtungen zur Laufzeit und zum Hauptspeicherverbrauch gehe ich davon aus, dass die Funktion  $\text{rand}$  eine konstante Laufzeit und einen konstanten Hauptspeicherverbrauch hat.

Im Folgenden untersuche ich verschiedene Verteiler gemäß folgender Kriterien:

**Zeiteffizienz:** Die Zeiteffizienz beschreibt die Laufzeit eines Verteilers. Ich betrachte, wie sich die Laufzeit für unterschiedliche Konfigurationen verhält, also speziell wie die Laufzeit von der Anzahl der Festplatten und der Anzahl der Datenelemente abhängig ist.

**Speichereffizienz:** Die Speichereffizienz beschreibt den erwarteten Hauptspeicherverbrauch eines Verteilers. Ich betrachte, wie sich der Speicherverbrauch für unterschiedliche Konfigurationen verhält, also speziell wie der Hauptspeicherverbrauch von der Anzahl der Festplatten und der Anzahl der platzierten Datenelemente abhängt.

**Redundanz:** Die Redundanz beschreibt die Anzahl der platzierten Kopien jedes Blocks. Folgend der Definition 3.1.1 besitzt ein  $[n, k]$ -Verteiler eine Redundanz von  $k$ . Einen  $[n, 1]$ -Verteiler bezeichne ich auch als nicht redundant.

**Fairness:** Die Last der virtuellen Festplatten soll gemäß der Kapazitäten der physikalischen Festplatten möglichst gleichmäßig verteilt sein. Ein optimaler  $[n, k]$ -Verteiler sollte im erwarteten Fall jeder Festplatte  $d_i$  mit  $i \in [1, \dots, n]$  einen Faktor von  $\lambda = k \cdot \frac{c_i}{C}$  der Datenlast zuweisen. Die Fairness beschreibt den Faktor, um den ein Verteiler im erwarteten Fall schlechter verteilt, als ein optimaler Verteiler. Hierbei ist der Erwartungswert bezüglich der zufälligen Abbildung mittels der soeben beschriebenen Funktion  $\text{rand}$  gemeint. Ein Verteiler ist  $\alpha$ -fair wenn für jede Festplatte  $d_i$  für  $i \in [1, \dots, n]$  gilt:

$$E[L_i] \leq \alpha \cdot k \cdot \frac{c_i}{C}$$

**Adaptivität:** Wird eine Festplatte mit der Kapazität  $c^{\text{neu}}$  einer Konfiguration hinzugefügt, so schreibt die Fairness vor, dass diese Festplatte im Idealfall einen Anteil von  $\lambda = k \cdot \frac{c^{\text{neu}}}{C + c^{\text{neu}}}$  der auf dem Speichersystem abgelegten Daten aufnehmen muss. Äquivalent dazu müsste eine zu entfernende Festplatte  $d_i$  einen Anteil von  $\lambda = k \cdot \frac{c_i}{C}$  der Daten enthalten. Ein

bezüglich der Adaptivität und Fairness perfektes Verfahren müsste genau diese Daten im Speichersystem bewegen, wenn eine Festplatte zu einer Konfiguration hinzugefügt oder von ihr entfernt wird.

Die Adaptivität beschreibt den Faktor an Replatzierungen, die vorgenommen werden müssen, wenn einer Konfiguration eine Festplatte hinzugefügt oder entfernt werden soll. Ich bezeichne  $r$  als die Replatzierungsschritte des Verfahrens beim Hinzufügen oder Entfernen einer Festplatte. Ein Verfahren ist  $\alpha$ -adaptiv, wenn dabei folgendes gilt:

$$E[r] = \alpha \cdot \lambda$$

Hierbei ist wieder der Erwartungswert bezüglich der zufälligen Abbildung mittels der soeben beschriebenen Funktion  $\text{rand}$  gemeint.

**Heterogenität:** Ich bezeichne eine Konfiguration als homogen, wenn sich die Kapazitäten der Festplatten der Konfiguration nicht unterscheiden. Andernfalls spreche ich von einer heterogenen Konfiguration. Ein heterogener Verteiler kann die Eigenschaften der Fairness auch für heterogene Festplattenkonfigurationen aufrecht erhalten, ein homogener Verteiler nur für homogene Konfigurationen.

Die Adaption der Datenverteilung nach dem Hinzufügen oder Entfernen einer Festplatte folgt einem festem Schema. Jeder Block, welcher mittels eines Verteilers  $v$  bezüglich der ursprünglichen Konfiguration platziert wurde, wird mittels eines Verteilers  $v'$  bezüglich der neuen Konfiguration verteilt. Wurde eine Kopie mittels  $v$  auf einer anderen Festplatte platziert als mittels  $v'$ , so ist eine Replatzierung notwendig.

Ein  $[n, k]$ -Verteiler berechnet für jeden Aufruf für ein  $b_j$  mit  $j \in [1, \dots, m]$  einen Tupel  $x = \{x_1, \dots, x_k\}$  mit Indizes von Festplatten. Werden auf jeder der  $k$ -Festplatten Kopien des Blocks  $b_j$  gespeichert, so ist es nicht nötig, die Kopien voneinander unterscheidbar zu machen. Daher kann ein Verteiler die Reihenfolge der Festplatten, die er für  $b_j$  zurückliefert, beliebig permutieren.

Wie bereits in der Einleitung dieses Kapitels erwähnt, soll ein  $[n, k]$ -Verteiler allerdings auch verwendet werden können, um eine komplexere Kodierung der Daten zu ermöglichen. Beispielsweise sollen auch *Stripes*, welche durch die in Abschnitt 2.1.1 vorgestellten *Erasur Codes* kodiert wurden, verteilt werden können. Da in diesem Fall die Festplatten unterschiedliche Daten erhalten, ist es notwendig, für  $b_j$  immer das gleiche Tupel von Festplatten zu erhalten, die Reihenfolge darf also nicht verändert werden. Wird die Reihenfolge der Festplatten im Tupel doch verändert, so müssen auch die zugehörigen Daten verschoben werden.

Ich unterteile bei der Betrachtung der Adaptivität also zwischen folgenden Anforderungen:

**Adaptivität ohne Beachtung der Reihenfolge der Kopien:** Die Reihenfolge der Festplatten, auf denen Kopien eines Blocks platziert werden sollen, darf beliebig permutiert

### 3 Strategien zur redundanten Datenverteilung

werden. Die einzelnen Kopien sind nicht unterscheidbar. Die Verschiebung einer Kopie innerhalb des Ergebnistupels zählt nicht als Umplatzierung.

**Adaptivität mit Beachtung der Reihenfolge der Kopien:** Die Reihenfolge der Festplatten, auf denen Kopien eines Blocks platziert werden sollen, darf nicht permutiert werden. Die einzelnen Kopien sind unterscheidbar. Die Verschiebung einer Kopie innerhalb des Ergebnistupels zählt als Umplatzierung.

## 3.2 Stand der Technik

Wie bereits in der Einleitung dieses Kapitels erklärt, stelle ich in Abschnitt 3.4 eine Hashfunktion zur redundanten Platzierung von Daten vor. Ich gebe in diesem Abschnitt einen Überblick über die wichtigsten Verfahren und gehe auf zwei Verfahren näher ein.

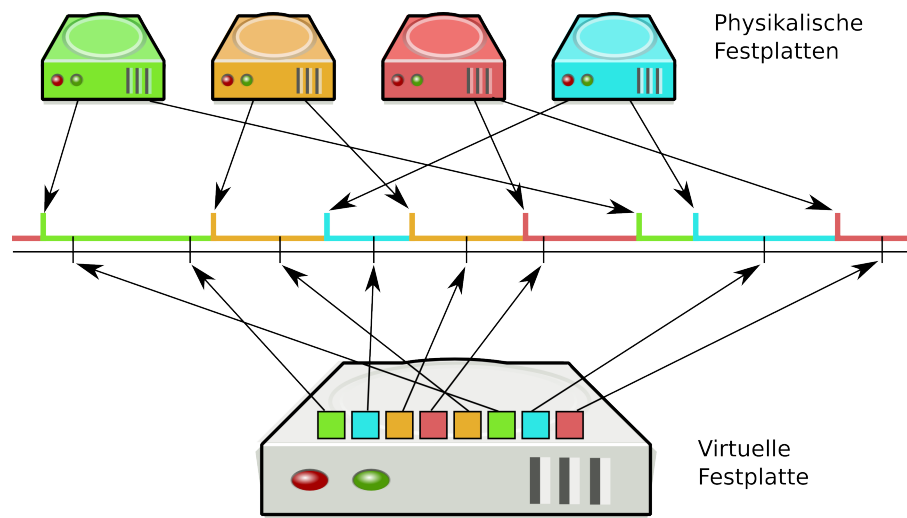
Generell kann die Verteilung von Daten über homogenen Festplatten mittels recht einfacher Hashfunktionen gelöst werden. Eine solche Hashfunktion zur Platzierung könnte wie folgt aussehen:

$$h(b_i) = (p_1 \cdot b_i + p_2) \bmod n$$

Mit unterschiedlichen Primzahlen  $p_1$  und  $p_2$  liefert  $h(b_i)$  eine Verteilungsfunktion zur Platzierung von Daten auf homogenen Festplatten. Ändert sich bei dieser Strategie allerdings die Anzahl der verwendeten Festplatten  $n$ , so führt dies unweigerlich zu sehr vielen Umplatzierungen der Daten auf den Festplatten, damit ihre Ordnung wieder der Hashfunktion folgen.

Die pseudorandomisierte Verteilung von Daten zur Lastenverteilung wurde besonders im Bereich der Bereitstellung von Videodaten untersucht, um möglichst unabhängige Zugriffsmuster auf Daten zu erreichen. Beispielsweise haben Alemany und Thathacher in [Alemany und Thathacher, 1997] die pseudorandomisierte Verteilung von Videodaten für Nachrichten auf Abruf vorgestellt. Dabei speichern sie die Videodaten auf mehreren zufällig ausgewählten Servern, um eine gleichmäßige Auslastung zu erreichen. Santos u.a. haben in [Santos und Muntz, 1998a,b; Santos u. a., 2000] auch Festplatten mit unterschiedlicher Kapazität und Bandbreite in die Betrachtung aufgenommen. Durch die Ablage einer temporären oder permanenten Kopie einzelner Blöcke versuchen sie zu große Zugriffsmengen auf einzelne Festplatten zu vermeiden. Keines dieser Verfahren ist auf eine dynamische Umgebung ausgelegt, das Hinzufügen und Entfernen von Festplatten erfordert einen nicht absehbaren Umplatzierungsaufwand.

Sanders hat in [Sanders, 2001] ein Modell zur Simulation von Festplatten mit mehreren Köpfen vorgestellt. Dabei verteilt er Daten pseudorandomisiert über mehrere Festplatten. Durch die Ablage von Kopien jedes Blocks erhöht er die Wahrscheinlichkeit, dass Blöcke sofort gelesen oder geschrieben werden können. Er betrachtet allerdings weder heterogene Festplatten noch die Adaptivität.

Abbildung 3.2: Datenverteilung bei *Consistent Hashing*

Adaptive Hashfunktionen lösen dieses Problem. Im Folgenden bezeichne ich eine Hashfunktion als adaptiv, wenn sie eine von  $m$  unabhängige Adaptivität besitzt.

Um eine adaptive Hashfunktion zu erzeugen, ist es notwendig die Abbildung der Festplatten auf den Wertebereich der verwendeten Hashfunktion zu lösen. Die wahrscheinlich verbreitetste Hashfunktion, die diese Möglichkeit bietet, ist das von Karger u. a. vorgestellte *Consistent Hashing* [Karger u. a., 1997; Lawin, 1998; Karger u. a., 1999]. Diese Funktion stelle ich näher vor, da sie später weitere Verwendung findet.

Bei der Datenverteilung mittels *Consistent Hashing* werden zunächst  $k$  Kopien jeder Festplatte in einem  $[0, 1)$ -Intervall platziert. Die Größe von  $k$  ist beliebig und kann von  $n$  abhängig sein. Dazu wird eine uniforme Hashfunktion  $g(\{0, \dots, k-1\}, \{0, \dots, N-1\}) \rightarrow [0, 1)$  verwendet. Diese Hashfunktion muss den gleichen Anforderungen genügen, wie die in Abschnitt 3.1 beschriebene Funktion  $\text{rand}$ . In Abbildung 3.2 werden auf diese Art zwei Kopien von vier physikalischen Festplatten verteilt.

Aufbauend auf der Verteilung der Festplatten verwendet *Consistent Hashing* eine zweite unabhängige Hashfunktion  $f : \{0, \dots, M-1\} \rightarrow [0, 1)$ , welche die Blöcke uniform in ein  $[0, 1)$ -Intervall abbildet. Um einen Block  $b_i$  zu platzieren errechnet *Consistent Hashing* mittels der Funktion  $f$  einen Punkt in dem  $[0, 1)$  Intervall. Der Block wird dann auf der Festplatte platziert, welcher mittels  $g$  der nächst kleinere Wert zugeordnet wurde. Dabei betrachtet *Consistent Hashing* das  $[0, 1)$ -Intervall als Ring, die an letzter Stelle im  $[0, 1)$  Intervall platzierte Festplatte ist also auch für den Bereich von 0 bis zur an erster Stelle platzierten Festplatte zuständig. Abbildung 3.2 zeigt dies beispielhaft für eine virtuelle Festplatte. Karger u. a. haben gezeigt, dass bei Platzierung von  $k = \Theta(\log n)$  Kopien jeder Festplatte *Consistent Hashing* mit hoher Wahrscheinlichkeit eine Verteilung mit einer Fairness von  $1 + \varepsilon$  erreicht, wobei  $\varepsilon$  von der Anzahl der

### 3 Strategien zur redundanten Datenverteilung

platzierten Kopien jeder Festplatte abhängt.

Karger u. a. konnten zeigen, dass *Consistent Hashing* in konstanter Zeit einen Block platzieren kann. Hierzu wird das  $[0, 1)$  Intervall in  $k \cdot N \cdot \log N$  gleich große Teilintervalle unterteilt. Bei der Platzierung eines Elements kann nun in konstanter Laufzeit bestimmt werden, in welchem Intervall es liegt, und es müssen nur noch die Kopien des Intervalls untersucht werden. Im erwarteten Fall ist dies nur eine Kopie, welche sich somit in konstanter Laufzeit bestimmen lässt.

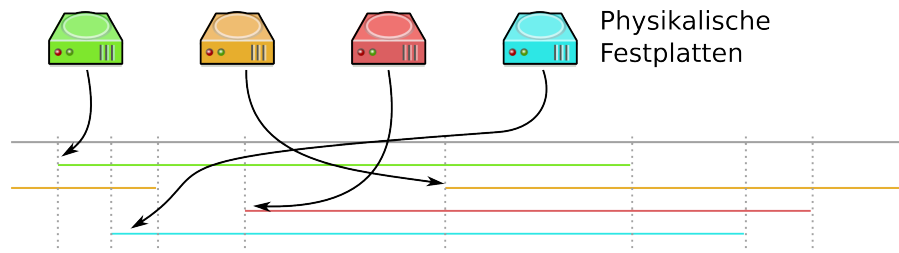
*Consistent Hashing* ist 1-adaptiv. Wird eine neue Festplatte eingeführt, so wird sie mit den anderen Festplatten im  $[0, 1)$ -Intervall platziert. Die Datenblöcke innerhalb der Intervalle der neuen Festplatte werden nun auf diese verschoben. Alle anderen Blöcke bleiben davon unberührt. Umgekehrt werden beim Entfernen einer Festplatte nur die Datenblöcke der zu entfernenden Festplatte auf die restlichen Festplatten verteilt.

Brinkmann u. a. haben 2000 in [Brinkmann u. a., 2000] ein anderes Verfahren zur Verteilung von Daten über homogenen Festplatten vorgestellt. Salzwedel hat dieses Verfahren in seiner Dissertation [Salzwedel, 2004] als *Cut-and-Paste* bezeichnet. Auch bei diesem Verfahren werden die Blöcke durch eine Pseudorandomisierung auf den  $[0, 1)$ -Raum abgebildet. Jeder Festplatte wird ein Bereich gleicher Größe dieses Intervalls zugeordnet. Wird eine neue Festplatte eingefügt, so erhält sie einen Teilbereich des Intervalls jeder ursprünglichen Festplatte. Gegenüber *Consistent Hashing* hat dieses Verfahren den Vorteil, dass jede Festplatte den exakt gleichen Anteil am  $[0, 1)$ -Intervall besitzt. Dafür entsteht bei dem Verfahren eine mit der Anzahl der Festplatten ansteigende Fragmentierung des  $[0, 1)$ -Raums beim Hinzufügen weiterer Festplatten. Um diese Fragmentierung beim Entfernen einer Festplatte  $d_i$  mit  $i \in \{1, \dots, n\}$  nicht weiter anwachsen zu lassen, entfernen Brinkmann u. a. immer zunächst die Festplatte  $d_n$  und verschieben alle Daten von  $d_i$  auf  $d_n$ . Abschließend vertauschen  $d_i$  und  $d_n$  ihre Identitäten. So erreicht *Cut-and-Paste* eine Adaptivität von 2 beim Entfernen einer Festplatte.

Die bisher vorgestellten Verfahren verteilen die Daten lediglich an Hand der Anzahl der verwendeten Festplatten. Dadurch erhält jede Festplatte die gleiche Last. Bei heterogenen Festplatten ist es aber nötig, jeder Festplatte einen Anteil gemäß ihrer Kapazität an den zu speichernden Daten zu geben.

Um dieses Problem zu lösen, kann die Anzahl an Kopien, welche *Consistent Hashing* von einer Festplatte platziert, abhängig von der Größe der jeweiligen Festplatte gewählt werden. Brinkmann u. a. haben in [Brinkmann u. a., 2002] gezeigt, dass dafür insgesamt  $\Omega(\min(\frac{c_{\max}}{c_{\min}}, m))$  Kopien verteilt werden müssen, wobei  $c_{\max}$  die Größe der größten Festplatte ist und  $c_{\min}$  die Größe der kleinsten Festplatte. Dies würde einen Speicherverbrauch von  $\Theta(m)$  bedeuten, also einen Bruch der Anforderung an die Speichereffizienz.

Das Verteilungsverfahren *Share* [Brinkmann u. a., 2002; Salzwedel, 2004] erreicht eine faire Verteilung über heterogenen Festplatten indem es zwei Phasen durchläuft. In der ersten Phase wird dieses heterogene Problem in eine Menge homogener Probleme überführt, welche dann

Abbildung 3.3: Platzierung der heterogenen Festplatten bei *Share*

z.B. mittels *Consistent Hashing* gelöst werden können. Zum Auflösen der Heterogenität wird eine Hashfunktion benötigt, welche ähnlich der Funktion in *Consistent Hashing* jeder Festplatte pseudorandomisiert einen Wert im  $[0, 1)$ -Intervall zuordnet. Dieser Wert wird als Startpunkt der Festplatte angenommen. Weiterhin wird jeder Festplatte ein Bereich der Länge  $\frac{c_i \cdot s}{C}$  zugeordnet.  $s \geq 1$  ist der Dehnungsfaktor und für alle Festplatten gleich. Er stellt sicher, dass mit hoher Wahrscheinlichkeit jedem Wert im  $[0, 1)$ -Intervall mindestens eine Festplatte zugeordnet wird. Brinkmann u. a. haben gezeigt, dass dazu ein Faktor von  $s = l \cdot \ln N$  mit  $l \geq 3$  ausreicht.

In Abbildung 3.3 wurden auf diese Art vier Festplatten platziert. Im nächsten Schritt unterteilt *Share* das  $[0, 1)$ -Intervall an jedem Start- oder Endpunkt einer physikalischen Festplatte. Auf diese Weise entstehen maximal  $2 \cdot n$  Intervalle. Zu jedem dieser Intervalle wird eine weitere Hashfunktion erstellt, welche eine uniforme homogene Verteilung über die Festplatten vornimmt, die das jeweilige Intervall abdecken. Dabei kann es vorkommen, dass eine Festplatte ein Intervall mehrfach überdeckt, da  $\frac{c_i \cdot s}{C} > 1$  sein kann. Überdeckt eine Festplatte ein Intervall  $x$ -fach, so werden  $x$  Abbilder von ihr in der homogenen Hashfunktion verwendet.

Soll nun ein Block der virtuellen Festplatte platziert werden, so wird dieser zunächst mittels einer uniformen Hashfunktion in dem  $[0, 1)$ -Intervall platziert. Als Ergebnis liefert *Share* in dieser ersten Runde ein Intervall. Mittels der Hashfunktion für das Intervall wird entschieden, auf welcher physikalischen Festplatte der Block platziert werden soll. Brinkmann u. a. zeigen, dass sich auch dieser Algorithmus in einer erwarteten Zeit von  $O(1)$  realisieren lässt, falls die homogenen Hashfunktionen für die Intervalle einen Block in konstanter Zeit platzieren können.

Wie Brinkmann u. a. in [Brinkmann u. a., 2002] gezeigt haben, besitzt *Share* eine Fairness von  $1 + \varepsilon$  falls die uniformen Hashfunktionen für die Intervalle 1-fair sind. Dazu ist ein Dehnungsfaktor von  $s = \frac{6 \ln N}{\sigma^2}$  mit  $\sigma = \frac{\varepsilon}{1 + \varepsilon}$  nötig. Die Adaptivität von *Share* liegt bei  $2 + \varepsilon$ . Der Hauptspeicherverbrauch liegt bei  $O(s \cdot n \cdot h)$ , wobei  $h$  den Hauptspeicherverbrauch einer homogenen Hashfunktionen zur Verteilung der Daten in einem Intervall beschreibt.

Neben der Strategie *Share* haben Brinkmann u. a. in [Brinkmann u. a., 2002] auch die Strategie *Sieve* vorgestellt. Bei dieser Strategie werden  $L$  unabhängige Hashfunktionen  $H_l$  mit  $l \in \{1, \dots, L\}$  verwendet. In jeder der Hashfunktionen ist die Hälfte des  $[0, 1)$ -Raums durch die Festplatten gemäß ihrer Kapazität abgedeckt. Weiterhin werden  $L$  unabhängige Hashfunktionen  $\text{rand}_l(\{1, \dots, M\}) \rightarrow [0, 1)$  zur Pseudorandomisierung der Blöcke erstellt. Sei  $b_j$  ein zu plat-

### 3 Strategien zur redundanten Datenverteilung

zierender Block, so wird zunächst geprüft, ob  $\text{rand}_1 b_j$  den Block in  $H_1$  auf eine Festplatte  $d_i$  abbildet. Ist dies der Fall, so wird der Block auf  $d_i$  platziert. Wurde in Schritt  $l \in \{1, \dots, L-1\}$  ein freier Bereich in  $H_l$  getroffen, so wird im nächsten Schritt die Abbildung  $\text{rand}_{l+1}(b_j)$  auf  $H_{j+1}$  betrachtet. Wurden in allen  $L$  Schritten freie Bereiche der Hashfunktionen  $H$  getroffen, so wird eine zuvor ausgewählte Standardfestplatte gewählt. Durch Anpassung relativer Kapazitäten in den Hashfunktionen  $H$  konnten Brinkmann u. a. trotzdem eine faire Verteilung im erwarteten Fall erreichen.

Schomaker und Schindelhauer haben in [Schindelhauer und Schomaker, 2005] weitere Hashfunktionen vorgestellt. Schomaker hat in [Schomaker, 2007] und [Schomaker, 2008] gezeigt, dass sie sich zur Verteilung von Daten über heterogenen Konfigurationen eignen. Diese Verfahren erzeugen für jede Festplatte eine Funktion, welche als Distanzmaß für jeden platzierten Block dient. Für einen zu platzierenden Block werden die Distanzen der einzelnen Festplatten berechnet und die Festplatte mit der geringsten Distanz verwendet.

Die bisher vorgestellten Verfahren ermöglichen die Verteilung von Blöcken eines virtuellen Speichers über mehrere physikalische Festplatten unter Einhaltung der Kriterien Adaptivität und Speichereffizienz. Während *Consistent Hashing* dies für homogene Festplatten schafft, kann *Share* auch heterogene Festplatten verwalten. Beide Verfahren arbeiten jedoch ohne Redundanz. Bei Verwendung einer homogenen Konfiguration ist es möglich,  $k$  Kopien zu platzieren. Dazu werden unabhängige Experimente durchgeführt, bis die verwendete Hashfunktion  $k$  unterschiedliche Festplatten zurück gegeben hat. Wie ich in Abschnitt 3.3.2 zeige, liefert dieses Vorgehen bei einigen heterogenen Konfigurationen eine beliebig schlechte Fairness.

Innerhalb ihres regelbasierten Speichersystems *CRUSH* [Weil u. a., 2006b] verwenden Weil u. a. eine leicht abgewandelte Version der Hashfunktion  $h(b_i)$  um Daten redundant über homogenen Festplatten zu verteilen. Ihre Funktion ist abgeleitet von Honickys und Millers in *RUSH* [Honicky und Miller, 2004] verwendeten Funktionen, um mehrere Kopien eines Elements über homogenen Festplatten zu verteilen:

$$h(b_i, k) = (\text{rand}(b_i) + k \cdot p) \bmod n$$

$\text{rand}(b_i)$  ist eine Hashfunktion zur Pseudorandomisierung, wie ich sie in Abschnitt 3.1 eingeführt habe.  $p$  ist eine Primzahl größer als  $n$  und  $k$  die Kopie, die platziert werden soll. Auch diese Funktion unterliegt den zuvor beschriebenen Beschränkungen.

2004 haben Honicky und Miller in [Honicky und Miller, 2003] ein Verfahren zur redundanten Platzierung von Daten über heterogenen Speichersystemen veröffentlicht. Dieses Verfahren sichert allerdings nicht zu, dass keine zwei Kopien eines Blocks auf dem gleichen Speichersystem platziert werden, womit es für den Einsatz über einzelnen Festplatten ungeeignet ist.

Innerhalb von *CRUSH* stellen Weil u. a. auch verschiedene andere Hashfunktionen zur redundanten Platzierung von Daten über heterogenen Festplatten vor. Allerdings erreichen ihre Ver-



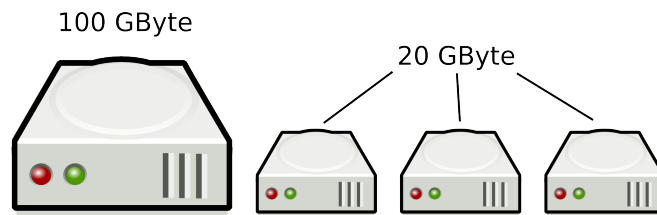


Abbildung 3.4: Beispiel einer zu heterogenen Konfiguration

teiler eine Fairness nur, indem sie regelbasiert die Verteilungsalgorithmen anpassen, beispielsweise wenn einige Festplatten drohen an ihre Kapazitätsgrenze zu stoßen. Außerdem ist die Erweiterung des Speichers nur in *chunks* möglich. Jeder dieser *chunks* muss aus  $k$  homogenen Servern bestehen, denn er muss alle Kopien jedes Blocks redundant speichern können.

Zeitgleich zur Entwicklung des später vorgestellten Verfahrens *Redundant Share* entstand die Verteilungsstrategie *SPREAD* [Mense und Scheideler, 2008; Mense, 2008], welche auch eine redundante Verteilung von Daten über heterogenen Festplatten erlaubt.

## 3.3 Probleme der redundanten Datenverteilung

Im Verlauf dieses Abschnitts erläutere ich, wo die Probleme beim Platzieren mehrerer Kopien auf heterogenen Festplatten liegen und warum die bisherigen Verfahren dieses Problem nicht lösen können ohne gegen eine der anderen Anforderungen zu verstoßen. Dazu verwende ich ein Modell, in dem  $k$  Kopien jedes Blocks auf unterschiedlichen physikalischen Festplatten gespeichert werden sollen. Die hier vorgestellten Ergebnisse wurden in [Brinkmann u. a., 2007] veröffentlicht.

### 3.3.1 Kapazitätseffizienz

Das große Problem bei der redundanten Datenplatzierung ist die Einhaltung der Kapazitätseffizienz. Diese Anforderung ist eng verwandt mit der Fairness und beschreibt die Fähigkeit eines Verfahrens, den gesamten physikalischen Speicher verwenden zu können.

Diese Anforderung betrachte ich näher. Ich gehe zur Vereinfachung davon aus, dass die Gesamtmenge des Speichers ein Vielfaches von  $k$  ist, andernfalls müssten Blöcke zerteilt werden. Eine triviale Anforderung wäre nun, dass der Algorithmus  $\frac{C}{k}$  verschiedene Elemente platzieren können muss. Leider ist dies bei heterogenen Festplatten nicht immer möglich.

Abbildung 3.4 zeigt eine Konfiguration mit vier Festplatten, bei welcher es nicht möglich ist, den gesamten Speicher zu verwenden wenn mehr als eine Kopie jedes Elements gespeichert werden soll. Angenommen es sollen zwei Kopien jedes Blocks gespeichert werden. Um die

### 3 Strategien zur redundanten Datenverteilung

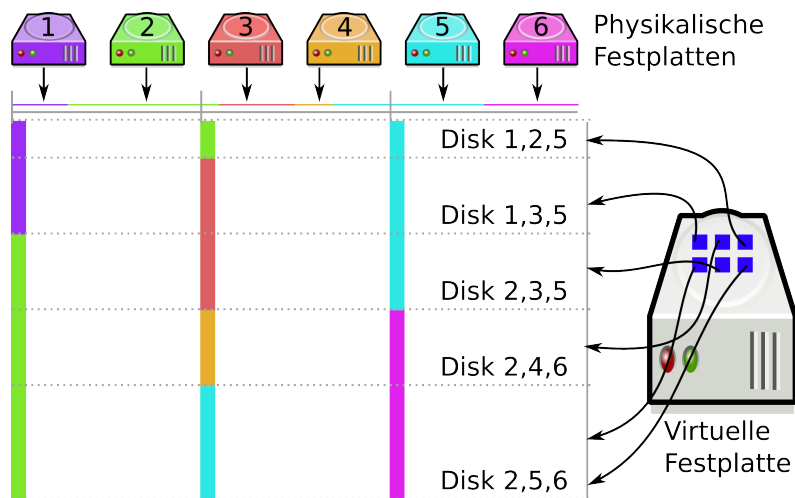


Abbildung 3.5: Platzierung von 3 Kopien in einem heterogenen System nach Schomaker [Schomaker, 2007]

erste Festplatte zu saturieren müssten hier 100 GByte abgelegt werden, Allerdings reicht der Speicherplatz der übrigen Festplatten nicht aus, um die zweiten Kopien aufzunehmen.

Folgendes Lemma gibt eine Einschränkung für die maximale Heterogenität.

**Lemma 3.3.1.** Sei  $d^{\max}$  die Festplatte mit der größten Kapazität einer gegebenen Konfiguration, also  $\forall_{i=1}^n c^{\max} \geq c_i$ . Ein fairer Algorithmus zur Datenplatzierung kann den vorhandenen Speicher nicht komplett nutzen, wenn gilt  $c^{\max} > C/k$ , also wenn der Anteil der größten Festplatte am Gesamtsystem mehr als  $1/k$  ist.

*Beweis.* Ich zeige, dass es nicht möglich ist, den gesamten Speicherplatz zu verwenden, falls  $c^{\max} > \frac{C}{k}$  ist. Soll in diesem Fall  $d^{\max}$  komplett genutzt werden, so müssten  $(k-1) \cdot c^{\max}$  Blöcke über die restlichen Festplatten verteilt werden. Dies ist aber nicht möglich, denn da  $C < k \cdot c^{\max}$  gilt  $C - c^{\max} < (k-1) \cdot c^{\max}$ . Die verbleibenden Festplatten bieten also nicht genügend Platz, um die Kopien aufzunehmen.  $\square$

Andererseits hat Schomaker in [Schomaker, 2007] einen Algorithmus vorgestellt der es schafft, den kompletten Speicherplatz zu nutzen, falls die Voraussetzung  $c^{\max} \leq \frac{C}{k}$  gilt. Er weist zunächst jeder Festplatte  $d_i$  eine Position  $p_i$  in einem  $[0, k)$  Intervall zu:

$$p_i = \begin{cases} 0 & \text{für } i = 0 \\ p_{i-1} + \frac{c_{i-1} \cdot k}{C} & \text{für } i > 0 \end{cases}$$

Bildlich gesprochen reiht er also die Festplatten im  $[0, k-1)$  Intervall hintereinander. Zur Platzierung der Daten verwendet er eine uniforme Hashfunktion  $h(\{0, M-1\}) \rightarrow [0, 1)$ , welche die

gleichen Eigenschaften wie die in Abschnitt 3.1 vorgestellte Funktion  $\text{rand}$  benötigt. Zur Platzierung des Elements  $b_j$  werden nun  $k$  Festplatten ausgewählt. Für  $l = 0 \dots k-1$  wird dazu die Festplatte  $d_l$  ausgewählt, welcher der nächst kleinste Wert  $p_l$  zu  $l + h(j)$  zugewiesen wurde. Da keine Festplatte mehr als  $\frac{C}{k}$  der Gesamtkapazität einnimmt, werden auf diese Weise immer  $k$  verschiedene Festplatten ausgewählt. Abbildung 3.5 zeigt dieses Verfahren beispielhaft für eine Konfiguration mit sechs Festplatten und  $k = 3$ .

Schomakers Hashfunktion zeigt, dass die Schranke aus 3.3.1 nicht nur eine obere, sondern auch eine untere Schranke der maximalen Heterogenität einer Konfiguration zur  $k$ -redundanten Platzierung darstellt.

Mit dieser Schranke kann ich berechnen, welchen Anteil einer beliebig heterogenen Verteilung der Kapazitäten der Festplatten ich verwenden kann. Für den Fall  $c^{\max} \leq \frac{C}{k}$  habe ich gezeigt, dass dies der gesamte Speicherplatz ist. Jetzt stellt sich also noch die Frage, welchen Teil ich für  $c^{\max} \geq \frac{C}{k}$  nutzen kann.

Um dieses Problem zu lösen sortiere ich die Festplatten zunächst absteigend gemäß ihrer Kapazität. Es ist wichtig festzustellen, dass maximal für  $d_i$  mit  $1 \leq i \leq k-1$  gelten kann, dass  $c_i > \frac{C}{k}$  gilt, also dass nur die ersten  $k-1$  Festplatten zu groß sind. Jede Festplatte, für die  $c_i > \frac{C}{k}$  gilt, muss von jedem Block genau eine Kopie erhalten, damit sie bestmöglichst saturiert werden kann.

**Lemma 3.3.2.** *Gegeben eine Konfiguration mit Festplatten absteigend sortiert nach ihrer Kapazität. Von jedem Block sollen  $k \geq 2$  Kopien auf unterschiedlichen Festplatten gespeichert werden. Die maximale Größe  $c'_i$ , welche von einer Festplatte  $d_i$  verwendet werden kann, berechnet sich wie folgt:*

$$c'_i = \begin{cases} c_i & \text{für } i > k-1 \\ \min \left( c_i, \frac{c_i + \sum_{j=i+1}^n c'_j}{k-i} \right) & \text{für } i \leq k-1 \end{cases}$$

Die maximal nutzbare Gesamtkapazität ist somit:

$$C' = \sum_{j=1}^n c'_j$$

*Beweis.* Als erstes ist zu zeigen, dass es nicht möglich ist mehr als  $C'$  des Speicherplatzes zu nutzen um  $k$  Kopien von Blöcken auf unterschiedlichen Festplatten zu speichern:

Um mehr als  $C'$  Blöcke speichern zu können müsste eine Festplatte  $d_i$  mit  $1 \leq i < k$  mehr Daten aufnehmen, da ich nur für diese Festplatten die Kapazität vermindere. Sei  $d_i$  die erste Festplatte, deren Kapazität angepasst wird, also  $c'_x = c_x$  für  $i < x \leq n$ . Da auf den Festplatten  $d_j$  mit  $j = 1 \dots i-1$  maximal  $i-1$  Kopien eines Blocks gespeichert werden können, müssen

### 3 Strategien zur redundanten Datenverteilung

---

**Algorithmus 1** optimalWeights( $k, \{c_0, \dots, c_{n-1}\}$ )

---

**Voraussetzung:**  $\forall i \in \{0, \dots, n-1\} : c_i \geq c_{i+1}$

- 1: **if**  $c_0 > \frac{1}{k-1} \sum_{i=1}^{n-1} c_i$  **then**
  - 2:     **if**  $k > 2$  **then**
  - 3:         optimalWeights ( $(k-1), \{c_1, \dots, c_{n-1}\}$ )
  - 4:     **end if**
  - 5:      $c_0 = \lfloor \frac{1}{k-1} \cdot \sum_{i=1}^{n-1} c_i \rfloor$
  - 6: **end if**
- 

mindestens  $k - i + 1$  Kopien über die Festplatten ab Position  $i$  verteilt werden. Da  $c'_i \neq c_i$  muss gelten:

$$c_i > \frac{c_i + \sum_{j=i+1}^n (c'_j)}{k - i}$$

Für jede Kopie, welche auf  $d_i$  gespeichert werden muss, müssen nun also mindestens  $k - i$  Kopien auf den Festplatten  $d_x$  mit  $i < x \leq n$  gespeichert werden. Dies ist allerdings nicht möglich, denn:

$$\begin{aligned} c_i &> \frac{c_i + \sum_{j=i+1}^n (c'_j)}{k - i + 1} \\ \Rightarrow c_i \cdot (k - i + 1) &> c_i + \sum_{j=i+1}^n (c'_j) \\ \Rightarrow c_i \cdot (k - i) &> \sum_{j=i+1}^n (c'_j) \end{aligned}$$

Somit können also nicht alle Kopien platziert werden.

Als nächstes ist zu zeigen, dass es möglich ist den gesamten Speicherplatz  $C'$  zu nutzen. Die Berechnungsvorschrift stellt sicher, dass für  $i \in [1, \dots, n]$  gilt  $c'_i \leq \frac{C'}{k}$ . Mittels des zuvor vorgestellten Algorithmus von Schomaker aus [Schomaker, 2007] ist es in diesem Fall möglich, eine Verteilung mit einer Fairness von 1 zu erreichen.  $\square$

Im weiteren Verlauf dieses Kapitels werde ich voraussetzen, dass die Kapazitäten der Festplatten der Forderung aus Lemma 3.3.1 genügen. Ist dies nicht der Fall, so können die Kapazitäten gemäß Lemma 3.3.2 mittels Algorithmus 1 angeglichen werden.

#### 3.3.2 Trivialer Ansatz

Die bisher betrachteten Verteiler verteilen Daten über einer homogenen oder heterogenen Konfiguration. Speziell in *Peer-2-Peer*-Umgebungen sind hier Hashfunktionen wie die bereits beschriebenen *Consistent Hashing* und *Share* verbreitet. Ein Ansatz zur Platzierung von redundanten Kopien könnte nun sein, solche nicht redundanten  $[n]$ -Verteiler mehrfach zu verwenden.

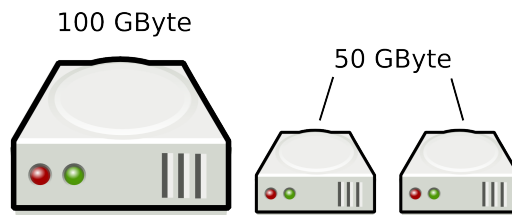


Abbildung 3.6: Beispiel einer Konfiguration, die für 2 Kopien komplett genutzt werden kann

Dabei würde z.B. mittels *Share* zunächst eine Festplatte für die erste Kopie ausgewählt um dann rekursiv über die restlichen Festplatten neue Hashfunktionen aufzubauen um die restlichen Kopien zu platzieren.

**Definition 3.3.3.** Ein trivialer Ansatz zur Platzierung mehrerer Kopien über einer Menge von Festplatten erfüllt folgende Bedingungen:

1. Die  $k$  Kopien werden platziert, indem  $k$  Platzierungen mittels nicht redundanter Verteiler vorgenommen werden.
2. Zur Platzierung der ersten Kopie stehen alle Festplatten zur Verfügung.
3. Zur Platzierung der Kopie  $i + 1$  stehen genau die Festplatten zur Verfügung, die für keine vorherige Kopie verwendet wurden.
4. Ein  $[n]$ -Verteiler entscheidet die Platzierung der Kopie  $i \in [1 \dots k]$  lediglich an Hand der Größen der Festplatten, welche zur Platzierung der Kopie zur Verfügung stehen. Speziell arbeitet er unabhängig von  $k$  und  $i$ .

Leider ist es nicht möglich mittels solch eines trivialen Ansatzes eine Verteilung von  $k \geq 2$  Kopien vorzunehmen, ohne dabei die Anforderung an die Fairness zu verletzen. Dies lässt sich an dem Beispiel aus Abbildung 3.6 erklären. Das dort abgebildete System besteht aus drei Festplatten. Die Kapazität der zweiten und der dritten Festplatte ist je die Hälfte der Kapazität der ersten Festplatte. Mittels des Verfahrens von Schomaker ist es in dieser Konfiguration möglich zwei Kopien zu platzieren und dabei den kompletten Speicher zu verwenden. Dazu muss eine Kopie auf der ersten Festplatte gespeichert werden, während die andere entweder auf der zweiten oder dritten Festplatte platziert wird.

Eine triviale Strategie würde gemäß Definition zunächst die erste Kopie gemäß der Größen der Festplatten verteilen. Dadurch würde die erste Kopie mit einer Wahrscheinlichkeit von  $\frac{1}{2}$  auf der ersten Festplatte platziert werden und mit einer Wahrscheinlichkeit von  $\frac{1}{2}$  auf einer der anderen beiden. Wird die erste Kopie nicht auf der ersten Festplatte platziert, so würde die zweite Kopie über die erste Festplatte und die verbleibende Festplatte verteilt werden, wieder gemäß ihrer Kapazitäten. In diesem Fall würde also die erste Festplatte die zweite Kopie mit einer Wahrscheinlichkeit von  $\frac{2}{3}$  erhalten, während mit einer Wahrscheinlichkeit von  $\frac{1}{3}$  die verbleibende Kopie auf der anderen Festplatte platziert würde, und die erste Festplatte keine Kopie erhalten

### 3 Strategien zur redundanten Datenverteilung

würde. Insgesamt ist die Wahrscheinlichkeit, dass die erste Festplatte keine Kopie eines Blocks erhält damit  $\frac{1}{2} \cdot \frac{1}{3} = \frac{1}{6}$ , womit ein Zwölftel der gesamten Kapazität verloren wäre.

Das folgende Lemma zeigt, dass der triviale Ansatz speziell für eine geringe Anzahl von Festplatten zu Einbußen in der Speicherkapazität führt. Der hier geführte Beweis entspricht nicht der Beweisführung aus [Brinkmann u. a., 2007]. Ich halte den hier geführten Beweis für besser verständlich.

**Lemma 3.3.4.** *Ein trivialer  $[n, k]$ -Verteiler kann nicht für beliebige Konfigurationen innerhalb der Schranke aus Lemma 3.3.1 eine faire Verteilung erreichen.*

*Beweis.* Ich definiere  $p_i^l$  für  $i \in [1, \dots, n]$  und  $l \in [1, \dots, k]$  als die Wahrscheinlichkeit, dass Kopie  $l$  eines Blocks auf  $d_i$  platziert wird.  $p_i^{\leq l}$  ist die Wahrscheinlichkeit, dass eine Kopie  $x$  mit  $1 \leq x \leq l$  auf  $d_i$  platziert wird. Äquivalent definiere ich  $L_i^l$  als die Last von  $d_i$  zur Speicherung von Kopien  $l$  und  $L_i^{\leq l}$  als die Last von  $d_i$  zur Speicherung von Kopien  $x$ .

Damit der triviale  $[n, k]$  Verteiler eine faire Verteilung erreicht, müsste für jede Konfiguration die Wahrscheinlichkeit, dass eine der  $k$  Kopien eines Datums auf einer Festplatte  $i \in \{1, \dots, n\}$  platziert wird, wie folgt beschaffen sein:

$$p_i^{\leq k} = k \cdot \frac{c_i}{C}$$

Da es sich um ein Bernoulli-Experiment handelt, ist der Erwartungswert für die zu erhaltene Last der Festplatte  $d_i$  für ein zu platzierendes Experiment wie folgt definiert:

$$E[L_i^{\leq k}] = p_i^{\leq k}$$

Die Wahrscheinlichkeit, dass eine Festplatte eine der Kopien erhält ist identisch zu der Summe der Wahrscheinlichkeiten, dass sie genau eine Kopie  $l \in \{1, \dots, k\}$  erhält:

$$p_i^{\leq k} = \sum_{l=1}^k p_i^l$$

$$E[L_i^{\leq k}] = \sum_{l=1}^k E[L_i^l]$$

Ich wähle nun eine Konfiguration, bei der  $c_1 > \frac{C}{n}$  und für alle  $i \in \{2, \dots, n\}$   $c_i < \frac{C}{n}$  gilt. Dies ist beispielsweise bei einer Konfiguration der Fall, bei der alle Festplatten außer der ersten die gleiche Kapazität besitzen und die Kapazität der ersten größer als die der restlichen Festplatten ist. Ich zeige nun, dass in diesem Fall  $d_1$  im erwarteten Fall einen zu kleinen Anteil der Last erhält. Dazu verwende ich eine vollständige Induktion über  $k$ . Ich zeige zunächst, dass  $d_i$  für

$k = 2$  im erwarteten Fall nicht genügend Daten erhält. Danach gehe ich auf den Induktionsschritt  $k - 1 \rightarrow k$  ein.

Zunächst gilt:

$$\begin{aligned} E[L_1^{\leq 2}] &= \sum_{l=1}^2 E[L_1^l] \\ &= E[L_1^1] + E[L_1^2] \\ &= p_1^1 + p_1^2 \end{aligned}$$

Die Fairness des verwendeten  $[n]$ -Verteilens zum Platzieren des ersten Elements sichert, dass die erste Kopie mit einer Wahrscheinlichkeit von  $p_i^1 = \frac{c_i}{C}$  auf der Festplatte  $d_i$  platziert wird. Vorausgesetzt die erste Kopie wurde auf  $d_i$  platziert sichert mir die Fairness des  $[n-1]$ -Verteilens zur Platzierung der zweiten Kopie weiterhin zu, dass mit einer Wahrscheinlichkeit  $\frac{c_j}{C-c_i}$  die zweite Kopie auf  $d_j$  gespeichert wird, also dass  $p_j^2 = \sum_{i=1, i \neq j}^n \frac{c_i}{C} \cdot \frac{c_j}{C-c_i}$  gilt. Daher gilt:

$$\begin{aligned} E[L_1^{\leq 2}] &= p_1^1 + p_1^2 \\ &= \frac{c_1}{C} + \sum_{j=2}^n \frac{c_j}{C} \cdot \frac{c_1}{C-c_j} \\ &= \frac{c_1}{C} + \frac{c_1}{C} \cdot \sum_{j=2}^n \frac{c_j}{C-c_j} \\ &= \frac{c_1}{C} \cdot \left(1 + \sum_{j=2}^n \frac{c_j}{C-c_j}\right) \end{aligned}$$

Ich habe definiert, dass für  $2 \leq i \leq n$  gilt  $c_i < \frac{C}{n}$ . Daher gilt:

### 3 Strategien zur redundanten Datenverteilung

$$\begin{aligned}
 E[L_1^{\leq 2}] &= \frac{c_1}{C} \cdot \left(1 + \sum_{j=2}^n \frac{c_j}{C - c_j}\right) \\
 &< \frac{c_1}{C} \cdot \left(1 + \sum_{j=2}^n \frac{\frac{C}{n}}{C - \frac{n}{C}}\right) \\
 &= \frac{c_1}{C} \cdot \left(1 + \sum_{j=2}^n \frac{\frac{C}{n}}{\frac{C}{n}(n-1)}\right) \\
 &= \frac{c_1}{C} \cdot \left(1 + \sum_{j=2}^n \frac{1}{n-1}\right) \\
 &= \frac{c_1}{C} \cdot \left(1 + (n-1) \cdot \frac{1}{n-1}\right) \\
 &= \frac{c_1}{C} \cdot 2
 \end{aligned}$$

Da also  $E[L_1^{\leq 2}] < 2 \cdot \frac{c_1}{C}$ , erhält  $d_1$  in der angegebenen Konfiguration zu wenige Kopien von Blöcken.

Um zu zeigen, dass bei der selben Konfiguration auch bei Platzierung von  $k > 2$  Kopien die erste Festplatte benachteiligt wird, definiere ich zunächst  $T_{2,n}^l$  als die Menge aller  $l$ -Tupel aus den Zahlenbereich  $[2, n]$  mit  $l$  unterschiedlichen ganzzahligen Werten. Ferner definiere ich  $\text{len}(T_{2,n}^l)$  als die Anzahl der verschiedenen existierenden unterschiedlichen Tupel. Für ein  $t \in T_{2,n}^l$  definiere ich  $\text{len}(t) = l$  als die Länge eines Tupels und  $t_1, \dots, t_l$  als die Elemente des Tupels.  $\text{len}(T_{2,n}^l)$  ist die Anzahl der Möglichkeiten aus  $n-1$  vielen Zahlen  $l$  unterschiedliche Tupel auszuwählen. Daher gilt:

$$\text{len}(T_{2,n}^l) = \frac{(n-1)!}{(n-l+1)!}$$

Ich zeige durch eine vollständige Induktion, dass für  $k \geq 2$  gilt:

$$E[L_1^{\leq k}] < k \cdot \frac{c_1}{C}$$

Die Induktionsvoraussetzung für  $k = 2$  habe ich bereits gezeigt, nun folgt also der Induktionsschritt  $k-1 \rightarrow k$ . Dazu zerlege ich zunächst wieder den Erwartungswert:



$$\begin{aligned} E[L_1^{\leq k}] &= \sum_{l=1}^k kE[L_1^l] \\ &= E[L_1^{\leq k-1}] + E[L_1^k] \end{aligned}$$

Nach Induktionsvoraussetzung gilt nun:

$$\begin{aligned} E[L_1^{\leq k}] &= E[L_1^{\leq k-1}] + E[L_1^k] \\ &< \frac{c_1}{C} \cdot (k-1) + E[L_1^k] \\ &= \frac{c_1}{C} \cdot (k-1) + p_1^k \end{aligned}$$

Damit muss ich also noch zeigen, dass  $p_1^k \leq \frac{c_1}{C}$  gilt.  $T_{2,n}^{k-1}$  beschreibt die Menge aller Möglichkeiten, um  $k-1$  Kopien über alle Festplatten  $d_j$  mit  $2 \leq j \leq n$  zu verteilen. Ich wähle nun ein beliebiges  $t \in T_{2,n}^{k-1}$  und nehme an, dass die ersten  $k-1$  Kopien auf den Festplatten  $d_{t_1}, \dots, d_{t_{k-1}}$  abgelegt wurden. Ich definiere  $p_t$  als die Wahrscheinlichkeit, dass die Festplatten  $d_{t_1}, \dots, d_{t_{k-1}}$  in dieser Reihenfolge ausgewählt wurden. Zur vereinfachten Schreibweise definiere ich dabei  $\sum_{u=1}^0 c_u = 0$ .

$$\begin{aligned} p_t &= \prod_{v=1}^{\text{len}(t)} \frac{c_{t_v}}{C - \sum_{u=1}^{v-1} c_{t_u}} \\ &= \frac{1}{C} \cdot \frac{\prod_{v=1}^{\text{len}(t)} c_{t_v}}{\prod_{v=2}^{\text{len}(t)} C - \sum_{u=1}^{v-1} c_{t_u}} \end{aligned}$$

Da  $\text{len}(t) = k-1$  für  $t \in T_{2,n}^{k-1}$  kann ich nun  $p_1^k$  wie folgt bestimmen:

$$\begin{aligned} p_1^k &= \sum_{t \in T_{2,n}^{k-1}} p_t \cdot \frac{c_1}{C - \sum_{u=1}^{k-1} c_{t_u}} \\ &= \sum_{t \in T_{2,n}^{k-1}} \frac{1}{C} \cdot \frac{\prod_{v=1}^{k-1} c_{t_v}}{\prod_{v=2}^{k-1} C - \sum_{u=1}^{v-1} c_{t_u}} \cdot \frac{c_1}{C - \sum_{u=1}^{k-1} c_{t_u}} \\ &= \frac{c_1}{C} \cdot \sum_{t \in T_{2,n}^{k-1}} \frac{\prod_{v=1}^{k-1} c_{t_v}}{\prod_{v=2}^k C - \sum_{u=1}^{v-1} c_{t_u}} \end{aligned}$$

Wie bereits für  $k=2$  nutze ich, dass für  $2 \leq j \leq n$  gilt  $c_j < \frac{C}{n}$ .

### 3 Strategien zur redundanten Datenverteilung

$$\begin{aligned}
p_1^k &= \frac{c_1}{C} \cdot \sum_{t \in T_{2,n}^{k-1}} \frac{\prod_{v=1}^{k-1} c_{t_v}}{\prod_{v=2}^k C - \sum_{u=1}^{v-1} c_{t_u}} \\
&< \frac{c_1}{C} \cdot \sum_{t \in T_{2,n}^{k-1}} \frac{\prod_{v=1}^{k-1} \frac{C}{n}}{\prod_{v=2}^k C - \sum_{u=1}^{v-1} \frac{C}{n}} \\
&= \frac{c_1}{C} \cdot \frac{\frac{C^{k-1}}{n^{k-1}}}{\frac{C^{k-1}}{n^{k-1}} \prod_{v=2}^k n - v - 1} \cdot \text{len } T_{2,n}^{k-1} \\
&= \frac{c_1}{C} \frac{1}{\prod_{v=1}^{k-1} n - v} \text{len } T_{2,n}^{k-1} \\
&= \frac{c_1}{C} \cdot \frac{(n-k)!}{(n-1)!} \cdot \text{len } T_{2,n}^{k-1} \\
&= \frac{c_1}{C}
\end{aligned}$$

Da  $p_1^k < \frac{c_1}{C}$  und nach Induktionsvoraussetzung  $p_1^{\leq k-1} < (k-1) \cdot \frac{c_1}{C}$  gilt auch  $p_1^{\leq k} < k \cdot \frac{c_1}{C}$ . Somit ist  $E[L_1^{\leq k}] < k \cdot \frac{c_1}{C}$ , die Festplatte  $d_1$  erhält also zu wenige Kopien von Blöcken.  $\square$

Aus diesem Beweis ist ersichtlich, dass es wichtig ist darauf zu achten, dass größere Festplatten ihren Anteil an der vorhandenen Last erhalten.

## 3.4 Redundant Share: Ein Algorithmus zur redundanten Datenverteilung

Die Analyse trivialer Strategien hat gezeigt, dass es bei der redundanten Speicherung von Blöcken besonders wichtig ist, großen Festplatten ihren Anteil an der Last zu geben. Dieses Problem löst das im Folgenden vorgestellte Verfahren *Redundant Share* [Brinkmann u. a., 2007; Brinkmann und Effert, 2008b], indem es große Festplatten bei der Platzierung von Elementen priorisiert.

Ich stelle das Verfahren in mehreren Stufen vor, um einen einfacheren Einstieg zu ermöglichen. In Abschnitt 3.4.1 betrachte ich die Platzierung von nur einer Kopie. In Abschnitt 3.4.2 erweitere ich das Verfahren zur Platzierung von zwei Kopien, um die prinzipiellen Probleme beim Aufbau der Redundanz zeigen zu können. In Abschnitt 3.4.3 erweitere ich das Verfahren, so dass es möglich ist  $k$  Kopien zu speichern. Alle Verfahren bis dahin haben eine lineare Laufzeit

---

**Algorithmus 2** LinPlace ( $\text{addr}, \{d_1, \dots, d_n\}, \{c_1, \dots, c_n\}$ )

---

**Voraussetzung:**  $\forall i \in \{1, \dots, n-1\} : c_i \geq c_{i+1}$

1:  $\forall i \in \{1, \dots, n\} : \check{c}_i = c_i / \sum_{j=i}^n c_j$

2:  $i \leftarrow 1$

3: **while**  $i \leq n$  **do**

4:    $\text{val} \leftarrow \text{rand}(\text{addr}, d_i) \in [0, 1)$

5:   **if**  $\text{val} < \check{c}_i$  **then**

6:     Platziere Element auf  $d_i$

7:     **return**

8:   **end if**

9:    $i \leftarrow i + 1$

10: **end while**

---

gemäß der Anzahl der Festplatten. Im nächsten Schritt stelle ich einen aufbauenden Verteiler vor, welcher in linearer Laufzeit gemäß der Anzahl der zu platzierenden Kopien arbeitet.

Das vorgestellte Verfahren liefert nicht nur eine faire Verteilung über die Festplatten, sondern kann diese sogar in einer dynamischen Umgebung aufrecht erhalten. Ich zeige, wie viele Umplatzierungen erwartet werden können, wenn eine Festplatte hinzugenommen oder entfernt wird. Wie in Abschnitt 3.1 beschrieben unterscheide ich zwischen der Adaptivität ohne und mit Berücksichtigung der Reihenfolge der Kopien.

Ich nehme schon einmal vorweg, dass *Redundant Share* eine gemäß der Kapazitäten absteigend sortierte Sequenz von Festplatten benötigt. Auf diese Art kann *Redundant Share* sicher stellen, dass auch Festplatten mit hoher Kapazität einen fairen Anteil an der zu verteilenden Last bekommen. Daher gehe ich in diesem Abschnitt und im folgenden Abschnitt 3.5 davon aus, dass die Festplatten gemäß ihrer Kapazität absteigend sortiert sind.






### 3.4.1 Platzierung ohne Redundanz

*LinPlace* ist ein Verfahren, welches Blöcke über einer heterogenen Konfiguration verteilen kann. Diese Strategie arbeitet noch nicht redundant, also nur für den Fall  $k = 1$ .

Der Algorithmus 2 zeigt das Verfahren *LinPlace* als Pseudocode. *LinPlace* benötigt als Eingabe neben der Adresse des zu platzierenden Blocks die Kapazitäten der Festplatten und ihre Identifizierer. Die Festplatten müssen absteigend gemäß ihrer Kapazität sortiert sein. Aus diesen Kapazitäten errechnet der Algorithmus zunächst den Anteil jeder Festplatte  $d_i$  mit  $i \in \{1, \dots, n\}$  am Speicherplatz der Festplatten, die ab  $d_i$  folgen. Diesen Wert bezeichne ich als  $\check{c}_i$ . Für die Festplatte  $d_i$  berechnet sich  $\check{c}_i$  wie folgt:

### 3 Strategien zur redundanten Datenverteilung

Tabelle 3.1: Beispiel zur Platzierung eines Elements über fünf heterogenen Festplatten

					
$c_i$	100	100	80	80	60
$C_i = \sum_{j=i}^n c_j$	420	320	220	140	60
$\check{c}_i = \frac{c_i}{C_i}$	0.24	0.31	0.36	0.57	1.0

$$\check{c}_i = \frac{c_i}{\sum_{j=i}^n c_j}$$

Tabelle 3.1 verdeutlicht die Berechnung von  $\check{c}_i$  an einem Beispiel mit 5 Festplatten. Die erste Zeile gibt die Kapazitäten der einzelnen Festplatten wieder. In der zweiten Zeile wird für jede Festplatte  $d_i$  die Summe der Kapazitäten aller Festplatten  $d_j$  mit  $i \leq j \leq n - 1$  berechnet. In der dritten Zeile wird die Kapazität der Platte durch die jeweilige Summe geteilt. Dieser Wert bestimmt nun die Wahrscheinlichkeit, dass der Block auf der Festplatte  $d_i$  zu platzieren ist, unter der Voraussetzung, dass er nicht auf einer Festplatte  $d_l$  mit  $l > i$  platziert wurde.

Das Verfahren geht in einer Schleife beginnend mit der ersten Festplatte alle Festplatten durch. Für jede Festplatte erzeugt es ein Experiment, welches den Block mit der Wahrscheinlichkeit  $\check{c}_i$  auf  $d_i$  platziert. Dazu wird die in Abschnitt 3.1 beschriebene pseudorandomisierte Abbildung `rand` verwendet. Diese Funktion wird hier benutzt, um für jede Kombination aus einer Blockadresse und dem Identifizierer einer Festplatte einen pseudorandomisierten Wert zu liefern. Ist dieser Wert kleiner als  $\check{c}_i$ , so ist das Experiment gelungen und der Block wird auf  $d_i$  platziert. Andernfalls bezeichne ich das Experiment als misslungen und die nächste Festplatte wird überprüft.

Im Folgenden untersuche ich dieses Verfahren auf seine Eigenschaften. Dazu ist es zunächst sehr wichtig zu beachten, dass es sich bei *LinPlace* um eine Hashfunktion handelt. Daher liefert *LinMirror* eine stabile Verteilung, das heißt bei gleicher Eingabe liefert *LinPlace* immer die gleiche Festplatte. Dies gelingt nur, weil auch die Funktion `rand` den Eigenschaften einer Hashfunktion folgt. Zwar liefert sie für jede Kombination aus Blockadresse und Identifizierer einer Festplatte einen unabhängigen, pseudorandomisierten Wert, aber für eine feste Blockadresse und eine feste Festplatte immer den gleichen. Ich verweise im Folgenden auf diesen Sachverhalt als Stabilität von *LinPlace*.

**Lemma 3.4.1.** *LinPlace hat im schlimmsten Fall eine Laufzeit und einen Hauptspeicherbedarf von  $O(n)$ .*

### 3.4 Redundant Share: Ein Algorithmus zur redundanten Datenverteilung

*Beweis.* Zunächst kann  $\check{c}_i$  für jede Festplatte im Voraus berechnet werden. Dazu benötigt das Verfahren  $O(n)$  Speicherplatz, denn für jede Festplatte muss dieser Wert vorrätig gehalten werden. Da laut Abschnitt 3.1 in meinem Modell beliebige reelle Zahlen in konstantem Speicher abgelegt werden können, hat *LinPlace* einen Speicherverbrauch von  $O(n)$ .

Die Laufzeit des Verfahrens wird von der Schleife bestimmt, welche Experimente für die einzelnen Festplatte durchführt bis das erste Experiment gelungen ist. Die Pseudorandomisierung kann laut Definition in konstanter Zeit berechnet werden. Die Schleife betrachtet jede Festplatte maximal einmal. Somit liegt die Laufzeit im schlimmsten Fall bei  $O(n)$ .  $\square$

**Lemma 3.4.2.** *LinPlace ist 1-fair.*

*Beweis.* Die Fairness von *LinPlace* lässt sich mittels einer vollständigen Induktion beweisen. Dazu zeige ich zunächst, dass die erste Festplatte ihren Anteil der zu platzierenden Blöcke erhält.

Laut Berechnungsvorschrift von  $\check{c}_i$  gilt  $\check{c}_1 = \frac{c_1}{C_1} = \frac{c_1}{C}$ . Mit dieser Wahrscheinlichkeit wird ein Block auf der Festplatte  $d_1$  platziert, wenn er auf keiner vorherigen Festplatte platziert wurde. Da es keine vorherigen Festplatten gibt gilt:

$$\begin{aligned} E[L_1] &= m \cdot \check{c}_1 \\ &= m \cdot \frac{c_1}{C} \end{aligned}$$

Damit erhält die erste Festplatte im erwarteten Fall ihren fairen Anteil.

Der Schritt  $i \rightarrow i + 1$  der Induktion baut nun zunächst darauf auf, dass die Festplatte  $d_i$  ihren fairen Anteil erhalten hat. Die Festplatte  $d_{i+1}$  wird nur untersucht, falls das Element nicht vorher platziert wurde. Somit ist auf der Festplatte  $d_{i+1}$  ihr fairer Anteil gegenüber der restlichen Festplatten zu speichern. Unter der Voraussetzung, dass die ersten  $i$  Festplatten ihren fairen Anteil an der Gesamtlast erhalten haben, muss  $d_{i+1}$  also folgenden Anteil der Restlast  $L_{i+1}^i$  erwarten:

$$\begin{aligned} E[L_{i+1}^i] &= m \cdot \frac{c_{i+1}}{C - \sum_{j=1}^i c_j} \\ &= m \cdot \frac{c_{i+1}}{\sum_{j=i+1}^n c_j} \\ &= m \cdot \check{c}_{i+1} \end{aligned}$$

$\square$

### 3 Strategien zur redundanten Datenverteilung

Die Adaptivität von *LinPlace* hängt von der Art der Festplatten und der Operation ab. Daher zeige ich die Adaptivität in mehreren Schritten. Zunächst beginne ich mit homogenen Konfigurationen und zeige die Kosten für das Hinzufügen und Entfernen einer Festplatte. Danach gehe ich auf heterogene Konfigurationen ein.

**Lemma 3.4.3.** *Sei  $d^{\text{neu}}$  eine neue hinzuzufügende Festplatte. Für die Kapazität von  $d^{\text{neu}}$  gelte  $c^{\text{neu}} \geq c_1$ , es gibt also keine Festplatte mit einer größeren Kapazität in der Konfiguration. *LinPlace* ist dann 1-adaptiv für das Einfügen von  $d^{\text{neu}}$ .*

*Beweis.* In Lemma 3.4.2 habe ich gezeigt, dass *LinPlace* im erwarteten Fall jeder Festplatte ihren fairen Anteil an der Gesamtlast zuweist. Damit muss ich nun noch zeigen, dass keine weiteren Replatzierungen vorgenommen werden.

Ich füge  $d^{\text{neu}}$  als erste Festplatte  $d_0$  in das System ein. Damit gilt  $\check{c}_0 = \frac{c_0}{c+c_0}$ , wobei  $c^{\text{neu}}$  die Kapazität der neuen Festplatte ist. Da  $\check{c}_i$  für  $1 \leq i \leq n$  nur von den Festplatten ab der Position  $i$  abhängt bleibt  $\check{c}_i$  für diese Festplatten unverändert. Damit wird, falls das Experiment gegen  $\check{c}_0$  gelingt, ein Block auf der neuen Festplatte platziert, sonst wird seine Position nicht verändert.  $\square$

Damit ist das Hinzufügen einer neuen homogenen Festplatte 1-adaptiv, da hier gilt  $c^{\text{neu}} = c_1$ , womit die Voraussetzung aus Lemma 3.4.3 hält. Analog lässt sich zeigen, dass das Entfernen der ersten Festplatte 1-adaptiv ist.

**Lemma 3.4.4.** *Das Entfernen der Festplatte  $d_1$  in *LinPlace* ist 1-adaptiv.*

*Beweis.* Lemma 3.4.3 besagt, dass *LinPlace* 1-adaptiv für das Hinzufügen einer Festplatte an erster Position ist. Gegeben die Konfiguration der Festplatten  $d_i$  für  $i \in [2, \dots, n]$  kann  $d_1$  ohne zusätzliche Replatzierungen hinzugefügt werden. Da *LinPlace* eine stabile Verteilung erzeugt, würde das anschließende Entfernen von  $d_1$  wieder zu exakt der gleichen Verteilung führen, wie vor dem Hinzufügen von  $d_1$ . Somit werden auch beim Entfernen von  $d_1$  nur die zuvor auf  $d_1$  platzierten Blöcke bewegt und *LinPlace* ist für das Entfernen von  $d_1$  1-adaptiv.  $\square$

Für homogene Konfigurationen lässt sich damit eine Grenze für das Entfernen einer beliebigen Festplatte bestimmen:

**Lemma 3.4.5.** *Gegeben eine Menge von  $n$  homogenen Festplatten mit der Kapazität  $c$ . Das Entfernen einer Festplatte mit *LinPlace* ist 3-adaptiv.*

*Beweis.* Ist die zu entfernende Festplatte an erster Position, so ist das Entfernen nach Lemma 3.4.4 1-adaptiv.

Ist dies nicht der Fall, so werden zunächst die Elemente auf der zu entfernenden Festplatte mit den Elementen auf  $d_1$  vertauscht. Die Fairness sichert zu, dass dazu im erwarteten Fall  $2 \cdot \frac{m}{n}$

### 3.4 Redundant Share: Ein Algorithmus zur redundanten Datenverteilung

Blöcke bewegt werden. Danach tauschen die beiden Festplatten ihre Identität und die erste Festplatte wird entfernt. Dies verursacht laut Lemma 3.4.4 nochmals Kosten von  $\frac{m}{n}$ . Somit sind die Gesamtkosten  $3 \cdot \frac{m}{n}$  und das Verfahren ist 3-adaptiv.  $\square$

Bisher habe ich nur Aussagen zu homogenen Systemen bzw. zu sehr speziellen Ereignissen bei heterogenen System gemacht. Nun betrachte ich, wie sich *LinPlace* für den heterogenen Fall beim Hinzufügen und Entfernen beliebiger Festplatten verhält.

**Lemma 3.4.6.** *LinPlace ist  $\ln n$ -adaptiv für das Hinzufügen einer beliebigen Festplatte  $d^{neu}$  und das Entfernen einer beliebigen Festplatte  $d_i$  für  $i \in [1, \dots, n]$ .*

*Beweis.* Zunächst betrachte ich das Hinzufügen einer Festplatte:

$n$  sei die Anzahl der Festplatten vor dem Hinzufügen.  $d^{neu}$  soll gemäß ihrer Größe  $c^{neu}$  an der Position  $q \in [1 \dots n]$  einsortiert werden. Wird die Festplatte an Position 1 eingefügt, so ergibt sich laut Lemma 3.4.3 eine Adaptivität von 1. Im Folgenden setze ich daher  $q > 1$  voraus. Weitergehend besagt Lemma 3.4.3, dass Elemente der Festplatten ab Position  $q$  nur verschoben werden, falls diese auf  $d^{neu}$  platziert werden, da  $\check{c}_i$  für  $q < i \leq n$  unverändert bleibt.

Nun ist zu zeigen, wie viele Elemente maximal im Bereich  $j \in [1 \dots q - 1]$  verschoben werden. Ich definiere  $C_q = \sum_{i=q}^n c_i$ . Die relative Kapazität der Festplatte  $d_j$  in der alten Konfiguration war damit  $\check{c}_j^{alt} = \frac{c_j}{\sum_{l=j}^{q-1} c_l + C_q}$ . Nach dem Hinzufügen von  $d^{neu}$  erhalte ich  $\check{c}_j^{neu} = \frac{c_j}{\sum_{l=j}^{q-1} c_l + c^{neu} + C_q}$ . Die Wahrscheinlichkeit  $p_j$ , dass ein Block von der Festplatte  $d_j$  weg bewegt wird, ist nun proportional zum Unterschied zwischen  $\check{c}_j^{alt}$  und  $\check{c}_j^{neu}$ .

$$\begin{aligned}
 p_j &= \frac{\check{c}_j^{alt} - \check{c}_j^{neu}}{\check{c}_j^{alt}} \\
 &= 1 - \frac{\check{c}_j^{neu}}{\check{c}_j^{alt}} \\
 &= 1 - \frac{\sum_{l=j}^{q-1} c_l + C_q}{\sum_{l=j}^{q-1} c_l + c^{neu} + C_q} \\
 &= \frac{c^{neu}}{\sum_{l=j}^{q-1} c_l + c^{neu} + C_q} \\
 &\leq \frac{c^{neu}}{\sum_{l=j}^{q-1} c_l + c^{neu}} \\
 &= \frac{1}{1 + \frac{1}{c^{neu}} \cdot \sum_{l=j}^{q-1} c_l}
 \end{aligned}$$

### 3 Strategien zur redundanten Datenverteilung

Der schlechteste Fall tritt also ein, falls  $c^{\text{neu}}$  groß ist und die Festplatte weit hinten einsortiert wird. Mit  $L_j$  als Last von  $d_j$  ergibt sich die folgende erwartete Anzahl der Bewegungen:

$$\begin{aligned} E[P] &= \sum_{j=1}^{q-1} p_j \cdot E[L_j] \\ &\leq \sum_{j=1}^{q-1} \frac{E[L_j]}{1 + \frac{1}{c^{\text{neu}}} \cdot \sum_{l=j}^{q-1} c_l} \end{aligned}$$

Dieser Wert erreicht sein Maximum falls alle Festplatten  $1 \leq j \leq q$  die gleiche Kapazität haben, also  $c_j = c^{\text{neu}}$ . Die Fairness von *Redundant Share* besagt, dass für die erwartete Last von  $d_j$  gilt:

$$E[L_j] = m \cdot \frac{c_j}{C}$$

Somit lässt sich die erwartete Anzahl an Schritten begrenzen durch:

$$\begin{aligned} E[P] &= \sum_{j=1}^{q-1} \frac{E[L^{\text{neu}}]}{1 + \frac{1}{c^{\text{neu}}} \cdot \sum_{l=j}^{q-1} c^{\text{neu}}} \\ &= E[L^{\text{neu}}] \cdot \sum_{j=1}^{q-1} \frac{1}{1 + \frac{1}{c^{\text{neu}}} \cdot (q-j) \cdot c^{\text{neu}}} \\ &= E[L^{\text{neu}}] \cdot \sum_{j=1}^{q-1} \frac{1}{1 + q-j} \\ &= E[L^{\text{neu}}] \cdot \sum_{j=2}^{q+1} \frac{1}{j} \end{aligned}$$

Dies kann ich mittels der harmonischen Reihe abschätzen. Dieser Reihe fehlt der Startwert 1. Daher gilt:

$$P \leq E[L^{\text{neu}}] \cdot ((\ln q) - 1 + \gamma)$$

Dabei ist  $\gamma \approx 0,5772$  die Euler-Mascheroni Konstante. Da  $\gamma < 1$  gilt:

$$P \leq E[L^{\text{neu}}] \cdot \ln q$$



Daher ist das Verhältnis der Elemente die umplatziert werden, zu den Elementen auf der neuen Festplatte  $\ln n$ . Da *LinPlace* eine Fairness von 1 besitzt, gilt die  $\ln n$ -Adaptivität für das Hinzufügen einer beliebigen Festplatte.

Die Adaptivität beim Entfernen einer Festplatte ergibt sich direkt aus der Stabilität von *LinPlace*. Wenn das Einfügen einer beliebigen Festplatte  $\ln n$ -adaptiv ist, so muss das Entfernen der selben Festplatte auch  $\ln n$ -adaptiv sein. Da *LinPlace* die Verteilung der Daten nur gemäß der aktuellen Konfiguration vornimmt (speziell ohne Beachtung früherer Konfigurationen) hat *LinPlace* auch für das Entfernen einer beliebigen Festplatte eine Adaptivität von  $\ln n$ .  $\square$

#### 3.4.2 2-redundante Platzierung

Mit *LinPlace* habe ich ein Verfahren vorgestellt, um Daten über verschiedene Festplatten zu verteilen. Dieses Verfahren erweitere ich nun zur redundanten Platzierung von Daten. Dazu beginne ich in diesem Abschnitt mit dem Fall  $k = 2$ , ich platziere also zwei Kopien jedes Blocks.

Um diesen und den folgenden Abschnitte einfacher verständlich zu machen, nummeriere ich die Kopien entgegen der Reihenfolge, in der sie gefunden werden. Das bedeutet, dass zuerst die Kopie  $k$  platziert wird. Danach wird die Kopie  $k - 1$  platziert, bis zum Schluss die Kopie eins platziert wird. Da in diesem Abschnitt  $k = 2$  ist, wird also zuerst die Kopie zwei, dann die Kopie eins platziert.

Das Verfahren *LinMirror* löst das Problem zwei Kopien jedes Blocks auf unterschiedlichen physikalischen Festplatten zu speichern. Algorithmus 3 zeigt das Verfahren als Pseudocode. *LinMirror* geht ähnlich zu *LinPlace* vor. Die Festplatten sind im Vorhinein wieder absteigend gemäß ihrer Kapazität sortiert. Auch die relative Anzahl gegenüber den restlichen Festplatten  $\check{c}_i$  wird wie zuvor berechnet.

Das Verfahren beginnt damit die Kopie zwei zu platzieren. Dazu wird mittels der Pseudorandomisierung ein Experiment für jede Festplatte durchgeführt, welches mit einer Wahrscheinlichkeit von  $2 \cdot \check{c}_i$  gelingt. Dieses Experiment berechnet wie bei *LinPlace* für einen zu platzierenden Block zu jeder Festplatte einen Zufallswert. Ist dieser Wert größer als  $\check{c}_i$ , so gilt das Experiment als gelungen. Andernfalls ist es misslungen. Auf der ersten Festplatte, für die das Experiment gelungen ist, wird die Kopie zwei platziert. Die Kopie eins wird mittels eines Verfahrens ohne Redundanz über die anderen Festplatten verteilt. Hierzu verwende ich das zuvor vorgestellte *LinPlace*.

Es ist wichtig zu bemerken, dass die Funktion *rand* in *LinPlace* und *LinMirror* die gleiche Abbildung erzeugt. Ich verwende in beiden Fällen die gleiche Hashfunktion und die Eingabe hängt nur von der Blockadresse und dem Identifizierer der Festplatte ab. Diese Eigenschaft wird insbesondere bei der Betrachtung der Adaptivität sehr wichtig.

Tabelle 3.2 erweitert die Tabelle 3.1 um die Berechnung von  $2 \cdot \check{c}_i$ . Dies ist die Wahrscheinlichkeit, dass eine Kopie zwei auf  $d_i$  platziert werden soll, sofern sie auf keiner vorherigen

### 3 Strategien zur redundanten Datenverteilung

---

**Algorithmus 3** LinMirror (addr,  $\{d_1, \dots, d_n\}$ ,  $\{c_1, \dots, c_n\}$ )

---

**Voraussetzung:**  $\forall i \in \{1, \dots, n-1\} : c_i \geq c_{i+1}$

**Voraussetzung:**  $\forall i \in \{1, \dots, n\} : 2 \cdot c_i \leq C$






```

1:  $\forall i \in \{1, \dots, n\} : \check{c}_i = c_i / \sum_{j=i}^n c_j$ 
2:  $i \leftarrow 1$ 
3: while  $i < n$  do
4:    $\text{val} \leftarrow \text{rand}(\text{addr}, d_i) \in [0, 1)$ 
5:   if  $\text{val} < 2 \cdot \check{c}_i$  then
6:     Platziere erste Kopie auf Festplatte  $d_i$ 
7:      $c^* \leftarrow c_{i+1}$ 
8:     if  $2 \cdot \check{c}_i < 1$  and  $2 \cdot \check{c}_{i+1} > 1$  then
9:        $c^* \leftarrow \text{adjust}(i, c_1, \dots, c_n)$ 
10:    end if
11:    Platziere zweite Kopie:
12:    LinPlace(addr,  $\{d_{i+1}, \dots, d_n\}$ ,  $\{c^*, c_{i+2}, \dots, c_n\}$ )
13:    return
14:  end if
15:   $i \leftarrow i + 1$ 
16: end while

```

---

Tabelle 3.2: Beispiel zur Platzierung eines Elements mit zwei Kopien über fünf heterogene Festplatten

					
$c_i$	100	100	80	80	60
$C_i = \sum_{j=i}^n c_j$	420	320	220	140	60
$\check{c}_i = \frac{c_i}{C_i}$	0.24	0.31	0.36	0.57	1.0
$2 \cdot \check{c}_i$	0.48	0.61	0.72	1.14	2.0

Festplatte abgelegt wurde. Die restlichen Zeilen der Tabelle bleiben unverändert gegenüber der nicht redundanten Verteilung.

Die letzte Zeile zeigt also, mit welcher Wahrscheinlichkeit eine Kopie zwei auf der jeweiligen Festplatte platziert wird, falls sie auf keiner vorherigen platziert wurde. Die erste Festplatte erhält 48 % aller Kopien zwei. Die zweite Festplatte erhält 61 % der Kopien zwei, welche nicht auf der ersten Festplatte gespeichert werden. Die dritte Festplatte erhält 72 % der nicht vorher

### 3.4 Redundant Share: Ein Algorithmus zur redundanten Datenverteilung

abgelegten Kopien zwei und die restlichen Kopien zwei werden auf der vierten Festplatte platziert. Da die Festplatten gemäß ihrer Kapazität absteigend sortiert sind, wird auf jeden Fall eine der ersten  $n - 1$  Festplatten zur Platzierung der Kopie zwei verwendet und mittels *LinPlace* kann aus den restlichen Festplatten eine Festplatte zur Platzierung der Kopie eins ausgewählt werden. Damit liefert der Algorithmus in jedem Fall ein gültiges Ergebnis.

Betrachtet man die Werte aus dem Beispiel zur Platzierung der Kopie zwei genauer, so fällt auf, dass die vierte Festplatte 114 % der Kopien zwei aufnehmen soll, die sie erreichen. Dies liegt daran, dass an dieser Stelle die rekursive Voraussetzung  $c_i \leq \frac{1}{k} \cdot \sum_{j=i}^{n-1} c_j$  nicht erfüllt ist. Daher erwartet diese Festplatte zu viele Kopien zwei und bekommt dadurch zu wenig Kopien eins.

Dieses Problem kann bei der 2-redundanten Platzierung nur bei der ersten Festplatte  $d_i$  auftreten, für die  $2 \cdot \check{c}_i \geq 1$  gilt. Für alle vorherigen Festplatten hält die Voraussetzung. Dies sichert die Sortierung der Festplatten zu. Ist  $2 \cdot \check{c}_i = 1$ , so bekommt die Festplatte genau die richtige Anzahl an Kopien. Andernfalls ist es nötig eine Justierungskapazität  $c^*$  zu berechnen. Mittels dieser Justierungskapazität weise ich  $d_i$  mehr Kopien eins zu, falls Kopie zwei auf  $d_{i-1}$  platziert wurde. In Algorithmus 3 geschieht dies in der inneren if-Verzweigung.

Ich berechne den Anteil an Kopien eins, welche  $d_i$  für Blöcke erhält, deren Kopien zwei auf den Festplatten  $d_j$  mit  $j \in [1, \dots, i-2]$  platziert wurden. Zunächst definiere ich  $P_j$  als die Wahrscheinlichkeit, dass die Kopie zwei  $d_j$  erreicht:

$$P_j = \begin{cases} \prod_{v=1}^{j-1} (1 - 2 \cdot \check{c}_v) & \text{für } 1 < j \leq i \\ 1 & \text{für } j = 1 \end{cases}$$

Damit erhält  $d_j$  einen Anteil von  $P_j \cdot \check{c}_j$  der Kopien zwei für  $j \in [1, \dots, i-1]$ . Weiterhin ist  $C_j = \sum_{v=j}^n c_v$  die Summe der Kapazität aller Festplatten ab der Position  $j$ .  $\frac{c_i}{C_{j+1}}$  mit  $j > i$  ist der Anteil der Kopien eins, die  $d_i$  für Kopien zwei auf der Festplatte  $d_j$  erhält. Im erwarteten Fall erhält  $d_i$  daher folgende Last an Kopien eins, für Kopien zwei auf den Festplatten  $j \in [0 \dots i-2]$ :

$$s_i^{i-2} = \sum_{j=1}^{i-2} \left( P_j \cdot 2 \cdot \check{c}_j \cdot \frac{c_i}{C_{j+1}} \right)$$

Da  $2 \cdot \check{c}_i > 1$  gelingt das Experiment zur Platzierung von Kopie zwei gegen  $d_i$  immer. Daher erhält  $d_i$  im erwarteten Fall einen Anteil  $P_i$  der Kopien zwei. Damit kann ich berechnen, welcher Anteil von  $d_i$  mit Kopien eins belegt werden muss für Kopien zwei auf  $d_{i-1}$ :

$$s_i = 2 \cdot \frac{c_i}{C} - s_i^{i-2} - P_i$$

Das Ziel ist es nun  $c^*$  so zu berechnen, dass die Festplatte genügend Kopien eins aufnimmt.  $2 \cdot \check{c}_{i-1} \cdot P_{i-1}$  ist der Anteil an Kopien zwei, welche auf  $d_{i-1}$  platziert werden.  $c^*$  muss so gewählt werden, dass  $\frac{c^*}{c^* + C_{i+1}}$  der zu platzierenden Kopien eins  $s_i$  ergibt, also:

### 3 Strategien zur redundanten Datenverteilung

$$s_i = \frac{c^*}{c^* + C_{i+1}} \cdot 2 \cdot \check{c}_{i-1} \cdot P_{i-1}$$

Diese Formel muss ich nun nur noch nach  $c^*$  auflösen:

$$\begin{aligned}
 & s_i = \frac{c^*}{c^* + C_{i+1}} \cdot 2 \cdot \check{c}_{i-1} \cdot P_{i-1} \\
 \Leftrightarrow & \frac{s_i}{2 \cdot \check{c}_{i-1} \cdot P_{i-1}} = \frac{c^*}{c^* + C_{i+1}} \\
 \Leftrightarrow & \frac{s_i}{2 \cdot \check{c}_{i-1} \cdot P_{i-1}} \cdot (c^* + C_{i+1}) = c^* \\
 \Leftrightarrow & \frac{s_i}{2 \cdot \check{c}_{i-1} \cdot P_{i-1}} \cdot c^* + \frac{s_i}{2 \cdot \check{c}_{i-1} \cdot P_{i-1}} \cdot C_{i+1} = c^* \\
 \Leftrightarrow & \frac{s_i}{2 \cdot \check{c}_{i-1} \cdot P_{i-1}} \cdot C_{i+1} = c^* - \frac{s_i}{2 \cdot \check{c}_{i-1} \cdot P_{i-1}} \cdot c^* \\
 \Leftrightarrow & \frac{s_i}{2 \cdot \check{c}_{i-1} \cdot P_{i-1}} \cdot C_{i+1} = c^* \cdot \left(1 - \frac{s_i}{2 \cdot \check{c}_{i-1} \cdot P_{i-1}}\right) \\
 \Leftrightarrow & \frac{\frac{s_i}{2 \cdot \check{c}_{i-1} \cdot P_{i-1}} \cdot C_{i+1}}{\left(1 - \frac{s_i}{2 \cdot \check{c}_{i-1} \cdot P_{i-1}}\right)} = c^*
 \end{aligned}$$

**Lemma 3.4.7.** *Es ist möglich  $c^*$  in Rechenzeit  $O(n^2)$  mit Speicherbedarf  $O(n)$  zu berechnen.*

*Beweis.* Die Funktion  $P_j$  kann in linearer Zeit durch eine Iteration über die ersten  $j$  Festplatten berechnet werden. Jeder Iterationsschritt braucht konstante Zeit. Damit hat  $P_j$  eine Laufzeit von  $O(j) = O(n)$ . Da  $P_j$  die Kapazitäten der Festplatten benötigt, liegt der Speicherbedarf bei  $O(n)$ .

Die Funktion  $s_i^{i-2}$  iteriert einmal über die ersten  $i - 2$  Festplatten. Während jedes Iterationsschritts wird ein  $P_j$  berechnet, die restlichen Berechnungen brauchen konstante Zeit. Damit hat  $s_i^{i-2}$  eine Laufzeit von  $O(n^2)$  und einen Speicherbedarf von  $O(n)$ .

Die Funktion  $s_i$  selbst kann in konstanter Laufzeit implementiert werden, nachdem  $s_i^{i-2}$  und  $P_i$  berechnet wurden. Damit hat  $s_i$  eine Laufzeit von  $O(n^2)$  und einen Speicherbedarf von  $O(n)$ .

$c^*$  kann in konstanter Laufzeit berechnet werden, nachdem  $s_i$  und  $P_{i-1}$  berechnet wurden. Damit kann  $c^*$  in Laufzeit  $O(n^2)$  bei einem Speicherbedarf von  $O(n)$  bestimmt werden.  $\square$

Es ist möglich die Rechenzeit zur Berechnung von  $c^*$  zu verbessern, indem mehr Hauptspeicher eingesetzt wird. Da  $P_{v+1} = P_v \cdot (1 - 2 \cdot \check{c}_{v+1})$  gilt, ist es möglich diese Werte in linearer Zeit und mit linearem Speicheraufwand vorzuhalten. Danach kann auf  $P_j$  für  $j \in [1, \dots, n]$  in konstanter Zeit zugegriffen werden. Dadurch wird die Laufzeit auf  $O(n)$  verbessert.

Nun untersuche ich *LinMirror* auf seine Eigenschaften, wie bereits zuvor *LinPlace*. Wie *LinPlace* liefert auch *LinMirror* eine stabile Verteilung, bei gleicher Konfiguration und gleicher

### 3.4 Redundant Share: Ein Algorithmus zur redundanten Datenverteilung

Blockadresse werden also immer die gleichen Festplatten ausgewählt. Diese Stabilität der Verteilung werde ich wieder bei der Betrachtung der Adaptivität nutzen.

Ich beginne mit der Betrachtung der Laufzeit und des Speicherverbrauchs.

**Lemma 3.4.8.** *Das Verfahren LinMirror hat im schlimmsten Fall Laufzeit und Speicherbedarf  $O(n)$ .*

*Beweis.*  $c^*$  lässt sich im Voraus berechnen und kann mittels  $O(k)$  Speicher vorgehalten werden. Der Beweis der Zeiteffizienz ist dann äquivalent zu Lemma 3.4.1. Als einziger Unterschied wird hier nach Auffinden der Kopie zwei ein Algorithmus zur Platzierung der Kopie eins aufgerufen. Dazu verwende ich *LinPlace*. *LinPlace* kann die für *LinMirror* berechneten  $\check{c}_i$  verwenden, und benötigt damit keinen zusätzlichen Speicher. Somit liegen Laufzeit und Speicherbedarf in  $O(n)$ .  $\square$

*LinMirror* ist 1-fair. Voraussetzung ist lediglich, dass die Heterogenität der Festplatten innerhalb der Schranke aus Lemma 3.3.1 bleibt, also dass die größte Festplatte nicht mehr als die Hälfte des Gesamtspeichers ausmacht.

**Lemma 3.4.9.** *Der Algorithmus LinMirror hat eine Fairness von 1 falls  $\frac{c_1}{C} \leq \frac{1}{2}$  gilt.*

*Beweis.* Ähnlich zu dem Beweis von 3.4.2 nutze ich auch hier die rekursive Natur des Verfahrens.

Die erwartete Last einer Festplatte  $d_i$  mit  $i \in [0 \dots n-1]$  setzt sich aus den Kopien zwei und eins zusammen, welche auf der Festplatte platziert werden. Eine Kopie eins kann auf einer Festplatte  $d_j$  mit  $j \in \{i+1, \dots, n\}$  nur platziert werden, falls die zweite Kopie auf einer vorherigen Festplatte  $i$  mit  $1 \leq i < j \leq n$  platziert wurde.

Ich beginne den Beweis, indem ich die Festplatte  $d_1$  betrachte. Diese Festplatte kann nur Kopien zwei erhalten, welche sie mit dem Experiment gegen  $2 \cdot \check{c}_1$  erhält. Somit erhält  $d_1$  im erwarteten Fall seine faire Last:

$$\begin{aligned} E[L_1] &= m \cdot 2 \cdot \frac{c_1}{C_1} \\ &= m \cdot 2 \cdot \frac{c_1}{C} \end{aligned}$$

An dieser Stelle ist es wichtig, dass die Kopien eins mittels *LinPlace* fair über die verbleibenden Festplatten  $\{2 \dots n\}$  verteilt werden. Dadurch kann ich das Problem in der nächsten Runde auf eine faire Verteilung von zwei Kopien über die Festplatten  $\{1, \dots, n-1\}$  reduzieren, welches auf die gleiche Weise gelöst wird, wie bereits die Verteilung für die erste Festplatte. So die

### 3 Strategien zur redundanten Datenverteilung

Festplatte  $d_i$  mit  $i \in \{1, \dots, n-1\}$  ihren fairen Anteil an der Last erhalten hat, erhält auch  $d_{i+1}$  im erwarteten Fall eine faire Last an Kopien zwei  $L_{i+1}^{i,2}$ :

$$\begin{aligned} E[L_{i+1}^{i,2}] &= m \cdot 2 \cdot \frac{c_{i+1}}{C - \sum_{j=1}^i c_j} \\ &= m \cdot 2 \cdot \frac{c_{i+1}}{\sum_{j=i+1}^n c_j} \\ &= m \cdot 2 \cdot \check{c}_{i+1} \end{aligned}$$

Dieser rekursive Ansatz endet, wenn  $2 \cdot \check{c}_i > 1$  wird. In diesem Fall ist es nicht möglich, die Festplatte mit einem Anteil von  $2 \cdot \check{c}_i$  an Kopien zwei zu versehen, da nicht genügend Kopien zwei diese Festplatte erreichen. Dies wird ausgeglichen durch die Anpassung der Anzahl der zugewiesenen Kopien eins, wenn die Kopie zwei auf der Festplatte  $d_{i-1}$  platziert wird. □

Als nächstes gehe ich auf die Adaptivität von *LinMirror* ein. Wie bereits bei *LinPlace* betrachte ich auch bei *LinMirror* zunächst den homogenen Fall. Zur Erinnerung: Für das Hinzufügen einer Festplatte bedeutet das, dass sie immer an erster Position eingefügt werden kann. Um dies näher zu untersuchen, zeige ich zunächst was mit den Kopien eines Blocks passiert, wenn eine neue Festplatte hinzugenommen wird.

**Lemma 3.4.10.** *Sei  $d^{\text{neu}}$  eine neu einzufügende Festplatte mit  $c^{\text{neu}} \geq c_1$ . Sei  $d_i$  eine Festplatte deren Kopie zwei auf  $d^{\text{neu}}$  verschoben wird, mit  $1 \leq i < n$ . Sei  $d_j$  die Festplatte mit der zugehörigen Kopie eins mit  $i < j \leq n$ . Nach Hinzufügen von  $d^{\text{neu}}$  wird die Kopie eins auf  $d_l$  oder  $d_i$  platziert.*

*Beweis.* Der Problemstellung folgend betrachte ich Blöcke, deren Kopie zwei auf  $d^{\text{neu}}$  platziert werden muss.  $d_i$  speicherte zuvor die Kopie zwei,  $d_l$  die Kopie eins. Ich zeige nun, dass die Kopie eins entweder auf  $d_l$  verbleibt oder auf  $d_i$  verschoben wird.

Zunächst kann ich ausschließen, dass die Kopie eins auf eine Festplatte  $d_j$  mit  $l < j \leq n$  verschoben wird. Da  $\check{c}_l$  unverändert bleibt wird spätestens diese Festplatte zur Platzierung ausgewählt.

Weiterhin kann ich mit dem gleichen Argument ausschließen, dass die Kopie eins auf einer Festplatte  $d_j$  mit  $i < j < l$  platziert wird. Auch die Gewichtung  $\check{c}_j$  jeder solchen Festplatte ist gleich geblieben, und die Kopie hätte schon zuvor dort platziert werden müssen.

Nun bleibt noch zu zeigen, dass die Kopie eins auf keiner Festplatte  $d_j$  mit  $0 < j < i$  platziert werden kann. Gehen wir dazu zunächst davon aus, eine Kopie eins sollte auf  $d_j$  gespeichert werden. Dies würde für den berechneten Wert der Pseudorandomisierung  $\text{rand}$  aus dem Algorithmus bedeuten, dass  $\text{rand} < \check{c}_j$  sein müsste. Da aber bereits die Kopie zwei nicht auf dieser

### 3.4 Redundant Share: Ein Algorithmus zur redundanten Datenverteilung

Festplatte abgelegt wurde, gilt  $rand > 2 \cdot \check{c}_j$ . Daher kann auch auf keiner solchen Festplatte  $d_j$  die Kopie eins platziert werden.  $\square$

Mit dieser Erkenntnis kann ich abschätzen, wie sich *LinMirror* beim Hinzufügen einer neuen Festplatte an erster Position verhält.

**Lemma 3.4.11.** *Sei  $d^{\text{neu}}$  eine neu einzufügende Festplatte mit  $c^{\text{neu}} \geq c_1$ . *LinMirror* ist 1,5-adaptiv mit Beachtung der Reihenfolge der Kopien für das Einfügen von  $d^{\text{neu}}$ .*

*Beweis.* Die neue Festplatte wird an erster Position eingefügt. Daher kann sie nur Kopien zwei erhalten. Die in Lemma 3.4.9 gezeigte Fairness sichert zu, dass im erwarteten Fall  $2 \cdot \frac{c^{\text{neu}}}{C+c^{\text{neu}}}$  der Daten auf  $d^{\text{neu}}$  verschoben werden.

Wie bereits in Lemma 3.4.3 beschrieben, bleiben die Kopien zwei, welche nicht auf der neuen Festplatte platziert werden, unberührt. Auch die Kopien eins dieser nicht verschobenen zweiten Kopien werden nicht verschoben. Damit muss ich nur Blöcke betrachten, deren Kopien zwei auf  $d^{\text{neu}}$  zu platzieren sind.

Lemma 3.4.10 hat bereits gezeigt, dass solche Kopien eins entweder an ihrer bisherigen Position  $l$  verbleiben oder auf die Festplatte  $i$  verschoben werden, auf welcher bisher die Kopie zwei abgelegt war. Ferner gilt  $rand < 2 \cdot \check{c}_i$ , da vorher die Kopie zwei hier platziert wurde. Die Kopie eins wird nun auf diese Platte verschoben, falls  $rand < \check{c}_i$  gilt, also mit einer Wahrscheinlichkeit von  $\frac{1}{2}$ .  $\square$

Falls die beiden Kopien wirklich identisch sind, so ist es, wie in Abschnitt 3.1 beschrieben, egal welche Kopie auf welcher der ausgewählten Festplatten platziert wurde. In diesem Fall kann die Reihenfolge der Kopien ausgetauscht werden, um das Verschieben von Blöcken zu verhindern.

**Lemma 3.4.12.** *Sei  $d^{\text{neu}}$  eine neu einzufügende Festplatte mit  $c^{\text{neu}} \geq c_1$ . *LinMirror* ist 1-adaptiv ohne Beachtung der Reihenfolge der Kopien für das Einfügen von  $d^{\text{neu}}$ .*

*Beweis.* Wie bereits im Lemma 3.4.11 gezeigt können die auf  $(d_i, d_l)$  gespeicherten Kopien nach einer Verschiebung nur auf  $(d^{\text{neu}}, d_i)$  oder  $(d^{\text{neu}}, d_l)$  abgelegt werden. Verschiebt man jeweils die Kopie der herausfallenden Festplatte auf  $d^{\text{neu}}$ , so ist keine weitere Umplatzierung notwendig.  $\square$

Nun betrachte ich das Entfernen einer homogenen Festplatte. Auch hier beginne ich wieder mit dem Entfernen der Festplatte an erster Position.

**Lemma 3.4.13.** **LinMirror* ist 1-adaptiv für das Entfernen der Festplatte  $d_1$  ohne Beachtung der Reihenfolge der Kopien und 1,5-adaptiv mit Beachtung der Reihenfolge.*

### 3 Strategien zur redundanten Datenverteilung

*Beweis.* Wie schon im Beweis zu Lemma 3.4.4 für *LinPlace* nutze ich die Stabilität von *LinMirror*. Diese sichert zu, dass das Hinzufügen und Entfernen der ersten Festplatte die gleiche Adaptivität hat. Somit ist das Entfernen von  $d_1$  ohne Beachtung der Reihenfolge 1-adaptiv, weil unter gleicher Voraussetzung das Hinzufügen von  $d_1$  zu der Konfiguration bestehend aus  $d_i$  mit  $i \in [2, \dots, n]$  1-adaptiv ist. Das Hinzufügen von  $d_1$  zu der Konfiguration bestehend aus  $d_i$  mit  $i \in [2, \dots, n]$  ist 1,5-adaptiv mit Berücksichtigung der Reihenfolge der Kopien, somit ist unter gleicher Voraussetzung auch das Entfernen von  $d_1$  1,5-adaptiv.  $\square$

**Lemma 3.4.14.** *LinMirror ist 3-adaptiv für das Entfernen einer homogenen Festplatte ohne Beachtung der Reihenfolge der Kopien. Mit Beachtung der Reihenfolge ist LinMirror 3,5-adaptiv.*

*Beweis.* Um dies zu gewährleisten vertausche ich zunächst wieder den Inhalt und die Identität der zu entfernenden Festplatte mit der ersten Festplatte. Danach wird die erste Festplatte aus dem System entfernt.  $\square$

Nun komme ich zum heterogenen Fall. Lemma 3.4.11 hat bereits eine Schranke für das Einfügen einer Festplatte an erster Position gezeigt. Nun zeige ich, wie sich *LinMirror* verhält, wenn eine beliebige, heterogene Festplatte hinzugefügt wird.

**Lemma 3.4.15.** *LinMirror ist  $\ln n$ -adaptiv beim Einfügen und Entfernen einer Festplatte ohne Beachtung der Reihenfolge der Kopien.*

*Beweis.* Der Beweis folgt unmittelbar aus dem Beweis von Lemma 3.4.6. Ich verwende dort lediglich die Veränderung der relativen Kapazitäten  $\check{c}_i$  für  $i \in \{1, \dots, n\}$  und die erwartete Last der Festplatten. Laut Lemma 3.4.9 besitzt *LinMirror* eine Fairness von 1, daher ist die erwartete Last jeder Festplatte  $d_i$  mit  $i \in \{1, \dots, n\}$ :

$$E[L_i] = 2 \cdot m \cdot \frac{c_i}{C}$$

Die erwartete Anzahl an Replatzierungen ist unabhängig der wirklichen Last und setzt lediglich die Fairness voraus. Daher ist *LinMirror*  $\ln n$ -adaptiv ohne Beachtung der Reihenfolge der Kopien.  $\square$

**Lemma 3.4.16.** *LinMirror ist  $1,5 \cdot \ln n$ -adaptiv beim Einfügen und Entfernen einer Festplatte mit Beachtung der Reihenfolge der Kopien.*

*Beweis.* Wie in Lemma 3.4.15 gezeigt, müssen ohne Beachtung der Reihenfolge  $\ln n$  der Daten gegenüber einer optimalen Adaptivität replaziert werden. Weiterhin habe ich in dem Beweis zu Lemma 3.4.11 gezeigt, dass mit Beachtung der Reihenfolge der Kopien mit einer Wahrscheinlichkeit von  $\frac{1}{2}$  die Kopie eines Blocks verschoben werden muss, falls seine Kopie zwei verschoben wurde.



---

**Algorithmus 4** RedundantShare ( $k$ , addr,  $\{d_1, \dots, d_n\}$ ,  $\{c_1, \dots, c_n\}$ )

---

**Voraussetzung:**  $\forall i \in \{1, \dots, n-1\} : c_i \geq c_{i+1}$

**Voraussetzung:**  $\forall i \in \{1, \dots, n\} : k \cdot c_i \leq C$

```

1:  $\forall i \in \{1, \dots, n\} : \check{c}_i = k \cdot c_i / \sum_{j=i}^n c_j$ 
2:  $i \leftarrow 1$ 
3: while  $i < n$  do
4:    $\text{val} \leftarrow \text{rand}(\text{addr}, d_i) \in [0, 1)$ 
5:   if  $\text{val} < k \cdot \check{c}_i$  then
6:     Platziere Kopie  $k$  auf Festplatte  $d_i$ 
7:      $c^* \leftarrow c_{i+1}$ 
8:     if  $k \cdot \check{c}_i < 1$  and  $k \cdot \check{c}_{i+1} > 1$  then
9:        $c^* \leftarrow \text{adjust}(k, i, c_0, \dots, c_{n-1})$ 
10:    end if
11:    if  $k \geq 2$  then
12:      RedundantShare( $k-1$ , addr,  $\{d_{i+1}, \dots, d_n\}$ ,  $\{c^*, c_{i+2}, \dots, c_{n-1}\}$ )
13:    end if
14:    return
15:  end if
16:   $i \leftarrow i + 1$ 
17: end while

```

---

Ich kann im allgemeinen Fall keine Aussage darüber treffen, wie viele der replazierten Daten Kopien zwei und wie viele Kopien eins sind. Im schlimmsten Fall sind alle replazierten Daten Kopien zwei. Somit liegt die Adaptivität mit Beachtung der Reihenfolge der Kopien bei  $1,5 \cdot \ln n$ . □

### 3.4.3 $k$ -redundante Platzierung

Nachdem ich mittels *LinMirror* das grundsätzliche Vorgehen zur redundanten Platzierung von Daten erläutert habe, erweitere ich nun das Verfahren, um eine beliebige, feste Menge von Kopien ablegen zu können. Algorithmus 4 zeigt *Redundant Share* als Pseudocode.

*Redundant Share* arbeitet sehr ähnlich zu *LinMirror*, allerdings wird diesmal die Zufallsvariable mit  $k \cdot \check{c}_i$  für die einzelnen Festplatten verglichen. Wurde eine Kopie  $k$  gefunden, arbeitet der Algorithmus rekursiv für  $k-1$  Kopien auf den restlichen Festplatten weiter. Für  $k=1$  verhält sich *Redundant Share* wie das in Abschnitt 3.4.1 vorgestellte Verfahren *LinPlace*. Für  $k=2$  verhält *Redundant Share* sich wie *LinMirror* aus Abschnitt 3.4.2.

Wie bereits bei den Erläuterungen für die 2-redundante Verteilung benenne ich die Kopien nicht nach der Reihenfolge, in der Sie gefunden werden, sondern nach dem  $k$ , mit dem das Verfahren

### 3 Strategien zur redundanten Datenverteilung

in der Rekursion aufgerufen wurde. Daher platziert der Algorithmus also zuerst die Kopie  $k$ , im nächsten Durchlauf die Kopie  $(k - 1)$  bis er zum Schluss die Kopie eins platziert.

Wie bereits bei *LinMirror* gezeigt, gibt es einen Fehler bei der jeweils letzten Festplatte, welche für eine Kopie  $l \in \{k, \dots, 2\}$  gewählt werden kann. Die Berechnung erfolgt analog zu der Berechnung für zwei Kopien, allerdings muss ich nun mit einfließen lassen, um die wievielte Kopie es sich handelt und wie viele Kopien es gibt.

Ich definiere für  $l \in \{k, \dots, 2\}$   $e_l$  als den Index der letzten Festplatte, welche zur Platzierung der Kopie  $l$  in Frage kommt. Dies ist die erste Festplatte, für die gilt  $l \cdot \check{c}_{e_l} \geq 1$ , für alle Festplatten  $d_i$  mit  $i \in \{1, \dots, e_l - 1\}$  muss also  $l \cdot \check{c}_i < 1$  gelten. Da im Falle von  $l \cdot \check{c}_{e_l} = 1$  nichts unternommen werden muss, gehe ich davon aus, dass  $l \cdot \check{c}_{e_l} > 1$  gilt.

Analog zu der Bestimmung von  $c^*$  für zwei Kopien berechne ich nun  $c_{e_l}^*$  für die ausgewählten Festplatten. In Algorithmus 4 findet sich diese Berechnung in der ersten inneren if-Verzweigung in Zeile 9.

Zunächst ist zu berechnen, welcher Anteil von  $d_{e_l}$  für welche Kopien verwendet wird. Zunächst definiere ich  $P_{a,b,l}$ . Angenommen eine Kopie  $l$  wird über die Festplatten ab Position  $a$  verteilt, so gibt  $P_{a,b,l}$  die Wahrscheinlichkeit an, dass diese Kopie auf keiner Festplatte  $d_x$  mit  $a \leq x < b$  platziert wird:

$$P_{a,b,l} = \begin{cases} \max(0, \prod_{v=a}^{b-1} (1 - l \cdot \check{c}_v)) & \text{für } a < b \\ 1 & \text{für } a = b \end{cases}$$

Die Funktion *max* verhindert, dass für Festplatten, die nicht zur Platzierung der Kopie  $l$  gewählt werden können, ein negatives Ergebnis zurück gegeben wird. Im nächsten Schritt berechne ich, wie viele Kopien  $k$  die Festplatte  $d_{e_l}$  erhält:

$$\begin{aligned} \text{Anteil}_k(e_l) &= \min(1, k \cdot \check{c}_{e_l}) \prod_{j=0}^{e_l-1} (1 - k \cdot \check{c}_j) \\ &= \min(1, k \cdot \check{c}_{e_l}) \cdot P_{0,e_l,k} \end{aligned}$$

Um im nächsten Schritt zu berechnen, welcher Anteil von  $d_{e_l}$  mit Kopien  $(k - 1)$  belegt wird, berechne ich für jede Festplatte  $d_j$  mit  $j \in \{0, \dots, e_l - 1\}$  wie viele Kopien  $k$  auf  $d_j$  platziert werden und wie viele Kopien  $(k - 1)$  dafür auf  $d_{e_l}$  platziert werden:

$$\text{Anteil}_{k-1,j}(e_l) = P_{0,j,k} \cdot \min(1, k \cdot \check{c}_j) \cdot P_{j+1,e_l,k-1} \cdot \min(1, (k - 1) \cdot \check{c}_{e_l})$$

### 3.4 Redundant Share: Ein Algorithmus zur redundanten Datenverteilung

Summiert man diese Anteile für alle  $d_j$  auf, so ergibt sich der Anteil, den  $d_{e_l}$  für Kopien  $(k-1)$  verbraucht:

$$\text{Anteil}_{k-1}(e_l) = \sum_{j=0}^{e_l-1} P_{0,j,k} \cdot \min(1, k \cdot \check{c}_j) \cdot P_{j+1,e_l,k-1} \cdot \min(1, (k-1) \cdot \check{c}_{e_l})$$

Analog kann ich auch den Anteil der Kopien  $(k-2)$  auf  $d_{e_l}$  berechnen, indem ich für jede  $d_j$  mit  $j \in \{1, \dots, e_l-2\}$  berechne, wie viele Kopien  $k$  dort platziert werden. Damit kann ich für jede Festplatte  $d_v$  mit  $v \in \{2, \dots, i-1\}$  berechnen, wie viele Kopien  $(k-1)$  dort platziert werden um schließlich zu bestimmen welchen Anteil die Kopien  $(k-2)$  von  $d_{e_l}$  belegen:

$$\begin{aligned} \text{Anteil}_{k-2}(e_l) = & \sum_{j=0}^{e_l-2} (P_{0,j,k} \cdot \min(1, k \cdot \check{c}_j) \cdot \\ & \sum_{v=j+1}^{e_l-1} (P_{j+1,v,k-1} \cdot \min(1, (k-1) \cdot \check{c}_v) \cdot \\ & P_{v+1,e_l,k-2} \cdot \min(1, (k-2) \cdot \check{c}_{e_l}))) \end{aligned}$$

Die rekursive Natur dieser Formel ausnutzend definiere ich  $r_{t,m,j,i}$ . Mit  $i = e_l$  berechnet diese Formel den Anteil an Kopien  $t$  auf  $d_{e_l}$  für Kopien  $m$  ab  $d_j$ .

$$r_{t,m,j,i} = \begin{cases} P_{j,i,m} \cdot \min(1, m \cdot \check{c}_i) & \text{für } m = t \\ \sum_{v=j}^{i-m+t} P_{j,v,m} \cdot \min(1, m \cdot \check{c}_v) \cdot r_{t,m-1,v+1,i} & \text{für } t < m \end{cases}$$

Mittels dieser Formel lässt sich der Anteil an beliebigen Kopien  $t \in \{1, \dots, k\}$  bestimmen:

$$\text{Anteil}_t(e_l) = r_{t,k,0,e_l}$$

Bei der Platzierung von zwei Kopien bekam  $d_i$  einen erhöhten Anteil an Kopien eins für Kopien zwei auf  $d_{i-1}$ . Entsprechend muss  $d_{e_l}$  bei  $k$  Kopien einen erhöhten Anteil an Kopien  $(l-1)$  erhalten, wenn die Kopie  $l$  auf  $d_{e_l-1}$  platziert wurde. Daher passe ich die Anteilsberechnung für  $t = l-1$  an, sodass in diesen Fall nur die Kopien  $(l-1)$  bis  $j = e_l-2$  berücksichtigt werden. Ich definiere  $r'_{t,m,j,i}$  so, dass mit  $i = e_l$   $d_{e_l-1}$  nicht mit in die Betrachtung einfließt, um damit  $\text{Anteil}_{l-1}(e_l)$  zu berechnen:

### 3 Strategien zur redundanten Datenverteilung

$$r'_{t,m,j,i} = \begin{cases} P_{j,i,m} \cdot \min(1, m \cdot \check{c}_i) & \text{für } m = t \\ \sum_{v=j}^{i-m+t-1} P_{j,v,m} \cdot \min(1, m \cdot \check{c}_v) \cdot r'_{t,m-1,v+1,i} & \text{für } t < m \end{cases}$$

Dies lasse ich nun in die Berechnung der Anteile einfließen, so dass ich folgende Funktion erhalte:

$$\text{Anteil}_t(e_l) = \begin{cases} r_{t,k,0,e_l} & \text{für } t \neq l-1 \\ r'_{t,k,0,e_l} & \text{für } k = l-1 \end{cases}$$

Somit kann nun  $k_{e_l}$  als der Anteil an  $d_{e_l}$  berechnet werden, der für die Kopien  $(l-1)$  verwendet werden muss:

$$k_{e_l} = k \cdot \frac{c_i}{C_0} - \left( \sum_{t=1}^k \text{Anteil}_t(e_l) \right)$$

Um zu erreichen, dass  $d_{e_l}$  genügend Kopien aufnimmt, ist die Größe  $c_{e_l}^*$  äquivalent zu  $k = 2$  zu berechnen. Dazu muss ich allerdings noch bestimmen, wie viele Kopien  $l$  auf  $d_{e_{l-1}}$  platziert werden. Das erledigt die Funktion  $r_{l,k,0,e_{l-1}}$ .

$$k_{e_l} = (l-1) \cdot \frac{c_{e_l}^*}{c_{e_l}^* + C_{e_{l+1}}} \cdot r_{l,k,0,e_{l-1}}$$

Diese Formel kann ich nun wieder nach  $c_{e_l}^*$  auflösen:

$$\begin{aligned} k_{e_l} &= (l-1) \cdot \frac{c_{e_l}^*}{c_{e_l}^* + C_{e_{l+1}}} \cdot r_{l,k,0,e_{l-1}} \\ \Leftrightarrow c_{e_l}^* &= \frac{\frac{k_{e_l}}{(l-1) \cdot r_{l,k,0,e_{l-1}}} \cdot C_{e_{l+1}}}{1 - \frac{k_{e_l}}{(l-1) \cdot r_{l,k,0,e_{l-1}}}} \end{aligned}$$

**Lemma 3.4.17.** *Es ist möglich, alle  $c_{e_l}^*$  in Rechenzeit  $O(k^2 \cdot n^k)$  und Speicherbedarf von  $O(k+n)$  zu berechnen.*

### 3.4 Redundant Share: Ein Algorithmus zur redundanten Datenverteilung

*Beweis.* Die Funktion  $P_{a,b,l}$  kann in linearer Zeit durch eine Iteration über die Festplatten  $j \in \{a, \dots, b-1\}$  berechnet werden. Jeder Iterationsschritt braucht konstante Zeit. Damit hat  $P_{a,b,l}$  eine Laufzeit von  $O(b-a) = O(n)$  und einen Speicherbedarf von  $O(n+k)$ , da neben des Speichers für diese Funktion die Identifizierer und Kapazitäten der Festplatten benötigt werden.

Die Funktion  $r_{t,m,j,e_l}$  besitzt einen rekursiven Aufruf der Tiefe  $m-t$ . Die Funktion  $T_x$  soll nun die Laufzeit von  $r_{t,m,j,e_l}$  mit  $x = m-t$  beschreiben. Zunächst gilt  $T_0 = O(n)$ , da für  $m=t$  neben  $P_{j,e_l,m}$  nur Berechnungen mit konstanter Laufzeit erfolgen. In jedem Rekursionsschritt wird einmal über einen Bereich der Festplatten iteriert. In jedem Schritt erfolgen neben der Berechnung von  $P_{j,v,m}$  und dem Rekursionsschritt nur Berechnungen mit konstanter Laufzeit. Damit hat jeder Schritt eine Laufzeit von  $O(n + T_{x-1})$  und folglich jeder Rekursionsschritt die Laufzeit  $T_x = O(n^2 + n \cdot T_{x-1}) = O(n^{x+1})$  für  $x > 0$ . Da  $x$  eine Variable in Abhängigkeit zu  $k$  ist ( $0 \leq x \leq k-1$ ) benötigt die Berechnung von  $r_{t,m,j,e_l}$  eine Laufzeit von  $O(n^k)$ . Jeder Rekursionsschritt hat einen konstanten Speicherbedarf, womit  $r_{t,m,j,e_l}$  einen Speicherbedarf von  $O(n+k)$  hat.

Die Funktion  $r'_{t,m,j,e_l}$  hat die gleiche Laufzeit und den gleichen Speicherbedarf wie  $r_{t,m,j,e_l}$ , da hier lediglich in jedem Schritt eine Festplatte weniger betrachtet wird.

Die Funktion  $\text{Anteil}_o(e_l)$  delegiert an  $r$  oder  $r'$  und hat damit auch eine Laufzeit von  $O(n^k)$  und einen Speicherbedarf von  $O(n+k)$ .

Die Funktion  $k_{e_l}$  besitzt  $k$  Aufrufe von  $\text{Anteil}_l$  und hat damit eine Laufzeit von  $O(k \cdot n^k)$  und einen Speicherbedarf von  $O(n+k)$ .

Um  $c_{e_l}^*$  für eine Festplatte zu berechnen wird  $k_{e_l}$  und ein Aufruf von  $r$  benötigt. Damit kann ein  $c_{e_l}^*$  in Laufzeit  $O(k \cdot n^k + n^k) = O(k \cdot n^k)$  berechnet werden, mit einem Speicherbedarf von  $O(n+k)$ .

Da es in einer Konfiguration maximal  $k-1$  Positionen geben kann, für welche ein  $c_{e_l}^*$  berechnet werden muss, können alle  $c_{e_l}^*$  in Laufzeit  $O(k^2 \cdot n^k)$  mit  $O(n+k)$  Speicherbedarf berechnet werden.  $\square$

Kommen wir nun zu den Eigenschaften von *Redundant Share*. Zunächst möchte ich wieder auf die Stabilität der Verteilung mittels *Redundant Share* hinweisen. Zu einer festen Konfiguration liefert *Redundant Share* für einen Block  $b_j$  mit  $j \in \{1, \dots, m\}$  immer die gleichen Festplatten.

Ich betrachte nun die Laufzeit und den Hauptspeicherverbrauch.

**Lemma 3.4.18.** *Redundant Share kann in Laufzeit und Speicherbedarf  $O(n)$  implementiert werden.*

*Beweis.* Der Beweis der Zeiteffizienz verläuft äquivalent zu Lemma 3.4.8. Die verschiedenen  $c_{e_l}^*$  können im Voraus berechnet werden und nehmen  $O(k) \leq O(n)$  Speicher ein. Damit ist der Algorithmus auch vom Speicherverbrauch in  $O(n)$ .  $\square$

### 3 Strategien zur redundanten Datenverteilung

Der nächste Punkt behandelt die Fairness von *Redundant Share*.

**Lemma 3.4.19.** *Redundant Share ist 1-fair falls  $c_1 \leq \frac{C}{k}$  gilt.*

*Beweis.* Der Beweis ergibt sich direkt aus dem Beweis von Lemma 3.4.9. Ich verwende eine vollständige Induktion über  $k$ . Den Induktionsanfang für  $k = 1$  habe ich bereits in Lemma 3.4.2 gezeigt. Der Induktionsschritt  $k - 1 \rightarrow k$  ist nun eine Verallgemeinerung des Beweises des Lemmas 3.4.9.

Zunächst erhält  $d_1$  nur Kopien  $k$ . Eine Kopie  $k$  wird mit einer Wahrscheinlichkeit von  $k \cdot \check{c}_1$  auf  $d_1$  platziert. Damit erhält  $d_1$  im erwarteten Fall einen fairen Anteil der Last:

$$\begin{aligned} E[L_1] &= m \cdot k \cdot \frac{c_1}{C_1} \\ &= m \cdot k \cdot \frac{c_1}{C} \end{aligned}$$

Laut Induktionsvoraussetzung werden die Kopien  $l \in \{k - 1, \dots, 1\}$  fair über die restlichen Festplatten verteilt. So kann ich das Problem in der nächsten Runde auf eine faire Verteilung von  $k$  Kopien über die Festplatten  $\{d_2, \dots, d_n\}$  reduzieren. So die Festplatte  $d_i$  mit  $i \in \{1, \dots, n - 1\}$  ihren fairen Anteil der Last erhalten hat, erhält auch  $d_{i+1}$  im erwarteten Fall eine faire Last  $L_{i+1}^{i,k}$  an Kopien  $k$ :

$$\begin{aligned} E[L_{i+1}^{i,2}] &= m \cdot k \cdot \frac{c_{i+1}}{C - \sum_{j=1}^i c_j} \\ &= m \cdot k \cdot \frac{c_{i+1}}{\sum_{j=i+1}^n c_j} \\ &= m \cdot k \cdot \check{c}_{i+1} \end{aligned}$$

Dieser rekursive Ansatz endet, wenn  $k \cdot \check{c}_i > 1$  wird. Dann ist es nicht möglich, die Festplatte mit einem Anteil von  $k \cdot \check{c}_i$  an Kopien  $k$  zu versehen, da nicht genügend Kopien  $k$  die Festplatte  $d_i$  erreichen. Dies wird ausgeglichen durch die Anpassung der Anzahl der zugewiesenen Kopien  $k - 1$ , wenn die Kopie  $k$  auf der Festplatte  $d_{i-1}$  platziert wird.  $\square$

Die Adaptivität untersuche ich wieder zuerst im homogenen Fall. Im homogenen Fall kann eine neue Festplatte immer an erster Position eingefügt werden. Deshalb betrachte ich, wie sich *Redundant Share* für das Einfügen einer neuen Festplatte an erster Position verhält.

**Lemma 3.4.20.** *Sei  $d^{\text{neu}}$  eine neu einzufügende Festplatte mit  $c^{\text{neu}} \geq c_1$ . Sei  $b_z$  mit  $z \in \{1, \dots, m\}$  ein Block dessen Kopie  $w + 1$  mit  $w \in \{k - 1, \dots, 1\}$  von einer Festplatte  $d_i$  mit  $i \in \{1, \dots, n - 1\}$  weg bewegt wird. Sei  $d_j$  mit  $j \in \{1, \dots, i - 1\}$  oder  $d^{\text{neu}}$  die Festplatte, auf die die Kopie bewegt wird. Sei  $d_l$  mit  $l \in \{i + 1, \dots, n\}$  die Festplatte mit der Kopie  $w$  von  $b_z$ . Nach Hinzufügen von  $d^{\text{neu}}$  wird die Kopie  $w$  auf  $d_l$  oder  $d_j$  platziert.*

### 3.4 Redundant Share: Ein Algorithmus zur redundanten Datenverteilung

*Beweis.* Ich kann die neue Festplatte  $d^{\text{neu}}$  wieder als Festplatte  $d_0$  verwenden, da es keine Festplatte mit größerer Kapazität in der Konfiguration gibt. Somit bleibt  $\check{c}_i$  für  $i \in \{1, \dots, n\}$  unverändert.

In Lemma 3.4.10 habe ich bereits den Beweis für  $w = 1$  gezeigt. Ich erweitere diesen Beweis nun für ein allgemeines  $w \in \{k-1, \dots, 1\}$ . Hierzu verwende ich eine vollständige Induktion. Ich zeige zunächst, dass die Behauptung für  $w = k-1$  gilt. Danach zeige ich als Induktionsschritt, dass die Behauptung für  $w \in \{k-2, \dots, 1\}$  gilt, falls sie für  $w+1$  gilt.

Sei  $w = k-1$ . In diesem Fall kann die Kopie  $k$  nur auf  $d^{\text{neu}}$  verschoben worden sein, da die Kopie auch zuvor auf keiner Festplatte  $d_t$  für  $t \in \{1, \dots, i-1\}$  platziert wurde. Die Kopie  $k-1$  kann auf keine Festplatte  $d_t$  mit  $t \in \{1, \dots, i\}$  verschoben worden sein, da für diese Festplatten gilt  $\text{rand}(b_z, d_t) > k \cdot \check{c}_t > (k-1) \cdot \check{c}_t$ . Die Festplatten  $d_t$  mit  $t \in \{i+1, \dots, l-1\}$  wurden bereits mittels der ursprünglichen Konfiguration nicht ausgewählt. Spätestens für  $d_i$  muss das Ergebnis der Pseudozufallsfunktion kleiner als  $(k-1) \cdot \check{c}_i$  sein, da mit der ursprünglichen Konfiguration die Kopie  $k-1$  hier platziert wurde. Somit kann die Kopie  $k-1$  nur auf  $d_l$  oder  $d_i$  platziert werden.

Ich betrachte nun den Iterationsschritt  $w+1 \rightarrow w$  für  $w \in \{k-2, \dots, 1\}$ . Durch den Iterationsschritt weiß ich, dass die Kopie  $w+1$  zuvor von  $d_j$  weg bewegt worden sein muss. Sei  $d_q$  die Festplatte, auf welche die Kopie  $w+1$  verschoben wurde. Zur Platzierung der Kopie  $w$  wurden somit die Festplatten  $d_p$  mit  $p \in \{q+1, \dots, n\}$  mittels des rekursiven Aufrufs übergeben. Die Kopie  $w$  kann auf keiner Festplatte  $d_t$  mit  $t \in \{q+1, \dots, i-1\}$  platziert werden, da für diese Festplatten  $\text{rand}(b_z, d_t) > (w+1) \cdot \check{c}_t > w \cdot \check{c}_t$  gilt. Die Festplatten  $d_t$  für  $t \in \{i+1, \dots, l-1\}$  schließen sich aus, da sie auch zuvor nicht verwendet wurden. Somit bleiben nur die Festplatten  $d_i$  und  $d_l$  übrig.  $\square$

**Lemma 3.4.21.** *Sei  $d^{\text{neu}}$  eine neu einzufügende Festplatte mit  $c^{\text{neu}} \geq c_1$ . Redundant Share ist  $\frac{k+1}{2}$ -adaptiv mit Beachtung der Reihenfolge der Kopien für das Einfügen von  $d^{\text{neu}}$ .*

*Beweis.* Der Beweis baut auf den Beweis von Lemma 3.4.11 auf.

Da die neue Festplatte an erster Position eingefügt wird, kann sie nur Kopien  $k$  enthalten. Alle Kopien von Blöcken, deren Kopie  $k$  nicht auf der neuen Festplatte platziert werden, verbleiben auf ihren Festplatten, da sich  $\check{c}_i$  für die vorhandenen Festplatten nicht verändert.

Lemma 3.4.20 besagt, dass sich die restlichen Kopien nur innerhalb der Festplatten, die auch zuvor verwendet wurden, um eine Festplatte nach vorne bewegen können. Die Wahrscheinlichkeit, dass eine Kopie  $l$  mit  $2 \geq l \geq k$  bewegt wird, sofern die Kopie  $(l-1)$  zuvor von  $d_j$  weg bewegt wurde, ist:

### 3 Strategien zur redundanten Datenverteilung

$$\begin{aligned} P_{l,j} &= \frac{(k-l+1)\check{c}_j}{(k-l+2)\check{c}_j} \\ &= \frac{k-l+1}{k-l+2} \end{aligned}$$

Damit ist die Wahrscheinlichkeit, dass die Kopie  $l$  bewegt wird:

$$\prod_{u=2}^l \frac{k-v+1}{k-v+2} = \frac{1}{k}$$

Aufsummiert über alle  $k$  erhalte ich damit im erwarteten Fall folgende Anzahl an Bewegungen für jeden Block, dessen Kopie  $k$  auf der neuen Festplatte platziert wurden:

$$\sum_{i=1}^k \left( i \cdot \frac{1}{k} \right) = \frac{1}{k} \sum_{i=1}^k i = \frac{k-1}{2}$$

Somit ergibt sich eine Adaptivität von  $\frac{k+1}{2}$  im erwarteten Fall. □

**Lemma 3.4.22.** Sei  $d^{\text{neu}}$  eine neu einzufügende Festplatte mit  $c^{\text{neu}} \geq c_1$ . Redundant Share ist 1-adaptiv ohne Beachtung der Reihenfolge der Kopien für das Einfügen von  $d^{\text{neu}}$ .

*Beweis.*  $d^{\text{neu}}$  wird wieder als erste Festplatte in die Konfiguration eingefügt. Auf  $d^{\text{neu}}$  werden dann nur Kopien  $k$  platziert. Lemma 3.4.20 besagt, dass sich die restlichen Kopien über Festplatten verteilen, welche auch vorher schon eine Kopie erhalten haben. Damit gibt es genau eine Festplatte  $d_v$ , auf welcher mittels der ursprünglichen Konfiguration eine Kopie platziert wurde, mittels der neuen aber nicht. Wird diese Kopie auf  $d^{\text{neu}}$  verschoben, sind keine weiteren Verschiebungen notwendig. □

**Lemma 3.4.23.** Redundant Share ist für das Entfernen von  $d_1$  1-adaptiv ohne Beachtung der Reihenfolge der Kopien und  $\frac{k+1}{2}$ -adaptiv mit Beachtung der Reihenfolge der Kopien.

*Beweis.* In Lemma 3.4.21 und 3.4.22 habe ich gezeigt, dass eine Festplatte an erster Position einzufügen 1-adaptiv ohne Beachtung der Reihenfolge der Kopien und  $\frac{k+1}{2}$ -adaptiv mit Beachtung der Reihenfolge der Kopien ist. Die Stabilität von Redundant Share sichert, dass das Entfernen der Festplatte an erster Position die gleiche Adaptivität besitzt. □

**Lemma 3.4.24.** Redundant Share ist für das Entfernen einer Festplatte  $d_i$  mit  $i \in \{1, \dots, n\}$  aus einer homogenen Konfiguration ohne Beachtung der Reihenfolge der Kopien 3-adaptiv und mit Beachtung der Reihenfolge  $(2 + \frac{k+1}{2})$ -adaptiv.



### 3.4 Redundant Share: Ein Algorithmus zur redundanten Datenverteilung

*Beweis.* Zunächst vertausche ich den Inhalt der Festplatte  $d_0$  mit dem der Festplatte  $d_i$  und vertausche die Identitäten der beiden Festplatten. Danach wird  $d_i$ , welche nun an erster Position steht, entfernt. Da das Entfernen von  $d_0$  ohne Beachtung der Reihenfolge der Kopien 1-adaptiv ist, ist das Entfernen von  $d_i$  3-adaptiv. Mit Beachtung der Reihenfolge ist das Entfernen von  $d_0$   $\frac{k+1}{2}$ -adaptiv, daher ist das Entfernen von  $d_i$   $(2 + \frac{k+1}{2})$ -adaptiv.  $\square$

**Lemma 3.4.25.** *Redundant Share ist  $\ln n$ -adaptiv beim Einfügen oder Entfernen einer Festplatte ohne Beachtung der Reihenfolge der Kopien.*

*Beweis.* Der Beweis folgt unmittelbar aus dem Beweis von Lemma 3.4.6. Ich argumentiere dort lediglich über die Veränderung der relativen Kapazitäten  $\check{c}_i$  für  $i \in \{1, \dots, n\}$  und die erwartete Last der Festplatten. Laut Lemma 3.4.19 besitzt *Redundant Share* eine Fairness von 1, daher ist die erwartete Last jeder Festplatte  $d_i$  mit  $i \in \{1, \dots, n\}$ :

$$E[L_i] = k \cdot m \cdot \frac{c_i}{C}$$

Die erwartete Anzahl an Replatzierungen ist unabhängig von der wirklichen Last und setzt lediglich die Fairness voraus. Daher ist *Redundant Share*  $\ln n$ -adaptiv ohne Beachtung der Reihenfolge der Kopien bezüglich des Einfügens und Entfernens einer Festplatte.  $\square$

**Lemma 3.4.26.** *Redundant Share ist  $(\frac{k+1}{2} \cdot \ln n)$ -adaptiv beim Einfügen und Entfernen einer Festplatte mit Beachtung der Reihenfolge der Kopien.*

*Beweis.* Ich habe in Lemma 3.4.25 gezeigt, dass ohne Berücksichtigung der Reihenfolge der Kopien *Redundant Share*  $\ln n$ -adaptiv für das Einfügen einer Festplatte ist. Weiterhin habe ich in dem Beweis zu Lemma 3.4.21 gezeigt, dass mit Berücksichtigung der Reihenfolge im erwarteten Fall  $\frac{k-1}{2}$  Replatzierungen nötig sind, wenn eine Kopie  $k$  auf eine frühere Festplatte verschoben wird.

Wie zuvor auch im Beweis zu Lemma 3.4.2 kann ich keinerlei Annahmen darüber treffen, aus wie vielen Kopien  $x$  mit  $k \leq x \leq 1$  sich die Replatzierungen ohne Beachtung der Reihenfolge der Kopien zusammensetzen. Daher muss ich wieder vom schlimmsten Fall ausgehen. Dieser Fall tritt ein, wenn alle verschobenen Daten Kopien  $k$  waren. Dies ergibt eine Adaptivität von  $\frac{k+1}{2} \cdot \ln n$ .  $\square$

#### 3.4.4 $k$ -redundante Platzierung in Laufzeit $O(k)$

Das im vorherigen Abschnitt vorgestellte Verfahren *Redundant Share* hat eine lineare Zeit- und Hauptspeichereffizienz bezüglich der Anzahl der Festplatten. Durch Verwendung von mehr Hauptspeicher kann ein darauf aufbauendes Verfahren mit linearer Zeiteffizienz bezüglich der

### 3 Strategien zur redundanten Datenverteilung

Anzahl der zu platzierenden Kopien entwickelt werden. In diesem Abschnitt werde ich zeigen, wie sich eine solche Beschleunigung erreichen lässt.

Die Idee hinter *Fast Redundant Share* ist, für das Auffinden der einzelnen Kopien zusätzliche, nicht redundante Verteiler zu verwenden. Verwendet man hierbei Verteiler mit konstanter Laufzeit, so können die  $k$  Kopien eines Blocks in Laufzeit von  $O(k)$  platziert werden. Solche Verteiler könnten beispielsweise mittels des in Abschnitt 3.2 beschriebenen Verfahrens *Share* implementiert werden. Innerhalb dieser Betrachtung gehe ich davon aus, dass die verwendeten  $[n, 1]$ -Verteiler eine Fairness und Adaptivität von eins besitzen. Solch einem Verteiler kann man sich mittels *Share* beliebig annähern, indem genügend große Dehnungsfaktoren gewählt werden.

Um das Verfahren einfacher verdeutlichen zu können, fange ich wieder mit der Verteilung einer Kopie an. Diese Kopie soll auf einer der Festplatten abgelegt werden. *Redundant Share* würde hierzu beginnend mit der ersten Festplatte für jede Festplatte  $d_i$  mit  $i \in \{1, \dots, n\}$  ein Experiment durchführen, welches mit Wahrscheinlichkeit  $\check{c}_i$  gelingt. Die erste Festplatte, für die das Experiment gelingt, würde den Block erhalten.

Das gleiche kann durch einen nicht redundanten  $[n, 1]$ -Verteiler erreicht werden. Dazu wird jeder Festplatte  $d_i$  mit  $i \in \{1, \dots, n\}$  eine virtuelle Größe von  $P_{0,i,1} \cdot \check{c}_i = \frac{c_i}{C}$  zugewiesen. Dabei verwende ich die in Abschnitt 3.4.3 definierte Funktion  $P_{a,b,l}$ . Die Funktion gibt die Wahrscheinlichkeit wieder, dass eine Kopie  $l$  auf keiner Festplatte  $d_x$  mit  $a < x < b$  platziert wurde, falls die Kopie  $l + 1$  auf  $d_a$  abgelegt wurde. Dieser Argumentation folgend gehe ich bei der Definition von  $P_{a,b,l}$  davon aus, dass eine imaginäre Kopie  $k + 1$  immer auf einer imaginären Festplatte  $d_0$  gespeichert wurde. Somit wird die Kopie  $k$  ab der Festplatte  $d_1$  platziert.

Sollen nun mehrere Kopien erstellt werden, ist dies auf einem ähnlichen Weg möglich. Hierzu verwende ich mehrere nicht redundante Verteiler. Ich bezeichne  $H_i^l$  als den Verteiler zur Platzierung von Kopie  $l \in \{k, \dots, 1\}$ , falls Kopie  $l + 1$  auf der Festplatte  $d_{i-1}$  platziert wurde. Speziell ist  $H_1^k$  der Verteiler, der zur Platzierung der Kopie  $k$  verwendet wird.

Um  $H_1^k$  zu berechnen erstelle ich zu jeder Festplatte  $d_i$  eine virtuelle Kapazität  $c_i^l = P_{0,i,k} \cdot \min(1, k \cdot \check{c}_i)$ .  $H_1^k$  ist ein nicht redundanter Verteiler, welcher Daten gemäß dieser virtuellen Kapazitäten verteilt. Dabei nutze ich aus, dass  $P_{a,b,l}$  so definiert wurde, dass es für Festplatten, die keine Kopie  $l$  erhalten, 0 als Ergebnis liefert. Damit haben solche Festplatten auch eine virtuelle Kapazität von 0.

Für jede Festplatte  $d_{i-1}$  auf welcher eine Kopie  $l + 1$  mit  $l \in \{k - 1, \dots, 1\}$  platziert werden kann, wird nun ein Verteiler  $H_i^l$  vorbereitet. Dabei erhält jede Festplatte  $d_j$  mit  $j \in \{1, \dots, i - 1\}$  eine virtuelle Kapazität von  $c_j^l = 0$ . Jede Festplatte  $d_j$  mit  $j \in \{i, \dots, n\}$  erhält eine virtuelle Kapazität von  $c_j^l = P_{j+1,i,l} \cdot \min(1, l \cdot \check{c}_j)$ .

Wie in Abschnitt 3.4.3 gezeigt, muss für jedes  $l \in \{k, \dots, 2\}$  für die erste Festplatte  $d_{e_l}$  mit  $e_l \in \{1, \dots, n\}$ , für die  $l \cdot \check{c}_{e_l} \geq 1$  gilt, eine Justierungskapazität  $c_{e_l}^*$  berechnet werden, falls  $l \cdot \check{c}_{e_l} \neq$

### 3.4 Redundant Share: Ein Algorithmus zur redundanten Datenverteilung

1. Diese Justierungskapazität wird verwendet, falls die Kopie  $l + 1$  auf der Festplatte  $d_{e_{l-1}}$  platziert wurde. Entsprechend muss diese angepasste Kapazität von  $d_{e_l}$  auch zur Berechnung der virtuellen Kapazität von  $d_{e_l}$  für den Verteiler  $H_{e_l}^l$  verwendet werden. Somit erhält  $d_{e_l}$  folgende virtuelle Kapazität:

$$c_{e_l}^l = l \cdot \frac{c_{e_l}^*}{c_{e_l}^* + C_{e_{l+1}}}$$

Die Eigenschaften von *Fast Redundant Share* hängen von den Eigenschaften der  $[n, 1]$ -Verteiler ab. Ich setze voraus, dass diese Verteiler eine stabile Verteilung liefern. Wie zuvor definiert müssen sie dazu bei gleicher Konfiguration für einen Block  $b_j$  für  $j \in \{1, \dots, m\}$  immer die gleiche Festplatte wählen. Unter dieser Voraussetzung liefert auch *Fast Redundant Share* eine stabile Verteilung.

Ich betrachte als erstes den Hauptspeicherverbrauch und die Laufzeit von *Fast Redundant Share*.

**Lemma 3.4.27.** *Gegeben ein nicht redundantes Verfahren zur Datenplatzierung, welche für einen Block in Laufzeit  $O(1)$  bei einem Hauptspeicherverbrauch von  $O(v)$  aus  $n$  Festplatten eine auswählt. Dann kann Fast Redundant Share so implementiert werden, dass es in Laufzeit  $O(k)$  bei einem Hauptspeicherverbrauch von  $O(k \cdot n \cdot v)$   $k$  Kopien eines Blocks platziert.*

*Beweis.* Die erste Kopie kann in konstanter Zeit mittels  $H_1^k$  platziert werden. Danach platziere ich die Kopien  $l \in [k - 1, \dots, 1]$  der Reihe nach. Wurde die Kopie  $l + 1$  auf  $d_{i-1}$  platziert, so verwende ich  $H_i^l$  um Kopie  $l$  in konstanter Zeit zu platzieren. Somit können die  $k$  Kopien in Laufzeit  $O(k)$  platziert werden.

Für die Platzierung von Kopie  $k$  wird lediglich ein Verteiler vorrätig gehalten. Für jede andere Kopie müssen bis zu  $n - k + 1 = O(n)$  Verteiler bereit gehalten werden. Somit müssen insgesamt bis zu  $O(k \cdot n)$  Verteiler im Hauptspeicher gehalten werden. Da jeder Verteiler einen Hauptspeicherverbrauch von  $O(v)$  hat, liegt der Hauptspeicherverbrauch von *Fast Redundant Share* in  $O(k \cdot n \cdot v)$ .  $\square$

Als nächstes untersuche ich die Fairness von *Fast Redundant Share*.

**Lemma 3.4.28.** *Gegeben nicht redundante, heterogene Verteiler in  $H$  mit einer Fairness von 1. Dann hat Fast Redundant Share eine Fairness von 1.*

*Beweis.* Ich definiere zunächst  $L_i^{a,l}$  als die Last, die  $d_i$  an Kopien  $l \in \{k, \dots, 1\}$  erhält, falls die Kopie  $l - 1$  auf  $d_{a-1}$  platziert wurde. Speziell definiere ich  $L_i^{1,k}$  als die Last, die  $d_i$  für Kopien  $k$  erhält.

Mittels  $H_1^k$  werden die Kopien  $k$  verteilt. Dabei sind die virtuellen Kapazitäten gerade so berechnet, dass jede Festplatte die gleiche erwartete Last an Kopien  $k$  erhält, wie mittels der Verteilung

### 3 Strategien zur redundanten Datenverteilung

durch *Redundant Share* aus Abschnitt 3.4.3. Da  $H_1^k$  eine Fairness von 1 besitzt, ist  $E[L_i^{1,k}]$  bei *Redundant Share* und *Fast Redundant Share* identisch.

Wurde eine Kopie  $l + 1$  für  $l \in \{k - 1, \dots, 1\}$  auf der Festplatte  $d_{i-1}$  platziert, so sichert die Berechnungsvorschrift der virtuellen Kapazitäten von  $H_i^l$  zu, dass die Kopie  $l$  auf einer Festplatte  $d_j$  mit  $j \in \{i, \dots, n\}$  platziert wird. Als virtuelle Kapazität einer Festplatte  $d_j$  für  $H_i^l$  mit  $j \in \{1, \dots, n\}$  habe ich exakt die Wahrscheinlichkeit gewählt, dass sie bei der linearen Verteilung die Kopie  $l$  eines Blocks erhält, sofern Kopie  $l + 1$  auf  $d_{i-1}$  platziert wurde. Da  $H_i^l$  eine Fairness von 1 hat, ist die erwartete Last  $E[L_j^{i,l}]$  mittels *Redundant Share* und *Fast Redundant Share* identisch.

Da für jede Festplatte  $d_i$  die erwarteten Teillasten  $E[L_i^{a,l}]$  zwischen *Redundant Share* und *Fast Redundant Share* übereinstimmen, stimmt auch die sich daraus summierende erwartete Gesamtlast  $E[L_i]$  jeder Festplatte bei beiden Verfahren überein. Da *Redundant Share* eine Fairness von 1 hat und alle Verteiler in  $H$  eine Fairness von 1 haben, muss daher auch *Fast Redundant Share* eine Fairness von 1 haben.  $\square$

Um nun die Adaptivität von *Fast Redundant Share* zu zeigen, fordere ich von der nicht redundanten heterogenen Hashfunktion, dass sie eine Adaptivität von 1 bezüglich Änderungen an dem unterliegenden Speicher aufweist. Diese Forderung unterscheidet sich gegenüber der Adaptivität, wie ich sie in Abschnitt 3.1 beschrieben habe. Ändern sich die Kapazitäten  $c'_1, \dots, c'_n$  einer Konfiguration zu  $c''_1, \dots, c''_n$ , so darf ein Verteiler mit einer Adaptivität von 1 bezüglich Änderungen an der Konfiguration nur die Daten bewegen, die auf Grund der geänderten Kapazitäten zur Erhaltung der Fairness bewegt werden müssen.

**Lemma 3.4.29.** *Fast Redundant Share ist bezüglich des Einfügens und Entfernens einer Festplatte ohne Beachtung der Reihenfolge der Kopien  $\ln n$ -adaptiv, falls die unterliegende Hashfunktion bezüglich Änderungen der Konfiguration 1-adaptiv ist.*

*Beweis.* Ich gehe auf die Adaptivität beim Einfügen einer Festplatte ein, die Adaptivität beim Entfernen einer Festplatte ergibt sich dann aus der Stabilität von *Fast Redundant Share*.

Bereits in Lemma 3.4.25 habe ich gezeigt, dass *Redundant Share*  $\ln n$ -adaptiv bezüglich des Einfügens einer neuen Festplatte ist. Dieser Beweis ging zurück auf Lemma 3.4.6. Dort habe ich die Replatzierung der Daten betrachtet, unabhängig der Anzahl der Kopien von Blöcken, aus welchen sich diese Daten zusammen setzen.

Für eine Festplatte  $d_i$  habe ich mit  $\check{c}_i^{\text{alt}}$  die relative Kapazität vor dem Hinzufügen der neuen Festplatte bezeichnet. Als  $\check{c}_i^{\text{neu}}$  habe ich die relative Kapazität nach dem Hinzufügen der Festplatte bezeichnet. Dann habe ich betrachtet, mit welcher Wahrscheinlichkeit ein Element von einer Festplatte  $d_i$  weg bewegt wird:

$$p_i = \frac{\check{c}_i^{\text{alt}} - \check{c}_i^{\text{neu}}}{\check{c}_i^{\text{alt}}}$$

Für eine Festplatte  $d_i$  beschreibt  $p_i$  genau die Änderung ihrer relativen Kapazität  $\check{c}_i$ . Ich habe in dem Beweis zu Lemma 3.4.6 gezeigt, dass die Summe der Änderungen der relativen Kapazitäten maximal  $\ln n$  ist.

Die virtuellen Kapazitäten der Festplatten, die in den Verteilern  $H$  verwendet werden, hängen nun genau von den relativen Kapazitäten ab. Da die Verteiler laut Voraussetzung diese Änderungen 1-adaptiv umsetzen können, ist *Fast Redundant Share*  $\ln n$ -adaptiv.  $\square$

## 3.5 Peer Replikation: Lesen ohne komplette Sicht

*Peer-to-Peer (P2P)* Umgebungen stellen besondere Anforderungen an Verteilungsverfahren. Stellen wir uns vor, die Daten eines Speichersystems sind verteilt auf verschiedene Server, wobei nicht jedem Server die komplette Konfiguration bekannt ist. Will nun ein Server Daten wieder finden, so sucht er mittels einer fehlerhaften Verteilungsfunktion. Diesem Problem widmet sich dieser Abschnitt.

In aktuellen *P2P*-Umgebungen wird sehr häufig das bereits in Abschnitt 3.2 vorgestellte *Consistent Hashing* [Karger u. a., 1997] als Verteilungsfunktion eingesetzt. Dieses Verfahren liefert allerdings nur für homogene Festplatten mit hoher Wahrscheinlichkeit eine faire Verteilung. Daher wird in diesem Abschnitt *Redundant Share* auf seine Eignung zum Einsatz in diesem Umfeld untersucht.

Ich gehe dazu davon aus, dass zuvor Daten über einer kompletten Konfiguration  $K$  verteilt wurden. Zu dieser Konfiguration gehe ich von einer fehlerhaften Konfiguration  $S$  aus, welche eine Teilmenge der Festplatten aus  $K$  enthält. Die Reihenfolge der Festplatten bleibt erhalten, die Anzahl der Festplatten bezeichne ich mit  $n'$ . Ich untersuche, unter welchen Voraussetzungen *Redundant Share* Kopien von Blöcken in  $S$  nicht wieder findet, welche mit der Konfiguration von  $K$  platziert wurden.

Abschließend stelle ich mit *Peer Replikation* eine auf dieses Model angepasste Version von *Redundant Share* vor. Dabei gehe ich davon aus, dass von den einzelnen Festplatten ausgelesen werden kann, ob auf ihnen eine Kopie eines Blocks platziert wurde.

Die in diesen Abschnitt vorgestellten Ergebnisse beruhen auf den in [Brinkmann und Effert, 2008a] vorgestellten Ergebnissen.

#### 3.5.1 Modell

Ich arbeite weiter auf dem bisherigen Modell, welches in Abschnitt 3.1 vorgestellt wurde. Als Erweiterung definiere ich zusätzlich die Sicht eines Servers:

**Definition 3.5.1.** Die Sicht  $S$  eines Servers bezeichnet die für ihn sichtbare Teilmenge aller Festplatten einer Konfiguration  $K$ .  $\bar{S} = K - S$  ist die Menge aller Festplatten, die der Server nicht kennt.

$$C^S = \sum_{d_i \in S} c_i \leq C$$

ist die Gesamtkapazität der Sicht und

$$C^{\bar{S}} = \sum_{d_i \in \bar{S}} c_i \leq C$$

ist die fehlende Gesamtkapazität. Ferner ist  $\tau = \frac{C^S}{C}$  der Anteil der Sicht am gesamten Speicher von  $K$ .

Steht einem Server nur eine unvollständige Sicht  $S$  auf dem Speicher zur Verfügung, so kann er nicht alle Daten erreichen, welche mit einer kompletten Sicht  $K$  platziert wurden. Somit können nicht mehr alle Kopien jedes Blocks gefunden werden. Daher sinkt auch die Wahrscheinlichkeit, dass eine der Kopien eines Blocks gefunden werden kann.

Ein  $[n, k]$ -Verteiler liefert eine Menge  $v$  von  $k$  unterschiedlichen Festplatten. Ebenso liefert ein  $[n', k]$ -Verteiler eine Menge  $v'$  von  $k$  Festplatten einer unvollständigen Sicht. Vergleicht man  $v$  und  $v'$ , so können sie sich in verschiedenen Punkten unterscheiden. Zunächst kann  $v$  Festplatten enthalten, welche in  $v'$  fehlen. Ist so eine Festplatte in  $S$  enthalten, so spreche ich von einem *falschen Nein*, weil der  $[n', k]$ -Verteiler eine sichtbare Kopie nicht gefunden hat. Weiterhin kann  $v'$  auch Festplatten enthalten, welche nicht in  $v$  enthalten sind. In diesem Fall spreche ich von einem *falschen Ja*.

#### 3.5.2 Redundant Share mit unvollständiger Sicht

Wie in Abschnitt 3.2 beschrieben verteilt *Consistent Hashing* Kopien der Festplatten über einem  $[0, 1)$ -Intervall. Über dem gleichen Intervall werden auch die Blöcke verteilt, wobei jeder Block auf der Festplatte mit dem nächst kleineren Wert platziert wird. *Consistent Hashing* kann kein falsches Nein erzeugen, denn dazu müsste durch die fehlerhafte Sicht eine andere Festplatte näher an dem Block platziert werden, als dies bei der kompletten Sicht passiert. Da allerdings alle Festplatten unabhängig von der Sicht immer an gleicher Position abgelegt werden, ist dies nicht möglich. Aus dem gleichen Grund kann *Consistent Hashing* ein falsches Ja nur erzeugen, falls die Festplatte, welche den Block gespeichert hat, nicht in der Sicht des Servers ist. Das Verhalten von *Consistent Hashing* bezüglich einer unvollständigen Sicht ist somit perfekt.

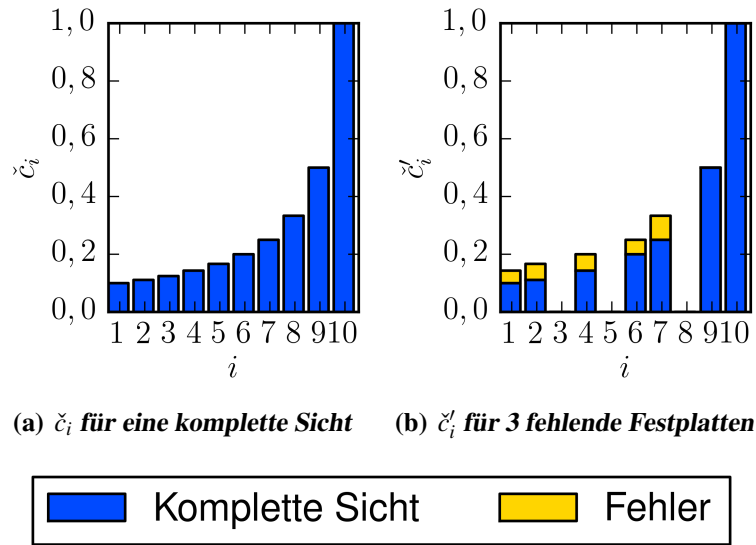


Abbildung 3.7:  $\check{c}_i$  und  $\check{c}'_i$  für zehn Festplatten mit identischer Kapazität

Betrachtet man im Gegensatz dazu *Redundant Share*, so fällt recht schnell auf, dass sowohl ein falsches Ja als auch ein falsches Nein auftreten können. Die Pseudorandomisierung *rand* liefert einen Wert abhängig von dem Block und der Identität der jeweiligen Festplatte. Dies geschieht unabhängig von der restlichen Konfiguration. Somit liefert die Pseudorandomisierung für die gleiche Festplatte den gleichen Wert, egal ob die komplette Konfiguration  $K$  oder die unvollständige  $S$  verwendet wird.

Für die Festplatten  $d_i$  mit  $i \in \{1, \dots, n\}$  werden die relativen Kapazitäten wie folgt berechnet:

$$\check{c}_i = \frac{c_i}{C_i}$$

Für eine Festplatte  $d_i$  mit  $i \in \{1, \dots, n\}$  hängt ihre relative Kapazität  $\check{c}_i$  also nur von den Festplatten  $d_j$  mit  $j \in \{i, \dots, n\}$  ab. Sei  $\check{c}'_i$  die relative Kapazität der Festplatte  $d_i$  über der eingeschränkten Sicht. Das Fehlen einer Festplatte  $d_l$  mit  $l \in \{1, \dots, i-1\}$  hat keine Auswirkung auf  $\check{c}'_i$ . Fehlt eine Festplatte, so verringert sich die Restkapazität und  $\check{c}'_i$  wird größer. Es gilt immer  $\check{c}_i \leq \check{c}'_i$ , da  $c_i$  konstant ist und die Restkapazität durch fehlende Festplatten nur kleiner werden kann.

Abbildung 3.7 stellt beispielhaft die Berechnung von  $\check{c}_i$  dar. In Abbildung 3.7(a) wird  $\check{c}_i$  für zehn homogene Festplatten dargestellt. Für Abbildung 3.7(b) wurde  $\check{c}'_i$  berechnet, hier waren die Festplatten 3, 5 und 8 nicht Teil der Sicht. Der gelbe Teil der Balken stellt den Bereich dar, um den das einzelne  $\check{c}'_i$  größer als  $\check{c}_i$  ist.

Mit diesem Wissen kann ich nun zwei wichtige Eigenschaften von *Redundant Share* auf einer fehlerhaften Sicht zeigen:

### 3 Strategien zur redundanten Datenverteilung

**Lemma 3.5.2.** *Mittels Redundant Share kann auf einer nicht kompletten Sicht ein falsches Nein nur erzeugt werden, wenn vorher ein falsches Ja aufgetreten ist.*

*Beweis.* Wie bereits dargestellt gilt für alle Festplatten  $d_i$ , dass  $\check{c}_i \leq \check{c}'_i$  ist. Bei der Bestimmung der Festplatten für einen Block erzeugt *Redundant Share* in den einzelnen Schritte einen pseudozufälligen Wert für jede Festplatte. Wie zuvor ausgeführt wird hier unabhängig von der restlichen Konfiguration für eine feste Festplatte und einen festen Block der gleiche Wert gewählt. Daher kann nur dann eine Festplatte fälschlicherweise nicht ausgewählt werden, falls  $k$  bei dem Experiment gegen diese Festplatte zu klein ist. Dies kann nur passieren, wenn  $k$  zuvor durch ein falsches Ja verkleinert wurde.  $\square$

**Lemma 3.5.3.** *In Redundant Share kann pro falschem Ja maximal ein falsches Nein auftreten.*

*Beweis.* Wie im Beweis von Lemma 3.5.2 gezeigt wurde, entsteht ein falsches Nein durch ein fehlerhaft dekrementiertes  $k$ . Da bei einem falschen Nein allerdings  $k$  nicht dekrementiert wird, obwohl dies bei kompletter Sicht hätte passieren müssen, ist damit die Auswirkung eines falschen Ja aufgehoben.  $\square$

Diese beiden Eigenschaften werden später in Abschnitt 3.5.3 benötigt, um *Redundant Share* an die besonderen Bedürfnisse von *P2P*-Umgebungen anzupassen. Ich betrachte weiter, wie sich *Redundant Share* ohne Modifikationen verhält.

Ich habe bereits gezeigt, dass  $\check{c}'_i \neq \check{c}_i$  nur gilt, falls mindestens eine Festplatte  $d_l$  mit  $l \in \{i, \dots, n\}$  nicht Teil der Sicht ist. Damit ist erkennbar, dass es relevant ist, an welcher Stelle die fehlenden Festplatten sich befinden.

**Lemma 3.5.4.** *Bei Verwendung von Redundant Share ist für eine Sicht mit einem festen Anteil am Gesamtspeicher  $\tau$  im erwarteten Fall die Anzahl an falschen Ja maximal, wenn der am Ende einsortierte Speicher nicht Teil der Sicht ist.*

*Beweis.* Zu einer Festplatte  $d_i$  mit  $i \in \{1, \dots, n\}$  bezeichnet  $\check{c}_i$  die relative Kapazität bei kompletter Sicht und  $\check{c}'_i$  die relative Kapazität bei eingeschränkter Sicht.  $d_i$  erhält bei fehlerhafter Sicht eine Kopie, die bei kompletter Sicht nicht auf  $d_i$  platziert wurde, wenn  $k \cdot \check{c}_i < \text{rand} \leq k \cdot \check{c}'_i$ , also wenn das Experiment für  $\check{c}_i$  fehlschlägt, aber für  $\check{c}'_i$  gelingt. In Abbildung 3.7(b) bedeutet dies, dass der Hashwert in den gelben Bereich fällt. Die Wahrscheinlichkeit, dass  $d_i$  ein falsches Ja erzeugt ist damit  $p_{i,k}^{\text{falsch}} = k \cdot \check{c}'_i - k \cdot \check{c}_i$ . Dieser Wert maximiert sich für jedes  $d_i$  wenn möglichst viel des nicht sichtbaren Speichers in die Berechnung von  $\check{c}_i$  eingeflossen ist, also wenn der gesamte fehlende Speicher am Ende einsortiert war.  $\square$



---

**Algorithmus 5** PeerReplication ( $k, \text{addr}, \{d_1, \dots, d_n\}, \{c_1, \dots, c_n\}$ )

---

**Voraussetzung:**  $\forall i \in \{1, \dots, n-1\} : b_i \geq b_{i+1}$ 
**Voraussetzung:**  $k \cdot c_0 < C$ 
**Voraussetzung:**  $\forall i \in \{1, \dots, n-1\} : c_i \geq c_{i+1}$ 
**Voraussetzung:**  $\forall i \in \{1, \dots, n\} : k \cdot c_i \leq C$ 

1:  $\forall i \in \{1, \dots, n\} : \check{c}_i = k \cdot c_i / \sum_{j=i}^n c_j$ 

2:  $i \leftarrow 1$ 

3: **while**  $i < n$  **do**

4:    $\text{val} \leftarrow \text{rand}(\text{addr}, d_i) \in [0, 1)$ 

5:   **if**  $\text{val} < k \cdot \check{c}_i$  **then**

6:     **Frage Block von Festplatte  $d_i$  ab**

7:     **if Block wurde auf  $d_i$  gefunden then**

8:       Setze  $d_i$  als Kopie  $k$ 

9:        $c^* \leftarrow c_{i+1}$ 

10:       **if**  $k \cdot \check{c}_i < 1$  and  $k \cdot \check{c}_{i+1} > 1$  **then**

11:           $c^* \leftarrow \text{adjust}(k, i, c_0, \dots, c_{n-1})$ 

12:       **end if**

13:       **if**  $(k > 1)$  **then**

14:          peerReplication( $k-1, \text{addr}, \{d_{i+1}, \dots, d_n\}, \{c^*, c_{i+2}, \dots, c_n\}$ )

15:       **end if**

16:       **return**

17:     **end if**

18:   **end if**

19:    $i \leftarrow i + 1$ 

20: **end while**


---

### 3.5.3 Peer Replikation

*Redundant Share* eignet sich auf Grund der auftretenden falschen Ja und den daraus resultierenden falschen Nein nicht sehr gut für den Einsatz in einer P2P-Umgebung mit einer beschränkten Sicht. Allerdings ist es recht einfach möglich durch etwas mehr Kommunikation das Auftreten eines falschen Ja zu unterbinden. Dazu muss für jedes gelungene Experiment geprüft werden, ob die entsprechende Festplatte eine Kopie des Blocks hält. Ist dies nicht der Fall, so wird das Ergebnis des Experiments ignoriert und das Verfahren läuft mit unverändertem  $k$  weiter. Algorithmus 5 zeigt das modifizierte Verfahren als Pseudocode.

**Lemma 3.5.5.** Peer Replication erzeugt kein falsches Ja und kein falsches Nein.

*Beweis.* Das Erzeugen eines falschen Ja wird unterbunden, indem der Algorithmus für jedes gelungene Experiment prüft, ob auf der Festplatte wirklich eine Kopie vorliegt. Da somit kein

### 3 Strategien zur redundanten Datenverteilung

falsches Ja auftreten kann, kann nach Lemma 3.5.2 auch kein falsches Nein erzeugt werden. Somit verhält sich *Peer Replication* optimal.  $\square$

Durch dieses modifizierte Verfahren ist es also möglich, auch in einer fehlerhaften Konfiguration alle noch vorhandenen Daten wieder zu finden.

Für die Betrachtung der Laufzeit und des Speicherverbrauchs zum Auffinden von Kopien mittels *Peer Replikation* setze ich voraus, dass in konstanter Laufzeit und mit konstantem Hauptspeicherverbrauch überprüft werden kann, ob eine Festplatte eine Kopie eines Blocks gespeichert hat. Abgesehen von dieser Überprüfung verhält sich *Peer Replikation* wie *Redundant Share*. Gegeben eine unvollständige Konfiguration mit  $n'$  Festplatten hat *Peer Replikation* daher eine Laufzeit zum Auffinden der Kopien von  $O(n')$  bei einem Hauptspeicherverbrauch von  $O(n')$ .

*Peer Replikation* eignet sich lediglich zum Auffinden bereits platzierter Kopien und nimmt keine Änderungen an dem unterliegenden Speicher vor. Zu jeder ausgewählten Festplatte überprüft *Peer Replikation*, ob eine Kopie des Blocks auf der Festplatte platziert wurde. Für noch nicht platzierte Blöcke schlägt dies immer fehl. Somit ist es nicht möglich die Fairness oder die Adaptivität von *Peer Replikation* zu betrachten.

## 4 Evaluation der Strategien

Um den praktischen Einsatz der verschiedenen Verteilungsverfahren testen zu können, habe ich eine C++-Implementierung von *Consistent Hashing*, *Share* und *Redundant Share* erstellt.

Alle Implementierungen halten sich an die Verfahren, wurden allerdings auf Integer-Arithmetik optimiert. Das bedeutet, dass die Verfahren statt in einem  $[0, 1)$  Raum in einem ganzzahligen  $[0, 2^{64})$  Raum agieren. Als Ersatz für eine uniforme Hashfunktion zur Randomisierung habe ich *SHAI* [Nationale Institute for Standards and Technology (NIST), 2004] mit angepassten Nachrichten verwendet.

Abweichend von der Beschreibung in Abschnitt 3.4.3 halte ich bei der Implementierung von *Redundant Share* zur Platzierung von  $k$  Kopien neben den  $\check{c}_i$  für  $l \in \{1, \dots, k\}$  auch  $l \cdot \check{c}_i$  vor, um diese Multiplikationen einzusparen. Einen festen Hauptspeicherverbrauch für Zahlen und Identifizierer vorausgesetzt erhöhe ich so den Hauptspeicherverbrauch von *Redundant Share* auf  $O(n \cdot k)$ . Im Gegenzug kann ich die Geschwindigkeit erheblich erhöhen.

Ich habe auch das in Abschnitt 3.4.4 beschriebene Verfahren *Fast Redundant Share* implementiert. Als nicht redundante Verteiler verwende ich dabei *Share*. Ich zeige in Abschnitt 4.1, dass auch für Konfigurationen mit wenig Festplatten der Hauptspeicherverbrauch so groß ist, dass eine sinnvolle Vermessung dieses Verfahrens nicht möglich ist.

*Consistent Hashing* habe ich um die Möglichkeit erweitert, mehrere Kopien einer Festplatte abhängig von ihrer Kapazität zu platzieren. Dazu kann ich der Implementierung einen Heterogenitätsdivisor  $v$  übergeben. Mit  $\alpha$  als Anzahl der Kopien, die pro Anteil einer Festplatte platziert werden sollen habe ich dann  $\lceil \frac{c_i}{v} \rceil \cdot \alpha$  Kopien jeder Festplatte  $d_i$  mit  $i \in \{1, \dots, n\}$  platziert.

Sowohl *Share* als auch *Consistent Hashing* habe ich um die Möglichkeit erweitert,  $k$  Kopien zu platzieren. Ich habe die Verfahren dazu so modifiziert, dass sie für jeden Block mehrere unabhängige Experimente durchführen, bis  $k$  verschiedene Festplatten ausgewählt wurden.

Die Anzahl der platzierten Kopien jeder Festplatte in *Consistent Hashing* und der Dehnungsfaktor von *Share* sind in meiner Implementierung konfigurierbar. Falls ich keine andere Aussage treffe, so platziere ich in *Consistent Hashing*  $400 \cdot n \cdot \lceil \log n \rceil$  Kopien der  $n$  Festplatten einer Konfiguration. Bei den Vermessungen von *Share* verwende ich einen Dehnungsfaktor von  $5 \cdot \log n$ , falls ich nichts anderes schreibe.

Für die Implementierung habe ich neben der C++ *Standard Template Library* verschiedene Bibliotheken verwendet. Für die Verteilung relevant ist hierbei die *GNU Multi Precision Arith-*

## 4 Evaluation der Strategien

*metic Library*<sup>1</sup>, welche ich zur Berechnung der angepassten Kapazitäten  $c^*$  in *Redundant Share* verwende. *GMP* erlaubt Berechnungen mit beliebiger Genauigkeit. Ich verwende *GMP* nur während der Initialisierung und bilde die Werte danach auf 64-Bit Integer Werte ab.

Alle Tests, bei denen die zugrunde liegende Hardware relevant ist, also die Messungen von Hauptspeicherverbrauch und Laufzeiten, wurden auf Knoten des *BisGrid*-Clusters<sup>2</sup> des *Paderborn Center for Parallel Computing (PC<sup>2</sup>)* der Universität Paderborn berechnet. Jeder Knoten dieses Clusters ist mit vier *dual-core Opteron* Prozessoren mit 2.8 GHz ausgestattet, es stehen also acht Kerne zur Verfügung. Ferner hat jeder Knoten 64 GByte Hauptspeicher. Als Betriebssystem verwenden die Knoten *RedHat Enterprise Linux 5*. Einige Tests, welche sich lediglich auf die Verteilungsgüte beziehen und damit unabhängig von der verwendeten Hardware sind, wurden auf anderen Systemen durchgeführt.

Im Folgenden betrachte ich die einzelnen Verfahren unter verschiedenen Gesichtspunkten. Zunächst vergleiche ich in Abschnitt 4.1 die Verfahren gemäß ihrer Laufzeit und ihres Hauptspeicherverbrauchs. Dazu initialisiere ich die Verfahren *Consistent Hashing*, *Share* und *Redundant Share* mit unterschiedlichen Konfigurationen und messe den Hauptspeicherverbrauch. Ferner berechne ich die Laufzeit pro platziertem Block.

In Abschnitt 4.2 betrachte ich dann die Fairness der verschiedenen Verteilungsverfahren. Dabei zeige ich zunächst, wie viele Blöcke über eine Konfiguration verteilt werden müssen, damit sich das Speichersystem eingeschwungen hat. Danach betrachte ich die Fairness der Implementierungen der Verteilungsverfahren über unterschiedlichen Konfigurationen und bei verschiedenen Parametrisierungen der Verfahren.

In Abschnitt 4.3 gehe ich dann auf die Adaptivität der verschiedenen Hashfunktionen ein. Dabei erweitere ich bestehende Konfigurationen um zusätzliche Festplatten und messe, wie viele Elemente gegenüber einer optimalen Verteilung unplatziert wurden.

### 4.1 Hauptspeicherverbrauch und Laufzeit

Die verschiedenen Verfahren zur Datenverteilung stellen sehr unterschiedliche Anforderungen an die Hardware. Zunächst unterscheiden sie sich in der Laufzeit. Während *Redundant Share* eine Laufzeit von  $O(n)$  besitzt, kommen die Verfahren *Share*, *Consistent Hashing* und *Fast Redundant Share* mit einer Laufzeit von  $O(k)$  im erwarteten Fall aus.

Dafür hat *Redundant Share* in meinem Model einen Hauptspeicherverbrauch von  $O(n \cdot k)$ , denn ich halte bei Platzierung von  $k$  Kopien auch die Vielfachen  $l \cdot \check{c}_i$  für  $l \in \{1, \dots, k\}$  vor, um die Platzierung zu beschleunigen. *Consistent Hashing* hat dagegen einen Speicherverbrauch von

<sup>1</sup>GMP - <http://gmp.lib.org>, am 4.11.2010

<sup>2</sup><http://pc2.uni-paderborn.de/hpc-systems-services/available-systems/bisgrid-cluster/>, am 4.11.2010

$O(\alpha \cdot n \cdot \lceil \log n \rceil)$ , wobei  $\alpha \cdot \lceil \log n \rceil$  die Anzahl der Kopien beschreibt, die von jeder Festplatte platziert werden. *Share* besitzt gar einen Speicheraufwand von  $O(s \cdot \alpha \cdot n^2 \cdot \lceil \log n \rceil)$  bei Verwendung von *Consistent Hashing* zur uniformen Verteilung. Dabei ist  $s$  der gewählte Dehnungsfaktor. *Fast Redundant Share* erreicht mit *Share* als nicht redundanten Verteiler einen Hauptspeicherverbrauch von  $O(s \cdot \alpha \cdot k \cdot n^3 \cdot \log n)$ . Ich zeige in Abschnitt 4.1.1, dass dieser Verbrauch zu groß ist, um *Fast Redundant Share* in skalierbaren Speichersystemen einzusetzen.

In diesem Abschnitt präsentiere ich die Ergebnisse der Vermessung der Laufzeit und des Speicherverbrauchs meiner Implementierung der verschiedenen Verfahren. Dabei betrachte ich im ersten Schritt homogene Konfigurationen, im zweiten heterogene.

Während jedes Tests habe ich zunächst die Verteilungsalgorithmen initialisiert. Im Fall von *Consistent Hashing* habe ich wie in Abschnitt 3.2 beschrieben die Kopien der Festplatten platziert und eine Hashfunktion erstellt, welche in erwarteter Laufzeit  $O(1)$  eine Festplatte für einen zu platzierenden Block auswählt. Die Initialisierung von *Share* umfasst die Platzierung der heterogenen Festplatten, die Initialisierung der uniformen Hashfunktionen für die Intervalle und die Initialisierung der Hashfunktion zur Auswahl eines Intervalls in Laufzeit  $O(1)$ . Als uniforme Hashfunktion habe ich die Implementierung von *Consistent Hashing* verwendet. Im Fall von *Redundant Share* umfasst die Initialisierung die Berechnung der in Abschnitt 3.4.3 beschriebenen  $\check{c}_i$  und ihrer Vielfachen, wie am Anfang dieses Kapitels beschrieben. Ferner ist die Berechnung der in Abschnitt 3.4.3 eingeführten Korrekturwerte  $\check{c}_{e_i}^*$  Teil der Initialisierung, in den verwendeten Beispielen war dies allerdings auf Grund des Aufbaus der Konfigurationen nicht notwendig. Bei *Fast Redundant Share* umfasst die Initialisierung neben den schon für *Redundant Share* gezeigten Schritten die Vorbereitung der nicht redundanten Hashfunktionen. Als nicht redundante Hashfunktionen habe ich die Implementierung von *Share* verwendet.









Nach der Initialisierung der Verfahren habe ich den Hauptspeicherverbrauch des jeweiligen Prozesses gespeichert. Dazu habe ich die Prozess ID (PID) des Prozesses ermittelt. Danach habe ich die virtuelle Datei `/proc/PID/status` gesichert. Diese Datei enthält verschiedene Statusinformationen über einen Prozess, unter anderem den aktuell verbrauchten Hauptspeicher und den maximal verwendeten Hauptspeicher. Der Eintrag `VmRSS` hält den aktuellen vor. `VmPeak` enthält den maximalen Hauptspeicherverbrauch des Prozesses. Im Gegensatz zu `VmRSS` beinhaltet `VmPeak` auch den Hauptspeicherverbrauch des Codes von Bibliotheken, welche sich der Prozess mit anderen Prozessen teilen kann<sup>3</sup>.

Nach der Initialisierung habe ich in jedem Test 1.000.000 Blöcke platziert. Für die Platzierung jedes Blocks  $b_i$  habe ich die Zeit  $t_i$  gemessen, die zur Platzierung des Blocks benötigt wurde. Ich habe  $T$  als Summe dieser Zeiten und  $Q$  als Summe der Quadrate der Zeiten gespeichert:

<sup>3</sup>Solche Bibliotheken sind als *Shared Libraries* bekannt.

## 4 Evaluation der Strategien

Tabelle 4.1: Legende der Diagramme zur Messung der Laufzeit und des Hauptspeicherverbrauchs

	VmRSS	VmPeak	$\frac{T}{m}$
eine Kopie			$\diamond$
zwei Kopien			$\diamond$
vier Kopien			$\diamond$
acht Kopien			$\diamond$

$$T = \sum_{i=1}^m t_i$$

$$Q = \sum_{i=1}^m (t_i)^2$$

Auf diese Weise musste ich lediglich die Summen der Zeiten zur Platzierung und ihrer Quadrate speichern, um mittels folgender Formel die Varianz  $\sigma^2$  und daraus die Standardabweichung  $\sigma$  zu berechnen:

$$\sigma^2 = \frac{m \cdot Q - m \cdot T^2}{n \cdot (n - 1)}$$

$$\sigma = \sqrt{\sigma^2}$$

Ich habe in unterschiedlichen Testläufen  $k \in \{1, 2, 4, 8\}$  Kopien jedes Blocks platziert und in Diagrammen die Werte VmRSS für den aktuellen Hauptspeicherverbrauch sowie VmPeak für den maximalen Hauptspeicherverbrauch während der Initialisierung eingezeichnet. Ferner habe ich die durchschnittlich zur Platzierung eines Blocks verbrauchte Zeit  $\frac{T}{m}$  sowie die Standardabweichung eingezeichnet. Wenn ich im Folgenden von der Laufzeit eines Verteilers zur Platzierung eines Blocks spreche, so meine ich immer die durchschnittliche Laufzeit  $\frac{T}{m}$ , wie sie auch in den Abbildungen zu sehen ist. Tabelle 4.1 zeigt die Legende der Diagramme.

### 4.1.1 Homogene Konfigurationen

In diesem Abschnitt untersuche ich den Hauptspeicherverbrauch und die Laufzeit für homogenen Konfigurationen. Dazu verwende ich Konfigurationen bestehend aus  $n = 2^v$  Festplatten mit  $v \in \{3, \dots, 13\}$ , also acht bis 8192 Festplatten. Jede Festplatte hat eine Kapazität von 500.000 Blöcken. Für die unterschiedlichen Messungen initialisiere ich die Verteilungsalgorithmen zur

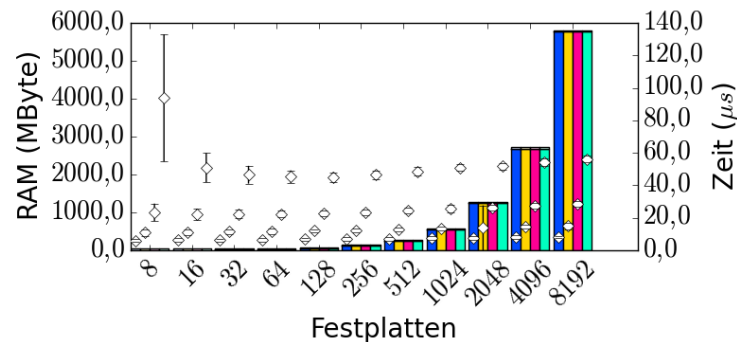


Abbildung 4.1: Hauptspeicherverbrauch und Laufzeit von *Consistent Hashing* bei unterschiedlicher Anzahl homogener Festplatten

Platzierung von  $k \in \{1, 2, 4, 8\}$  Kopien jedes Blocks und messe den Hauptspeicherverbrauch wie zu Anfang dieses Abschnitts beschrieben. Danach platziere ich 1.000.000 Blöcke und messe die durchschnittliche Platzierungszeit und die Standardabweichung.

## Consistent Hashing

Der Hauptspeicherverbrauch von *Consistent Hashing* hängt von der Anzahl der Festplatten in der Konfiguration und der Anzahl der platzierten Kopien jeder Festplatte ab. In Abbildung 4.1 ist der Hauptspeicherverbrauch und die Laufzeit von *Consistent Hashing* bei Platzierung von  $400 \cdot n \cdot \lceil \log n \rceil$  Kopien jedes Blocks zu sehen.

Der Hauptspeicherverbrauch von *Consistent Hashing* stieg mit der Anzahl der Festplatten in der jeweiligen Konfiguration. Bei der nicht redundanten Platzierung stieg der Hauptspeicherverbrauch nach der Initialisierung von 5,94 MByte bei acht Festplatten auf 5,64 GByte bei 8192 Festplatten. Bis zu einer Konfiguration mit 64 Festplatten war das Wachstum des Hauptspeicherverbrauchs sublinear, hier überwog der statische Hauptspeicherverbrauch. Ab dieser Größe der Konfiguration ist das von Karger u. a. in [Karger u. a., 1997] gezeigte Wachstum von  $O(n \cdot \log n)$  zu sehen.

Für die Platzierung mehrerer Kopien verwende ich die gleiche Hashfunktion. Ich erzeuge lediglich mehrere unabhängige Abbildungen jedes Blocks bis ich  $k$  unterschiedliche Festplatten erhalten habe. Daher ist der Hauptspeicherverbrauch unabhängig von der Anzahl der zu platzierenden Kopien.

VmPeak lag in allen Konfigurationen um ca. 33 MByte über VmRSS. Ich habe die Verteiler in einer eigenen Bibliothek implementiert, welche dynamisch eingebunden werden kann. Die 33 MByte Unterschied ergeben sich aus den Anforderungen für den ausführbaren Code dieser Bibliothek.

#### 4 Evaluation der Strategien

Die nicht redundante Platzierung eines Blocks über acht Festplatten benötigte  $5,92 \mu\text{s}$  bei einer Standardabweichung von  $0,82$ . Zur Platzierung eines Blocks über 8192 Festplatten benötigte *Consistent Hashing*  $8,17 \mu\text{s}$  bei einer Standardabweichung von  $0,76$ . Dabei stieg die benötigte Laufzeit in etwa logarithmisch gegenüber der Anzahl der Festplatten. Bei jeder Verdoppelung der Anzahl der Festplatten stieg die Laufzeit um ca.  $0,2 \mu\text{s}$ .

Dieser Anstieg erklärt sich durch den erhöhten Hauptspeicherverbrauch. Bei Verwendung von acht Festplatten passt ein großer Teil der Daten des Prozesses in den ein MByte großen Cache des Prozessors. Je mehr Hauptspeicher verwendet wird, desto häufiger müssen Daten zwischen dem Prozessor und dem RAM ausgetauscht werden, wodurch die Laufzeit beeinträchtigt wird.

Die Platzierung mehrerer Kopien eines Blocks realisiere ich durch mehrere unabhängige Experimente. Dies resultiert natürlich in einem Anstieg der Laufzeit. In meinen Tests stieg die Laufzeit bei der Platzierung von zwei Kopien monoton von  $10,84 \mu\text{s}$  bei einer Standardabweichung von  $1,86$  auf  $14,94 \mu\text{s}$  bei einer Standardabweichung von  $1,92$  bei 8192 Festplatten.

Die Platzierung von vier Kopien über acht Festplatten benötigte  $23,32 \mu\text{s}$  bei einer Standardabweichung von  $5,43$ . Bei 16 verwendeten Festplatten fiel die Laufzeit auf  $22,00 \mu\text{s}$  mit einer Standardabweichung von  $3,45$  und stieg dann bis auf  $28,65 \mu\text{s}$  bei einer Standardabweichung von  $1,55$  bei 8192 Festplatten.

Bei der Platzierung von acht Kopien lag die Laufzeit bei acht verwendeten Festplatten bei  $93,79 \mu\text{s}$  bei einer Standardabweichung von  $39,16$ . Diese benötigte Laufzeit fiel bis zu einer Konfiguration mit 128 Festplatten auf  $45,13 \mu\text{s}$  bei einer Standardabweichung von  $2,89$ . Bei größeren Konfigurationen stieg die benötigte Laufzeit dann wieder und lag bei 8192 Festplatten bei  $56,28 \mu\text{s}$  bei einer Standardabweichung von  $2,29$ .

Die Laufzeit zur Platzierung mehrerer Kopien bestimmt sich in erster Linie durch die Anzahl der Experimente. Dabei durchläuft die Implementierung eine Schleife. In jedem Schritt wird eine unabhängige Abbildung des Blocks mittels *Consistent Hashing* erzeugt. Danach wird überprüft, ob die erhaltene Festplatte bereits Teil der Ergebnismenge ist. Falls nicht, wird sie hinzugefügt, andernfalls wird das Experiment verworfen. Diese Schleife endet, wenn  $k$  Kopien gefunden wurden.

Die Laufzeit meiner Modifizierung von *Consistent Hashing* zur Platzierung mehrerer Kopien hängt somit von der Anzahl der benötigten Experimente ab, um die  $k$  unterschiedliche Festplatten zu finden. Liegt die Anzahl der Festplatten und die Anzahl der Kopien nahe beieinander, so ist die Wahrscheinlichkeit groß, dass die selbe Festplatte mehrmals gefunden wird. Dies erhöht die erwartete Anzahl an Experimenten. Daher ergibt sich die erhöhte Laufzeit und die erhöhte Standardabweichung bei der Platzierung vieler Kopien über wenigen Festplatten. Vernachlässige ich diese Werte, so benötigt meine Implementierung zur Platzierung von  $k$  Kopien ungefähr die  $k$ -fache Zeit gegenüber der nicht redundanten Platzierung bei gleicher Konfiguration.

Der Hauptspeicherverbrauch von *Consistent Hashing* ist nicht nur abhängig von der Anzahl der verwendeten Festplatten, sondern auch von der Anzahl der Kopien, die von jeder Festplatte



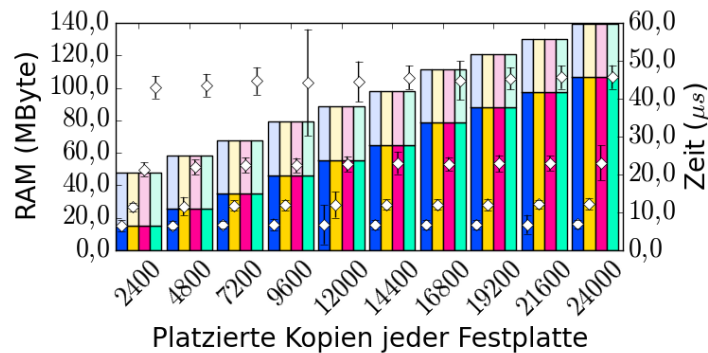


Abbildung 4.2: Hauptspeicherverbrauch und Laufzeit von *Consistent Hashing* bei unterschiedlicher Anzahl platzierter Kopien jeder Festplatte

platziert werden. Abbildung 4.2 zeigt, wie sich der Speicherverbrauch von *Consistent Hashing* bei Platzierung einer steigenden Anzahl von Kopien verhält. Dazu habe ich eine Konfiguration mit 256 homogenen Festplatten gewählt. Für  $i \in \{1, \dots, 10\}$  habe ich von jeder Festplatte  $i \cdot 300 \cdot \lceil \log 256 \rceil = i \cdot 2400$  Kopien platziert. Ich habe wie zuvor den Speicherverbrauch und die Laufzeit vermessen.

Der Speicherverbrauch in dieser Parametrisierung stieg linear mit der Anzahl der platzierten Kopien jeder Festplatte an. Unabhängig von der Anzahl der zu platzierenden Kopien der Blöcke benötigte die Implementierung von *Consistent Hashing* bei Platzierung von 2400 Kopien jeder Festplatte 14,98 MByte. Für 4800 Kopien jeder Festplatte stieg der Hauptspeicherverbrauch um etwa 10,50 MByte auf 25,40 MByte. Mit im Schnitt 10,20 MByte pro 2400 Kopien jeder Festplatte stieg der Wert weiter bis auf 106,86 MByte bei 24.000 Kopien jeder Festplatte.

Die Laufzeit stieg wieder mit der Menge des verwendeten Hauptspeichers. Für die nicht redundante Platzierung bei Verwendung von 2400 Kopien jeder Festplatte benötigte meine Implementierung 6,45  $\mu$ s bei einer Standardabweichung von 1,49. Bei Verwendung von 24.000 Kopien jeder Festplatte stieg der Wert auf 6,89  $\mu$ s bei einer Standardabweichung von 1,19. Die Platzierung von  $k$  Kopien jedes Blocks benötigte wieder in etwa die  $k$ -fache Zeit der nicht redundanten Platzierung.

## Share

Wie in Abschnitt 3.2 beschrieben hängt der Hauptspeicherverbrauch von *Share* von der Anzahl der Festplatten in der verwendeten Konfiguration und dem gewählten Dehnungsfaktor  $s$  ab. Ich habe *Share* mit einem Dehnungsfaktor von  $5 \cdot \log n$  vermessen, wobei  $n$  die Anzahl der Festplatten in der jeweils verwendeten Konfiguration ist. Die unterliegenden *Consistent Hashing* Verteilungen platzierten  $400 \cdot n'$  Kopien jedes Abbilds einer Festplatte. Dabei ist  $n'$  die Menge

#### 4 Evaluation der Strategien

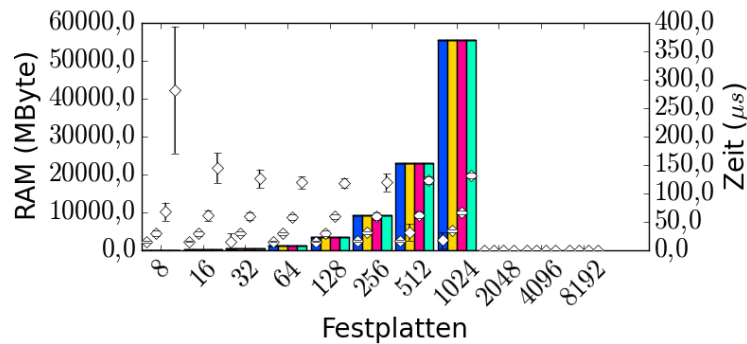


Abbildung 4.3: Hauptspeicherverbrauch und Laufzeit von *Share* bei unterschiedlicher Anzahl homogener Festplatten

der Abbilder von Festplatten, welche der jeweilige Verteiler als Eingabe erhielt. Die Ergebnisse sind in Abbildung 4.3 dargestellt.

Bei der nicht redundanten Platzierung über acht Festplatten benötigte der Prozess nach der Initialisierung 47,04 MByte. Dieser Wert stieg auf 54,19 GByte bei einer Konfiguration mit 1024 Festplatten. Größere Konfigurationen konnte ich nicht vermessen, da die 64 GByte Hauptspeicher der verwendeten Rechner dazu nicht ausreichten.

Brinkmann u. a. haben in [Brinkmann u. a., 2002] für *Share* einen Speicherverbrauch von  $O(s \cdot n)$  gezeigt, wobei der Speicherbedarf der homogenen Verteiler außen vor gelassen wurde.  $s$  ist dabei der gewählte Dehnungsfaktor. Salzwedel hat in [Salzwedel, 2004] den Speicherbrauch mit  $O(s \cdot n \cdot h)$  inklusive der homogenen Verteiler gezeigt. Dabei hat er einen Speicherverbrauch von  $h$  für jeden homogenen Verteiler angenommen. In der Realität hängt  $h$  von  $s$  ab. Je größer der Dehnungsfaktor  $s$  desto mehr Abbilder der  $n$  Festplatten gehen in die homogenen Verteiler ein. Da  $s$  in meinen Tests sublinear vergrößert wurde, stieg auch  $h$  sublinear an. Gleichzeitig wuchs aber die Anzahl der verwendeten homogenen Verteiler linear.

Der Hauptspeicherverbrauch meiner Implementierung von *Share* ist unabhängig von der Anzahl der zu platzierenden Kopien. *Share* beachtet diese ebenso wenig wie *Consistent Hashing*. Mehrere Kopien werden durch unabhängige Experiment gegenüber der gleichen Hashfunktion realisiert.

Wie bereits zuvor *Consistent Hashing* hatte auch *Share* einen um 33 MByte höheren Wert für  $VmPeak$  gegenüber  $VmRSS$  in allen Messungen. Dieser Speicher wird wieder durch die dynamische Bibliothek belegt.

Die Laufzeit von *Share* bei Platzierung ohne Redundanz über acht Festplatten liegt bei 15,01  $\mu s$  bei einer Standardabweichung von 0,98. Die Laufzeit steigt bei zunehmender Anzahl an Festplatten auf 18,02  $\mu s$  mit einer Standardabweichung von 13,94. Dieses Wachstum der Laufzeit erklärt sich wieder durch den Anstieg des Hauptspeicherverbrauchs.

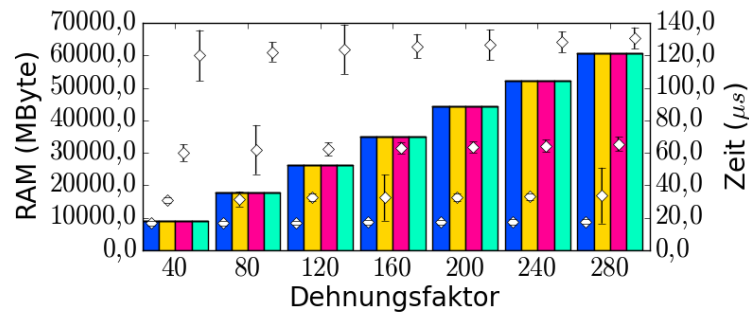


Abbildung 4.4: Hauptspeicherverbrauch und Laufzeit von *Share* bei unterschiedlichem Dehnungsfaktor

Die Laufzeit zur Platzierung von zwei Kopien über acht Festplatten lag bei  $29,96 \mu\text{s}$  bei einer Standardabweichung von  $5,15$ . Diese Laufzeit sank bis auf  $29,51 \mu\text{s}$  mit einer Standardabweichung von  $3,21$  bei der Verteilung über 32 Festplatten. Bis zu einer Konfiguration mit 1024 Festplatten stieg die Laufzeit danach auf  $34,40 \mu\text{s}$  bei einer Standardabweichung von  $2,30$ .

Zur Platzierung von vier Kopien über acht Festplatten benötigte die Implementierung  $67,95 \mu\text{s}$  bei einer Standardabweichung von  $15,81$ . Die Laufzeit sank bis auf  $58,18 \mu\text{s}$  mit einer Standardabweichung von  $4,79$  bei Verwendung von 64 Festplatten. Danach stieg die Laufzeit auf  $66,83 \mu\text{s}$  bei einer Standardabweichung von  $3,19$  für 1024 Festplatten.

Die Platzierung von acht Kopien über acht Festplatten benötigte  $281,94 \mu\text{s}$  bei einer Standardabweichung von  $111,57$ . Bis zu der Konfiguration mit 128 Festplatten fiel die Laufzeit auf  $118,58 \mu\text{s}$  bei einer Standardabweichung von  $7,85$ . Danach stieg die Laufzeit auf  $132,40 \mu\text{s}$  bei einer Standardabweichung von  $5,09$  bei der Platzierung über 1024 Festplatten.

Die redundante Platzierung mittels *Share* habe ist auf ähnliche Weise implementiert, wie die redundante Verteilung mittels *Consistent Hashing*. Für jeden Block werden so lange unabhängige Experimente durchgeführt, bis  $k$  unterschiedliche Festplatten gefunden wurden. Dadurch steigt auch die Laufzeit von *Share*, wenn  $n$  und  $k$  nahe beieinander liegen. Wie die Ergebnisse der Messungen zeigen, wirkt sich dies allerdings bei *Share* stärker aus, als bei *Consistent Hashing*, was sich auch sehr gut an den teilweise extrem großen Standardabweichungen erkennen lässt. Während diese bei konstanter Anzahl an Kopien mit zunehmender Anzahl Festplatten in den Konfigurationen immer geringer werden, gibt es einige Ausreißer, beispielsweise bei der Platzierung von acht Kopien über 256 Festplatten. Davon unabhängig nähert sich die benötigte Zeit zur Platzierung von  $k$  Kopien sehr an das  $k$ -fache der Zeit für die nicht redundanten Platzierung an.

Der Hauptspeicherverbrauch und die Verteilungsgüte von *Share* hängen von dem gewählten Dehnungsfaktor ab. Daher habe ich *Share* mit einer festen Konfiguration aus 256 homogenen

## 4 Evaluation der Strategien

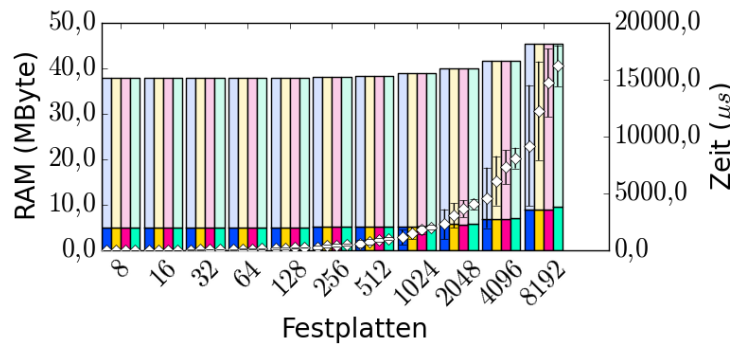


Abbildung 4.5: Hauptspeicherverbrauch und Laufzeit von *Redundant Share* bei unterschiedlicher Anzahl homogener Festplatten

Festplatten und unterschiedlichen Dehnungsfaktoren vermessen. Bisher habe ich einen Dehnungsfaktor von  $5 \cdot \log n$  verwendet. Bei 256 Festplatten ergibt dies einen Dehnungsfaktor von 40. In meinen Tests habe ich Vielfache dieses Werts vermessen, bis zu einem Dehnungsfaktor von  $35 \cdot \log 256 = 280$ . Abbildung 4.4 zeigt die Ergebnisse.

Unabhängig von der Anzahl der zu platzierenden Kopien stieg der Hauptspeicherverbrauch linear mit dem Dehnungsfaktor. Bei einem Dehnungsfaktor von 40 konnte ich einen Hauptspeicherverbrauch von 8,94 GByte nach der Initialisierung messen. Bei einem Dehnungsfaktor von 80 hat sich dieser Wert nahezu verdoppelt auf 17,42 GByte. Bei einem Dehnungsfaktor von 280 war mit einem Hauptspeicherverbrauch von 59,12 GByte der RAM des Rechners saturiert.

Wie zuvor wuchs die Laufzeit zur nicht redundanten Platzierung mit steigendem Hauptspeicherverbrauch leicht an. Bei einem Dehnungsfaktor von 40 benötigte der Prozess durchschnittlich  $16,83 \mu s$  mit einer Standardabweichung von 1,38 pro Block. Dies stieg bis auf  $17,71 \mu s$  bei einem Dehnungsfaktor von 280.

Die Platzierung von  $k$  Kopien benötigte in etwa die  $k$ -fache Zeit der nicht redundanten Platzierung. Für einen Dehnungsfaktor von 280 stieg die Platzierungszeit für  $k = 2$  auf  $33,65 \mu s$  bei einer Standardabweichung von 17,16. Für  $k = 4$  wurden  $65,56 \mu s$  bei einer Standardabweichung von 4,38 benötigt. Die Platzierung von acht Kopien eines Blocks benötigte  $130,61 \mu s$  bei einer Standardabweichung von 6,60.

### Redundant Share

Der Hauptspeicherverbrauch von *Redundant Share* ist wesentlich geringer, als bei *Share* und *Consistent Hashing*, wie Abbildung 4.5 zeigt. Zur nicht redundanten Platzierung von Blöcken über einer Konfiguration aus acht Festplatten benötigte die Implementierung von *Redundant Share* lediglich 4,95 MByte. Durch Verdoppelung der Anzahl der Festplatten auf 16 erhöhte

sich der Hauptspeicherverbrauch um 8 KByte, eine weitere Verdoppelung erhöhte den Hauptspeicherverbrauch um weitere 8 KByte. Der Hauptspeicherverbrauch stieg bis auf 9,03 MByte für die Verteilung über 8192 Festplatten. Daran ist sehr gut zu erkennen, dass der Hauptspeicherverbrauch hauptsächlich durch einen statischen Anteil geprägt ist, welcher nicht von der Anzahl der verwendeten Festplatten abhängt.

Im Gegensatz zu *Share* und *Consistent Hashing* ist der Hauptspeicherverbrauch von *Redundant Share* auch abhängig von der Anzahl der zu platzierenden Kopien jedes Blocks. Meine Implementierung hat einen Speicherbedarf von  $O(k \cdot n)$ , da ich, wie in der Einleitung dieses Kapitels erwähnt, für jede Festplatte  $k$  Werte für die Zufallsexperimente speichere. Damit werden nur sehr wenig Daten zusätzlich pro Kopie vorrätig gehalten. Erst ab 1024 Festplatten ist ein Unterschied in den Messergebnissen zu erkennen. Zur nicht redundanten Platzierung belegte der Prozess hier 5304 KByte. Zur Platzierung von acht Kopien benötigte er mit 5352 KByte 48 KByte mehr. Bei Verwendung von 8192 Festplatten stieg der Hauptspeicherverbrauch meiner Implementierung auf 9,6 MByte zur Platzierung von acht Kopien, also um ca. 0,57 MByte gegenüber der nicht redundanten Platzierung.

Die benötigte Laufzeit zur Platzierung von Blöcken ist bei *Redundant Share* wesentlich höher, als bei *Consistent Hashing* und *Share*. Wie in Abschnitt 3.4.3 gezeigt, hat *Redundant Share* eine lineare Laufzeit gegenüber der Anzahl der Festplatten. Im schlimmsten Fall muss für jede Festplatte ein Zufallsexperiment durchgeführt werden. Abbildung 4.5 belegt dieses Wachstum auch in der Praxis. Zur nicht redundanten Platzierung eines Blocks über acht Festplatten brauchte meine Implementierung von *Redundant Share* 12,57  $\mu$ s bei einer Standardabweichung von 5,11. Durch Verdoppelung der Anzahl der Festplatten verdoppelte sich auch die benötigte Zeit zur nicht redundanten Platzierung und erreichte bei Verwendung von 8192 Festplatten einen Wert von 9207,19  $\mu$ s bei einer Standardabweichung von 5321,92.

Die Laufzeit von *Redundant Share* stieg mit der Anzahl der platzierten Kopien jedes Blocks. Die Platzierung von acht Kopien über acht Festplatten benötigte 22,36  $\mu$ s bei einer Standardabweichung von 5,17. Dieser Wert stieg auf 16223,96  $\mu$ s bei einer Standardabweichung von 1815,31 bei der Verteilung über 8192 Festplatten.

Ich habe im Abschnitt 3.4.3 gezeigt, dass *Redundant Share* im schlimmsten Fall ein Experiment für jede Festplatte machen muss, die Laufzeit also  $O(n)$  ist. Dabei bin ich davon ausgegangen, dass für jede Festplatte ein Experiment durchgeführt werden muss. Im erwarteten Fall ist die Anzahl der durchzuführenden Experimente geringer, nähert sich mit steigender Anzahl zu platzierender Kopien allerdings  $n$  an.

### Fast Redundant Share

Meine Implementierung von *Fast Redundant Share* benutzt *Share* als nicht redundanten Verteiler mit konstanter Laufzeit. Dabei habe ich einen Dehnungsfaktor von  $s = 3 \cdot \log n$  verwendet.

#### 4 Evaluation der Strategien

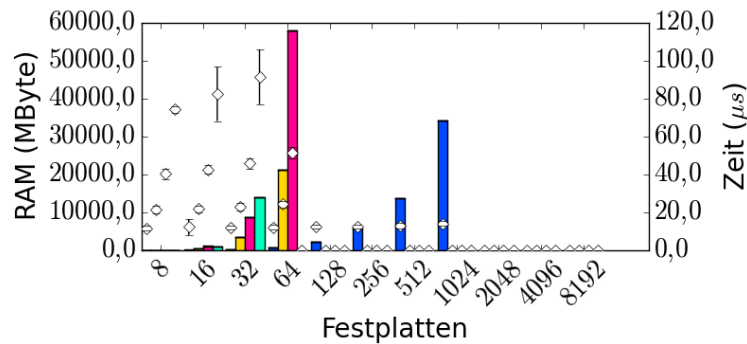


Abbildung 4.6: Hauptspeicherverbrauch und Laufzeit von *Fast Redundant Share* bei unterschiedlicher Anzahl homogener Festplatten

Die Ergebnisse der Vermessung der Laufzeit und des Speicherverbrauchs sind in Abbildung 4.6 dargestellt.

Bei der nicht redundanten Platzierung benötigte *Fast Redundant Share* zur Verteilung von Daten über acht Festplatten 31,62 MByte Hauptspeicher. Dieser Bedarf stieg kontinuierlich auf 33,40 GByte bei der Verteilung über 1024 Festplatten. Verteiler für größere Systeme konnten auf Grund des auf 64 GByte begrenzten Hauptspeichers der verwendeten Rechner nicht durchgeführt werden.

Der Hauptspeicherverbrauch stieg mit der Anzahl der zu platzierenden Kopien. Eine Initialisierung des Verteilers für zwei und vier Kopien war mit bis zu 64 Festplatten möglich. Der Hauptspeicherverbrauch lag bei zwei Kopien bei 20,73 GByte und bei acht Kopien bei 56,70 GByte. Die Verteilung von acht Kopien konnte nur über Konfigurationen mit bis zu 32 Festplatten simuliert werden, der Hauptspeicherverbrauch lag bei 13,62 GByte.

Der maximal Hauptspeicherverbrauch lag in allen Konfigurationen ungefähr 33 MByte über dem Verbrauch nach der Initialisierung. Dies zeigt wieder den benötigten Hauptspeicher für die Bibliotheken und das ausführbare Programm.

Die Laufzeit stieg bei gleicher Anzahl platzierter Kopien leicht mit der verwendeten Menge an Festplatten. Bei der nicht redundanten Verteilung stieg die Laufzeit von 11,73  $\mu s$  mit einer Standardabweichung von 1,27 für acht Festplatten auf 14,04  $\mu s$  mit einer Standardabweichung von 1,27 für 1024 Festplatten.

Bei der Platzierung mehrerer Kopien stieg die Laufzeit proportional. Bei der Verteilung von zwei Kopien stieg die Laufzeit von 21,34  $\mu s$  bei einer Standardabweichung von 2,03 für acht Festplatten auf 24,36  $\mu s$  bei einer Standardabweichung von 1,86 für 64 Festplatten. Bei der Platzierung von vier Kopien stieg die Laufzeit von 40,35  $\mu s$  bei einer Standardabweichung von 2,67 für acht Festplatten auf 51,54  $\mu s$  bei einer Standardabweichung von 2,99 für 64 Festplatten. Bei der Platzierung von acht Kopien stieg die Laufzeit von 74,26  $\mu s$  bei einer Standardabweichung von 2,99 für 64 Festplatten.

chung von 1,53 für acht Festplatten auf 91,56  $\mu$ s bei einer Standardabweichung von 14,45 für 32 Festplatten.

Für die nicht redundante Verteilung erstellt *Fast Redundant Share* lediglich eine Hashfunktion. Somit sind die Ergebnisse in diesem Bereich äquivalent zu der Vermessung von *Share*. Durch die redundante Platzierung steigt die Anzahl der benötigten Hashfunktionen um  $O(k \cdot n)$ . Bereits ein einzelner Verteiler vom Typ *Share* hat einen nicht zu vernachlässigenden Hauptspeicherverbrauch, wie ich zuvor gezeigt habe. Der Hauptspeicherverbrauch von *Fast Redundant Share* ist damit so hoch, dass ich nur sehr kleine Konfigurationen vermessen könnte. Daher eignet sich dieses Verfahren nicht zur Verteilung von Daten in Speichernetzen. Ich werde auf weitere Vermessung dieses Verfahrens verzichten.

Die Laufzeit stieg bei konstanter Anzahl an Kopien durch die größere Menge an verwendeten Hauptspeicher. Dieses Problem habe ich bereits bei den vorherigen Verteilern beobachtet und beschrieben. Davon abgesehen wächst die benötigte Laufzeit von *Fast Redundant Share* wie erwartet mit der Anzahl der zu platzierenden Kopien.

### 4.1.2 Heterogene Konfigurationen

Werden Speichersysteme um neue Festplatten erweitert, so haben diese häufig nicht die gleiche Kapazität, wie die Festplatten der ursprünglichen Konfiguration. Auf diese Weise entstehen heterogene Speichersysteme, mit denen die Verteilungsfunktionen umgehen können müssen. Im den folgenden Tests vermesse ich meine Implementierungen von *Consistent Hashing*, *Share* und *Redundant Share* über solchen heterogenen Umgebungen.

Ich habe die Konfigurationen wie folgt aufgebaut: Im ersten Schritt habe ich 128 Festplatten mit einer Kapazität von 500.000 Blöcken verwendet und für  $k \in \{1, 2, 4, 8\}$  Kopien vermessen. Dieser Test ist identisch mit dem homogenen Test für 128 Festplatten. Danach habe ich in jedem Schritt 128 Festplatten hinzugefügt, alle mit einer 1,5-fachen Kapazität der im vorherigen Schritt hinzugefügten Festplatten. Ergab die Kapazität keine ganze Zahl, so habe ich abgerundet. Auf diese Weise habe ich Konfigurationen mit  $n = 128 \cdot i$  Festplatten für  $i \in \{1, \dots, 10\}$  erstellt und vermessen. Die Abbildungen folgen wieder der in Tabelle 4.1 dargestellten Legende.

#### Consistent Hashing

Die Ergebnisse der Vermessung von *Consistent Hashing* mit den angegebenen Konfigurationen zeigt Abbildung 4.7. Wie bereits im homogenen Fall habe ich  $400 \cdot \lceil \log n \rceil$  Kopien jeder Festplatte platziert.

Über einer Konfiguration von 128 Festplatten belegte der Prozess zur nicht redundanten Platzierung nach der Initialisierung 52,64 MByte. Der Hauptspeicherverbrauch stieg monoton und

## 4 Evaluation der Strategien

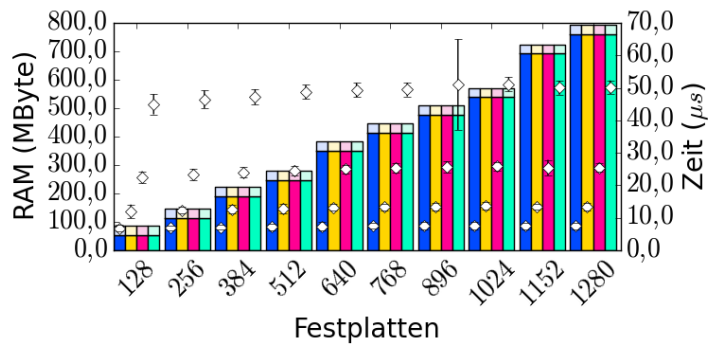


Abbildung 4.7: Hauptspeicherverbrauch und Laufzeit von *Consistent Hashing* bei unterschiedlicher Anzahl heterogener Festplatten

lag bei 1280 Festplatten bei 760,63 MByte. Der belegte Hauptspeicher für 128, 256, 512 und 1024 Festplatten stimmte exakt mit den Werten für homogene Konfigurationen überein.

Der Hauptspeicherverbrauch entwickelt sich wie erwartet gemäß der in [Karger u. a., 1997] gezeigten Schranke  $O(n \cdot \log n)$ . *Consistent Hashing* beachtet die Größe der Festplatten nicht, somit ergibt sich der gleiche Hauptspeicherverbrauch unabhängig der Kapazitäten der Festplatten. Auch die Anzahl der zu platzierenden Kopien hat keinen Einfluss, wie ich bereits in Abschnitt 4.1.1 erläutert habe.

Zur nicht redundanten Platzierung über 128 Festplatten brauchte meine Implementierung von *Consistent Hashing* 6,81  $\mu$ s bei einer Standardabweichung von 1,06. Diese Laufzeit stieg auf 7,45  $\mu$ s bei einer Standardabweichung von 1,03 bei der Platzierung über 1280 Festplatten.

Um zwei Kopien über 1280 Festplatten zu verteilen stieg die benötigte Laufzeit der Implementierung auf 13,42  $\mu$ s bei einer Standardabweichung von 1,03, für vier Kopien auf 25,52  $\mu$ s bei einer Standardabweichung von 1,39 und für acht Kopien auf 50,25  $\mu$ s bei einer Standardabweichung von 2,03.

Die Laufzeit entwickelte sich wie erwartet. Da sich in diesen Tests die Anzahl der verwendeten Festplatten und damit der belegte Speicher nicht mehr exponentiell sondern nur noch linear veränderte, war nur ein geringer Anstieg der Laufzeit im Vergleich zur nicht redundanten Platzierung zu erkennen. Die Implementierung benötigt zur Platzierung von  $k$  Kopien ungefähr die  $k$ -fache Zeit gegenüber der nicht redundanten Platzierung.

Die hier vermessene Parametrisierung meiner Implementierung von *Consistent Hashing* behandelt alle Festplatten gleich und unterscheidet nicht nach ihrer Kapazität. Damit entspricht sie dem von Karger u. a. in [Karger u. a., 1997] vorgestellten Verfahren. Wie ich später in Abschnitt 4.2.3 noch in praktischen Tests belege, wird so keine faire Verteilung gemäß der Kapazitäten der Festplatten erreicht. Daher habe ich *Consistent Hashing* um die Möglichkeit erweitert, gemäß der Kapazitäten der Festplatten eine unterschiedliche Anzahl an Kopien zu platzieren. Bei



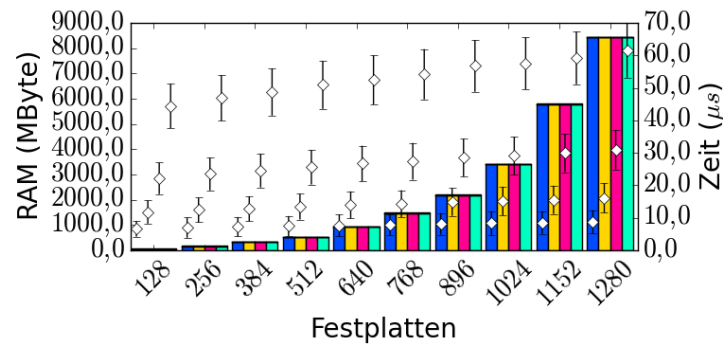


Abbildung 4.8: Hauptspeicherverbrauch und Laufzeit von *Consistent Hashing* bei unterschiedlicher Anzahl heterogener Festplatten und kapazitätsabhängiger Anzahl platzierter Kopien jeder Festplatte

gegebenen Kapazitäten  $c_i$  lege ich dann von einer Festplatte  $d_i$  mit  $i \in \{1, \dots, n\}$   $\alpha_i^{\text{het}}$  Kopien gemäß folgender Formel an:

$$\alpha_i^{\text{het}} = \left\lceil \frac{c_i}{100.000} \right\rceil \cdot 80 \cdot \lceil \log n \rceil$$

Im vorherigen Test wurden von jeder Festplatte  $d_i$  genau  $\alpha_i^{\text{hom}} = 400 \cdot \lceil \log n \rceil$  Kopien abgelegt. Da die ersten 128 Festplatten eine Kapazität von 500.000 Blöcken haben, wird von jeder exakt die gleiche Anzahl an Kopien platziert wie zuvor. Alle Festplatten mit einer größeren Kapazität erhalten entsprechend mehr Kopien. Abbildung 4.8 zeigt den Hauptspeicherverbrauch und die Laufzeit meiner Implementierung im beschriebenen Fall.

Der Hauptspeicherverbrauch nach der Initialisierung und die Laufzeit stimmen für 128 Festplatten mit den vorherigen Messungen überein. Für die nicht redundante Platzierung über 256 Festplatten wurden nun 150,96 MByte Hauptspeicher benötigt. Für 1280 Festplatten erreichte der Hauptspeicherverbrauch 8,22 GByte. Der Hauptspeicherverbrauch blieb unabhängig von der Anzahl der zu platzierenden Kopien.

Der benötigte Hauptspeicher nach der Initialisierung stieg nun wesentlich stärker, da in jedem Schritt von den 128 hinzukommenden Festplatten auch eine wachsende Anzahl an Kopien platziert wurde. Die in Schritt  $s$  hinzukommenden Festplatten hatten in etwa eine Kapazität  $1,5^s$  der im ersten Schritt benutzten Festplatten, genauer von  $\lceil 1,5^s \cdot 500.000 \rceil$ . Die Kapazität wuchs also exponentiell. Im gleichen Maß stieg auch der Speicherverbrauch.

Um einen Block nicht redundant über 256 Festplatten zu platzieren brauchte meine Implementierung nun 6,96  $\mu\text{s}$  bei einer Standardabweichung von 3,11. Für die nicht redundante Platzierung über 1280 Festplatten benötigte die Implementierung 8,80  $\mu\text{s}$  bei einer Standardabweichung von 3,54.

## 4 Evaluation der Strategien

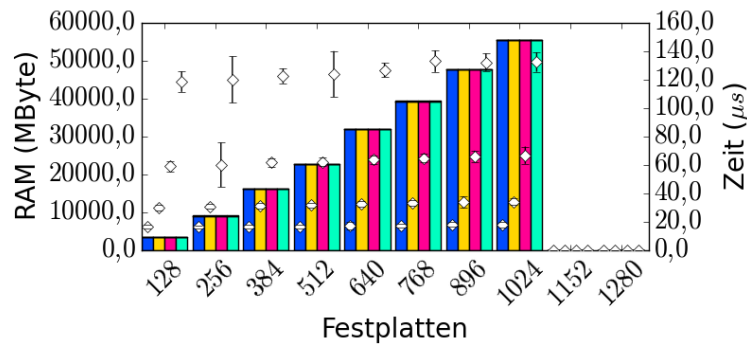


Abbildung 4.9: Hauptspeicherverbrauch und Laufzeit von *Share* bei unterschiedlicher Anzahl heterogener Festplatten

Zur Platzierung von zwei Kopien eines Blocks benötigt die Implementierung  $16,11 \mu s$  bei einer Standardabweichung von  $4,58$ . Für vier Kopien steigt die benötigte Zeit auf  $30,96 \mu s$  bei einer Standardabweichung von  $6,09$  und für acht Kopien auf  $61,53 \mu s$  bei einer Standardabweichung von  $8,35$ .

### Share

Wie im homogenen Fall habe ich *Share* mit einem Dehnungsfaktor von  $5 \cdot \log n$  vermessen. Die Ergebnisse sind in Abbildung 4.9 dargestellt.

Zur Platzierung über 128 Festplatten wurden  $3,35$  GByte Hauptspeicher durch den Prozess belegt. Dieser Wert deckt sich exakt mit dem entsprechenden Test über einer homogenen Konfiguration. Zur Platzierung über 256 Festplatten wurden mit  $8,89$  GByte etwa  $50$  MByte weniger Hauptspeicher benötigt, als bei der homogenen Konfiguration über 256 Festplatten. Für 1024 Festplatten wurden mit  $54,21$  GByte ca.  $20$  MByte mehr benötigt als bei dem entsprechenden homogenen Test. Testläufe mit mehr als 1024 Festplatten waren auf Grund des auf  $64$  GByte begrenzten Hauptspeichers der verwendeten Rechner nicht möglich. Der belegte Hauptspeicher war weiterhin unabhängig von der Anzahl der zu platzierenden Kopien.

Zur nicht redundanten Platzierung eines Blocks über 128 Festplatten wurden  $11,77 \mu s$  bei einer Standardabweichung von  $1,05$  benötigt. Für 1024 Festplatten stieg die Laufzeit auf  $13,62 \mu s$  bei einer Standardabweichung von  $2,55$ . Die benötigte Zeit zur Platzierung mehrerer Kopien entwickelt sich äquivalent.

Somit sind der Hauptspeicherverbrauch und die Laufzeit von *Share* bei den Messungen über homogenen und heterogenen Konfigurationen annähernd identisch.

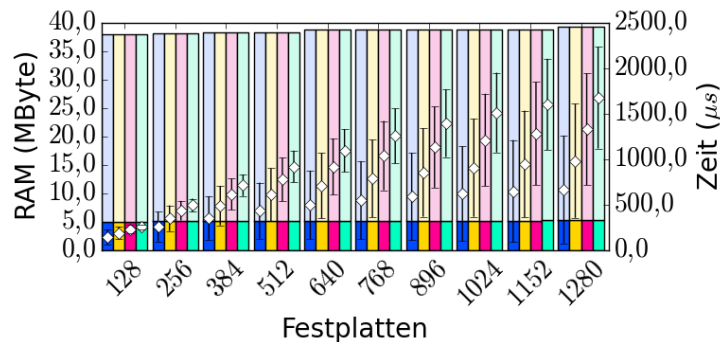


Abbildung 4.10: Hauptspeicherverbrauch und Laufzeit von *Redundant Share* bei unterschiedlicher Anzahl heterogener Festplatten

### Redundant Share

Abbildung 4.10 zeigt den Hauptspeicherverbrauch und die Laufzeit von *Redundant Share* bei Verwendung der heterogenen Konfigurationen.

Über 128 Festplatten verbrauchte die Implementierung von *Redundant Share* 5176 KByte zur Platzierung ohne Redundanz. Für größere Konfigurationen stieg der Hauptspeicherverbrauch und lag bei 1280 Festplatten bei 5432 KByte. Im Schnitt stieg der Speicherbedarf in jedem Schritt um 28,44 KByte.









Der Hauptspeicherverbrauch von *Redundant Share* ist unabhängig von der Heterogenität der Festplatten. Daher ist wie bei den homogenen Tests ein linearer Anstieg des benötigten Hauptspeichers zu sehen, welcher allerdings durch den statischen Anteil des Prozesses dominiert wird.

Der zusätzlich pro Kopie benötigte Hauptspeicher ist mit den verwendeten Konfigurationen kaum erkennbar. Zur Platzierung von acht Kopien über 1280 Festplatten benötigt die Implementierung mit 5492 KByte genau 60 KByte mehr Hauptspeicher als zur nicht redundanten Platzierung.

Zur nicht redundanten Platzierung über 256 Festplatten ist die Verteilung über die heterogene Konfiguration mit durchschnittlich 260,31  $\mu s$  bei einer Standardabweichung von 162,53 in etwa 26,5  $\mu s$  schneller als über der homogene Konfiguration. Über 1024 heterogenen Festplatten konnte meine Implementierung nicht redundant Blöcke in 621,89  $\mu s$  mit einer Standardabweichung von 522,57 platzieren. Bei dem Test über die gleiche Anzahl homogener Festplatten benötigte die Implementierung dazu mit 1139,02  $\mu s$  fast doppelt so lange.

Die geringere Laufzeit begründet sich in der Art, wie mittels *Redundant Share* Daten platziert werden. Sortiert nach ihrer Größe wird für jede Festplatte ein Experiment durchgeführt, bis alle Kopien platziert wurden. Dabei gelingt das Experiment für frühe Festplatten häufiger, wenn

Tabelle 4.2: Legende der Diagramme zur Messung der Fairness

	mininum	maximum
eine Kopie		
zwei Kopien		
vier Kopien		
acht Kopien		

diese einen höheren Anteil an der Gesamtkapazität tragen. Somit erreicht *Redundant Share* bei fester Anzahl an Festplatten eine bessere Laufzeit über heterogenen Konfigurationen als über homogenen Konfigurationen.

## 4.2 Fairness

In diesem Abschnitt betrachte ich, wie gut die verschiedenen Verfahren die Last über die Festplatten verteilen. Hierzu zeige ich, wie sich die Verfahren bei der Platzierung von einer unterschiedlichen Menge von Blöcken pro Festplatte verhalten. Danach betrachte ich, wie sich die Verteilung bei Konfigurationen mit unterschiedlicher Anzahl an Festplatten verhält.

In jedem Test habe ich untersucht, wie sich das Speichersystem bei der Platzierung von  $k \in \{1, 2, 4, 8\}$  Kopien von  $m$  Blöcken verhält. Für jede Festplatte habe ich gezählt, wie viele Kopien eines Blocks sie erhalten hat. Eine Verteilung, welche absolut fair gemäß der Kapazitäten der Festplatten arbeitet, müsste einer beliebigen Festplatte  $d_i$  die folgende Anzahl an Kopien von Blöcken zuweisen:

$$\text{perf}_i = \frac{c_i}{C} \cdot m \cdot k$$

Zu jeder Festplatte berechne ich den Faktor, um den ihre Last von der perfekten Verteilung entfernt ist. Mit  $l_i$  als die auf  $d_i$  platzierte Last habe ich dazu folgende Formel verwendet:

$$\text{faktor}_i = \frac{l_i}{\text{perf}_i}$$

In den folgenden Diagrammen habe für jeden Test den minimalen und darüber den maximalen dieser Faktoren eingetragen. Tabelle 4.2 zeigt die Legende der Diagramme. Weiterhin habe ich zu jedem Test die Standardabweichung eingezeichnet.

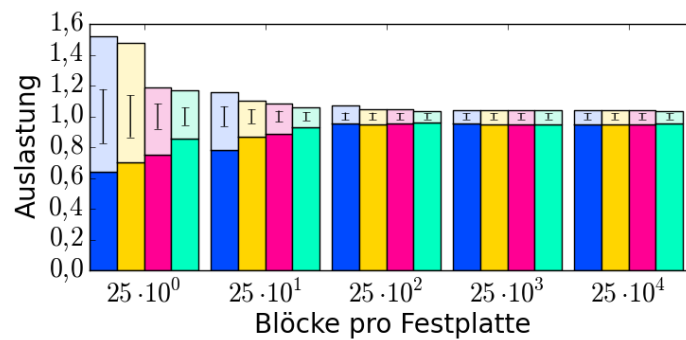


Abbildung 4.11: Unterschiedliche Anzahl Blöcke über einer Konfiguration homogener Festplatten mit *Consistent Hashing*

### 4.2.1 Einfluss der Anzahl der platzierten Blöcke

Die von mir vorgestellten Verteiler folgen dem Prinzip *Balls into Bins*. Bei diesem Prinzip werden eine Menge von Bällen zufällig über eine Menge von Körben verteilt. Abgebildet auf ein Speichersystem sind die Blöcke die Bälle und die Körbe die Festplatten. Unter Verwendung eines randomisierten Verteilers ist es notwendig, dass die Menge an verteilten Daten wesentlich größer ist, als die Anzahl der verwendeten Festplatten, damit die erwartete Abweichung der Last der Festplatten von der erwarteten Fairness begrenzt ist. Raab und Steger haben in [Raab und Steger, 1998] gezeigt, dass falls  $m > n \cdot \log n$  ist, bei einer homogenen Verteilung kein Korb mehr als die folgende Anzahl an Bällen hält:

$$\frac{m}{n} + \Theta\left(\sqrt{\frac{m \cdot \ln n}{n}}\right)$$

Somit ist die Qualität einer Verteilung abhängig der platzierten Bälle. Ich habe untersucht, wie sich die verschiedenen Verteilungsverfahren bei der Platzierung von einer steigenden Anzahl von Blöcken verhalten. Dazu habe ich eine homogene Konfiguration mit 64 Festplatten gewählt. Über diese Konfiguration habe ich  $k = \{1, 2, 4, 8\}$  Kopien einer steigenden Anzahl an Blöcken platziert. Wie bereits beschrieben habe ich in den folgenden Abbildungen den Faktor der am geringsten und am höchsten ausgelasteten Festplatte dargestellt. In jedem Test habe ich pro Festplatte 25, 250, 2500, 25.000 und 250.000 Blöcke platziert, hier also jeweils 64 mal den angegebenen Wert.

## 4 Evaluation der Strategien

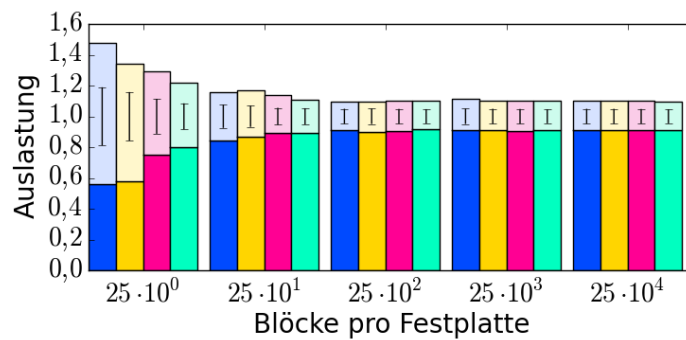


Abbildung 4.12: Unterschiedliche Anzahl Blöcke über einer Konfiguration homogener Festplatten mit *Share*

### Consistent Hashing

Abbildung 4.11 zeigt die Ergebnisse für *Consistent Hashing*. Von jeder der 64 homogenen Festplatten wurden  $400 \cdot \log n = 2400$  Kopien platziert.

Die Lastfaktoren bei der nicht redundanten Platzierung von  $25 \cdot 64$  Blöcke lagen zwischen 0,64 und 1,52 bei einer Standardabweichung von 0,18. Durch Erhöhung der Anzahl der platzierten Blöcke verringerte sich die Abweichung vom Ideal. Nach der Verteilung von  $25.000 \cdot 64$  Blöcken lagen die Lastfaktoren zwischen 0,95 und 1,04 bei einer Standardabweichung von 0,02. Eine weitere Erhöhung der platzierten Blöcke brachte keinen signifikanten Gewinn mehr.

Durch die Platzierung mehrerer Kopien jedes Blocks wird die Abweichung vom Idealwert vermindert. Bei der Platzierung von acht Kopien von  $25 \cdot 64$  Blöcken lagen die Lastfaktoren zwischen 0,86 und 1,17 bei einer Standardabweichung von 0,06. Bei der Verteilung von  $25.000 \cdot 64$  Blöcken lagen die Lastfaktoren wie bei der nicht redundanten Verteilung zwischen 0,95 und 1,04 bei einer Standardabweichung von 0,02.

Wie nicht anders zu erwarten verbessert sich die Verteilung der Daten je mehr Daten platziert werden. Dabei kann eine Erhöhung der verteilten Datenmenge durch die Verteilung von mehr Blöcken oder durch die Verteilung von mehr Kopien je Block erreicht werden. Dieses Verhalten war mit  $25.000 \cdot 64$  Blöcken weitestgehend saturiert. Die restliche Abweichung ist als konstanter Fehler in *Consistent Hashing* zu sehen. Der konstante Fehler entsteht auf Grund der pseudorandomisierte Abbildung der Festplatten, welche nicht sicherstellen kann, dass jede Festplatte den gleichen Anteil am Abbildungsbereich erhält. Dieser Fehler kann vermindert werden, indem mehr Kopien jeder Festplatte verteilt werden, wie ich in Abschnitt 4.2.2 noch zeige.

## Share

Ich habe die gleichen Messungen über identische Konfigurationen mittels *Share* durchgeführt. Die Ergebnisse sind Abbildung 4.12 zu entnehmen. Ich habe einen Dehnungsfaktor von  $5 \cdot \log n = 30$  gewählt. Wurden mittels *Share n'* Abbilder von Festplatten in einem Intervall platziert, so verwendete die entsprechende Instanz von *Consistent Hashing*  $400 \cdot \lceil \log n' \rceil$  Kopien jedes Abbilds.

Die Lastfaktoren bei der nicht redundanten Platzierung von  $25 \cdot 64$  Blöcken lagen zwischen 0,56 und 1,48 bei einer Standardabweichung von 0,19. Die geringste Abweichung vom Ideal konnte ich bei der Platzierung von  $2500 \cdot 64$  Blöcken messen. Die Lastfaktoren lagen hier zwischen 0,91 und 1,10 bei einer Standardabweichung von 0,05.

Zwar konnte bei wenigen platzierten Blöcken die Erhöhung der Anzahl der platzierten Kopien eine geringere Abweichung der Lastfaktoren vom Ideal bewirken, allerdings bleiben alle Lastfaktoren innerhalb eines Bereichs von 0,91 und 1,10. Auch die Standardabweichung blieb ähnlich.

*Share* erreicht schneller eine Sättigung als *Consistent Hashing*. Dies erscheint zunächst verwunderlich, werden doch zwei Hashfunktionen verwendet, erklärt sich allerdings durch eine nähere Betrachtung der Hashfunktionen. Die homogenen Hashfunktionen erhalten auf Grund des geringen Dehnungsfaktors nur eine geringe Menge an Festplattenabbildern. Somit sind die einzelnen homogenen Hashfunktionen recht schnell gesättigt. Das gleicht sich auch nicht durch die Menge der homogenen Hashfunktionen aus, da diese nicht unabhängig sind. Die Festplattenabbilder der selben Festplatte werden jeweils identisch abgebildet. Diese Eigenschaft ist notwendig, um die Adaptivität von *Share* zu gewährleisten.

Der höhere konstante Fehler von *Share* entsteht durch die Hashfunktion zur Abbildung der heterogenen Festplatten auf die Intervalle, dieser addiert sich zu dem konstanten Fehler von *Consistent Hashing*. Durch eine Erhöhung des Dehnungsfaktors kann dieser Fehler vermindert werden.

## Redundant Share

Abbildung 4.13 zeigt die Ergebnisse der Vermessung von *Redundant Share*. Die Lastfaktoren bei der Platzierung von  $25 \cdot 64$  Blöcken ohne Redundanz lagen zwischen 0,52 und 1,48 bei einer Standardabweichung von 0,19. Durch Erhöhung der Anzahl der platzierten Blöcke wurde eine kontinuierliche Verminderung der Abweichung vom Idealwert erreicht. Bei der Platzierung von  $250.000 \cdot 64$  Blöcken ohne Redundanz lagen die Lastfaktoren zwischen 0,9953 und 1,0038 bei einer Standardabweichung von 0,0017, sie wichen also kaum noch vom Ideal ab.

Auch für *Redundant Share* lässt sich eine zusätzliche Verbesserung der Abweichung durch Platzierung mehrerer Kopien erreichen. Die Platzierung von acht Kopien von  $25 \cdot 64$  Blöcken lieferte

## 4 Evaluation der Strategien

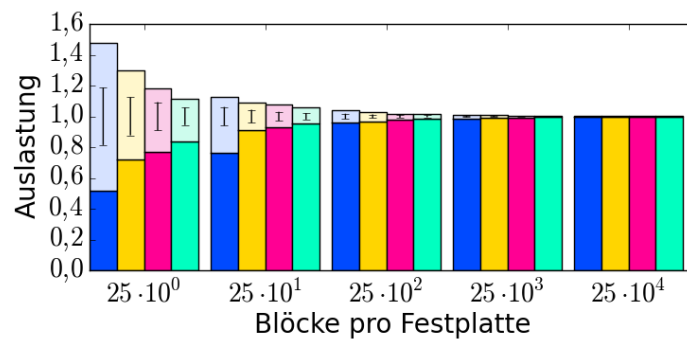


Abbildung 4.13: Unterschiedliche Anzahl Blöcke über einer Konfiguration homogener Festplatten mit *Redundant Share*

Lastfaktoren zwischen 0,84 und 1,12 bei einer Standardabweichung von 0,06. Für  $250.000 \cdot 64$  Blöcke verminderte sich die Abweichung auf Lastfaktoren zwischen 0,9976 und 1,0015 bei einer Standardabweichung von 0,0007.

Im Gegensatz zu *Share* und *Consistent Hashing* verwendet *Redundant Share* kein Zufallsexperiment zur Bildung der Hashfunktion. Lediglich die Platzierung der Daten geschieht randomisiert. Daher bringt *Redundant Share* keinen konstanten Fehler ein, und die Abweichung der Lastfaktoren konvergiert gegen 0 bei steigender Menge platzierter Daten.

### 4.2.2 Einfluss der Anzahl homogener Festplatten

Eine wichtige Eigenschaft von Speichersystemen ist die Skalierung über eine große Anzahl an Festplatten. Daher habe ich in den folgenden Tests untersucht, wie sich die Verfahren bei Konfigurationen mit unterschiedlich vielen Festplatten verhalten. Innerhalb dieses Abschnitts betrachte ich dabei nur Konfigurationen, in welchen alle Festplatten die gleiche Kapazität haben, also homogene Konfigurationen. Ich verwende die gleichen Konfigurationen wie in Abschnitt 4.1.1. Im Abschnitt 4.2.3 gehe ich auch auf heterogene Konfigurationen ein.

Im vorherigen Abschnitt habe ich gezeigt, dass das Speichersystem mit 64 Festplatten bei  $25.000 \cdot 64$  platzierten Blöcken gut eingeschwungen hat. *Consistent Hashing* und *Share* hatten in diesen Bereich bereits die besten Ergebnisse erreicht, *Redundant Share* hatte eine Abweichung von weniger als 0,02. Mit  $n$  als Anzahl der Festplatten im System habe ich in den folgenden Tests jeweils  $25.000 \cdot n$  Blöcke platziert. Wie zuvor betrachte ich den Einfluss einer unterschiedlichen Anzahl an Kopien je Block. Die Legende aus Tabelle 4.2 gilt weiterhin. Zu jedem Test habe ich wieder die Standardabweichung eingetragen.



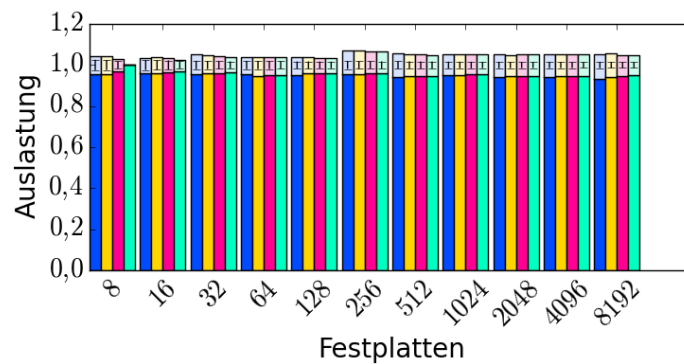


Abbildung 4.14: Fairness von *Consistent Hashing* bei unterschiedlicher Anzahl homogener Festplatten

### Consistent Hashing

Abbildung 4.14 zeigt die Fairness von *Consistent Hashing* bei Verwendung einer steigenden Anzahl von Festplatten. In jedem Fall wurden hier  $400 \cdot \lceil \log n \rceil$  Kopien jeder Festplatte platziert.

Die minimale und maximale gemessene relative Last einer Festplatte bleibt bei der Verwendung einer unterschiedlichen Anzahl von Festplatten annähernd gleich, allerdings fiel die Standardabweichung monoton. Bei Verwendung von acht Festplatten lag der Lastfaktor zwischen 0,96 und 1,04 bei einer Standardabweichung von 0,0256. Bei der Verteilung über 8192 Festplatten lagen die Lastfaktoren zwischen 0,93 und 1,05 bei einer Standardabweichung von 0,0152. Der geringste vorkommende Lastfaktor über alle Tests mit nicht redundanter Platzierung wurde bei 8192 Festplatten erreicht. Der höchste Lastfaktor wurde bei Verwendung von 256 Festplatten erreicht. Hier lagen die Lastfaktoren zwischen 0,95 und 1,07 bei einer Standardabweichung von 0,0192.

Die Platzierung einer höheren Anzahl von Kopien hat nur einen Einfluss auf die Fairness, wenn  $k$  und  $n$  sehr nah beieinander liegen. Im Extremfall  $k = n$  liefert *Consistent Hashing* eine absolut faire Verteilung. Dies ist auch nicht verwunderlich, da hier jede gültige Antwort aus allen Festplatten besteht. Bei vier Kopien liegt der Lastfaktor zwischen 0,97 und 1,03 bei einer Standardabweichung von 0,0177. Bei der Platzierung von zwei Kopien hat sich der Lastfaktor normalisiert und liegt zwischen 0,96 und 1,04 bei einer Standardabweichung von 0,0245.

Vernachlässige ich die Werte für acht Festplatten, so bleiben die minimalen und maximalen Lastfaktoren auch über mehrere Kopien hinweg konstant. Allerdings sinkt die Standardabweichung monoton bei steigender Anzahl von Kopien und fester Anzahl an Festplatten. Ebenso sinkt sie monoton bei steigender Anzahl an Festplatten und konstanter Anzahl zu platzierender Kopien, da in beiden Fällen die Menge der platzierten Daten erhöht wird.

Wie bereits im Abschnitt 4.1.1 erwähnt hat die Anzahl der platzierten Kopien jeder Festplatte

#### 4 Evaluation der Strategien

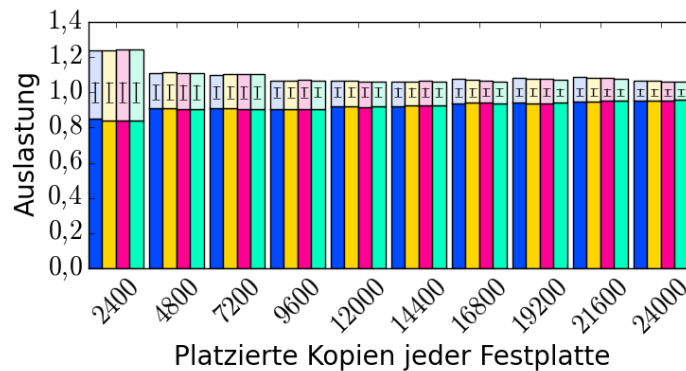


Abbildung 4.15: Fairness von *Consistent Hashing* bei unterschiedlicher Anzahl platzierter Kopien jeder Festplatte

Einfluss auf den Hauptspeicherverbrauch und die Fairness. Nachdem ich dort gezeigt habe, dass der Hauptspeicherverbrauch linear mit der Anzahl der platzierten Kopien jeder Festplatte steigt, zeige ich nun, wie sich die Fairness in diesem Fall verhält. Wie im entsprechenden Test in Abschnitt 4.1.1 habe ich wieder 256 homogene Festplatten verwendet. Von jeder Festplatte habe ich 2400 bis 24.000 Kopien platziert und die Verteilung von  $25.000 \cdot 256$  Blöcken vermessen. Die Ergebnisse sind in Abbildung 4.15 zu sehen.

Die Abweichung vom Ideal sinkt mit steigender Anzahl platzierter Kopien monoton. Bei nicht redundanter Platzierung liegen die gemessenen Lastfaktoren bei 2400 platzierten Kopien jeder Festplatte zwischen 0,85 und 1,24 bei einer Standardabweichung von 0,06. Bei 4800 platzierten Kopien jeder Festplatte liegen die gemessenen Lastfaktoren zwischen 0,91 und 1,11 bei einer Standardabweichung von 0,04, also deutlich dichter am Ideal. Bei der Platzierung von 24.000 Kopien jeder Festplatte lagen die Lastfaktoren noch zwischen 0,95 und 1,07 bei einer Standardabweichung von 0,02.

Die Anzahl der platzierten Kopien jedes Blocks hat nur einen sehr geringen Einfluss auf die Fairness. Bei 2400 platzierten Kopien jeder Festplatte schwanken die gemessenen Lastfaktoren leicht, allerdings in einem Bereich von weniger als 0,01.

Wie ich in Abschnitt 3.2 geschrieben habe, konnten Karger u. a. zeigen, dass *Consistent Hashing* eine Adaptivität von 1 bezüglich des Hinzufügens oder Entfernens einer Festplatte hat. Dies gilt allerdings nur, wenn immer die gleiche Anzahl von Kopien von jeder Festplatte platziert wird. Nach Karger u. a. ([Karger u. a., 1997]) müsste schon bei Erstellung eines Speichersystems ein Wert  $N$  festgelegt werden, welcher die maximale Anzahl an Festplatten festlegt, die jemals gleichzeitig in dem Speichersystem existieren. Die Anzahl platzierter Kopien je Festplatte wäre dann  $\alpha \cdot \lceil \log N \rceil$ .

Oftmals ist es schwer, schon zum Zeitpunkt der Erstellung eines Speichersystems zu sagen, wie viele Festplatten es in einigen Jahren umfassen soll. So kann es passieren, dass irgendwann mehr

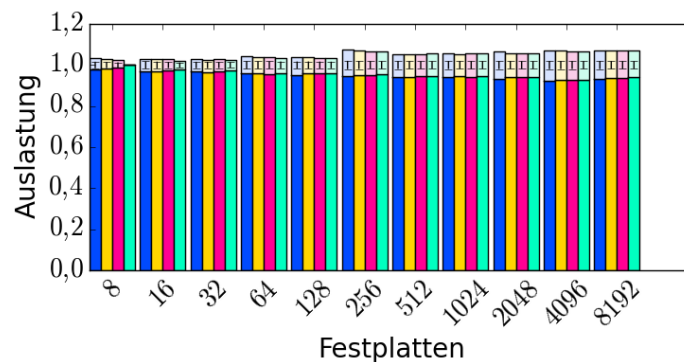


Abbildung 4.16: Fairness von *Consistent Hashing* bei unterschiedlicher Anzahl homogener Festplatten und fester Anzahl an Kopien jeder Festplatte

als  $N$  Festplatten in dem Speichersystem verwendet werden müssen. Daher habe ich für eine wachsende Anzahl an Festplatten die Fairness von *Consistent Hashing* bei einer festen Anzahl von  $400 \cdot \lceil \log 128 \rceil = 2800$  platzierter Kopien jeder Festplatte vermessen. Dieser Verteiler ist also auf maximal 128 Festplatten ausgelegt. Ich habe vermessen, wie er sich für bis zu 8192 Festplatten verhält. Die Ergebnisse sind in Abbildung 4.16 dargestellt.

Unter diesen Bedingungen erzeugen Konfigurationen mit vielen Festplatten eine größere Abweichung der Lastfaktoren vom Idealwert als Konfigurationen mit wenigen Festplatten. Bei der nicht redundanten Platzierung lagen die Lastfaktoren bei einer Verteilung über acht Festplatten zwischen 0,98 und 1,03 bei einer Standardabweichung von 0,02. Für 128 Festplatten lagen die Lastfaktoren zwischen 0,95 und 1,04 bei einer Standardabweichung von 0,02. Für 8192 Festplatten stieg die Abweichung vom Ideal. Die Lastfaktoren lagen zwischen 0,93 und 1,07. Die Platzierung mehrerer Kopien jedes Blocks brachte keine signifikante Änderung.

Selbst ein extremes Überschreiten von  $N$  ist also nur mit einer begrenzten Verschlechterung der Lastfaktoren verbunden. Falls die Anzahl der Festplatten und die Anzahl der zu platzierenden Kopien nicht nah beieinander liegen, ist die Fairness meiner Implementierung von *Consistent Hashing* unabhängig von der Anzahl der platzierten Kopien jedes Blocks.

## Share

Wie bereits in Abschnitt 4.1.1 gezeigt, benötigt *Share* mit einer Konfiguration von mehr als 1024 Festplatten mehr Hauptspeicher, als die 64 GByte RAM, die mir in meiner Testumgebung zur Verfügung stand. Daher habe ich *Share* nur für Konfigurationen bis zu dieser Größe untersucht. In Abbildung 4.17 habe ich die Bereiche für größere Konfigurationen leer gelassen, um dies zu verdeutlichen.

Die Lastfaktoren bei der nicht redundanten Platzierung über acht Festplatten lagen zwischen

#### 4 Evaluation der Strategien

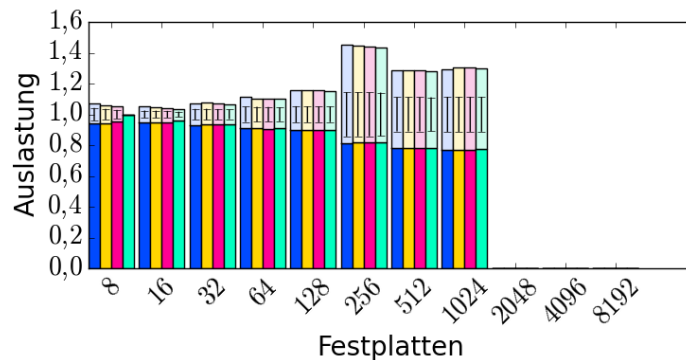


Abbildung 4.17: Fairness von *Share* bei unterschiedlicher Anzahl homogener Festplatten

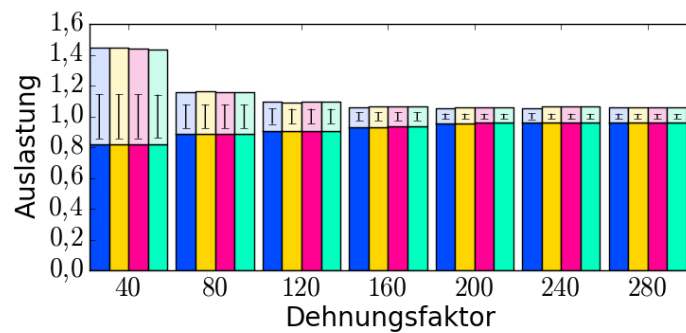
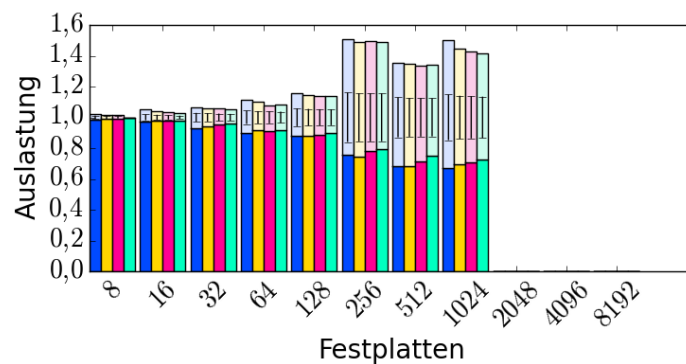
0,94 und 1,07 bei einer Standardabweichung von 0,04. Für Konfigurationen mit bis zu 32 Festplatten blieben die Lastfaktoren in ähnlichen Bereichen, danach stieg die Abweichung vom Ideal. Mit Lastfaktoren zwischen 0,81 und 1,45 bei einer Standardabweichung von 0,14 konnte ich bei der Verteilung über 256 Festplatten den größten Lastfaktor messen. Der geringste Lastfaktor trat bei der Verteilung über 1024 Festplatten auf. Die Lastfaktoren lagen hier zwischen 0,77 und 1,29 bei einer Standardabweichung von 0,11.

Die schlechte Fairness von *Share* begründet sich über den geringen Dehnungsfaktor. In Abschnitt 3.2 habe ich die Fairness von *Share* abhängig von dem gewählten Dehnungsfaktor angegeben. Dem folgend müsste ein Dehnungsfaktor von  $726 \cdot \ln n$  gewählt werden, um Lastfaktoren zwischen 0,9 und 1,1 zu erhalten, vorausgesetzt die homogenen Hashfunktionen würden mit einer Fairness von 1 verteilen. Für Lastfaktoren zwischen 0,5 und 1,5 wäre immer noch ein Dehnungsfaktor von  $54 \cdot \ln n$  nötig. Ich habe in Abschnitt 4.1.1 gezeigt, dass der Speicherbedarf von *Share* bei solchen Dehnungsfaktoren zu hoch ist, um dieses Verfahren zur Verteilung über viele Festplatten zu verwenden.

Der Einfluss der platzierten Kopien ist nur signifikant, wenn die Anzahl der Kopien nahe der Anzahl der Festplatten in der verwendeten Konfiguration ist. Hier macht sich wieder bemerkbar, dass die Implementierung bei vielen Kopien eine geringere Wahlfreiheit hat.

Um den Einfluss des gewählten Dehnungsfaktors  $s$  zu untersuchen, habe ich *Share* mit unterschiedlich großem Dehnungsfaktor vermessen. Den Hauptspeicherverbrauch und die Laufzeit für diese Parametrisierung habe ich bereits in Abschnitt 4.1.1 betrachtet. Diesen Messungen entsprechend habe ich die Fairness von *Share* bei der Verteilung von Daten über 256 Festplatten vermessen. Ich habe *Share* mit unterschiedlichen Dehnungsfaktoren zwischen  $s = 5 \cdot \log n = 40$  und  $s = 35 \cdot \log n = 280$  vermessen. Größere Dehnungsfaktoren konnte ich auf Grund des Hauptspeicherverbrauchs nicht vermessen. Die 64 GByte RAM des Testrechners wären damit überfordert gewesen. Die Ergebnisse habe ich in Abbildung 4.18 dargestellt.

Die Abweichung der gemessenen Lastfaktoren vom Ideal sinkt bei steigendem Dehnungsfaktor.

Abbildung 4.18: Fairness von *Share* bei unterschiedlichem DehnungsfaktorAbbildung 4.19: Fairness von *Share* bei unterschiedlicher Anzahl homogener Festplatten und festem Dehnungsfaktor

Bei der nicht redundanten Platzierung lagen die Lastfaktoren zwischen 0,82 und 1,45 bei einer Standardabweichung von 0,14 und einem Dehnungsfaktor von 40. Bis zu einem Dehnungsfaktor von 160 sank die Abweichung vom Ideal recht schnell, die Lastfaktoren lagen zwischen 0,93 und 1,06 bei einer Standardabweichung von 0,03. Danach blieben die Lastfaktoren allerdings in einem ähnlichen Bereich.

Diese Betrachtung deckt sich mit den Ergebnissen aus [Brinkmann u. a., 2002]. Durch einen größeren Dehnungsfaktor wird die Fairness von *Share* verbessert. Um weitere Verbesserungen zu erreichen, sind wesentlich größere Dehnungsfaktoren notwendig, da ein angestrebter Fehler quadratischen Einfluss auf den Dehnungsfaktor hat.

Ähnlich zu der Anzahl der platzierten Kopien bei *Consistent Hashing* muss auch bei *Share* der Dehnungsfaktor konstant sein, um die Adaptivität zu gewährleisten. Daher habe ich *Share* mit einem festen Dehnungsfaktor mit Konfigurationen bestehend aus einer ansteigenden Anzahl an Festplatten vermessen. Ich habe dazu einen Dehnungsfaktor von  $5 \cdot \log 128 = 35$  gewählt, den Verteilungsalgorithmus also auf maximal  $N = 128$  Festplatten ausgelegt. Für die Verteilung

## 4 Evaluation der Strategien

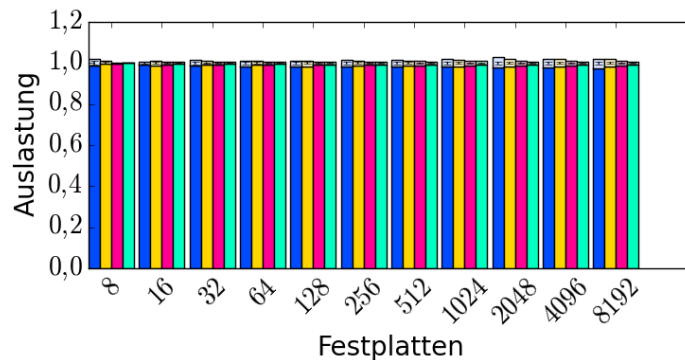


Abbildung 4.20: Fairness von *Redundant Share* bei unterschiedlicher Anzahl homogener Festplatten

innerhalb der Intervalle habe ich die Anzahl der Kopien, die *Consistent Hashing* von jedem Abbild einer Festplatte platziert, auf  $400 \cdot \lceil \log 128 \rceil = 2800$  festgelegt. Die Ergebnisse habe ich in Abbildung 4.19 dargestellt.

Die Lastfaktoren bei der nicht redundanten Platzierung über acht Festplatten lagen zwischen 0,99 und 1,02 bei einer Standardabweichung von 0,01. Die Abweichung vom Ideal stieg bei steigender Anzahl an Festplatten an. Bei der Verteilung über 256 Festplatten lagen die Lastfaktoren zwischen 0,76 und 1,51 bei einer Standardabweichung von 0,16. Die Lastfaktoren bei der Verteilung über größere Konfigurationen bleiben in ähnlichen Bereichen. Die Anzahl der platzierten Kopien hatte keinen signifikanten Einfluss auf die Lastfaktoren, falls die Anzahl der Festplatten wesentlich größer war, als die Anzahl der platzierten Kopien.

Bereits für die Konfiguration mit 128 Festplatten ist der Dehnungsfaktor sehr klein gewählt. Wie die Ergebnisse der Messungen zeigen, ist ein Überschreiten von  $N$  bei *Share* wesentlich problematischer, als bei *Consistent Hashing*.

### Redundant Share

Abbildung 4.20 zeigt die Ergebnisse der Vermessung von *Redundant Share*. Da die Ergebnisse von *Redundant Share* teils extrem nah am Ideal liegen, werde hier die Faktoren und Standardabweichung mit vier Nachkommastellen angeben. Andernfalls wären kaum Entwicklungen sichtbar.

Bei der nicht redundanten Platzierung lagen die Lastfaktoren zwischen 0,9962 und 1,0189 bei einer Standardabweichung von 0,0096. Der Abstand wuchs je mehr Festplatten verwendet wurden und lag bei Verwendung von 8192 Festplatten zwischen 0,9738 und 1,0219 bei einer Standardabweichung von 0,0063.

Je mehr Kopien jedes Blocks platziert wurden, desto geringer wurden die Abstände der Lastfak-

toren vom Ideal. Bei der Verteilung über 8192 Festplatten lagen die Lastfaktoren zur Platzierung von acht Kopien jedes Blocks zwischen 0,9927 und 1,0078 bei einer Standardabweichung von 0,0022.

Die Abweichung der Lastfaktoren vom Ideal ist bei *Redundant Share* wesentlich geringer als bei *Consistent Hashing* und *Share*. *Redundant Share* besitzt keinen konstanten Fehler und kann daher beliebig nah an eine perfekt faire Verteilung kommen, indem genügend Daten platziert werden. Daher bewirkt eine Erhöhung der Anzahl der platzierten Kopien auch eine bessere Verteilung. Diese macht sich allerdings kaum noch bemerkbar. Um sie sichtbar zu machen musste ich die Lastfaktoren bis zur vierten Nachkommastelle betrachten.

Der leichte Anstieg der Abweichung der Lastfaktoren vom Ideal bei steigender Anzahl an Festplatten begründet sich in der Anzahl der platzierten Blöcke. Die Anzahl der platzierten Blöcke wächst linear mit der Anzahl der verwendeten Festplatten. Dies genügt nicht, um einen konstanten Fehler zu erhalten.

### 4.2.3 Einfluss der Anzahl heterogener Festplatten

Wie bereits in Abschnitt 4.1.2 beschrieben ist es für skalierbare Speichersysteme extrem wichtig, auch Festplatten mit unterschiedlicher Kapazität fair belasten zu können. In diesem Abschnitt untersuche ich daher die Implementierung der verschiedenen Verfahren hinsichtlich ihrer Fairness über heterogenen Konfigurationen. Ich verwende dabei wieder die bereits in Abschnitt 4.1.2 vorgestellten Konfigurationen. Wie bei den vorherigen Messungen platziere ich auch hier  $k \in \{1, 2, 4, 8\}$  Kopien jedes Blocks. Allerdings passe ich die Anzahl der platzierten Blöcke an die Heterogenität an. Mit  $C$  als Gesamtmenge des physikalischen Speichers habe ich bei der Vermessung der homogenen Konfigurationen jeweils  $25.000 \cdot n = \frac{C}{20}$  Blöcke verteilt. Bei der Vermessung der heterogenen Konfigurationen habe ich dem folgend  $\frac{C}{20}$  Blöcke verteilt, um die Auswirkungen der heterogenen Verteilung besser beobachten zu können. Auch die Diagramme in diesem Abschnitt halten sich an die Legende aus Tabelle 4.2.

#### Consistent Hashing

Wie Abbildung 4.21 zeigt, eignet sich *Consistent Hashing* in der einfachen Form nicht wirklich zur Verteilung von Daten über heterogenen Festplatten. *Consistent Hashing* weist jeder Festplatte den gleichen Anteil der zu verteilenden Last zu, womit die großen Festplatten benachteiligt werden. Im Schritt  $s_1$  wird eine Konfiguration bestehend aus 128 homogenen Festplatten mit einer Kapazität  $c = 500.000$  Blöcken vermessen. In Schritt  $i \in \{2, \dots, 10\}$  kommen 128 Festplatten mit einer Kapazität von  $c \cdot 1,5^{i-1}$  zu der Konfiguration aus Schritt  $s_{i-1}$  hinzu. Insgesamt hat das Speichersystem daher in Schritt  $s_i$  folgende Kapazität  $C_i$ :

#### 4 Evaluation der Strategien

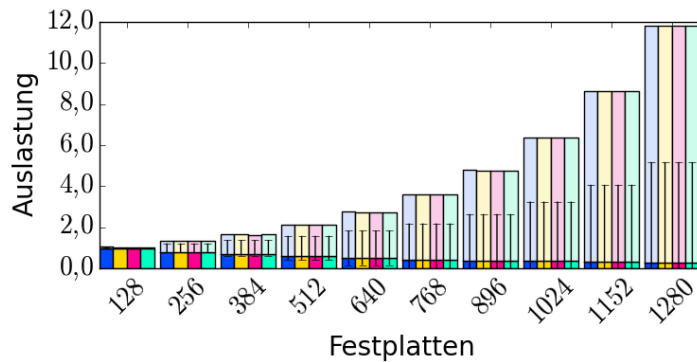


Abbildung 4.21: Fairness von *Consistent Hashing* bei unterschiedlicher Anzahl heterogener Festplatten

$$C_i = \sum_{j=1}^i 128 \cdot c \cdot 1,5^{i-1}$$

Die größten Festplatten in Schritt  $s_i$  haben eine Kapazität von  $c \cdot 1,5^{i-1}$ . Daher müssten sie je eine Last von  $\frac{c \cdot 1,5^{i-1}}{C_i}$  erhalten. Bei einer Verteilung ohne Beachtung der Kapazitäten erhält jede Festplatte allerdings einen Anteil von  $\frac{1}{128 \cdot i}$  der Last, da es  $128 \cdot i$  Festplatten in der Konfiguration des jeweiligen Schritts gibt. Dies führt zu folgendem erwarteten Lastfaktor für die größten Festplatten:

$$\begin{aligned}
 F_i^g &= \frac{\frac{1}{128 \cdot i}}{\frac{c \cdot 1,5^{i-1}}{C_i}} \\
 &= \frac{C_i}{c \cdot 1,5^{i-1} \cdot 128 \cdot i} \\
 &= \frac{\sum_{j=1}^i 128 \cdot c \cdot 1,5^{i-1}}{c \cdot 1,5^{i-1} \cdot 128 \cdot i} \\
 &= \frac{\sum_{j=1}^i 1,5^{i-1}}{1,5^{i-1} \cdot i}
 \end{aligned}$$

Auf der anderen Seite haben die kleinsten Festplatten eine Kapazität von  $c$  und müssten daher eine Last von  $\frac{c}{C_i}$  erhalten. Ihr erwarteter Lastfaktor lässt sich wie folgt bestimmen:



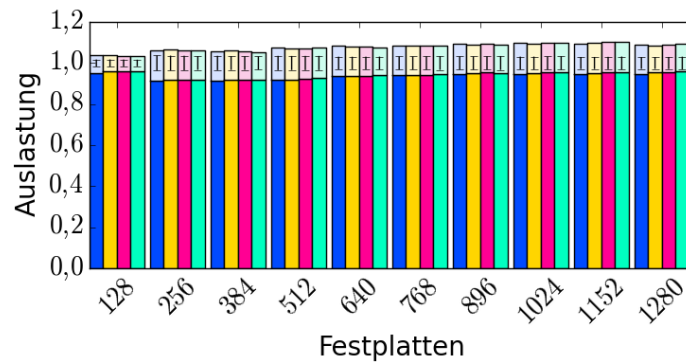


Abbildung 4.22: Fairness von *Consistent Hashing* bei unterschiedlicher Anzahl heterogener Festplatten und kapazitätsabhängiger Anzahl platzierter Kopien jeder Festplatte

$$\begin{aligned}
 F_i^k &= \frac{1}{\frac{128 \cdot i}{c}} \\
 &= \frac{c}{128 \cdot i} \\
 &= \frac{c \cdot 128 \cdot i}{\sum_{j=1}^i 128 \cdot c \cdot 1,5^{i-1}} \\
 &= \frac{c \cdot 128 \cdot i}{\sum_{j=1}^i 1,5^{i-1}} \\
 &= \frac{1}{i}
 \end{aligned}$$

In der Konfiguration mit 1280 Festplatten erwarte ich daher einen durchschnittlichen Lastfaktor von  $F_{10}^g \approx 0,29$  für die größten Festplatten und von  $F_{10}^k \approx 11,33$  für die kleinsten Festplatten. Dies deckt sich mit den in Abbildung 4.21 dargestellten Ergebnissen meiner Messungen. Bei der nicht redundanten Platzierung lag der kleinste gemessene Lastfaktor bei 0,28 und der größte gemessene Lastfaktor bei 11,84.

Bereits in Abschnitt 4.1.2 habe ich vorgestellt, wie meine Implementierung von *Consistent Hashing* Festplatten heterogen abbildet. Dazu platziert sie von jeder Festplatte eine unterschiedliche Anzahl an Kopien, abhängig von ihrer Kapazität. Auf diesem Weg kann auch für heterogene Festplatten eine gute Verteilung erreicht werden. Wie in Abschnitt 4.1.2 beschrieben habe ich *Consistent Hashing* so parametrisiert, dass von jeder Festplatte pro 100.000 Blöcken Kapazität  $80 \cdot \lceil \log n \rceil$  Kopien platziert werden. Dabei ist  $n$  die Anzahl der Festplatten in der Konfiguration. Die Ergebnisse sind in Abbildung 4.22 dargestellt.

Die Konfiguration aus 128 Festplatten bestand nur aus Festplatten mit der gleichen Kapazität. Weiterhin wurde von jeder Festplatte die gleiche Anzahl an Kopien platziert, wie bei den

#### 4 Evaluation der Strategien

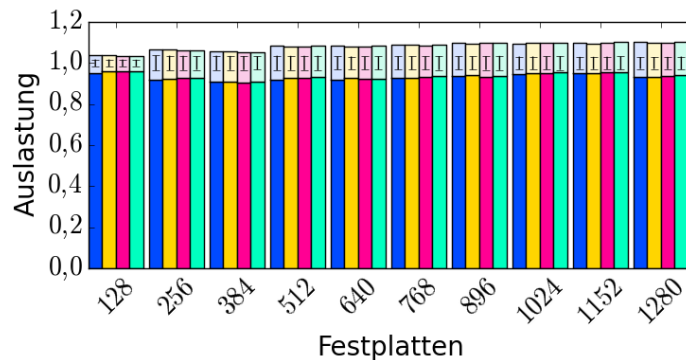


Abbildung 4.23: Fairness von *Consistent Hashing* bei unterschiedlicher Anzahl heterogener Festplatten und kapazitätsabhängiger Anzahl platzierter Kopien jeder Festplatte unabhängig von der Anzahl der Festplatten

homogenen Tests. Somit wurden die gleichen Lastfaktoren wie bei der homogenen Verteilung erreicht. Diese lagen bei der nicht redundanten Platzierung zwischen 0,95 und 1,04 bei einer Standardabweichung von 0,02.

Beginnend mit der Konfiguration mit 256 Festplatten existieren einige Festplatten, deren Größe kein Vielfaches von 100.000 ist. Dies sorgt automatisch für eine schlechtere Verteilung der Daten. Die nicht redundante Verteilung über 256 Festplatten lieferte Lastfaktoren zwischen 0,91 und 1,06 bei einer Standardabweichung von 0,04. Unabhängig von der Anzahl der Kopien lagen die Lastfaktoren für alle weiteren Konfigurationen in einem Bereich von 0,11 um den Idealwert.

In Abschnitt 4.2.2 habe ich bereits erklärt, dass es für die Adaptivität von *Consistent Hashing* wichtig ist, die Anzahl der platzierten Kopien einer Festplatte unabhängig von den restlichen Festplatten in der Konfiguration zu wählen. Um auch für heterogene Konfigurationen eine gute Verteilung zu erreichen, muss trotzdem die Anzahl der zu platzierenden Kopien jeder Festplatte abhängig von ihrer Kapazität bestimmt werden. Ich habe die zuvor verwendeten Konfigurationen übernommen und pro 100.000 Blöcken Kapazität von jeder Festplatte  $80 \cdot \lceil \log 128 \rceil = 560$  Kopien der Festplatte platziert. Die Ergebnisse habe ich in Abbildung 4.23 dargestellt.

Bereits bei der Betrachtung von *Consistent Hashing* über homogenen Festplatten und fester Anzahl an Kopien in Abschnitt 4.2.2 ist aufgefallen, dass sich die Fairness von *Consistent Hashing* bei steigender Anzahl an Festplatten nur sehr langsam verschlechtert. Innerhalb der dort vorgestellten Konfiguration hat sich die Anzahl der verwendeten Festplatten exponentiell auf 8192 erhöht. Innerhalb der hier lediglich linear auf 1280 Festplatten ansteigenden Konfiguration ist daher nur ein sehr geringer Unterschied zwischen den in Abbildung 4.23 dargestellten Tests mit einer festen Anzahl an Kopien je Festplatten und den in Abbildung 4.22 dargestellten Tests mit einer von  $n$  abhängigen Anzahl an Kopien jeder Festplatte erkennbar.

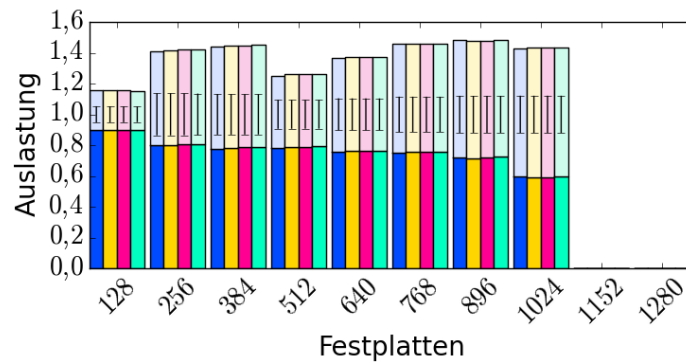


Abbildung 4.24: Fairness von *Share* bei unterschiedlicher Anzahl heterogener Festplatten

Vergleiche ich die Ergebnisse für eine Konfiguration mit 1280 Festplatten so lagen die Lastfaktoren bei der nicht redundanten Platzierung zwischen 0,93 und 1,10 bei einer Standardabweichung von 0,03. Bei dem zuvor durchgeführten Test wurden bei dieser Konfiguration Lastfaktoren zwischen 0,95 und 1,09 bei einer Standardabweichung von 0,03 gemessen.

## Share

*Share* habe ich zunächst wieder mit einem Dehnungsfaktor von  $5 \cdot \log n$  vermessen. Die homogenen Verteiler haben dabei  $400 \cdot \lceil \log n' \rceil$  Kopien der  $n'$  Abbildungen der Festplatten im jeweiligen Intervall platziert. Die Ergebnisse sind in Abbildung 4.24 dargestellt.

Die Konfiguration über 128 Festplatten besteht lediglich aus homogenen Festplatten und stimmt exakt mit der Konfiguration über 128 Festplatten aus Abschnitt 4.2.2 überein. Folglich liefert *Share* auch die gleiche Verteilung. Wie schon bei den homogenen Tests beobachtet vergrößert sich die Abweichung der Lastfaktoren vom Ideal ab 256 Festplatten. Bei nicht redundanter Platzierung lagen die Lastfaktoren zwischen 0,80 und 1,41 bei einer Standardabweichung von 0,14. Diese Abweichung vergrößerte sich fast durchgängig. Über der Konfiguration mit 1024 Festplatten lagen die Lastfaktoren zwischen 0,60 und 1,43 bei einer Standardabweichung von 0,12. Die Platzierung mehrerer Kopien hatte keinen signifikanten Einfluss auf die Lastfaktoren.

Bereits in Abschnitt 4.2.2 habe ich beschrieben, dass der Dehnungsfaktor zu gering gewählt ist, um die Abweichung der Lastfaktoren vorhersagen zu können. Dies macht sich auch bei den heterogenen Konfigurationen bemerkbar.

Um die Adaptivität zu gewährleisten ist es weiterhin nötig, den Dehnungsfaktor unabhängig von der Anzahl der verwendeten Festplatten zu wählen. Wie zuvor bei der Vermessung von *Consistent Hashing* habe ich den Dehnungsfaktor auf den für  $N = 128$  Festplatten gewählten Wert festgesetzt. Der Dehnungsfaktor lag also bei  $5 \cdot \log 128 = 35$ . Abbildung 4.25 zeigt die Ergebnisse der Vermessung von *Share* bei einem festen Dehnungsfaktor von 35.

#### 4 Evaluation der Strategien

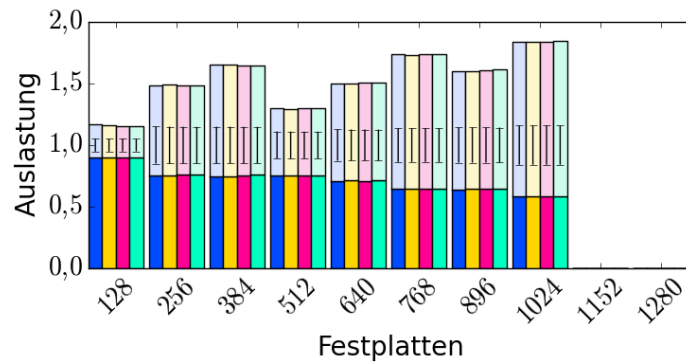


Abbildung 4.25: Fairness von *Share* bei unterschiedlicher Anzahl heterogener Festplatten und festem Dehnungsfaktor

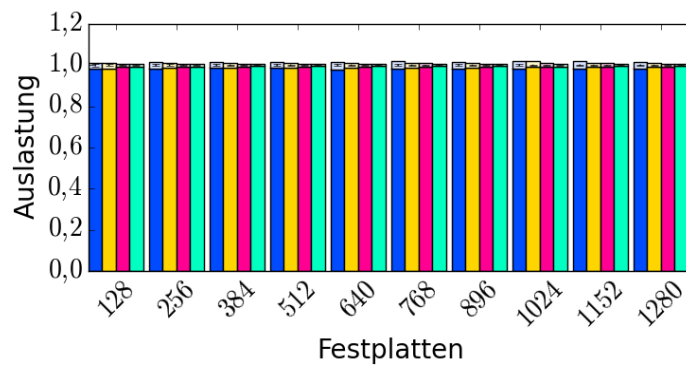


Abbildung 4.26: Fairness von *Redundant Share* bei unterschiedlicher Anzahl heterogener Festplatten

Gegenüber den vorherigen Tests mit einem von der Anzahl der Festplatten abhängigen Dehnungsfaktor hat sich die Abweichung vom Ideal bei Verwendung von mehr als 128 Festplatten sichtlich erhöht. Bei der nicht redundanten Platzierung lagen die gemessenen Lastfaktoren bei Verwendung von 256 Festplatten bereits zwischen 0,75 und 1,49 bei einer Standardabweichung von 0,15. Bei der Verwendung von 1024 Festplatten lagen die Lastfaktoren zwischen 0,58 und 1,84 bei einer Standardabweichung von 0,16. Die Platzierung mehrerer Kopien jedes Blocks resultierte in keinen bemerkenswerten Unterschieden.

#### Redundant Share

Abbildung 4.26 zeigt die Ergebnisse der Vermessung der Fairness von *Redundant Share* über heterogenen Festplatten.

Bei der nicht redundanten Verteilung von Daten über 128 Festplatten mittels *Redundant Share*

lagen die Lastfaktoren zwischen 0,9812 und 1,0127 bei einer Standardabweichung von 0,0058. Diese Lastfaktoren schwankten ein wenig bei unterschiedlichen Konfigurationen, was sich aber nur in der dritten Nachkommastelle bemerkbar machte.

Die Abweichung der Lastfaktoren vom Ideal schrumpfte je mehr Kopien platziert wurden. Bei der Platzierung von acht Kopien über 8192 Festplatten lagen die Lastfaktoren zwischen 0,9961 und 1,0060 bei einer Standardabweichung von 0,0012.

*Redundant Share* unterscheidet nicht zwischen homogenen und heterogenen Festplatten. Wie bereits in Abschnitt 4.2.3 ausgeführt gibt es auch keinen eigenen Fehler. Daher ist die Qualität der Verteilung lediglich von der Menge der platzierten Daten abhängig. Die leichte Steigerung der Abweichung der Lastfaktoren vom Ideal bei großen Konfigurationen war hier nicht mehr bemerkbar. Die Anzahl der platzierten Blöcke wuchs mit dem Gesamtspeicher stärker als bei den homogenen Messungen. So wurde der Fehler durch die Randomisierung ausgeglichen.













## 4.3 Adaptivität

Eine der Anforderungen, die ich an ein Speichersysteme gestellt habe, war die Skalierbarkeit. Reicht die Kapazität des vorhandenen Speichers nicht mehr aus, so sollte das Speichersystem einfach durch das Hinzufügen weiterer Festplatten vergrößert werden können. Damit eine Anpassung an eine veränderte Konfiguration schnell umgesetzt werden kann, sollten die Verteilungsverfahren so wenig Daten innerhalb des Speichersystems bewegen wie möglich. Im Idealfall sollten nur die Daten bewegt werden, welche auf dem hinzukommenden Speicher platziert werden müssen um eine faire Verteilung zu erhalten. Hat der hinzukommende Speicher einen Anteil von  $\lambda$  am Gesamtspeicher, so sollte um die Fairness zu gewährleisten ein Anteil von  $\lambda$  der Gesamtlast auf den neuen Speicher verschoben werden.

Innerhalb der in diesem Abschnitt dargestellten Tests verwende ich als Ausgangsbasis eine Konfiguration mit 128 homogenen Festplatten. Ich verteile über diesen Festplatten  $k \in \{1, 2, 4, 8\}$  Kopien von  $250.000 \cdot 128$  Blöcken. Danach messe ich, wie viele Daten, also Kopien von Blöcken, innerhalb des Speichersystems bewegt werden, wenn  $n' \in \{1, 2, 3, 5, 7, 11, 13\}$  Festplatten hinzugefügt werden. In Abschnitt 4.3.1 vermesse ich homogene Speichersysteme und füge Festplatten mit der gleichen Kapazität wie die der ursprünglichen Festplatten hinzu. In Abschnitt 4.3.2 erzeuge ich ein heterogenes Speichersystem, indem ich Festplatten mit einer 1,5-fachen Kapazität der anfangs verwendeten Festplatten hinzufüge.

In den Abbildungen der Testergebnisse trage ich für jeden Test ein, welcher Faktor an Daten gegenüber einer absolut fairen Verteilung auf den neuen Festplatten platziert wurde. Hat der hinzukommende Speicher einen Anteil an  $\lambda$  des Gesamtspeichers und wurde ein Anteil von  $\lambda'$  der Daten auf den hinzukommenden Speicher bewegt, so errechnet sich der Faktor aus  $\frac{\lambda'}{\lambda}$ .

Tabelle 4.3: Legende der Diagramme zur Messung der Adaptivität

	$\frac{\lambda'}{\lambda}$	$\frac{\eta}{\lambda}$	$\frac{\eta'}{\lambda}$
eine Kopie			
zwei Kopien			
vier Kopien			
acht Kopien			

In vielen Fällen werden die Verteilungsverfahren neben den Daten, die sie auf die neuen Festplatten verschieben, auch Daten zwischen den schon vorher vorhandenen Festplatten verschieben. Innerhalb meiner Messungen habe ich auch protokolliert, wie viele Daten insgesamt im Speichersystem verschoben wurden. Dabei unterscheide ich wie in Abschnitt 3.1 beschrieben zwischen den notwendigen Replatzierungen mit und ohne Beachtung der Reihenfolge der Kopien. Musste ein Anteil von  $\eta$  der Daten ohne Beachtung der Reihenfolge der Kopien replaziert werden, so trage ich in die Abbildungen diesen Wert relativ zu  $\lambda$  ein, also  $\frac{\eta}{\lambda}$ . Musste ein Anteil von  $\eta'$  der Daten mit Beachtung der Reihenfolge der Kopien replaziert werden, so trage ich auch diesen Wert relativ zu  $\lambda$  ein, also  $\frac{\eta'}{\lambda}$ .

Da die Daten, welche auf den hinzukommenden Speicher verschoben werden müssen eine Teilmenge der Daten sind, die ohne Beachtung der Reihenfolge der Kopien replaziert wurden, gilt  $\lambda' \leq \eta$ . Werden ohne Beachtung der Reihenfolge der Kopien Daten in einem Speichersystem replaziert, so müssen diese Daten auf jeden Fall auch mit Berücksichtigung der Reihenfolge replaziert werden, daher gilt  $\eta \leq \eta'$ . Innerhalb der Tests zur Adaptivität trage ich die Faktoren übereinander ein. Tabelle 4.3 zeigt die Legende der Abbildungen zur Adaptivität.

### 4.3.1 Adaptivität beim Einfügen homogener Festplatten

Innerhalb dieses Abschnitts zeige ich die Ergebnisse der Vermessung homogener Speichersysteme. Die hinzugefügten Festplatten haben daher die gleiche Kapazität wie die ursprünglichen Festplatten.

#### Consistent Hashing

Zunächst habe ich die Adaptivität von *Consistent Hashing* vermessen. Dabei habe ich von jeder der  $n$  Festplatten  $400 \cdot \lceil \log n \rceil$  Kopien platziert. Die Ergebnisse sind in Abbildung 4.27 dargestellt.

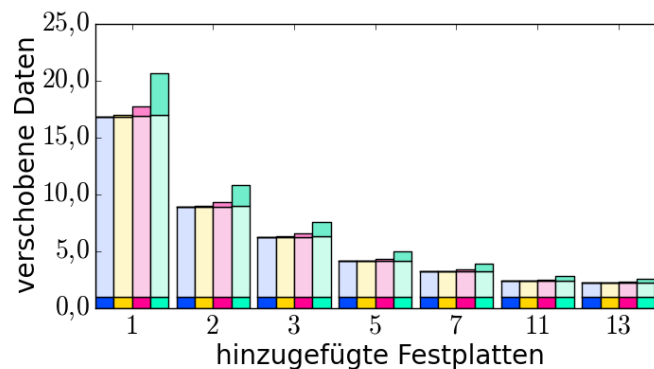


Abbildung 4.27: Adaptivität von *Consistent Hashing* beim Hinzufügen homogener Festplatten

In Abschnitt 4.2.3 habe ich die Fairness von *Consistent Hashing* bei Verwendung einer variablen Anzahl an homogenen Festplatten vermessen. Die Lastfaktoren bei einer Konfiguration mit 128 Festplatten lagen bei diesen Tests zwischen 0,95 und 1,04. Innerhalb dieses Bereichs liegt bei sämtlichen Messungen auch der Faktor der auf die neuen Festplatten verschobenen Daten.

Zusätzlich zu den Verschiebungen auf die neuen Festplatten wurden bei meiner Implementierung von *Consistent Hashing* auch Daten zwischen den schon vorher vorhandenen Festplatten bewegt. Bei der nicht redundanten Platzierung lag der Faktor  $\frac{\eta}{\lambda}$  beim Hinzufügen einer Festplatte bei 16,82. Wurden die Anzahl der hinzugefügten Festplatten größer, so stieg die Anzahl der replazierten Daten nur leicht, so dass der Faktor der replazierten Daten pro Festplatte sank. Bei zwei hinzugefügten Festplatten hat sich der Faktor annähernd halbiert auf 8,89. Diese Entwicklung setzt sich fort. Beim Hinzufügen von 13 Festplatten lag der Faktor der zusätzlich bewegten Daten bei 2,21.

Ohne Berücksichtigung der Reihenfolge der Kopien hat sich der Faktor der replazierten Daten minimal erhöht, je mehr Kopien jedes Blocks platziert wurden. Beim Hinzufügen einer Festplatte ist der Faktor für zwei Kopien auf 16,86, für vier Kopien auf 16,91 und für acht Kopien auf 16,99 gestiegen. Beim Hinzufügen von 13 Festplatten ist der Wert beim Platzieren von zwei Kopien auf 2,21, für vier Kopien auf 2,22 und für acht Kopien auf 2,23 gestiegen.

Zusätzlich gab es auch Replatzierungen innerhalb der Kopien einzelner Blöcke. Der Faktor der replazierten Daten mit Berücksichtigung der Reihenfolge der Kopien hing sowohl von der Anzahl der hinzugefügten Festplatten als auch von der Anzahl der Kopien ab. Bei der Platzierung von zwei Kopien lag der Faktor beim Hinzufügen einer Festplatte bei 16,99 und sank monoton auf 2,23 beim Hinzufügen von 13 Festplatten. Bei der Platzierung von vier Kopien jedes Blocks sank der Faktor monoton von 17,71 auf 2,30 und beim Platzieren von acht Kopien von 20,67 auf 2,60.

Wie beschrieben habe ich in jedem Testlauf  $400 \cdot \lceil \log n \rceil$  Kopien jeder Festplatte platziert. Daher habe ich in der ursprünglichen Konfiguration  $400 \cdot \lceil \log 128 \rceil = 400 \cdot 7 = 2800$  Kopien jeder

#### 4 Evaluation der Strategien

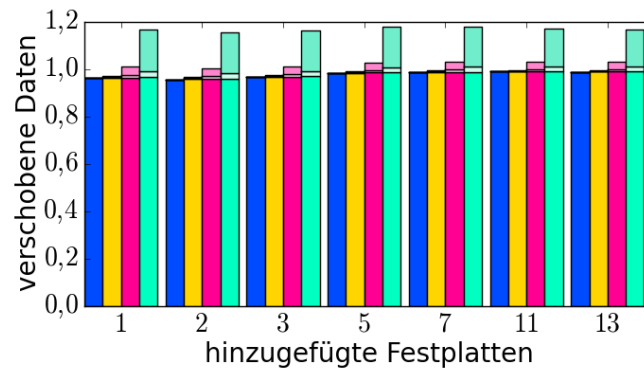


Abbildung 4.28: Adaptivität von *Consistent Hashing* mit fester Anzahl an Kopien jeder Festplatte beim Hinzufügen homogener Festplatten

Festplatte platziert. Nach dem Hinzufügen der neuen Festplatten hat sich diese Anzahl auf  $400 \cdot \lceil \log(128 + n') \rceil = 400 \cdot 8 = 3200$  erhöht, da  $\lceil \log(128 + 1) \rceil = 8 = \lceil \log(128 + 13) \rceil$  und  $1 \leq n' \leq 13$ . Dementsprechend werden neben den 3200 Kopien der neuen Festplatten 400 Kopien jeder ursprünglichen Festplatte platziert. Damit kann ein Block entweder auf eine neue Festplatte oder auf eine der hinzugekommenen Kopien der ursprünglichen Festplatten verschoben werden. Wird eine Festplatte nicht mehr für eine Kopie  $l \leq k$  verwendet, so kann sie stattdessen für eine Kopie  $l < l' \leq k$  verwendet werden, wodurch zusätzlich Replatzierungen mit Beachtung der Reihenfolge entstehen.

Wie Karger u. a. in [Karger u. a., 1997] gezeigt haben, ist *Consistent Hashing* 1-adaptiv. Daher werden beim Hinzufügen von Festplatten nur die Daten bewegt, welche auf der hinzugefügten Festplatte platziert werden. Ihr Beweis beruht auf der Annahme, dass von jeder Festplatte eine feste Anzahl an Kopien platziert wird. Ferner gehen sie von einer Verteilung ohne Redundanz aus. Daher habe ich meine Implementierung von *Consistent Hashing* auch unter dieser Bedingung vermessen. Unabhängig von der Anzahl der Festplatten habe ich von jeder Festplatte  $400 \cdot \lceil \log 128 \rceil = 2800$  Kopien platziert und darüber Daten verteilt. Die Ergebnisse sind in Abbildung 4.28 dargestellt.

Wie zuvor erhielt jede Festplatte einen fairen Anteil der Last innerhalb der Fairness von *Consistent Hashing* bei dieser Parametrisierung. Bei der nicht redundanten Platzierung wurden exakt die Daten repliziert, die auf dem neu hinzugekommenen Speicher platziert wurden. Somit belegt meine Implementierung das von Karger u. a. gezeigte Verhalten.

Bei der Platzierung mehrerer Kopien jedes Blocks waren zusätzliche Replatzierungen notwendig. Bei der Platzierung von zwei Kopien lag der Faktor der Replatzierungen ohne Beachtung der Reihenfolge der Kopien um 0,0037 über dem Faktor der auf den neuen Festplatten platzierten Daten beim Hinzufügen einer Festplatte. Diese Differenz sank monoton auf 0,0031 beim Hinzufügen von 13 Festplatten. Bei der Platzierung von vier Kopien jedes Blocks lag diese



Differenz zwischen 0,0105 und 0,0091, beim Platzieren von acht Kopien zwischen 0,0234 und 0,198.

Weiterhin waren zusätzliche Replatzierungen mit Beachtung der Reihenfolge der Kopien der Festplatten nötig. Bei der Platzierung von zwei Kopien stieg der Faktor mit Beachtung der Reihenfolge der Kopien gegenüber dem Faktor ohne Beachtung der Reihenfolge um 0,0040 beim Hinzufügen einer Festplatte. Diese Differenz sank wieder monoton auf 0,0036 beim Hinzufügen von 13 Festplatten. Bei der Platzierung von vier Kopien lag diese Differenz zwischen 0,0339 und 0,0302, bei der Platzierung von acht Kopien zwischen 0,1780 und 0,1565.

Wie zu Anfang dieses Kapitels erklärt arbeitet meine Implementierung von *Consistent Hashing* mit einer Schleife. In jedem Schritt macht sie ein unabhängiges Zufallsexperiment und erhält dafür eine Festplatte gemäß der nicht redundanten Beschreibung von *Consistent Hashing* aus Abschnitt 3.2. Wurde auf dieser Festplatte noch keine Kopie des Blocks platziert, so wird nun auf der Festplatte eine Kopie platziert, andernfalls wird das Experiment verworfen. Der Algorithmus terminiert, wenn  $k$  unterschiedliche Festplatten gefunden wurden.

Wird eine Kopie nun auf eine der neuen Festplatten verschoben, bleiben die restlichen Kopien im Allgemeinen an ihrem bisherigen Platz. Es kann allerdings passieren, dass die zuvor ausgewählte Festplatte nun für eine spätere Kopie ausgewählt wird, falls sie zuvor mehrfach ausgewählt wurde. Dies würde in eine zusätzliche Replatzierung bei Beachtung der Reihenfolge der Kopien resultieren. Weiterhin sind zusätzliche Replatzierungen notwendig, falls eine neue Festplatte mehrere Punkte überdeckt, an welchen zuvor Kopien platziert wurden. In diesem Fall entstehen zusätzliche Experimente. Liefern diese Experimente eine bereits in der alten Konfiguration ausgewählte Festplatte, so ist nur bei Beachtung der Reihenfolge der Kopien eine Replatzierung notwendig. Wird dagegen eine zuvor nicht für den Block verwendete Festplatte ausgewählt, so ist auch ohne Berücksichtigung der Reihenfolge eine Replatzierung notwendig.

## Share

Für die Vermessung der Adaptivität meiner Implementierung von *Share* habe ich einen Dehnungsfaktor von  $5 \cdot \log n$  verwendet. Die unterliegenden *Consistent Hashing* Funktionen haben  $400 \cdot \lceil \log n' \rceil$  Kopien jeder im entsprechenden Intervall liegenden Abbildung einer Festplatte platziert.  $n'$  ist dabei die Anzahl der Abbildungen von Festplatten im entsprechenden Intervall. Mittels dieser Parameter habe ich *Share* mit den unterschiedlichen Konfigurationen initialisiert und die Fairness vermessen. Die Ergebnisse sind in Abbildung 4.29 dargestellt.

*Share* weist den neuen Festplatten einen fairen Anteil gemäß der in Abschnitt 4.2.2 gezeigten Bereiche für eine Konfiguration mit 128 Festplatten zu. Bei der nicht redundanten Platzierung lagen die Lastfaktoren zwischen 0,90 und 1,16 bei einer Standardabweichung von 0,05. Der Faktor der auf den neuen Festplatten platzierten Daten gegenüber dem Ideal  $\lambda$  lag über alle Tests hinweg zwischen 0,98 und 1,03.

#### 4 Evaluation der Strategien

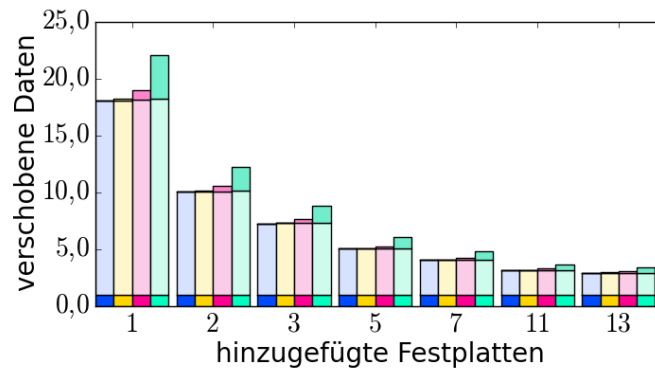


Abbildung 4.29: Adaptivität von *Share* beim Hinzufügen homogener Festplatten

Die zusätzlichen Replatzierungen ohne Beachtung der Reihenfolge stiegen bei konstanter Anzahl an Kopien jedes Blocks je mehr Festplatten hinzugefügt wurden. Allerdings stieg die Anzahl der Replatzierungen wesentlich langsamer, als die Anzahl der hinzugefügten Festplatten, so dass der Faktor der replazierten Daten gegenüber  $\lambda$  schrumpfte. Bei der nicht redundanten Platzierung wurde beim Hinzufügen einer Festplatte das 18,05-fache der im Ideal zu replazierenden Daten verschoben. Dieser Faktor sank monoton auf 2,95 beim Hinzufügen von 13 Festplatten.

Durch Platzierung mehrerer Kopien jedes Blocks erhöhte sich dieser Faktor leicht. Beim Hinzufügen einer Festplatte stieg er bei der Platzierung von zwei Kopien jedes Blocks auf 18,11, bei Platzierung von vier Kopien auf 18,16 und bei der Platzierung von acht Kopien auf 18,22. Beim Hinzufügen von 13 Festplatten wurde auch bei der redundanten Platzierung das 2,95-fache des Ideals ohne Beachtung der Reihenfolge der Kopien replaziert. Dieser Faktor schwankte nur in der dritten Nachkommastelle.

Bei der redundanten Platzierung von  $k \geq 2$  Kopien kamen zusätzliche Replatzierungen bei Beachtung der Reihenfolge der Kopien hinzu. Bei der Platzierung von zwei Kopien stieg der Faktor der Replatzierungen mit Beachtung der Reihenfolge der Kopien um 0,14 gegenüber dem Faktor ohne Beachtung der Reihenfolge. Diese Differenz stieg bei wachsender Anzahl platzierter Kopien. Bei vier platzierten Kopien lag die Differenz bei 0,85 und bei acht platzierten Kopien bei 3,90. Bei steigender Anzahl hinzugefügter Festplatten sank die Differenz bei konstanter Anzahl an Kopien monoton. Beim Hinzufügen von 13 Festplatten lag sie bei der Platzierung von zwei Kopien bei 0,02, bei der Platzierung von vier Kopien bei 0,11 und bei der Platzierung von acht Kopien bei 0,46.

Die zusätzlichen Replatzierungen treten aus verschiedenen Gründen auf. Zunächst entsteht ein Teil der Replatzierungen durch *Consistent Hashing* und die veränderliche Anzahl von Kopien jedes Festplattenabbilds. Dies habe ich bereits bei der Vermessung der Adaptivität von *Consistent Hashing* in diesem Abschnitt erklärt. Hinzu kommt noch der veränderliche Dehnungsfaktor

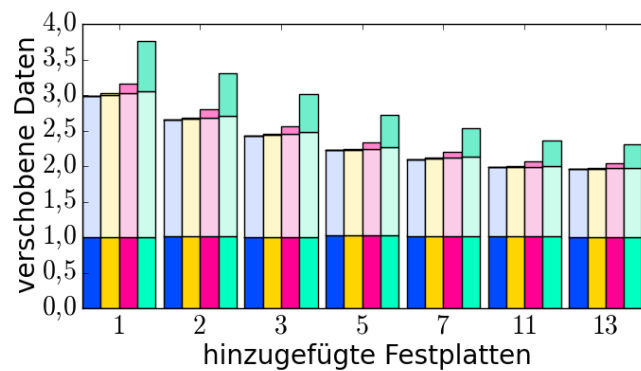


Abbildung 4.30: Adaptivität von *Share* mit festem Dehnungsfaktor beim Hinzufügen homogener Festplatten

von *Share*, welcher die Intervalle nachträglich verändert und daher weitere Replatzierungen verursacht.

Um die Adaptivität von *Share* zu beschränken ist es notwendig den Dehnungsfaktor unabhängig von der Anzahl der Festplatten zu wählen. Weiterhin ist es auch notwendig die Anzahl der Kopien jedes Festplattenabbilds in den *Consistent Hashing* Verteilungen unabhängig von der Anzahl der platzierten Abbilder (und unabhängig von der Anzahl der Festplatten) zu wählen. Daher habe ich *Share* mit einem festen Dehnungsfaktor von  $5 \cdot \log 128 = 35$  vermessen. Die Anzahl der platzierten Kopien jedes Festplattenabbildes habe ich fest auf  $400 \cdot \lceil \log 128 \rceil = 2800$  gesetzt. Die Ergebnisse sind in Abbildung 4.30 dargestellt.

Wie zuvor liegt der Faktor der replazierten Daten gegenüber dem Ideal im Bereich der in Abschnitt 4.2.2 gezeigten Fairness von *Share* bei entsprechender Konfiguration. Bei allen Tests lag  $\frac{\lambda'}{\lambda}$  zwischen 0,9982 und 1,0299.

Der Faktor der replazierten Daten ohne Beachtung der Reihenfolge der Kopien ist gesunken. Bei der nicht redundanten Platzierung lag der Faktor beim Hinzufügen einer Festplatte bei 2,99 und sank bei steigender Anzahl hinzugefügter Festplatten monoton auf 1,96 beim Hinzufügen von 13 Festplatten.

Durch Erhöhung der Anzahl der platzierten Kopien stieg der Faktor monoton. Beim Hinzufügen einer Festplatte lag der Faktor bei Platzierung von zwei Kopien bei 3,00, bei Platzierung von vier Kopien bei 3,02 und bei der Platzierung von acht Kopien bei 3,06. Bei konstanter Anzahl Kopien sank der Faktor auch bei Platzierung mehrerer Kopien bei zunehmender Anzahl hinzugefügter Festplatten. Beim Hinzufügen von 13 Festplatten lag der Faktor bei der Platzierung von zwei Kopien bei 1,96, bei der Platzierung von vier Kopien bei 1,97 und bei der Platzierung von acht Kopien bei 1,98.

Bei der redundanten Platzierung waren mit Beachtung der Reihenfolge der Kopien weitere Replatzierungen notwendig. Beim Hinzufügen einer Festplatte und der Platzierung von zwei Ko-

## 4 Evaluation der Strategien

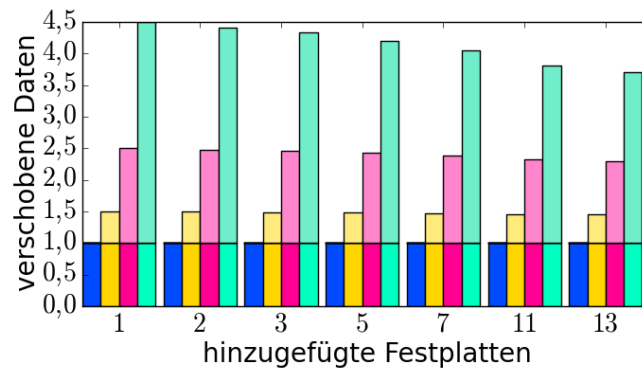


Abbildung 4.31: Adaptivität von *Redundant Share* beim Hinzufügen homogener Festplatten

Bei jedem Block lag der Faktor mit Beachtung der Reihenfolge der Kopien um 0,02 über dem Faktor ohne Beachtung der Reihenfolge. Diese Differenz stieg auf 0,14 bei der Platzierung von vier Kopien und auf 0,71 bei der Platzierung von acht Kopien. Durch Hinzufügen mehrerer Festplatten konnte die Differenz bei konstanter Anzahl an Kopien jedes Blocks monoton gesenkt werden. Beim Hinzufügen von 13 Festplatten lag die Differenz bei der Platzierung von zwei Kopien bei 0,01, bei der Platzierung von vier Kopien bei 0,07 und bei der Platzierung von acht Kopien bei 0,33.

Während *Consistent Hashing* bei der nicht redundanten Platzierung keine Replatzierungen außer auf die neu hinzukommenden Festplatten benötigt, kann *Share* dies nicht sicherstellen. Zwar ist der Dehnungsfaktor in *Share* konstant, allerdings verändern sich trotzdem die von jeder Festplatte abgedeckten Intervalle abhängig von der Anzahl der Festplatten in der Konfiguration. Wie die Messungen belegen können diese Bewegungen allerdings vermindert werden, indem der Dehnungsfaktor und die Anzahl der Kopien jedes Festplattenabbilds in den unterliegenden *Consistent Hashing* Implementierungen fest gesetzt werden.

### Redundant Share

*Redundant Share* ordnet, wie in Abschnitt 3.4 beschrieben, die Festplatten in einer Reihe an, um dann Experimente für die einzelnen Festplatten durchzuführen, bis alle Kopien platziert wurden. Die Festplatten sind dabei absteigend gemäß ihrer Kapazität sortiert. Wie in Abschnitt 3.4.1 gezeigt, unterscheidet sich die Adaptivität von *Redundant Share*, je nachdem an welcher Position neue Festplatten in diese Reihe eingefügt werden. Bei einer homogenen Konfiguration kann dies an beliebiger Stelle passieren. Die geringste Adaptivität wird erzielt, wenn die neuen Festplatten an erster Position eingefügt werden. Daher habe ich die Adaptivität von *Redundant Share* zunächst im besten Fall vermessen, habe also die neuen Festplatten vor die ursprünglichen Festplatten in der Reihe platziert. Die Ergebnisse sind in Abbildung 4.31 dargestellt.

*Redundant Share* hat jeder Festplatte ihren fairen Anteil gemäß der in Abschnitt 4.2.2 gezeigten

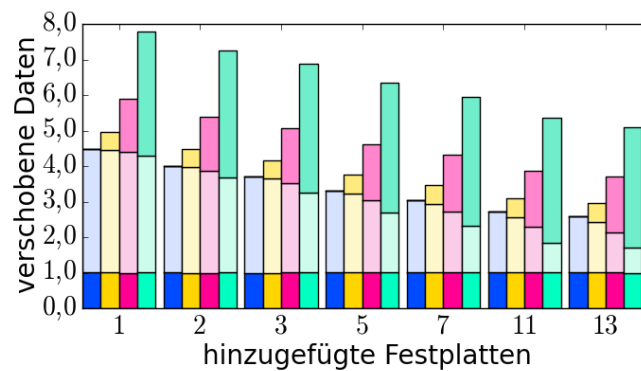


Abbildung 4.32: Adaptivität von *Redundant Share* beim Hinzufügen homogener Festplatten an schlechtesten Position

Abweichung für eine Konfiguration mit 128 Festplatten zugewiesen. Tatsächlich wurde sogar eine bessere Verteilung als bei der Vermessung der Fairness erzielt, da das Zehnfache an Daten platziert wurde.

Ohne Beachtung der Reihenfolge der Kopien waren keine zusätzlichen Replatzierungen notwendig. Dies deckt sich mit den Ergebnissen aus Abschnitt 3.4.3. Zusätzliche Bewegungen waren nur notwendig, wenn die Reihenfolge der Kopien beibehalten werden musste.

Mit Beachtung der Reihenfolge der Kopien war bei der Platzierung von zwei Kopien jedes Blocks beim Hinzufügen einer Festplatte ein Faktor von 1,50 Replatzierungen gegenüber einer optimalen Adaptivität nötig. Bei konstanter Anzahl platzierter Kopien sank dieser Faktor monoton auf 1,46 beim Hinzufügen von 13 Festplatten. Bei Platzierung von vier Kopien fiel der Faktor von 2,50 auf 2,29 und bei der Platzierung von acht Kopien von 4,49 auf 3,70.

Mit Beachtung der Reihenfolge der  $k$  Kopien sollte die Adaptivität beim Einfügen einer homogenen Festplatte bei  $\frac{k+1}{2}$  liegen. Dies habe ich in Lemma 3.4.21 in Abschnitt 3.4.3 gezeigt. Bei der Platzierung von zwei Kopien sollte der Faktor also bei 1,5, bei vier Kopien bei 2,5 und bei acht Kopien bei 4,5 liegen. Dies deckt sich mit den gemessenen Ergebnissen.

*Redundant Share* fügt neue homogene Festplatten immer an den Anfang der zu durchlaufenden Liste an. Dadurch wahrt *Redundant Share* die gezeigte Adaptivität. Ich habe allerdings in Abschnitt 3.4.1 in Lemma 3.4.6 gezeigt, dass der schlimmste Fall für die erwartete Adaptivität eintritt, wenn eine neue homogene Festplatte am Ende der Liste einsortiert wird. Um dies zu vermessen habe ich *Redundant Share* so abgewandelt, dass die neuen Festplatten am Ende der Liste eingefügt werden und diese Parametrisierung vermessen. Die Ergebnisse sind in Abbildung 4.32 dargestellt.

Auch in dieser Parametrisierung hat *Redundant Share* jeder Festplatte ihren fairen Anteil geliefert. Die Faktoren der auf die neuen Festplatten übertragenen Daten gegenüber einer idealen Verteilung lagen in den gleichen Bereichen wie zuvor.

#### 4 Evaluation der Strategien

Im Gegensatz zu den vorherigen Tests waren nun allerdings auch bei der nicht redundanten Platzierung zusätzliche Replatzierungen notwendig. Je nach Anzahl der hinzugefügten Festplatten sank der Faktor der Replatzierungen gegenüber dem Ideal von 4,48 beim Hinzufügen einer Festplatte monoton auf 2,59 beim Hinzufügen von 13 Festplatten.

Der Faktor an Replatzierungen ohne Beachtung der Reihenfolge der Kopien fiel leicht bei steigender Anzahl platzierter Kopien. Bei zwei platzierten Kopien jedes Blocks fiel der Faktor von 4,45 beim Hinzufügen einer Festplatte auf 2,43 beim Hinzufügen von 13 Festplatten, bei der Platzierung von vier Kopien von 4,39 auf 2,14 und bei der Platzierung von acht Kopien von 4,28 auf 1,71.

Ich habe in Abschnitt 3.4.3 Lemma 3.4.25 gezeigt, dass die Adaptivität von *Redundant Share* ohne Beachtung der Reihenfolge der Kopien bei  $\ln n$  liegt, falls zu  $n$  Festplatten eine weitere Festplatte hinzugefügt wird. Dies ergibt eine erwartete Adaptivität von  $\ln 128 \approx 4,85$ . Die gemessenen Ergebnisse decken sich mit diesem Wert.

Bei der redundanten Platzierung von  $k \leq 2$  Kopien jedes Blocks waren weitere Replatzierungen mit Beachtung der Reihenfolge der Kopien notwendig. Bei der Platzierung von zwei Kopien lag der Faktor bei 4,95 gegenüber dem Ideal, also um 0,50 über dem Faktor ohne Beachtung der Reihenfolge. Für eine steigende Anzahl an hinzugefügten Festplatten fiel der Faktor monoton auf 2,97 beim Hinzufügen von 13 Festplatten. Gleichzeitig stieg die Differenz zwischen dem Faktor bei Platzierung ohne gegenüber dem Faktor mit Beachtung der Reihenfolge der Kopien monoton auf 0,54. Bei der Platzierung von vier Kopien fiel der Faktor von 5,89 monoton auf 3,72. Die Differenz gegenüber der Adaptivität ohne Beachtung der Reihenfolge der Kopien lag damit bei einer hinzugefügten Festplatte bei 1,50. Diese Differenz stieg monoton auf 1,59 beim Hinzufügen von sieben Festplatten und fiel dann wieder monoton auf 1,58 beim Hinzufügen von 13 Festplatten. Bei Platzierung von acht Kopien fiel der Faktor der Replatzierungen mit Beachtung der Reihenfolge der Kopien von 7,78 monoton auf 5,10. Die Differenz zum Faktor ohne Beachtung der Reihenfolge der Kopien lag damit beim Hinzufügen einer Festplatte bei 3,50. Diese Differenz stieg monoton auf 3,66 beim Hinzufügen von fünf Festplatten und sank dann monoton auf 3,39 für das Hinzufügen von 13 Festplatten.

Gemäß Lemma 3.4.26 aus Abschnitt 3.4.3 sollte die Adaptivität beim Hinzufügen einer Festplatte und Platzierung von  $k$  Kopien bei  $\frac{k+1}{2} \cdot \ln n$  liegen. Dies würde bei der Platzierung von zwei Kopien eine Adaptivität von 7,28 bedeuten. Bei vier Kopien sollte die Adaptivität bei 12,13 liegen und bei acht Kopien bei 21,83. Die gemessenen Adaptivitäten bleiben weit unter diesen Werten. In dem Beweis von Lemma 3.4.26 habe ich den schlimmsten Fall angenommen, indem ich davon ausgegangen bin, dass die ohne Beachtung der Reihenfolge replatzierten Daten jeweils die zuerst gefundenen Kopien eines Blocks waren. Dies spiegelt sich jedoch in der Realität nicht wieder.

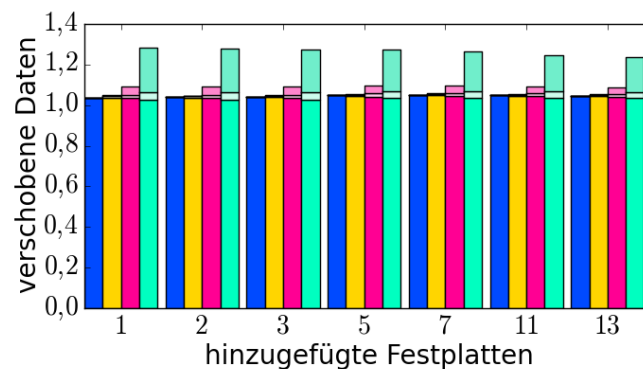


Abbildung 4.33: Adaptivität von *Consistent Hashing* mit fester Anzahl an Kopien jeder Festplatte beim Hinzufügen heterogener Festplatten

### 4.3.2 Adaptivität beim Einfügen heterogener Festplatten

Werden nachträglich Festplatten einem bestehenden Speichersystem hinzugefügt, so haben die neuen Festplatten oft nicht die gleiche Kapazität wie die ursprünglich im System verwendeten Festplatten. Da die durchschnittliche Kapazität einer Festplatte ständig steigt, ist es sehr wahrscheinlich, dass neue Festplatten eine höhere Kapazität haben als alte. Daher habe ich für diesen Abschnitt die Adaptivität der Implementierungen der verschiedenen Verteilungsalgorithmen vermessen, wenn Festplatten hinzugefügt werden, welche die 1,5-fache Kapazität der ursprünglichen Festplatten haben.

#### Consistent Hashing

Wie bereits in Abschnitt 4.2.3 gezeigt, benötigt *Consistent Hashing* eine kapazitätsabhängige Anzahl an Kopien jeder Festplatte um eine konstante Fairness auch über heterogenen Festplatten zu ermöglichen. Weiterhin habe ich in Abschnitt 4.3.1 gezeigt, dass die Adaptivität von *Consistent Hashing* verschlechtert wird, falls die Anzahl der pro Festplatte platzierten Kopien von der Anzahl der Festplatten abhängt. Daher habe ich für die Messungen mit heterogenen Konfigurationen *Consistent Hashing* so parametrisiert, dass von jeder Festplatte  $d_i \lceil \frac{c_i}{100.000} \rceil \cdot 560$  Kopien angelegt werden,  $c_i$  beschreibt dabei die Kapazität der Festplatte, also 500.000 für die ursprünglichen Festplatten und 750.000 für die neu hinzugefügten Festplatten. Die Ergebnisse dieses Tests sind in Abbildung 4.33 dargestellt.

*Consistent Hashing* weist den hinzugefügten Festplatten ihren fairen Anteil innerhalb der in Abschnitt 4.2.3 gezeigten Schranken zu. Allerdings ist dieser Anteil immer über dem Ideal. Bei der nicht redundanten Platzierung lag der Faktor der auf den neuen Festplatten platzierten Daten gegenüber dem Ideal zwischen 1,04 und 1,05. Über alle Tests hinweg lagen die Faktoren zwischen 1,03 und 1,05.

## 4 Evaluation der Strategien

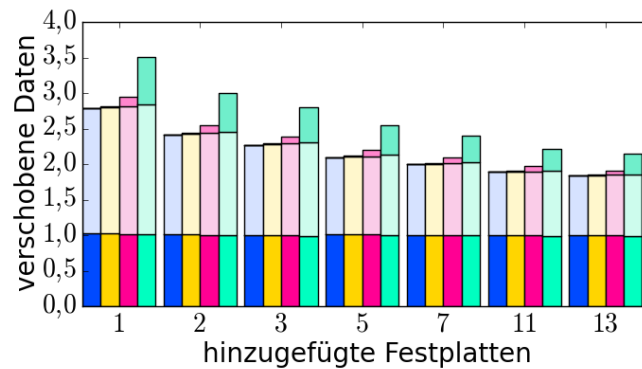


Abbildung 4.34: Adaptivität von *Share* mit festem Dehnungsfaktor beim Hinzufügen heterogener Festplatten

Die neu hinzugekommenen Festplatten haben eine Kapazität von 750.000 Blöcken. Meine Implementierung legt von jeder dieser Festplatten  $\lceil \frac{750.000}{100.000} \cdot 560 \rceil$  Kopien an. Damit werden die Festplatten von *Consistent Hashing* so behandelt, als hätten sie eine Kapazität von 800.000 Blöcken und erhalten dadurch mehr Daten, als im Idealfall.

Bei der nicht redundanten Platzierung gab es keine zusätzlichen Replatzierungen innerhalb des Speichersystems.

Beim Hinzufügen einer Festplatte zu einem 2-redundanten Verteiler lag der Faktor der Replatzierungen ohne Beachtung der Reihenfolge der Kopien um 0,0063 über dem Faktor der auf die neuen Festplatten verschobenen Daten. Diese Differenz sank monoton auf 0,0048 beim Hinzufügen von 13 Festplatten. Beim Platzieren von vier Kopien lag diese Differenz zwischen 0,0180 und 0,0138 und für acht Kopien zwischen 0,0402 und 0,0297. In allen Fällen wurde die Differenz monoton kleiner je mehr Festplatten hinzugefügt wurden.

Mit Beachtung der Reihenfolge der Kopien waren weitere Replatzierungen notwendig. Beim Hinzufügen von einer Festplatte und Platzierung von zwei Kopien jedes Blocks lag der Faktor der Replatzierungen mit Beachtung der Reihenfolge der Kopien um 0,0042 über dem Faktor ohne Beachtung der Reihenfolge. Diese Differenz sank monoton und lag beim Hinzufügen von 13 Festplatten bei 0,0036. Beim Platzieren von vier Kopien lag die Differenz zwischen 0,0404 und 0,0328 und beim Platzieren von acht Kopien zwischen 0,2181 und 0,1719.

Der Mehraufwand liegt innerhalb der bereits in Abschnitt 4.3.1 bei der Betrachtung von *Consistent Hashing* über homogene Festplatten erläuterten Bereiche.

### Share

Ich habe in Abschnitt 4.3.1 gezeigt, dass *Share* zur Wahrung einer begrenzten Adaptivität einen konstanten Dehnungsfaktor und eine konstante Anzahl an Kopien jedes Festplattenabbildes in



den unterliegenden *Consistent Hashing* Verteilungen benötigt. Daher habe ich *Share* wie zuvor mit einem festen Dehnungsfaktor von 35 und mit 2800 Kopien jedes Festplattenabbildes in den unterliegenden *Consistent Hashing* Implementierungen vermessen. Die Ergebnisse der Adaptivität von *Share* beim Hinzufügen von heterogenen Festplatten sind in Abbildung 4.34 dargestellt.

Wie zuvor ist der Faktor der tatsächlich auf die neuen Festplatten verschobenen Daten gegenüber dem Ideal innerhalb der Bereiche, die auch schon bei der Betrachtung der Fairness von *Share* bei einer Konfiguration mit 128 Festplatten zu beobachten waren. Der Faktor  $\frac{\lambda'}{\lambda}$  lag bei allen Tests zwischen 0,99 und 1,02.

Die Faktoren der replazierten Daten ohne Beachtung der Reihenfolge der Kopien gegenüber dem Ideal lag knapp unterhalb der im homogenen Fall gemessenen Faktoren. Beim Hinzufügen einer Festplatte lag der Faktor bei nicht redundanter Platzierung bei 2,78 und stieg bei steigender Anzahl der platzierten Kopien auf 2,79 bei zwei Kopien, auf 2,81 bei vier Kopien und auf 2,84 bei acht Kopien jedes Blocks. Bei konstanter Anzahl Kopien jedes Blocks sank der Faktor je mehr neue Festplatten hinzugefügt wurden. Bei 13 hinzugefügten Festplatten lag der Faktor ohne Redundanz bei 1,84 und bei acht platzierten Kopien bei 1,85.

Bei redundanter Speicherung der Blöcke kamen wieder zusätzliche Replatzierungen hinzu, wenn die Reihenfolge der Kopien beachtet werden musste. Der Faktor der Replatzierungen gegenüber dem Ideal lag mit Beachtung der Reihenfolge um 0,02 über dem Faktor der Replatzierungen ohne Beachtung der Reihenfolge bei der Platzierung von zwei Kopien wenn eine neue Festplatte hinzukam. Bei der Platzierung von vier Kopien lag die Differenz bei 0,13 und bei acht Kopien bei 0,66. Diese Differenz sank wieder je mehr Festplatten hinzugefügt wurden. Beim Hinzufügen von 13 Festplatten lag die Differenz bei der Platzierung von zwei Kopien bei 0,01, bei der Platzierung von vier Kopien bei 0,06 und bei der Platzierung von acht Kopien bei 0,29. Ich habe diese zusätzlichen Replatzierungen bereits in Abschnitt 4.3.1 für den homogenen Fall begründet.

### Redundant Share

Die Ergebnisse der Vermessung der Adaptivität von *Redundant Share* beim Hinzufügen heterogener Festplatten zeigt Abbildung 4.35. *Redundant Share* weist auch bei heterogenen Festplatten jeder Festplatte ihren fairen Anteil zu. Die Faktoren der auf die neuen Festplatten verschobenen Daten gegenüber dem Ideal lagen über alle Tests hinweg zwischen 0,9975 und 1,0004.

Ohne Beachtung der Reihenfolge der Kopien waren keine weitere Replatzierungen notwendig. Die neuen Festplatten sind größer als die ursprünglichen Festplatten, weshalb die in Abschnitt 3.4.3 Lemma 3.4.23 gezeigte Adaptivität von 1 gilt. Dies stimmt mit den gemessenen Daten überein.

#### 4 Evaluation der Strategien

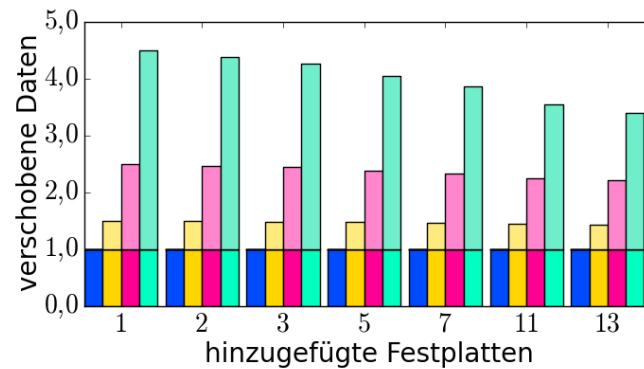


Abbildung 4.35: Adaptivität von *Redundant Share* beim Hinzufügen heterogener Festplatten

Mit Beachtung der Reihenfolge der Kopien waren bei der Platzierung von  $k \leq 2$  Kopien jedes Blocks zusätzliche Replatzierungen nötig. Bei der Platzierung von zwei Kopien fiel die gemessene Adaptivität von 1,50 beim Hinzufügen von einer Festplatte monoton auf 1,44 beim Hinzufügen von 13 Festplatten. Bei der Platzierung von vier Kopien sank der Faktor von 2,50 monoton auf 2,21 und bei der Platzierung von acht Kopien von 4,50 auf 2,41.

Auch diese Ergebnisse decken sich mit der in Abschnitt 3.4.3 gezeigten Adaptivität von *Redundant Share*. Da die neuen Festplatten größer als die ursprünglichen Festplatten sind, werden sie in jedem Fall an den Anfang der zu durchlaufenden Liste gestellt. Dadurch gilt für das Hinzufügen einer Festplatte mit Beachtung der Reihenfolge die in Lemma 3.4.23 gezeigte Adaptivität  $\frac{k+1}{2}$  bei der Platzierung von  $k$  Kopien.

## 5 Zusammenfassung

Ich habe mich in dieser Arbeit mit den Problemen skalierbarer Speichernetze beschäftigt. Ich habe gezeigt, welche verschiedenen Speichersysteme es gibt, wie verschiedene Speichernetze skalieren und bin anschließend tiefer auf verschiedene Verfahren zur Datenverteilung in skalierbaren Speichernetzen eingegangen.

In Kapitel 1 habe ich eine kurze Einführung in das Umfeld dieser Arbeit gegeben und die Ziele vorgestellt. Dort habe ich ins Besondere festgelegt, was ich unter einer *Cluster Blockvirtualisierung* und unter einem *Cluster Blockgerät* verstehe.

In Kapitel 2 habe ich verschiedene Speichersysteme betrachtet. In Abschnitt 2.1 habe ich einen Überblick über bestehende Speichersysteme gegeben. Speziell bin ich auf verschiedene *Erasur Codes*, Verfahren zur Cluster Blockvirtualisierung und auf objektbasierte Speichervirtualisierungen eingegangen.

In Abschnitt 2.2 habe ich einen Überblick über die Kernkomponenten von *V:Drive* gegeben. *V:Drive* ist eine Cluster Blockvirtualisierung, welche Cluster Blockgeräte erstellen kann, wie ich sie in Kapitel 1 eingeführt habe.

In Abschnitt 2.3 habe ich untersucht, wie eine Cluster Blockvirtualisierung über lokalen Festplatten skalieren kann. Dazu habe ich Speicherknoten mit lokalen Festplatten vorausgesetzt, deren Daten über ein Netzwerkprotokoll allen Speicherknoten zur Verfügung gestellt werden. Auf den unterschiedlichen Knoten werden Cluster Blockgeräte verwendet, deren Daten durch die Speicherknoten uniform über die Festplatten verteilt werden. Ich habe untersucht, wie so ein Speichersystem für verschiedene Arten der Redundanz skaliert. Dabei konnte ich zeigen, dass ein nicht redundantes Speichernetz mit einem halb-duplexen Netzwerk in etwa mit

$$\frac{n^2}{3 \cdot n - 2} \approx \frac{1}{3} \cdot n$$

und mit einem voll-duplexen Netzwerk mit

$$\frac{n^2}{2 \cdot n - 1} \approx \frac{1}{2} \cdot n$$

skaliert. Dabei ist  $n$  die Anzahl der Speicherknoten im Speichernetz. Bei der Platzierung von  $k$  Kopien jedes Blocks auf unterschiedlichen Speicherknoten skaliert so ein Speichernetz mit

$$\frac{n^2}{n + 2 \cdot k \cdot (n - 1)} \approx \frac{1}{1 + 2 \cdot k} \cdot n$$

## 5 Zusammenfassung

für ein halb-duplexes und mit

$$\frac{n^2}{(k+1) \cdot n - k} \approx \frac{1}{k+1} \cdot n$$

für ein voll-duplexes Netzwerk. Bei Verwendung eines *MDS-Codes*, welcher  $v$  unkodierte Blöcke auf  $u \geq v$  kodierte Blöcke erweitert, konnte ich zeigen, dass ein Speichernetz mit

$$\frac{v \cdot n^2}{u \cdot (3 \cdot n - 2)} \approx \frac{v}{3 \cdot u} \cdot n$$

mit einem halb-duplexen und mit

$$\frac{v \cdot n^2}{(v+u) \cdot n - u} \approx \frac{n}{v+u} \cdot n$$

mit einem voll-duplexen Netzwerk skaliert. Diese theoretischen Berechnungen habe ich mit der praktischen Vermessung von *V:Drive* in der passenden Konfiguration belegt. Dabei habe ich gezeigt, dass auf Grund der Implementierung des verwendeten *iSCSI-Targets* zusätzliche Kommunikation in dem Speichernetz notwendig war, wenn Daten mittels eines *MDS-Codes* redundant abgelegt werden sollten. Darunter litt auch die Skalierbarkeit des Speichernetzes.

Abgeschlossen habe ich Kapitel 2 indem ich in Abschnitt 2.4 auf die Probleme der redundanten Datenverteilung in *V:Drive* eingegangen bin. Ich habe gezeigt, dass die verwendete Datenverteilung keine Redundanz innerhalb einer Speichergruppe ermöglicht. Diese Problematik habe ich in Kapitel 3 aufgegriffen und gelöst.

Dazu habe ich in Abschnitt 3.1 ein Modell ausgearbeitet, welches meinen weiteren Überlegungen zu Grunde lag. Speziell habe ich auch die Anforderungen an ein Verfahren zur redundanten Datenplatzierung ausgearbeitet. Dies waren im Einzelnen die Zeit- und Speichereffizienz, die Fairness, die Adaptivität, die Redundanz und die Heterogenität.

In Abschnitt 3.2 habe ich verschiedene auf Hashfunktionen basierende Verfahren zur Datenverteilung in Speichernetzen vorgestellt. Keines dieser Verfahren konnte die Eigenschaft der Fairness bei der redundanten Datenplatzierung über einer heterogenen Konfiguration aufrecht erhalten. Vertiefend bin ich auf die Verfahren *Consistent Hashing* und *Share* eingegangen. *Consistent Hashing* eignet sich zur Datenverteilung über homogenen Festplatten. Dabei ist es auch möglich, mehrere Kopien fair zu verteilen. *Share* eignet sich zur nicht redundanten Verteilung über heterogenen Festplatten.

In Abschnitt 3.3 bin ich näher auf die Problematik der redundanten Datenverteilung eingegangen. Ich konnte zunächst eine Schranke für die maximale Heterogenität einer Konfiguration zeigen, wenn  $k$  Kopien von Blöcken darüber verteilt werden sollen. Der gesamte Speicherplatz der Festplatten lässt sich genau dann nutzen, wenn die größte Festplatte maximal einen Anteil von  $\frac{1}{k}$  des Gesamtspeichers ausmacht. Ich konnte auch zeigen, dass die mehrfache Verwendung nicht redundanter Verteiler zur redundanten Datenplatzierung keine faire Verteilung erreicht, falls eine heterogene Konfiguration verwendet wird.

Das Verfahren *Redundant Share* aus Abschnitt 3.4 löst das Problem der redundanten Datenverteilung über heterogenen Konfigurationen. Dazu habe ich zunächst vorgestellt, wie das Verfahren ohne Redundanz arbeitet. Danach bin ich auf die Platzierung von zwei Kopien jedes Blocks eingegangen, um zu zeigen wie *Redundant Share* die Redundanz herstellt. Dieses Verfahren habe ich dann erweitert und gezeigt, wie es zur Platzierung von  $k$  Kopien jedes Blocks verwendet werden kann. *Redundant Share* erfüllt alle Anforderungen an eine redundante Verteilungsfunktion, wie ich sie für eine *Cluster Blockvirtualisierung* wie V:Drive benötigt. Die Speichereffizienz ist mit  $O(1)$  hervorragend, allerdings ist die Laufzeit von  $O(n)$  ein Nachteil gegenüber anderen Verteilern. Ich konnte zeigen, dass *Redundant Share* eine Fairness von 1 besitzt. Weiterhin konnte ich zeigen, dass die Adaptivität je nach Einsatzart zwischen 1 und  $\frac{k+1}{2} \cdot \ln n$  liegt. Abgeschlossen habe ich den Abschnitt mit der Betrachtung von *Fast Redundant Share*. Während *Redundant Share* eine lineare Laufzeit gemäß der Anzahl der Festplatten benötigt, kommt *Fast Redundant Share* mit einer linearen Laufzeit gemäß der Anzahl der Kopien aus. Allerdings hat *Fast Redundant Share* dafür einen wesentlich höheren Hauptspeicherverbrauch von  $O(k \cdot n \cdot v)$ . Dabei ist  $v$  der Hauptspeicherbedarf eines nicht redundanten, heterogenen Verteilers.

Abgeschlossen habe ich Kapitel 3 indem ich in Abschnitt 3.5 *Redundant Share* mit einer fehlerhaften Sicht auf die Konfiguration betrachtet habe. Dies ist beispielsweise in *Peer-to-Peer* Umgebungen relevant. Dabei bin ich davon ausgegangen, dass ein Verteiler initialisiert wird, welcher nicht alle Festplatten eines ursprünglichen Verteilers erhält. Ich habe betrachtet, wie Daten wieder gefunden werden und *Peer Replication*, eine für dieses Szenario optimierte Adaption von *Redundant Share*, vorgestellt.

In Kapitel 4 habe ich die Ergebnisse der Vermessung von Implementierungen von *Consistent Hashing*, *Share*, *Redundant Share* und *Fast Redundant Share* dargestellt. Zunächst bin ich in Abschnitt 4.1 auf die Laufzeit und den Hauptspeicherverbrauch eingegangen. Die verschiedenen Implementierungen lagen dabei innerhalb der erwarteten Bereiche. Ich konnte hier zeigen, dass die Implementierung von *Fast Redundant Share* sich nicht zur redundanten Datenplatzierung in skalierbaren Speichernetzen eignet, da der Hauptspeicherverbrauch zu groß ist. Ich konnte auch zeigen, dass der Hauptspeicherverbrauch von *Share* kritisch ist, speziell wenn Dehnungsfaktoren eingesetzt werden, welche eine gute Fairness garantieren.

In Abschnitt 4.2 habe ich die Fairness der verschiedenen Implementierungen untersucht. Dabei bin ich zunächst auf den Einfluss der Anzahl der platzierten Blöcke und Kopien eingegangen. Danach habe ich verschiedene homogene und heterogene Konfigurationen untersucht und konnte die theoretisch gezeigten Schranken praktisch untermauern. *Redundant Share* konnte als einziges der eingesetzten Verfahren eine beliebig gute Fairness erreichen, indem genügend Daten verteilt wurden. Sowohl *Consistent Hashing* als auch *Share* konnten dies nicht leisten, da durch die pseudorandomisierte Anordnung der Festplatten ein konstanter Fehler übrig blieb.

In Abschnitt 4.3 habe ich die Adaptivität beim Einfügen von Festplatten betrachtet. Dazu habe ich einer bestehende Konfiguration eine unterschiedliche Menge homogener oder heterogener

Festplatten hinzugefügt. Auch hier konnte ich zeigen, dass die theoretischen Ergebnisse in der Praxis erkennbar sind. Speziell konnte ich für *Redundant Share* eine Adaptivität sehr nahe der zuvor gezeigten Grenzen darstellen. Lediglich die Adaptivität mit Beachtung der Reihenfolge für das Hinzufügen einer Festplatte in eine homogene Konfiguration an ungünstigster Stelle konnte ich nicht erreichen. In meinen Messung erreichte *Redundant Share* eine wesentlich bessere Adaptivität als  $\frac{k+1}{2} \cdot \ln n$ , wie ich sie in Abschnitt 3.4 gezeigt habe.

### 5.1 Ausblick

Es gibt auch nach dieser Arbeit noch verschiedene offene Probleme bei der redundanten Datenplatzierung in skalierbaren Speichernetzen. Ein Schwachpunkt von *Redundant Share* liegt in der notwendigen Berechnung adaptierter Kapazitäten von Festplatten um die Fairness zu gewährleisten. In Abschnitt 3.4.3 habe ich gezeigt, dass bei  $k$ -redundanter Platzierung bis zu  $k - 1$  adaptierte Kapazität berechnet werden müssen. Die Laufzeit dieser Berechnungen ist mit  $O(k^2 \cdot n^k)$  für  $n$  Festplatten zu hoch, um in großen Speichersystemen schnelle Anpassungen zu ermöglichen. Zwar kann die Notwendigkeit dieser Adaptionen in realen Systemen durch die Verwendung passender Konfigurationen vermieden werden, allerdings würde das Verfahren damit den Anspruch verlieren, auf beliebigen heterogenen Konfigurationen eine optimale Verteilung zu erzeugen. Hier wäre es sinnvoll zu untersuchen, ob sich der Fehler durch einen einfacher zu berechnenden Wert begrenzen lässt.

Auch die Laufzeit von *Redundant Share* ist in großen Speichernetzen problematisch. Ich habe in Abschnitt 4.2.2 gezeigt, dass *Redundant Share* auf den von mir verwendeten Rechnern durchschnittlich 16 ms benötigte, um die Platzierung von acht Kopien über 8192 Festplatten zu berechnen. Das Problem liegt in der linearen Laufzeit gemäß der Anzahl der Festplatten. Hier wäre zu untersuchen, ob sich Verfahren mit einer besseren Laufzeit finden lassen. Zwar konnte ich die Verteilung in Abschnitt 3.4.4 mittels nicht redundanter Hashfunktionen beschleunigen, allerdings habe ich in Abschnitt 4.1.1 gezeigt, dass sich dieses Verfahren wegen seines hohen Hauptspeicherverbrauchs nur für Konfigurationen mit wenigen Festplatten eignet.

In Abschnitt 3.4.3 konnte ich für *Redundant Share* eine Adaptivität von  $\frac{k+1}{k} \cdot \ln n$  bezüglich des Einfügens einer beliebigen Festplatte zeigen. Zur Berechnung dieser Schranke bin ich davon ausgegangen, dass alle Daten, die ohne Beachtung der Reihenfolge der Kopien replaziert wurden, Kopien  $k$  waren. Dadurch habe ich eine Schranke erhalten, welche ich in den praktischen Tests in Abschnitt 4.3.1 weit unterschreiten konnte. Ein Ansatz, um eine bessere Schranke zu finden, könnte darin liegen zu betrachten, wie viele Kopien  $l \in \{k, \dots, 1\}$  ohne Beachtung der Reihenfolge der Kopien verschoben werden. Dies hängt jedoch davon ab, an welcher Stelle eine Festplatte eingefügt wird. Im nächsten Schritt müsste die Anzahl der einzelnen Kopien mit  $\frac{l+1}{2}$  multipliziert werden, um so eine Schranke für die Umplatzierungen mit Beachtung der Reihenfolge der Kopien zu finden.

Neben diesen offenen Fragestellungen besteht auch noch ein Bedarf an der genaueren Vermessung der Verfahren im realen Umfeld. Eine Einbettung der Verfahren in *V:Drive*, mittels welcher sich die Verfahren innerhalb einer echten CBV einsetzen lassen, ist in Vorbereitung. Auch der Vergleich der vorgestellten Verfahren mit weiteren Verteilungsalgorithmen ist ein interessanter offener Punkt. Hier bieten sich die Verfahren des Speichersystems *CRUSH*, wie in [Weil u. a., 2006b] vorgestellt, oder *SPREAD*, wie in [Mense und Scheideler, 2008] vorgestellt, an.





# Abbildungsverzeichnis

1.1	Virtualisierung innerhalb eines Servers . . . . .	2
1.2	Mehrere Server mit gemeinsamem Speicher . . . . .	2
1.3	Mehrere Server mit gemeinsamem, zentral virtualisiertem Speicher . . . . .	3
1.4	Mehrere Server mit gemeinsamem, verteilt virtualisiertem Speicher . . . . .	3
2.1	Erasur Codierung eines virtuellen Laufwerks mit $v = 3$ und $u = 4$ . . . . .	11
2.2	Aufbau eines Speichersystems mit <i>V:Drive</i> . . . . .	22
2.3	Repräsentation der Speicherkonfigurationen im Metadatenserver . . . . .	24
2.4	Typische inter-Server Kommunikation bei Verwendung lokaler Festplatten . . . . .	28
2.5	Skalierung des Speichersystems ohne Redundanz . . . . .	37
2.6	Skalierung des Speichersystems mit langsamen Festplatten ohne Redundanz . . . . .	38
2.7	Skalierung des Speichersystems mit <i>RAID 1</i> . . . . .	39
2.8	Skalierung des Speichersystems mit <i>RAID 5</i> . . . . .	40
3.1	Aufbau der virtuellen Laufwerke in <i>V:Drive</i> . . . . .	44
3.2	Datenverteilung bei <i>Consistent Hashing</i> . . . . .	49
3.3	Platzierung der heterogenen Festplatten bei <i>Share</i> . . . . .	51
3.4	Beispiel einer zu heterogenen Konfiguration . . . . .	53
3.5	Platzierung von 3 Kopien in einem heterogenen System nach Schomaker [Schomaker, 2007] . . . . .	54
3.6	Beispiel einer Konfiguration, die für 2 Kopien komplett genutzt werden kann . . . . .	57
3.7	$\check{c}_i$ und $\check{c}'_i$ für zehn Festplatten mit identischer Kapazität . . . . .	91
4.1	Hauptspeicherverbrauch und Laufzeit von <i>Consistent Hashing</i> bei unterschiedlicher Anzahl homogener Festplatten . . . . .	99
4.2	Hauptspeicherverbrauch und Laufzeit von <i>Consistent Hashing</i> bei unterschiedlicher Anzahl platzierter Kopien jeder Festplatte . . . . .	101
4.3	Hauptspeicherverbrauch und Laufzeit von <i>Share</i> bei unterschiedlicher Anzahl homogener Festplatten . . . . .	102
4.4	Hauptspeicherverbrauch und Laufzeit von <i>Share</i> bei unterschiedlichem Dehnungsfaktor . . . . .	103
4.5	Hauptspeicherverbrauch und Laufzeit von <i>Redundant Share</i> bei unterschiedlicher Anzahl homogener Festplatten . . . . .	104

4.6	Hauptspeicherverbrauch und Laufzeit von <i>Fast Redundant Share</i> bei unterschiedlicher Anzahl homogener Festplatten . . . . .	106
4.7	Hauptspeicherverbrauch und Laufzeit von <i>Consistent Hashing</i> bei unterschiedlicher Anzahl heterogener Festplatten . . . . .	108
4.8	Hauptspeicherverbrauch und Laufzeit von <i>Consistent Hashing</i> bei unterschiedlicher Anzahl heterogener Festplatten und kapazitätsabhängiger Anzahl platzierter Kopien jeder Festplatte . . . . .	109
4.9	Hauptspeicherverbrauch und Laufzeit von <i>Share</i> bei unterschiedlicher Anzahl heterogener Festplatten . . . . .	110
4.10	Hauptspeicherverbrauch und Laufzeit von <i>Redundant Share</i> bei unterschiedlicher Anzahl heterogener Festplatten . . . . .	111
4.11	Unterschiedliche Anzahl Blöcke über einer Konfiguration homogener Festplatten mit <i>Consistent Hashing</i> . . . . .	113
4.12	Unterschiedliche Anzahl Blöcke über einer Konfiguration homogener Festplatten mit <i>Share</i> . . . . .	114
4.13	Unterschiedliche Anzahl Blöcke über einer Konfiguration homogener Festplatten mit <i>Redundant Share</i> . . . . .	116
4.14	Fairness von <i>Consistent Hashing</i> bei unterschiedlicher Anzahl homogener Festplatten . . . . .	117
4.15	Fairness von <i>Consistent Hashing</i> bei unterschiedlicher Anzahl platzierter Kopien jeder Festplatte . . . . .	118
4.16	Fairness von <i>Consistent Hashing</i> bei unterschiedlicher Anzahl homogener Festplatten und fester Anzahl an Kopien jeder Festplatte . . . . .	119
4.17	Fairness von <i>Share</i> bei unterschiedlicher Anzahl homogener Festplatten . . . . .	120
4.18	Fairness von <i>Share</i> bei unterschiedlichem Dehnungsfaktor . . . . .	121
4.19	Fairness von <i>Share</i> bei unterschiedlicher Anzahl homogener Festplatten und festem Dehnungsfaktor . . . . .	121
4.20	Fairness von <i>Redundant Share</i> bei unterschiedlicher Anzahl homogener Festplatten . . . . .	122
4.21	Fairness von <i>Consistent Hashing</i> bei unterschiedlicher Anzahl heterogener Festplatten . . . . .	124
4.22	Fairness von <i>Consistent Hashing</i> bei unterschiedlicher Anzahl heterogener Festplatten und kapazitätsabhängiger Anzahl platzierter Kopien jeder Festplatte . . . . .	125
4.23	Fairness von <i>Consistent Hashing</i> bei unterschiedlicher Anzahl heterogener Festplatten und kapazitätsabhängiger Anzahl platzierter Kopien jeder Festplatte unabhängig von der Anzahl der Festplatten . . . . .	126
4.24	Fairness von <i>Share</i> bei unterschiedlicher Anzahl heterogener Festplatten . . . . .	127
4.25	Fairness von <i>Share</i> bei unterschiedlicher Anzahl heterogener Festplatten und festem Dehnungsfaktor . . . . .	128

4.26	Fairness von <i>Redundant Share</i> bei unterschiedlicher Anzahl heterogener Festplatten . . . . .	128
4.27	Adaptivität von <i>Consistent Hashing</i> beim Hinzufügen homogener Festplatten . . . . .	131
4.28	Adaptivität von <i>Consistent Hashing</i> mit fester Anzahl an Kopien jeder Festplatte beim Hinzufügen homogener Festplatten . . . . .	132
4.29	Adaptivität von <i>Share</i> beim Hinzufügen homogener Festplatten . . . . .	134
4.30	Adaptivität von <i>Share</i> mit festem Dehnungsfaktor beim Hinzufügen homogener Festplatten . . . . .	135
4.31	Adaptivität von <i>Redundant Share</i> beim Hinzufügen homogener Festplatten . . . . .	136
4.32	Adaptivität von <i>Redundant Share</i> beim Hinzufügen homogener Festplatten an schlechtester Position . . . . .	137
4.33	Adaptivität von <i>Consistent Hashing</i> mit fester Anzahl an Kopien jeder Festplatte beim Hinzufügen heterogener Festplatten . . . . .	139
4.34	Adaptivität von <i>Share</i> mit festem Dehnungsfaktor beim Hinzufügen heterogener Festplatten . . . . .	140
4.35	Adaptivität von <i>Redundant Share</i> beim Hinzufügen heterogener Festplatten . . . . .	142



# Tabellenverzeichnis

1.1	Verwendete Einheiten . . . . .	7
2.1	Vermessung IO/s eines einzelnen Servers . . . . .	35
3.1	Beispiel zur Platzierung eines Elements über fünf heterogenen Festplatten . . .	64
3.2	Beispiel zur Platzierung eines Elements mit zwei Kopien über fünf heterogene Festplatten . . . . .	70
4.1	Legende der Diagramme zur Messung der Laufzeit und des Hauptspeicherver- brauchs . . . . .	98
4.2	Legende der Diagramme zur Messung der Fairness . . . . .	112
4.3	Legende der Diagramme zur Messung der Adaptivität . . . . .	130



# Algorithmenverzeichnis

1	<code>optimalWeights(k, {c<sub>0</sub>, ..., c<sub>n-1</sub>})</code> . . . . .	56
2	<code>LinPlace (addr, {d<sub>1</sub>, ..., d<sub>n</sub>}, {c<sub>1</sub>, ..., c<sub>n</sub>})</code> . . . . .	63
3	<code>LinMirror (addr, {d<sub>1</sub>, ..., d<sub>n</sub>}, {c<sub>1</sub>, ..., c<sub>n</sub>})</code> . . . . .	70
4	<code>RedundantShare (k, addr, {d<sub>1</sub>, ..., d<sub>n</sub>}, {c<sub>1</sub>, ..., c<sub>n</sub>})</code> . . . . .	77
5	<code>PeerReplication (k, addr, {d<sub>1</sub>, ..., d<sub>n</sub>}, {c<sub>1</sub>, ..., c<sub>n</sub>})</code> . . . . .	93





# Literaturverzeichnis

- [Abd-El-Malek u. a. 2005] ABD-EL-MALEK, M. ; COURTRIGHT, W. V. ; CRANOR, C. ; GANGER, G. R. ; HENDRICKS, J. ; KLOSTERMAN, A. J. ; MESNIER, M. ; PRASAD, M. ; SALMON, B. ; SAMBASIVAN, R. R. ; SINNAMOHIDEEN, S. ; STRUNK, J. D. ; THERESKA, E. ; WACHS, M. ; WYLIE, J. J.: Ursa minor: versatile cluster-based storage. In: *Proceedings of the 4th USENIX conference on File And Storage Technologies (FAST)*. Berkeley, CA, USA : USENIX Association, 2005, S. 59–72
- [Adya u. a. 2002] ADYA, A. ; BOLOSKY, W. J. ; CASTRO, M. ; CERMAK, G. ; CHAIKEN, R. ; DOUCEUR, J. R. ; HOWELL, J. ; LORCH, J. R. ; THEIMER, M. ; WATTENHOFER, R. P.: Farsite: federated, available, and reliable storage for an incompletely trusted environment. In: *Proceedings of the 5th USENIX symposium on Operating Systems Design and Implementation (OSDI)*. Berkeley, CA, USA : USENIX Association, 2002, S. 1–14
- [Alemany und Thathacher 1997] ALEMANY, J. ; THATHACHER, J.S.: Random Striping for News on Demand Servers / University of Washington, Department of Computer Science and Engineering. 1997. – Technical Report
- [Anderson u. a. 2002] ANDERSON, D. C. ; CHASE, J. S. ; VAHDAT, A. M.: Interposed request routing for scalable network storage. In: *ACM Transactions On Computer Systems (TOCS)* 20 (2002), Nr. 1, S. 25–48
- [Azagury u. a. 2003] AZAGURY, A. ; DREIZIN, V. ; FACTOR, M. ; HENIS, E. ; NAOR, D. ; RINETZKY, N. ; RODEH, O. ; SATRAN, J. ; TAVORY, A. ; YERUSHALMI, L.: Towards an Object Store. In: *Proceedings of the 11th NASA Goddard, 20st IEEE conference on Mass Storage Systems and Technologies (MSS)*. Washington, DC, USA : IEEE Computer Society, 2003, S. 165–176
- [Blaum u. a. 1994] BLAUM, M. ; BRADY, J. ; BRUCK, J. ; MENON, J.: EVENODD: An Optimal Scheme for Tolerating Double Disk Failures in RAID Architectures. In: *Proceedings of the 21st ACM IEEE International Symposium on Computer Architecture (ISCA)*. New York, NY, USA : ACM, 1994, S. 245–254
- [Blömer u. a. 1995] BLÖMER, J. ; KALFANE, M. ; KARP, R. ; KARPINSKI, M. ; LUBY, M. ; ZUCKERMAN, D.: An xor-based erasure-resilient coding scheme / Unversity of California at Berkley, International Computer Science Institute. 1995. – Technical Report

- [Brinkmann und Effert 2007a] BRINKMANN, A. ; EFFERT, S.: Inter-node Communication in Peer-to-Peer Storage Clusters. In: *Proceedings of the 24th IEEE IEEE conference on Mass Storage Systems and Technologies (MSST)*. Washington, DC, USA : IEEE Computer Society, 2007, S. 257–262
- [Brinkmann und Effert 2007b] BRINKMANN, A. ; EFFERT, S.: Snapshots and Continuous Data Replication in Cluster Storage Environments. In: *Proceedings of the 4th international workshop on Storage Network Architecture and Parallel I/Os (SNAPI)*. Washington, DC, USA : IEEE Computer Society, 2007, S. 3–10
- [Brinkmann und Effert 2008a] BRINKMANN, A. ; EFFERT, S.: Data Replication in P2P Environments. In: *Proceedings of the 20th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. New York, NY, USA : ACM, 2008, S. 191–193
- [Brinkmann und Effert 2008b] BRINKMANN, A. ; EFFERT, S.: Redundant Data Placement Strategies for Cluster Storage Environments. In: *Proceedings of the 12th international conference On Principles Of DIstributed Systems (OPODIS)*. Berlin, Germany : Springer-Verlag, 2008, S. 551–554
- [Brinkmann u. a. 2005] BRINKMANN, A. ; EFFERT, S. ; HEIDEBUER, M. ; VODISEK, M.: Distributed MD. In: *Proceedings of the 3rd international workshop on Storage Network Architecture and Parallel I/Os (SNAPI)*. Washington, DC, USA : IEEE Computer Society, 2005, S. 81 – 88
- [Brinkmann u. a. 2006a] BRINKMANN, A. ; EFFERT, S. ; HEIDEBUER, M. ; VODISEK, M.: Influence of Adaptive Data Layouts on Performance in dynamically changing Storage Environments. In: *Proceedings of the 14th Euromicro conference on Parallel, Distributed and network based Processing (PDP)*. Washington, DC, USA : IEEE Computer Society, 2006, S. 155–162
- [Brinkmann u. a. 2006b] BRINKMANN, A. ; EFFERT, S. ; HEIDEBUER, M. ; VODISEK, M.: Realizing Multilevel Snapshots in Dynamically Changing Virtualized Storage Environments. In: *Proceedings of the 5th IEEE International Conference on Networking (ICN)*. Berlin, Germany : Springer Verlag, 2006
- [Brinkmann u. a. 2007] BRINKMANN, A. ; EFFERT, S. ; MEYER AUF DER HEIDE, F. ; SCHEIDELER, C.: Dynamic and Redundant Data Placement. In: *Proceedings of the 27th IEEE International Conference on Distributed Computing Systems (ICDCS)*. Washington, DC, USA : IEEE Computer Society, 2007
- [Brinkmann u. a. 2004] BRINKMANN, A. ; HEIDEBUER, M. ; MEYER AUF DER HEIDE, F. ; RÜCKERT, U. ; SALZWEDEL, K. ; VODISEK, M.: V:Drive - Costs and Benefits of an Out-of-Band Storage Virtualization System. In: *Proceedings of the 12th NASA Goddard, 21st IEEE conference on Mass Storage Systems and Technologies (MSST)*. Washington, DC, USA : IEEE Computer Society, 2004

- [Brinkmann u. a. 2000] BRINKMANN, A. ; SALZWEDEL, K. ; SCHEIDELER, C.: Efficient, Distributed Data Placement Strategies for Storage Area Networks. In: *Proceedings of the 12th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. New York, NY, USA : ACM, 2000, S. 119–128
- [Brinkmann u. a. 2002] BRINKMANN, A. ; SALZWEDEL, K. ; SCHEIDELER, C.: Compact, adaptive placement schemes for non-uniform distribution requirements. In: *Proceedings of the 14th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. New York, NY, USA : ACM, 2002, S. 53–62
- [Callaghan 1999] CALLAGHAN, B.: *NFS Illustrated*. Amsterdam, Niederlande : Addison-Wesley Longman, 1999. – 544 S. – ISBN-13: 978-0201325706
- [Cooley u. a. 2003] COOLEY, J. A. ; MINEWEASER, J. L. ; SERVI, L. D. ; TSUNG, E. T.: Software-based erasure codes for scalable distributed storage. In: *Proceedings of the 11th NASA Goddard, 20st IEEE conference on Mass Storage Systems and Technologies (MSS)*. Washington, DC, USA : IEEE Computer Society, 2003, S. 157–164
- [Corbett u. a. 2004] CORBETT, P. ; ENGLISH, B. ; GOEL, A. ; GRCANAC, T. ; KLEIMAN, S. ; LEONG, J. ; SANKAR, S.: Row-Diagonal Parity for Double Disk Failure Correction. In: *Proceedings of the 3rd USENIX conference on File And Storage Technologies (FAST)*. Berkeley, CA, USA : USENIX Association, 2004, S. 1–14
- [Cortes und Labarta 2000] CORTES, T. ; LABARTA, J.: A case for heterogeneous disk arrays. In: *Proceedings of the IEEE international Conference on Cluster Computing*. Washington, DC, USA : IEEE Computer Society, 2000, S. 319–325
- [Cortes und Labarta 2001] CORTES, T. ; LABARTA, J.: Extending Heterogeneity to RAID level 5. In: *Proceedings of the 2001 USENIX Annual Technical Conference*. Berkeley, CA, USA : USENIX Association, 2001, S. 119–132
- [Fasheh 2006] FASHEH, M.: OCFS2: The Oracle Clustered File System, Version 2. In: *Proceedings of the 8th Linux symposium*, 2006, S. 289–301
- [Frølund u. a. 2003] FRØLUND, S. ; MERCHANT, A. ; SAITO, Y. ; SPENCE, S. ; VEITCH, A.: FAB: enterprise storage systems on a shoestring. In: *Proceedings of the 9th workshop on Hot Topics in Operating Systems (HOTOS)*. Berkeley, CA, USA : USENIX Association, 2003, S. 169–174
- [Ganger u. a. 2003] GANGER, G. R. ; STRUNK, J. D. ; KLOSTERMAN, A. J.: Self-\* Storage: Brick-based storage with automated administration / Carnegie Mellon University. 2003. – Technical Report
- [von zur Gathen und Gerhard 1999] GATHEN, J. von zur ; GERHARD, J.: *Modern Computer Algebra*. Cambridge, U.K. : Cambridge University Press, 1999. – 771 S. – ISBN-13: 978-0521826464

- [Gibson u. a. 1998] GIBSON, G. A. ; NAGLE, D. F. ; AMIRI, K. ; BUTLER, J. ; CHANG, F. W. ; GOBIOFF, H. ; HARDIN, C. ; RIEDEL, E. ; ROCHBERG, D. ; ZELENKA, J.: A cost-effective, high-bandwidth storage architecture. In: *ACM SIGOPS Operating Systems Review* 33 (1998), Nr. 11, S. 92–103
- [Gibson und Van Meter 2000] GIBSON, G. A. ; VAN METER, R.: Network attached storage architecture. In: *Communications of the ACM* 43 (2000), Nr. 11, S. 37–45
- [Gonzalez und Cortes 2004] GONZALEZ, J. L. ; CORTES, T.: Increasing the capacity of RAID5 by online gradual assimilation. In: *Proceedings of the 2nd international workshop on Storage Network Architecture and Parallel I/Os (SNAPI)*. New York, NY, USA : ACM, 2004, S. 17–24
- [Gonzalez und Cortes 2008] GONZALEZ, J. L. ; CORTES, T.: Distributing Orthogonal Redundancy on Adaptive Disk Arrays. In: *Proceedings of the International Conference on Grid computing, high-performAnce and Distributed Applications (GADA'08)*, 2008
- [Hafner 2005] HAFNER, J. L.: WEAVER codes: highly fault tolerant erasure codes for storage systems. In: *Proceedings of the 4th USENIX conference on File And Storage Technologies (FAST)*. Berkeley, CA, USA : USENIX Association, 2005
- [Hertel 2003] HERTEL, C. R.: *Implementing CIFS: The Common Internet File System*. Upper Saddle River, New Jersey, USA : Prentice Hall International, 2003. – 672 S. – ISBN-13: 978-0130471161
- [Holland und Gibson 1992] HOLLAND, M. ; GIBSON, G. A.: Parity declustering for continuous operation in redundant disk arrays. In: *ACM SIGPLAN Notices* 27 (1992), Nr. 9, S. 23–35
- [Honicky und Miller 2003] HONICKY, R. J. ; MILLER, E. L.: A fast algorithm for online placement and reorganization of replicated data. In: *Proceedings of the 17th IEEE Parallel and Distributed Processing Symposium (IPDPS)*. Washington, DC, USA : IEEE Computer Society, 2003
- [Honicky und Miller 2004] HONICKY, R. J. ; MILLER, E. L.: Replication under Scalable Hashing: A Family of Algorithms for Scalable Decentralized Data Distribution. In: *Proceedings of the 18th IEEE Parallel and Distributed Processing Symposium (IPDPS)*. Washington, DC, USA : IEEE Computer Society, 2004
- [Huang und Xu 2008] HUANG, Cheng ; XU, Lihao: STAR : An Efficient Coding Scheme for Correcting Triple Storage Node Failures. In: *IEEE Transactions on Computers* 57 (2008), Nr. 7, S. 889–901
- [Karger u. a. 1997] KARGER, D. ; LEHMAN, E. ; LEIGHTON, T. ; PANIGRAHY, R. ; LEVINE, M. ; LEWIN, D.: Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the World Wide Web. In: *Proceedings of the 29th ACM Symposium on Theory Of Computing (STOC)*. New York, NY, USA : ACM, 1997, S. 654–663

- [Karger u. a. 1999] KARGER, D. ; SHERMAN, A. ; BERKHEIMER, A. ; BOGSTAD, B. ; DHANIDINA, R. ; IWAMOTO, K. ; KIM, B. ; MATKINS, L. ; YERUSHALMI, Y.: Web caching with consistent hashing. In: *Computer Networks* 31 (1999), Nr. 11-16, S. 1203–1213
- [Kofler 2009] KOFLER, M.: *Linux 2010*. 9. München, Deutschland : Addison-Wesley, 2009. – 1216 S. – ISBN-13: 978-3827328779
- [Kubiatowicz u. a. 2000] KUBIATOWICZ, J. ; BINDEL, D. ; CHEN, Y. ; CZERWINSKI, S. ; EATON, P. ; GEELS, D. ; GUMMADI, R. ; RHEA, S. ; WEATHERSPOON, H. ; WELLS, C. ; ZHAO, B.: OceanStore: an architecture for global-scale persistent storage. In: *Proceedings of the 9th international conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. New York, NY, USA : ACM, 2000, S. 190–201
- [Lawin 1998] LAWIN, D. M.: *Consistent hashing and random trees: Algorithms for caching in distributed networks*, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Masterarbeit, 1998
- [Lee und Thekkath 1996] LEE, E. K. ; THEKKATH, C. A.: Petal: distributed virtual disks. In: *Proceedings of the 7th international conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. New York, NY, USA : ACM, 1996, S. 84–92
- [Luby 2002] LUBY, M.: LT codes. In: *Proceedings of the 43rd IEEE symposium on Foundations of Computer Science (FOCS)*. Washington, DC, USA : IEEE Computer Society, 2002, S. 271–280
- [MacCormick u. a. 2009] MACCORMICK, J. ; MURPHY, N. ; RAMASUBRAMANIAN, V. ; WIEDER, U. ; YANG, J. ; ZHOU, L.: Kinesis: A new approach to replica placement in distributed storage systems. In: *ACM Transactions On Storage (TOS)* 4 (2009), Nr. 4, S. 1–28
- [Mense 2008] MENSE, M.: *On Fault-Tolerant Data Placement in Storage Networks*. Paderborn, Deutschland, Universität Paderborn, Heinz Nixdorf Institut, Dissertation, 2008
- [Mense und Scheideler 2008] MENSE, M. ; SCHEIDELER, C.: SPREAD: An Adaptive Scheme for Redundant and Fair Storage in Dynamic Heterogeneous Storage Systems. In: *Proceedings of the 19th ACM-SIAM Symposium On Discrete Algorithms (SODA)*. Philadelphia, PA, USA : Society for Industrial and Applied Mathematics, 2008, S. 1135–1144
- [Mextorf und Schmidt 2009a] MEXTORF, Olaf ; SCHMIDT, Ulrike: *Ein Netz für Petaflops und Petabytes – Netzwerk- und Storage-Design für Supercomputer*. 2009. – URL <http://www.fz-juelich.de/jsc/files/docs/vortraege/jak-2009/jak-2009-netze.pdf>
- [Mextorf und Schmidt 2009b] MEXTORF, Olaf ; SCHMIDT, Ulrike: *Ein Netz für Petaflops und Petabytes – Teil2: JUST – von 1PB auf 5PB mit 66 GB/s*. 2009. – URL <http://www.fz-juelich.de/jsc/files/docs/vortraege/jak-2009/jak-2009-storage.pdf>
- [Meyer u. a. 2008] MEYER, D. T. ; AGGARWAL, G. ; CULLY, B. ; LEFEBVRE, G. ; FEELEY, M. J. ; HUTCHINSON, N. C. ; WARFIELD, A.: Parallax: virtual disks for virtual machines. In:

- Proceedings of the 3rd ACM SIGOPS/EuroSys European conference on Computer Systems (Eurosys)*. New York, NY, USA : ACM, 2008, S. 41–54
- [Nationale Institute for Standards and Technology (NIST) 2004] NATIONALE INSTITUTE FOR STANDARDS AND TECHNOLOGY (NIST): *Announcing the SECURE HASH STANDARD*. 2004. – FIPS-PUB 180-2
- [Olaru und Tichy 2005] OLARU, V. ; TICHY, W. F.: On the design and performance of kernel-level TCP connection endpoint migration in cluster-based servers. In: *Proceedings of the 5th IEEE symposium on Cluster Computing and the Grid (CCGrid)* Bd. 2. Washington, DC, USA : IEEE Computer Society, 2005, S. 1000–1007
- [Patterson u. a. 1988] PATTERSON, D. A. ; GIBSON, G. A. ; KATZ, R. H.: A Case for Redundant Arrays of Inexpensive Disks (RAID). In: *Proceedings of the 1988 ACM SIGMOD international conference on management of data*. New York, NY, USA : ACM, 1988, S. 109 – 116
- [Plank 2007] PLANK, J. S.: A New MDS Erasure Code for RAID-6 / University of Tennessee, Department of Computer Science. 2007. – Technical Report
- [Plank 2008] PLANK, J. S.: The RAID-6 Liberation Codes. In: *Proceedings of the 6th USENIX conference on File And Storage Technologies (FAST)*. Berkeley, CA, USA : USENIX Association, 2008, S. 97–110
- [Preslan u. a. 2000] PRESLAN, K. W. ; BARRY, A. P. ; BRASSOW, J. ; CATTELAN, R. ; MANTHEI, A. ; NYGAARD, E. ; VAN OORT, S. ; TEIGLAND, D. ; TILSTRA, M. ; O'KEEFE, M. T. ; ERICKSON, G. ; AGARWAL, M.: Implementing Journaling in a Linux Shared Disk File System. In: *Proceedings of the 8th NASA Goddard, 7th IEEE conference on Mass Storage Systems and Technologies (MSS)*. Washington, DC, USA : IEEE Computer Society, 2000, S. 351–378
- [Raab und Steger 1998] RAAB, M. ; STEGER, A.: Balls into Bins - A Simple and Tight Analysis. In: *Proceedings of the 2nd International Workshop on Randomization and Approximation Techniques in Computer Science (RANDOM)*. London, U.K. : Springer-Verlag, 1998, S. 159–170
- [Reed und Solomon 1960] REED, I. S. ; SOLOMON, G.: Polynomial Codes Over Certain Finite Fields. In: *Journal of the Society for Industrial and Applied Mathematics* 8 (1960), Nr. 2, S. 300–304
- [Rhea u. a. 2003] RHEA, S. ; EATON, P. ; GEELS, D. ; WEATHERSPOON, H. ; ZHAO, B. ; KUBIATOWICZ, J.: Pond: The OceanStore Prototype. In: *Proceedings of the 2nd USENIX conference on File And Storage Technologies (FAST)*. Berkeley, CA, USA : USENIX Association, 2003, S. 1–14
- [Saito u. a. 2004] SAITO, Y. ; FRØLUND, S. ; VEITCH, A. ; MERCHANT, A. ; SPENCE, S.: FAB: building distributed enterprise disk arrays from commodity components. In: *Procee-*

- dings of the 11th international conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. New York, NY, USA : ACM, 2004, S. 48–58
- [Salzwedel 2004] SALZWEDEL, K.: *Data Distribution Algorithms for Storage Networks*. Paderborn, Deutschland, Universität Paderborn, Heinz Nixdorf Institut, Dissertation, 2004. – ISBN 3-935433-62-X.
- [Sanders 2001] SANDERS, P.: Reconciling simplicity and realism in parallel disk models. In: *Proceedings of the 12th ACM-SIAM Symposium On Discrete Algorithms (SODA)*. Philadelphia, PA, USA : Society for Industrial and Applied Mathematics, 2001, S. 67–76
- [Santos und Muntz 1998a] SANTOS, J. R. ; MUNTZ, R.: Performance analysis of the RIO multimedia storage system with heterogeneous disk configurations. In: *Proceedings of the 6th ACM international conference on Multimedia (MULTIMEDIA)*. New York, NY, USA : ACM, 1998, S. 303–308
- [Santos und Muntz 1998b] SANTOS, J. R. ; MUNTZ, R. R.: Using Heterogeneous Disks on a Multimedia Storage System with Random Data Allocation / UCLA Computer Science Department. 1998. – Technical Report
- [Santos u. a. 2000] SANTOS, J. R. ; MUNTZ, R. R. ; RIBEIRO-NETO, B.: Comparing random data allocation and data striping in multimedia servers. In: *Proceedings of the 2000 ACM SIGMETRICS international conference on measurement and modeling of computer systems (SIGMETRICS)*. New York, NY, USA : ACM, 2000, S. 44–55
- [Satran u. a. 2004] SATRAN, J. ; METH, K. ; SAPUNTZAKIS, C. ; CHADALAPAKA, M. ; ZEIDNER, E.: *Internet Small Computer Systems Interface (iSCSI)*. 2004. – IETF RFC 3720
- [Schindelhauer und Schomaker 2005] SCHINDELHAUER, C. ; SCHOMAKER, G.: Weighted distributed hash tables. In: *Proceedings of the 17th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. New York, NY, USA : ACM, 2005, S. 218–227
- [Schmuck und Haskin 2002] SCHMUCK, F. ; HASKIN, R.: GPFS: A Shared-Disk File System for Large Computing Clusters. In: *Proceedings of the 1st USENIX conference on File And Storage Technologies (FAST)*. Berkeley, CA, USA : USENIX Association, 2002
- [Schomaker 2007] SCHOMAKER, G.: DHHT-RAID: A Distributed Heterogeneous Scalable Architecture for Dynamic Storage Environments. In: *Proceedings of the 21st international conference on Advanced Information Networking and Applications (AINA)*. Washington, DC, USA : IEEE Computer Society, 2007, S. 331–339
- [Schomaker 2008] SCHOMAKER, G.: *Distributed Resource Allocation and Management in Heterogeneous Networks*. Paderborn, Deutschland, Universität Paderborn, Heinz Nixdorf Institut, Dissertation, 2008
- [Schwan 2003] SCHWAN, P.: Lustre: Building a File System for 1,000-node Clusters. In: *Proceedings of the 5th Linux Symposium*, 2003, S. 380–386

- [Shepler u. a. 2000] SHEPLER, S. ; CALLAGHAN, B. ; ROBINSON, D. ; THURLOW, R. ; BEAME, C. ; EISLER, M. ; NOVECK, D.: *NFS Version 4 Protocol*. 2000. – IETF RFC 3010
- [Sinnamohideen u. a. 2010] SINNAMOHIDEEN, S. ; SAMBASIVAN, R. R. ; HENDRICKS, J. ; LIU, L. ; GANGER, G. R.: A Transparently-Scalable Metadata Service for the Ursa Minor Storage System. In: *Proceedings of the 2010 USENIX Annual Technical Conference*. Berkeley, CA, USA : USENIX Association, 2010
- [Stodolsky u. a. 1993] STODOLSKY, D. ; GIBSON, G. A. ; HOLLAND, M.: Parity logging overcoming the small write problem in redundant disk arrays. In: *Proceedings of the 20th ACM IEEE International Symposium on Computer Architecture (ISCA)*. New York, NY, USA : ACM, 1993, S. 64–75
- [Storage Networking Industry Association 2010] STORAGE NETWORKING INDUSTRY ASSOCIATION: *The 2010 SNIA Dictionary*. 2010. – URL [http://www.snia.org/education/dictionary/SNIADictionary\\_v\\_2010\\_1\\_ENG.pdf](http://www.snia.org/education/dictionary/SNIADictionary_v_2010_1_ENG.pdf)
- [Surhone u. a. 2010] SURHONE, L. M. ; TIMPLEDON, M. T. ; MARSEKEN, S. F.: *Object Storage Device*. Beau Batin, Mauritius : Betascript Publishing, 2010. – 88 S. – ISBN-13: 978-6130958671
- [T10 committee 2004] T10 COMMITTEE: *Object-Based Storage Device Commands*. 2004. – URL [http://www.t10.org/drafts.htm#OSD\\_Family](http://www.t10.org/drafts.htm#OSD_Family)
- [Troppens u. a. 2007] TROPPENS, U. ; ERKENS, R. ; MÜLLER, W.: *Speichernetze : Grundlagen und Einsatz von Fibre Channel SAN, NAS, iSCSI und InfiniBand*. Heidelberg, Deutschland : dpunkt.verlag, 2007. – 586 S. – ISBN-13: 978-3898643931
- [Vacca 2001] VACCA, J. R.: *The Essential Guide to Storage Area Networks*. Upper Saddle River, New Jersey, USA : Prentice Hall International, 2001. – 624 S. – ISBN-13: 978-0130935755
- [Weil u. a. 2006a] WEIL, S. A. ; BRANDT, S. A. ; MILLER, E. L. ; LONG, D. D. E. ; MALTZAHN, C.: Ceph: a scalable, high-performance distributed file system. In: *Proceedings of the 7th USENIX symposium on Operating Systems Design and Implementation (OSDI)*. Berkeley, CA, USA : USENIX Association, 2006, S. 307–320
- [Weil u. a. 2006b] WEIL, S. A. ; BRANDT, S. A. ; MILLER, E. L. ; MALTZAHN, C.: CRUSH: controlled, scalable, decentralized placement of replicated data. In: *Proceedings of the 2006 ACM/IEEE conference on Supercomputing (SC)*. New York, NY, USA : ACM, 2006, S. 122
- [Whitehouse 2007] WHITEHOUSE, Steven: The GFS2 Filesystem. In: *Proceedings of the 9th Linux Symposium, 2007*, S. 253–259
- [Wilkes u. a. 1996] WILKES, J. ; GOLDING, R. ; STAELIN, C. ; SULLIVAN, T.: The HP AutoRAID hierarchical storage system. In: *ACM Transactions On Computer Systems (TOCS)* 14 (1996), Nr. 1, S. 108–136