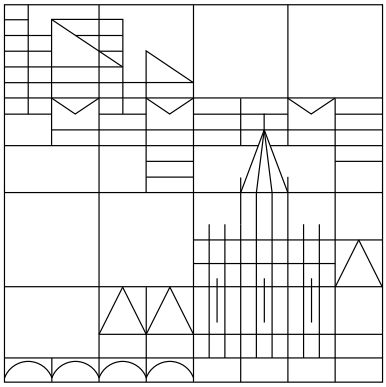
Universität Konstanz



Fachbereich für Informatik und Informationswissenschaft
Lehrstuhl für verteilte Systeme

Wissenschaftliche Arbeit
zur Erlangung des akademischen Grades eines Bachelor of Science (B.Sc.)
im Fachbereich Informatik & Informationswissenschaft der Universität Konstanz

Freiheitsgrade beim Einsatz Verteilter Disks

Degrees of Freedom in Distributed Storage

Verfasser:

Bastian Lemke

3. April 2008

Studienfach: Information Engineering
Schwerpunkt: Informatik der Systeme

Erstgutachter: Prof. Dr. M. Waldvogel
Zweitgutachter: Prof. Dr. M. Scholl
Betreuer: M.Sc. ETH M. Kramis

Konstanzer Online-Publikations-System (KOPS)
URL: http://www.ub.uni-konstanz.de/kops/volltexte/2008/5807/
URN: http://nbn-resolving.de/urn:nbn:de:bsz:352-opus-58078

Zusammenfassung

Diese Bachelorarbeit untersucht die Freiheiten und Spielräume beim Einsatz verteilter Disks. Anhand der Kriterien Performance, Flexibilität, Erweiterbarkeit und Skalierbarkeit werden verschiedene Möglichkeiten aufgezeigt. Der Schwerpunkt wird hierbei auf die Optimierung der Performance von Lese- und Schreiboperationen redundanter Zusammenschlüsse verteilter Disks gelegt. Neben der optimalen Ausnutzung von redundanter Datenhaltung werden Freiheiten beim Aufsplitten und Verteilen von Daten auf die unterschiedlichen physischen Disks untersucht.

Zur Verdeutlichung der Freiheiten werden Benchmarks einer Prototyp-Implementierung angefertigt. Anhand dieser Performancemessungen werden das Optimierungspotential bei unterschiedlich schnellen Disks sowie die Unterschiede verschiedener Strategien zur Verteilung der Daten auf die physischen Disks veranschaulicht.

Abstract

This bachelor thesis analyzes the degrees of freedom and scope in distributed storage. The different opportunities and tradeoffs are analyzed based on the criterions performance, flexibility, extensibility and scalability. The focus is placed on optimizing the performance of read and write operations using redundant combinations of distributed discs. Besides an optimal utilization of redundant data storage, the degrees of freedom affecting the segmentation of data on the different physical discs are examined.

To clarify the degrees of freedom, benchmarks of a prototype implementation are presented. Based on these performance measurements the optimization potential for discs with various speeds as well as the differences between several strategies for the distribution of data to the physical discs become clear.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

Durch die heutzutage immer größer werdenden Datenmengen sind Lösungen zur Gewährleistung der Datensicherheit und zur flexiblen Verwaltung von Speicherplatz unerlässlich.

Storage Area Networks (SANs) bilden einen zentralen Bestandteil großer, skalierbarer und hoch performanter Speicherlösungen. Durch SANs werden die vorhandenen Festplattensubsysteme[1] virtualisiert. Dadurch kann der gesamte verfügbare Speicherplatz im laufenden Betrieb flexibel verschiedenen Systemen zugeordnet werden.

Der Zugriff auf die Daten erfolgt blockbasiert und wird meist über das standardisierte SCSI[2]-Kommunikations-Protokoll, welches entweder auf *Fibre Channel* [Sta] oder *iSCSI* [SMS+04] als Transportprotokoll aufsetzt, realisiert. Bei den immer größer werdenden Speicherlösungen dürfen aber auch Verfügbarkeit und Ausfallsicherheit der einzelnen Komponenten nicht vernachlässigt werden.

Da es bei großen Speichernetzwerken nicht vermeidbar ist unterschiedliche Hardware einzusetzen, sind Verfahren notwendig, um die zur Verfügung stehenden Bandbreiten optimal zu nutzen und Latenzzeiten sowie den Energiebedarf zu minimieren. Heterogene Hardwareumgebungen entstehen in erster Linie durch die Erweiterung des Speichernetzwerks durch zusätzliche Hardware. Auch der Einsatz von *Solid State Disks* und *RAM-Disks* lässt heterogene Umgebungen entstehen. Die unterschiedlichen Zugriffsgeschwindigkeiten und auch der unterschiedliche Strombedarf bieten hier großes Optimierungspotenzial.

1.1 Gliederung der Arbeit

In Kapitel 2 werden zunächst die Grundlagen der *RAID*-Technologie zur Erhöhung von Zuverlässigkeit und Performance einzelner Festplatten erläutert. Da die Performance eines Speicherpools mit mehreren Festplatten auch durch die Verteilung der einzelnen Blö-

[1] Geräte, die mehrere Festplatten beinhalten
[2] Small Computer Systems Interface

cke auf die Festplatten entscheidend beeinflusst wird, folgt eine Einführung in das *Block-Mapping*. Anschließend wird der Einfluss von Übertragungsverzögerungen und verschiedener Latenzen erklärt.

Eine Analyse der Möglichkeiten zur Verbesserung der Performance verteilter Disks folgt in Kapitel 3. Nach der Betrachtung der Optimierungsmöglichkeiten von Leseoperationen werden die Auswirkungen verschiedener *Block-Mapping*-Strategien auf die Performance und Flexibilität von Schreiboperationen untersucht.

Die Analyse der Freiheiten bezüglich der Performance wird in Kapitel 4 zunächst durch die Vorstellung von Techniken zur Umverteilung von Daten weitergeführt. Neben dem Einsatz von Flashspeicher und *RAM-Disks* wird in diesem Kapitel außerdem auf die Minimierung des Energiebedarfs und unterschiedliche Verfügbarkeiten der Disks eingegangen.

Kapitel 5 beschreibt den Aufbau und die Implementierung des in Java geschriebenen Prototyps. Die Performancemessungen zur Darlegung des Optimierungspotentials werden anschließend in Kapitel 6 aufgezeigt und bewertet. Durch eine Zusammenfassung und einen Ausblick auf Verbesserungen des Prototyps wird die Arbeit mit Kapitel 7 abgeschlossen.

2 Grundlagen

2.1 Striping und Redundanz

Festplatten sind fehleranfällig und um ein Vielfaches langsamer als Prozessoren und Arbeitsspeicher. Techniken zur Erhöhung der Ausfall- und Datensicherheit sowie zur Verbesserung der Performance sind deshalb unbedingt notwendig. Diese beiden Ziele werden durch die seit mehr als 20 Jahren eingesetzten *RAID*[1]-Systeme [CLG+94] verfolgt. Durch die RAID-Techniken *Striping* und *Redundanz* können Performance und Verlässlichkeit von Festplatten stark verbessert werden. Des Weiteren können mehrere Festplatten (physische Disks) zu einer großen logischen Disk kombiniert werden.

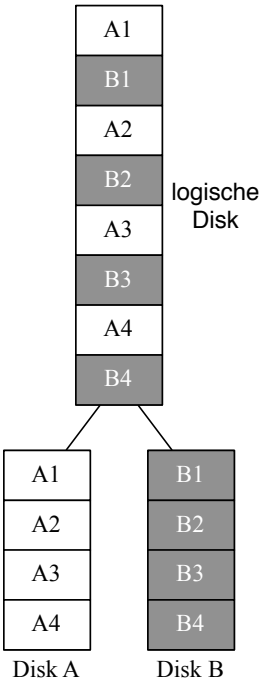


Abbildung 2.1: *RAID-0*

Durch Striping werden Daten bit-, byte- oder blockweise auf Festplatten verteilt. Die bekannteste RAID-Variante die Striping benutzt ist das in Abbildung 2.1 skizzierte RAID-0. Die einzelnen Festplatten werden hierbei zu einem großen zusammenhängenden Speicherbereich zusammengefasst. Bei Lese- und Schreiboperationen kann nun davon profitiert werden, dass es sich nicht um eine einzelne Festplatte sondern um mehrere handelt. Sequenzielle Anfragen können von mehreren Festplatten gleichzeitig bearbeitet werden, wodurch deren Datendurchsatz addiert wird. Nebenläufige Anfragen können gleichzeitig von unterschiedlichen Festplatten bearbeitet werden. Von Nachteil ist beim RAID-0 allerdings die gegenüber einer einzelnen Festplatte geringere Ausfallsicherheit. Beim Ausfall einer einzigen Festplatte sind sämtliche Daten des RAID-0 vom Datenverlust betroffen.

Die Performance der logischen Disk wird maßgeblich durch die beiden Konfigurationsparameter *Striping-Granularität* und *Striping-Breite* beeinflusst. Um einen bestmöglichen

[1] Redundant Array of Independent (Inexpensive) Disks

Datendurchsatz zu erzielen müssen die beiden Parameter auf die Größe der verwendeten Dateneinheiten und die Nebenläufigkeit angepasst werden.

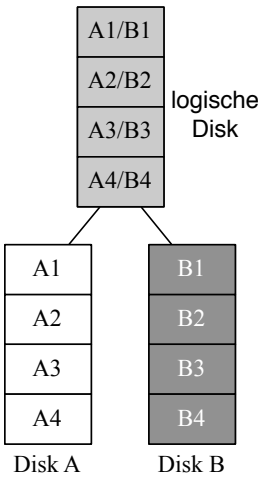


Abbildung 2.2: *RAID-1*

Durch die *Striping-Granularität* (auch *chunk size* genannt) wird die Größe der einzelnen Datenblöcke, welche auf die Festplatten verteilt werden, beschrieben. Feine Granularität gewährleistet eine hohe Parallelisierbarkeit von Anfragen, grobe Granularität verbessert auf der anderen Seite die Nebenläufigkeit.

Die *Striping-Breite* beschreibt die Anzahl der Festplatten über welche die Daten verteilt werden. Dabei gilt prinzipiell: Je größer die Striping-Breite eines RAID-0 ist, desto größer ist der Gesamtdurchsatz. Durch die Verteilung von Daten auf viele verschiedene Festplatten bei einer geringen *Striping-Granularität* können jedoch weniger Anfragen parallel ausgeführt werden.

Durch *Redundanz* wird vor allem eine höhere Datensicherheit gewährleistet. Man unterscheidet zwischen voller *Redundanz* (RAID-1) und Redundanz durch Paritätsinformationen (z.B. RAID-5). Da zusätzliche Daten geschrieben werden müssen nimmt der Datendurchsatz beim Schreiben ab. Leseoperationen können jedoch aufgeteilt und ähnlich wie beim Striping von mehreren Festplatten gleichzeitig gelesen werden. Die von den einzelnen Festplatten gelesenen Datenfragmente werden anschließend zusammengefügt.

2.2 Statisches Block-Mapping

Mit *Block-Mapping* ist die Abbildung beliebiger physischer Adressen auf logische Blockadressen gemeint, welche von Null aufsteigend durchnummeriert werden. Beim statischen Block-Mapping werden die logischen Adressen durch einen im Voraus festgelegten Algorithmus auf die physischen Adressen abgebildet.

Ein Einsatzgebiet des statischen Block-Mappings ist das in Kapitel 2.1 beschriebene Striping. Der Anwendung, die auf den logischen Adressen arbeitet, bleibt dadurch verborgen, dass es sich nicht um eine einzelne Festplatte sondern um mehrere handelt. Das Hinzufügen von Festplatten zu einem bereits bestehenden Verbund von Festplatten ist jedoch nicht möglich ohne das Block-Mapping neu zu berechnen, wodurch aber die ursprünglichen logischen Adressen ungültig werden würden.

Um eine Unabhängigkeit von der Geometrie der Festplatte (Aufteilung in Zylinder, Köpfe und Sektoren) zu erreichen werden bei ATA[2]-Festplatten die physischen dreidimensionalen Zylinder-Kopf-Sektor-Adressen in logische Blockadressen (LBA) umgerechnet [Law94]:

$$\mathrm{LBA} = ((\mathrm{C} \cdot \#\mathrm{H} + \mathrm{H}) \cdot \#\mathrm{S}) + \mathrm{S} - 1$$

C entspricht der Zylindernummer, #H der Anzahl der Leseköpfe, H der Lesekopfnummer, #S der Anzahl Sektoren pro Track und S der Sektornummer. Statisches Block-Mapping ist für die logische Blockadressierung sehr gut geeignet, da der Berechnungsaufwand – wie man an obiger Formel erkennen kann – gering ist und kein zusätzlicher Speicher benötigt wird.

2.3 Dynamisches Block-Mapping

Im Gegensatz zum statischen Block-Mapping kann mit der dynamischen Variante ein bestehendes Mapping geändert werden. Dazu wird das Block-Mapping für jeden Block (bzw. für Blockgruppen) in einer Tabelle gespeichert. Im Folgenden wird diese Technik anhand des auf Unixsystemen weit verbreiteten *Logical Volume Managers* (LVM) erläutert.

Einzelne Festplatten sind – vor allem in größeren Stückzahlen – sehr unflexibel. Einmal angelegte Partitionen können nur umständlich vergrößert oder verkleinert werden. Oft ist das äußerst zeitaufwendige Verschieben einer Partition und somit das Umkopieren aller beinhalteter Daten notwendig.

Um eine flexiblere Speicherverwaltung zu ermöglichen, bilden LVM [TM01, Leh99, Has01] eine Abstraktionsebene zwischen Festplatten und Dateisystemen. Festplatten (*Physical Volumes*, PVs) werden in *Volume Groups* (VGs) zusammengefasst, aus welchen dynamisch Speicherplatz in Form von *Logical Volumes* (LVs) angefordert werden kann. Sofern dies vom darauf verwendeten Dateisystem unterstützt wird, können diese LVs im Nachhinein beliebig vergrößert oder verkleinert werden. Anpassungen an bestehenden Dateisystemen sind nicht nötig, da aus Anwendungssicht kein Unterschied zwischen einem LV und einer herkömmlichen Festplattenpartition besteht.

Intern werden die PVs in *Physical Extents* (PEs) unterteilt, die normalerweise einige Megabytes groß sind aber auch bis zu einigen Gigabytes enthalten können. Bei der Erstel-

[2] Advanced Technology Attachment

lung eines LVs werden PEs von der VG angefordert. Für jedes PE wird genau ein *Logical Extent* (LE) angelegt, welches die logische Adresse innerhalb des LVs auf die physische Adresse abbildet. LVM Implementierungen, die Redundanz durch Mirroring unterstützen, ordnen einem LE jeweils zwei PEs zu.

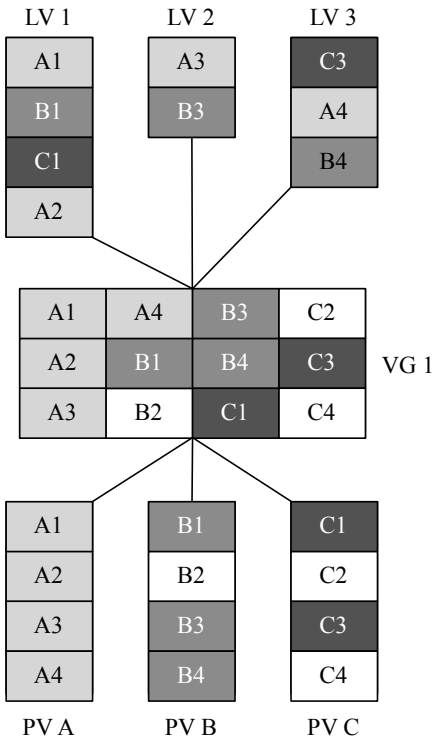


Abbildung 2.3: *Logical Volume Manager*

In Abbildung 2.3 ist der Aufbau eines LVM mit drei PVs A, B und C skizziert. Die weißen PEs stehen für nicht benutzte PEs, also freien Speicherplatz der VG, der für die Erstellung neuer LVs verwendet werden kann.

Beim Vergrößern eines LVs werden zusätzliche PEs von der VG angefordert sowie neue LEs erstellt. Das Verkleinern von LVs wird über das Entfernen freier LEs und die Freigabe der zugehörigen PEs in der VG realisiert. Die Mappingtabellen werden jeweils entsprechend vergrößert oder verkleinert.

Der für das Mapping der LEs auf die PEs benötigte Speicher ist vernachlässigbar. Die einzigen Informationen, die gespeichert werden müssen, sind jeweils das PV, die Anfangsadresse des PEs auf dem PV sowie die Anfangsadresse des LEs innerhalb des LVs. Durch die geringe Größe kann die Mappingtabelle ständig im Speicher gehalten werden. Da sie für jeden I/O benötigt wird, wird sie vom Prozessor vermutlich sogar direkt im internen Prozessorspeicher vorgehalten.

2.4 Verzögerungen und Übertragungsraten

Die Zugriffszeit auf Festplatten wird hauptsächlich von den mechanischen Komponenten bestimmt. Der Schreib-/Lesekopf der Festplatte muss über dem entsprechenden Zylinder positioniert werden (Positionierzeit). Anschließend fällt eine Rotationsverzögerung an, da gewartet werden muss bis der gewünschte Sektor unter dem Schreib-/Lesekopf durchrotiert. Erst dann können die gewünschten Sektoren gelesen und übertragen werden, was wiederum Zeit benötigt (Übertragungszeit).

Zusätzlich zu den festplattenspezifischen Verzögerungen müssen bei einem verteilten Speicherpool auch die Netzwerkverzögerungen mitberücksichtigt werden. Neben der Bearbeitungszeit in den einzelnen Netzwerkknoten und den Warteschlangenverzögerungen bei ausgelasteten Routern fällt eine von der Bandbreite abhängige Übertragungsverzögerung an. Die Ausbreitungsverzögerung ist abhängig von der Länge der physischen Verbindung sowie der Ausbreitungsgeschwindigkeit im jeweiligen Medium.

Diese 7 Verzögerungen lassen sich in zwei Kategorien einteilen. Im Folgenden werden alle Verzögerungen, die unabhängig von der zu übertragenden Datenmenge sind, zusammenfassend als *Latenz* (meist in Millisekunden) bezeichnet. Die von der Datenmenge abhängigen Verzögerungen werden durch den *Durchsatz* in Form einer Übertragungsrate (beispielsweise Mebibyte pro Sekunde) repräsentiert.

Die Schreib- und Leselatenzen können gemessen werden, indem die kleinstmögliche Datenmenge auf eine Disk geschrieben bzw. von dieser gelesen wird. Beim Durchsatz muss ebenfalls zwischen Lese- und Schreibdurchsatz unterschieden werden. Der Durchsatz kann durch die Übertragung von Datenmengen, bei denen die Bandbreite vollständig ausgenutzt wird, gemessen werden. Von der gemessenen Übertragungszeit muss noch die Latenz abgezogen werden.

3 Performance, Flexibilität und Erweiterbarkeit

In diesem Kapitel werden die Freiheiten in Bezug auf die Performance eines Speicherpools diskutiert, der aus Disks besteht, welche über ein Netzwerk verteilt sind. Die Datensicherheit in Form von Redundanz wird als vorgegeben angesehen. Als Performancemaße werden Latenz und Datendurchsatz betrachtet.

Je nach verwendeter Block-Mapping-Strategie stehen unterschiedliche Freiheiten in Bezug auf Performanceoptimierung, Flexibilität und Erweiterbarkeit eines Speicherpools zur Wahl. Der schwierigste Bereich ist hierbei die Optimierung von Schreiboperationen. Es gilt einen Kompromiss aus geringem Speicher- und Berechnungsoverhead sowie optimaler Blockverteilung und Lastausgleich der einzelnen Disks zu finden.

3.1 Optimierung von Leseoperationen

Da die Daten im Normalfall auf maximal zwei Disks redundant abgelegt sind, sind die Freiheiten beim Lesen relativ gering. Die einzige Optimierungsmöglichkeit besteht darin, die Anfragen auf redundant abgelegte Blöcke zu parallelisieren. Man unterscheidet in diesem Zusammenhang zwischen *Intra-Request-Parallelisierung* und *Inter-Request-Parallelisierung* (auch *Load Balancing*). Diese beiden Techniken werden in den folgenden Kapiteln 3.1.1 und 3.1.2 erklärt. Da das Block-Mapping in der Regel beim Lesen nicht verändert wird, sind diesbezüglich keine Optimierungen möglich. Eine Ausnahme stellen Techniken zur Umverteilung von Daten zum Lastausgleich der Disks dar (siehe Kapitel 4.1).

Auch Redundanz durch Paritätsinformationen, wie beispielsweise bei einem RAID-5, kann für die Optimierung des Lesedurchsatzes verwendet werden. Da jedoch bei einem RAID-5 mit n Disks für die Berechnung der gewünschten Daten von $n-1$ Disks gelesen werden muss, fällt der Optimierungsspielraum äusserst gering aus. Im Folgenden wird deshalb nur volle Redundanz, bei der alle Datenblöcke genau zwei mal vorliegen, be-

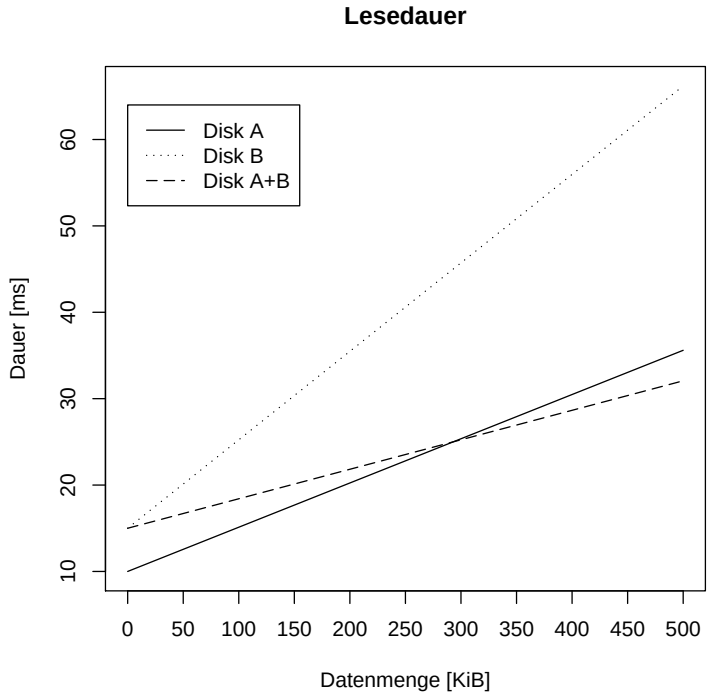


Abbildung 3.1: *Lesedauer von den einzelnen Disks, bzw. von beiden Disks für kleine Datenmengen.*
Die Latenz von Disk A (Disk B) beträgt 10 ms (15 ms), der Durchsatz 20000 Byte/ms (10000 Byte/ms)

trachtet. Latenz und Durchsatz werden als bekannt vorausgesetzt. Die Dauer Z, die für die Übertragung einer Datenmenge M bei einem Durchsatz D und einer Latenz L von einer Disk X benötigt wird kann folgendermaßen berechnet werden:

$$Z_X = \frac{M}{D} + L \tag{3.1}$$

Zwei Disks A und B, welche die redundanten Daten enthalten, benötigen also für die Übertragung der Datenmenge $M = M_A + M_B$ folgende Dauer:

$$Z_A = \frac{M_A}{D_A} + L_A \tag{3.2}$$

$$Z_B = \frac{M_B}{D_B} + L_B \tag{3.3}$$

Aus Gleichung 3.1 sowie aus Abbildung 3.1 wird ersichtlich, dass die Latenz für kleine Datenmengen stärker ins Gewicht fällt als der Datendurchsatz. Dies führt dazu, dass es

bis zu einem gewissen Schwellenwert schneller ist nur von einer Disk anstatt von beiden zu lesen. Dieser Schwellenwert wird hauptsächlich durch die Differenz der Latenzen der einzelnen Disks bestimmt.

Durch Einsetzen der Datenmenge in die Gleichungen 3.2 und 3.3 kann die Dauer der Leseoperation auf Disk A bzw. Disk B bestimmt werden. Für eine minimale Leseverzögerung müssen die zu lesenden Daten von dem Laufwerk mit dem kleineren Z_X gelesen werden.

3.1.1 Intra-Request-Parallelisierung

Ab einem gewissen Schwellenwert (in Abbildung 3.1: 300000 Byte) kann durch das Aufsplitten der Leseoperation auf zwei Disks die Leseverzögerung verkürzt werden. Der insgesamt beste Durchsatz und die kleinstmögliche Verzögerung kann erreicht werden, wenn beide Disks gleich lange mit dem Lesen der Daten beschäftigt sind (also $Z_A = Z_B$ gilt). Aus Gleichung 3.3 nach M_B aufgelöst und $M = M_A + M_B$ folgt:

$$M = M_A + D_B \cdot (Z_B - L_B) \tag{3.4}$$

Da $Z_A = Z_B$ erreicht werden soll kann Gleichung 3.2 in Gleichung 3.4 eingesetzt werden:

$$M = M_A + D_B \cdot \left(\frac{M_A}{D_A} + L_A - L_B \right) \tag{3.5}$$

Nach M_A aufgelöst folgt daraus schließlich:

$$M_A = \frac{M - D_B \cdot (L_A - L_B)}{1 + \frac{D_B}{D_A}} \tag{3.6}$$

Über Gleichung 3.6 kann der Anteil der Gesamtdatenmenge M berechnet werden, der von Disk A gelesen werden muss, um eine gleiche Auslastung beider Laufwerke zu erreichen. Für $M_A < 0$ würde das Lesen von beiden Disks länger dauern als das Lesen der gesamten Datenmenge von Disk B. Es wird deshalb $M_A = 0$ gesetzt. Für $M_A > M$ wird dementsprechend $M_A = M$ gesetzt. M_B lässt sich anschließend über $M_B = M - M_A$ berechnen.

Der Berechnungsoverhead ist gering, da D_A, D_B, L_A und L_B feste Größen sind. Für die Berechnung von M_A und M_B zu einem M sind somit zur Laufzeit nur 2 Subtraktionen und eine Division nötig.

3.1.2 Inter-Request-Parallelisierung

Bei der *Inter-Request-Parallelisierung* werden in erster Linie die Übertragungswarteschlangen beobachtet und die Lesezugriffe so verteilt, dass die Warteschlangen überall möglichst klein sind. Gleichung 3.6 berücksichtigt keine Warteschlangenverzögerungen. Um diese Verzögerungen mit einzubeziehen müssen die voraussichtlichen Verzögerungen anhand der Warteschlangenlänge Q und dem Durchsatz der Disk berechnet werden:

$$W_X = \frac{Q_X}{D_X} \tag{3.7}$$

Diese Verzögerung muss nun für einen Lastausgleich der Disks zu L_A bzw. L_B in Gleichung 3.6 hinzuaddiert werden:

$$M_A = \frac{M - D_B \cdot (L_A + W_A - L_B - W_B)}{1 + \frac{D_B}{D_A}} \tag{3.8}$$

3.2 Optimierung von Schreiboperationen

3.2.1 Statisches Block-Mapping

Die bereits in 2.2 beschriebenen Vorteile des statischen Block-Mappings sind der minimale Speicherverbrauch und der (je nach verwendetem Algorithmus) geringe Berechnungsaufwand. Ein Speicherpool mit statischem Block-Mapping ist zudem einfacher zu implementieren. Ohne die Speicherung zusätzlicher Informationen ist allerdings das nachträgliche Hinzufügen von Disks zum Speicherpool nicht möglich.

Um die Performance und die Datensicherheit zu verbessern, sollte in jedem Fall von den in Kapitel 2.1 beschriebenen Techniken Striping und Redundanz Gebrauch gemacht werden. Entscheidend für die Performance des Speicherpools sind die Striping-Breite, die Striping-Granularität und die Redundanz. Striping-Breite und Redundanz sind normalerweise über die Anzahl der Disks im Speicherpool bzw. die gewünschte Datensicherheit vorgegeben.

Der Gesamtdatendurchsatz des Speicherpools wird hauptsächlich durch die Striping-Granularität beeinflusst. Wie bereits in Kapitel 2.1 beschrieben beeinflusst dieser Parameter die Nebenläufigkeit und die Parallelisierbarkeit des Speicherpools. Um die optimale Striping-Granularität zu bestimmen muss deshalb die Art der Auslastung des Speicherpools (der Grad der Nebenläufigkeit sowie die Größen der Anfragen) bekannt sein. Ein Verfahren zur Bestimmung der Granularität ist in [CP90] beschrieben.

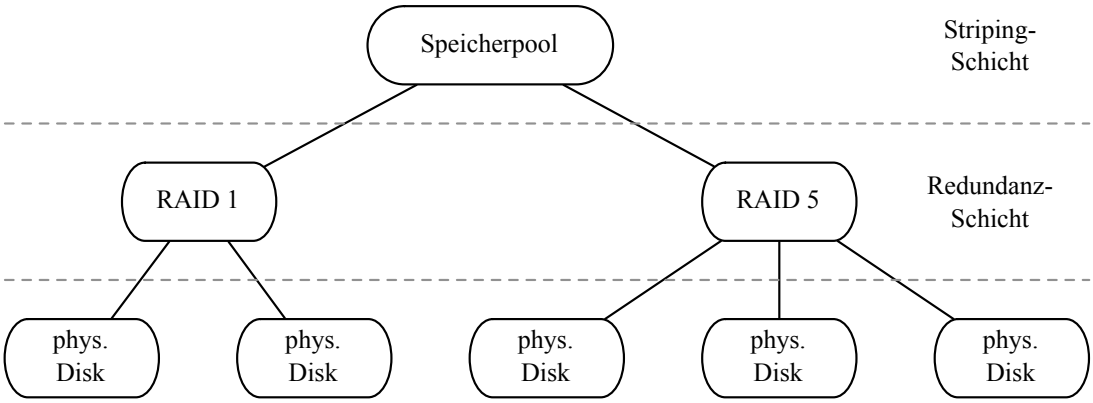


Abbildung 3.2: *3-Schichten-Modell*

Nach Erstellung des Speicherpools können Redundanz, Striping-Breite und Striping-Granularität bei statischem Block-Mapping nicht geändert werden. Der Speicherpool kann somit nicht auf Veränderungen der Anzahl der parallel zugreifenden Prozesse, der Latenzen oder der Lese- und Schreibdurchsätze angepasst werden. Ebenso ist ein nachträgliches Verschieben von Daten auf eine andere weniger stark ausgelastete Disk nicht durchführbar.

Sind die drei Parameter festgelegt müssen Überlegungen zur Verteilung der Daten auf die einzelnen Disks angestellt werden. Im Folgenden werden zwei Modelle zur Datenverteilung vorgestellt.

3-Schichten-Modell

In diesem Modell werden Striping und Redundanz getrennt voneinander implementiert. Wie aus Abbildung 3.2 ersichtlich wird werden die physischen Disks zu logischen RAID-Disks kombiniert. Je nach gewünschter Redundanz können beispielsweise RAID-1- und RAID-5-Verbünde erstellt werden. Der Vorteil der strikten Trennung von Redundanz und Striping ist die einfache Implementierbarkeit. Die vielen verfügbaren RAID Software- und Hardwarelösungen können in den Speicherpool integriert werden; Eine Neuimplementierung ist nicht nötig. Die zu RAID-Verbünden zusammengefassten Disks sollten jedoch möglichst gleich groß sein, da sonst Speicherplatz verloren geht. Für verteilte Disks, die zusammengefasst werden sollen, muss gegebenenfalls eine Softwarelösung implementiert werden.

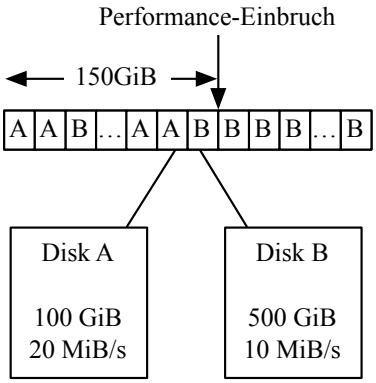


Abbildung 3.3: *Performanceeinbruch bei statischem Block-Mapping mit Beachtung des Durchsatzes*

Durch die Wahl unterschiedlicher RAID-Varianten kann unterschiedliche Redundanz für verschiedene Arten von Daten realisiert werden. Da die Datenverteilung auf die logischen RAID-Disks für die darüber liegende Schicht transparent ist, muss bei der Verwendung unterschiedlicher RAID-Systeme eine Methode für das Abfragen der Redundanz eines logischen Blocks implementiert werden. Andernfalls können die unterschiedlichen Redundanzen nicht genutzt werden, was zu Speicherplatz- und Performanceverlust sowie geringerer Datensicherheit führt.

Voraussetzung für die Nutzung unterschiedlicher Redundanzen ist eine Anwendung, die diese gezielt für Datensicherheit und Performance einsetzen kann. Daten, deren Datensicherheit eine untergeordnete Rolle spielen, können auf logischen Disks abgelegt werden, die eine geringere Redundanz – dafür aber einen höheren Schreibdurchsatz – bieten. Wenn bekannt ist, dass Daten häufig gelesen werden, kann auch der Lesedurchsatz der Blöcke bei der Blockauswahl berücksichtigt werden. Eine Möglichkeit ist, die Daten anhand des Schreibdurchsatzes – wie in Abbildung 3.3 skizziert – zu verteilen. Auf schnelleren Disks werden somit mehr Daten abgelegt als auf langsameren. Das Verhältnis der, auf den einzelnen Disks abgelegten, Datenmengen entspricht dem Verhältnis der Schreibdurchsätze. Die Latenz wird bei dieser Berechnung nicht berücksichtigt, da sie beim sequenziellen Lesen bzw. Schreiben von großen Datenmengen vernachlässigbar ist. Falls jedoch die Größe der Disks nicht im selben Verhältnis wie die Geschwindigkeit steht, sind die schnellen Disks mit zunehmender Speicherplatzbelegung des Speicherpools irgendwann komplett gefüllt, die langsamen Disks jedoch noch nicht. Ab diesem Zeitpunkt bricht die Performance stark ein.

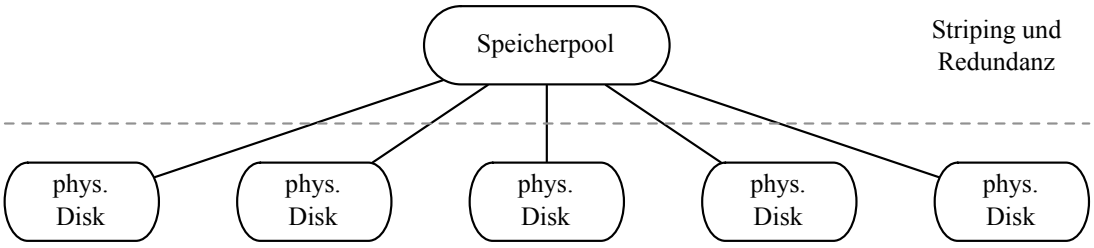


Abbildung 3.4: *2-Schichten-Modell*

Die Performance eines nach dem 3-Schichten-Modell aufgebauten Speicherpools ist größtenteils von den Fähigkeiten der verwendeten Anwendung abhängig. Kann diese Anwendung mit unterschiedlichen Redundanzen und Datendurchsätzen umgehen, können Daten gezielt auf langsame oder schnelle Blöcke verteilt werden; Der Performanceeinbruch kann so verhindert werden.

2-Schichten-Modell

Eine weitere Möglichkeit der Umsetzung eines Speicherpools ist die Kombination von Striping und Redundanz in derselben Schicht. Der Vorteil der beiden Schichten ist die bessere Ausnutzung des verfügbaren Speicherplatzes bei unterschiedlichen Laufwerksgrößen.

Für das Block-Mapping kann der Algorithmus des 3-Schichten-Modells übernommen werden. Die einzige notwendige Modifikation ist die Abbildung logischer Blöcke auf zwei (für volle Redundanz) statt auf nur einen physischen Block. Bei geringer Redundanz muss die Paritätsberechnung nun durch den Speicherpool durchgeführt werden. Von Hardware-RAID-Systemen und dem damit verbundenen geringeren Berechnungsaufwand kann nicht profitiert werden. Unterschiedliche Redundanzen sind mit diesem Modell nicht möglich.

3.2.2 Dynamisches Block-Mapping

Für eine flexiblere Speicherplatzverwaltung muss dynamisches Block-Mapping verwendet werden. Die Mappinginformationen werden in einer Tabelle abgespeichert. Durch die so gewonnene Flexibilität wird neben dem nachträglichen Hinzufügen von Laufwerken zum Speicherpool auch das Entfernen von Laufwerken unterstützt. Anpassungen auf Veränderungen bezüglich Latenz und Schreibdurchsatz können ebenfalls realisiert werden.

logische Adresse	Disk 1	Offset 1	Disk 2	Offset 2
0	0	0	1	0
100	2	0	3	0
200	0	100	1	100
300	2	100	3	100
⋮	⋮	⋮	⋮	⋮

Tabelle 3.1: *Mappingtabelle für 2-Schichten-Modell mit voller Redundanz*

Im Vergleich zu reinem statischem Block-Mapping bietet die dynamische Variante sehr viele Freiheiten. Im Prinzip kann jeder logische Block auf einen beliebigen physischen Block abgebildet werden. Die dadurch sehr groß werdende Mappingtabelle würde jedoch nicht mehr im Speicher vorgehalten werden können. Neben dem hohen Speicherplatzbedarf würden für das Auflösen der logischen in physische Adressen zusätzliche I/Os notwendig. Es gilt deshalb einen Kompromiss zwischen den Freiheiten des beliebigen Block-Mappings und dem Speicher- bzw. Mappingaufwand zu finden. Um dies zu realisieren werden die Laufwerke wie beim LVM in *Physische Extents* (PEs) aufgeteilt. Gleich große *logische Extents* (LEs) können durch einen statischen Algorithmus auf die physischen Extents abgebildet werden. Der Grad der nutzbaren Freiheiten wird maßgeblich durch die Größe dieser Extents beeinflusst.

Für die Erstellung des LE/PE-Mappings gibt es mehrere Möglichkeiten. Die Mappingtabelle kann entweder bei der Erstellung des Pools vollständig generiert werden. Alternativ kann zu Beginn eine leere Tabelle erstellt werden, die je nach Bedarf über *allocate-on-write* Stück für Stück ergänzt wird. Letztere Variante bietet den Vorteil, dass beim Hinzufügen von Laufwerken keine Änderungen an der Tabelle notwendig werden. Die Tabelle enthält außerdem keine unbenutzten Einträge. Somit wird weniger Speicherplatz beansprucht und logische Blockadressen können – da weniger Adressen überprüft werden müssen – schneller in physische Adressen aufgelöst werden.

Die Abbildung der LEs auf die PEs kann mit einem der beiden in Kapitel 3.2.1 beschriebenen Modelle realisiert werden. Das 2-Schichten-Modell profitiert von den zusätzlichen Freiheiten beim dynamischen Block-Mapping. Die Redundanz kann flexibel durch die Mappingtabelle sichergestellt werden. Eine beispielhafte Mappingtabelle ist in Tabelle 3.1 dargestellt. Jeweils der erste Block eines LEs wird in dieser Tabelle eingetragen. Die Adressen zwischen den Einträgen können über den verwendeten statischen Mappingalgorithmus bestimmt werden. Über diese Tabelle können auch unterschiedliche Redundanzen für die LEs dargestellt werden, womit dieser Nachteil des kombinierten

Modells beim Einsatz einer Mappingtabelle entfällt. Werden unterschiedliche statische Mappingalgorithmen verwendet, können die dazu nötigen Informationen und Parameter ebenfalls in dieser Tabelle gespeichert werden.

Dynamisches Block-Mapping ist als Ergänzung, nicht als Ersatz, für statisches Block-Mapping zu sehen. Durch die Speicherung des Block-Mappings in einer Tabelle können bei geringem Performance- und Speicheroverhead Erweiterbarkeit und Skalierbarkeit des Speicherpools gewährleistet werden.

3.2.3 Kein Block-Mapping

Eine weitere Möglichkeit ist der Verzicht auf Block-Mapping. In diesem Fall werden auf Blockebene für jede Adresse auf jeder Disk *Blockpointer* sowie die Performancedaten aller Disks bereitgestellt. Auf Basis der Performancedaten wird das Block-Mapping mithilfe der *Blockpointer* durch ein Dateisystem hergestellt.

Das Block-Mapping auf Dateisystemebene hat den Vorteil, dass Informationen über Datentypen verwendet werden können. Es können beispielsweise Metainformationen gezielt auf schnelleren Disks als die eigentlichen Daten plaziert werden. Von Nachteil ist allerdings, dass die Realisierung von Redundanz und auch Block-Mapping die Entwicklung von Dateisystemen verkompliziert.

Ein Beispiel für ein Dateisystem, das Redundanz sowie Block-Mapping umsetzt ist das ZFS [ZFS] von Sun Microsystems. Beim ZFS werden die äußeren (schnelleren) Regionen von Festplatten bevorzugt beschrieben. Über *copy-on-write* wird eine gleichmäßige Verteilung der Daten über alle, auch neu hinzugefügte, Disks erreicht.

3.3 Zusammenfassung / Fazit

Operation	**Freiheit**	**Kommentar**
Lesen	Intra-Request-Parallelisierung	Striping
	Inter-Request-Parallelisierung	Load Balancing
Schreiben	Block-Mapping	statisch / dynamisch / ohne
	Schichten-Modell	3-Schichten- / 2-Schichten-Modell

Tabelle 3.2: *Zusammenfassung der Freiheiten eines Speicherpools in Bezug auf die Performance*

Leseoperationen auf redundante Daten können durch Intra- und Inter-Request-Parallelisierung praktisch ohne zusätzlichen Berechnungsaufwand, unabhängig zu den Perfor-

mance-Eigenschaften der einzelnen physischen Disks, beschleunigt werden. Für kleine Datenmengen ist jedoch eine Umsortierung und asynchrone Ausführung der Leseoperationen notwendig, um unter Volllast einen optimalen Durchsatz des gesamten Systems zu erzielen. Durch diese Umsortierung der Leseoperationen muss versucht werden, die Bewegungen der Lese-/Schreibköpfe der Festplatten zu minimieren. Die Zugriffe müssen also möglichst sequenziell erfolgen.

Um eine optimale Lastverteilung bei Schreiboperationen zu erreichen, muss im Voraus bekannt sein, welche Daten wie oft geschrieben werden. Sequenzielle Schreibzugriffe können beschleunigt werden, indem die Verteilung der Blöcke dem Schreibdurchsatz der einzelnen Laufwerke angepasst wird.

4 Weitere Freiheiten

4.1 Umverteilung von Daten

Durch Algorithmen zur intelligenten Umverteilung von Daten [WZS91, SWZ98] kann ein Lastausgleich der einzelnen Festplatten sowohl bei Lese- als auch bei Schreiboperationen erreicht werden. Ein im Hintergrund laufendes Verschieben von Daten zwischen Festplatten erhöht jedoch deren Auslastung und verringert somit die Performance. Diese Technik ist demnach nur einsetzbar, wenn es „Ruhephasen“ gibt, in denen der Speicherpool nur eine geringe Auslastung hat, die für das Verschieben von Daten verwendet werden können.

Über das *copy-on-write*-Verfahren können Daten beim Schreiben ohne Performanceverlust auf weniger ausgelastete Disks „verschoben“ werden. Bei *copy-on-write*-Schreiboperationen werden Daten nicht *in-place* verändert, sondern an einer anderen Stelle neu gespeichert. Die alten Daten werden anschließend gelöscht. Diese Technik wird in *Sun Microsystem's* ZFS [ZFS] eingesetzt, um nach dem Einfügen zusätzlicher Disks eine gleichmäßige Belegung aller Laufwerke im Speicherpool zu erreichen.

4.2 Verwendung heterogener Disks

Große Speichernetzwerke bestehen nicht nur aus identischen Festplatten. Da gängige Speicherlösungen im Laufe der Zeit mit immer größer und auch schneller werdenden Festplatten ergänzt und erweitert werden, müssen diese unterschiedlichen Hardwareeigenschaften bei der Verteilung der Daten und dem Zugriff darauf beachtet werden.

Anstatt die unterschiedlichen Geschwindigkeiten der Festplatten, wie in Kapitel 3 beschrieben, auf Blockebene zu berücksichtigen, kann dies auch auf Dateisystemebene geschehen. Das verteilte Dateisystem *Ceph* [WBM+06] versucht beispielsweise mit einer speziellen Datenverteilungsfunktion die Metadaten von den Daten zu trennen und separat zu speichern.

Neben herkömmlichen Festplatten können auch RAM-Disks und Flashspeicher zum Einsatz kommen. Das hybride Dateisystem *Conquest* [WRPK02] verwendet persistenten RAM, um Lese- und Schreiboperationen zu beschleunigen. Zur Umsetzung dieses Dateisystems wurden Statistiken zu Dateigrößen und Dateizugriffen herangezogen:

1. Ein Großteil der Dateien eines Dateisystems hat eine geringe Größe.
2. Die meisten Dateizugriffe erfolgen auf kleine Dateien.
3. Der Großteil des Speichers wird durch große Dateien belegt.

Aus diesen Beobachtungen kann der Schluss gezogen werden, dass durch das Ablegen der kleinen Dateien auf einem schnellen Speicher ein Großteil der Dateizugriffe beschleunigt werden kann. *Conquest* speichert deshalb alle Dateien, deren Größe kleiner eines bestimmten Schwellenwerts ist, im schnellen RAM. Alle großen Dateien werden dagegen (möglichst sequenziell) auf den langsameren Festplatten abgelegt. Durch diese sehr einfache Regel können sowohl Lese- als auch Schreiboperationen mit äußerst geringem Overhead stark beschleunigt werden.

Statt der Integration könnten schnelle Disks wie *Solid State Disks (SSDs)* oder *nicht flüchtige RAM-Disks* durch ihren Geschwindigkeitsvorteil auch als reiner Cache eingesetzt werden. Dadurch würde jedoch der Vorteil, dass die Daten auch nach einem Stromausfall erhalten bleiben, nicht genutzt werden. Desweiteren sind *SSDs* aufgrund ihrer Eigenschaft nur eine begrenzte Anzahl an Löschvorgängen zu unterstützen, nicht geeignet. *SSDs* eignen sich vielmehr für einen Einsatz, bei welchem sie wenige Schreib- aber viele Leseoperationen durchführen müssen.

4.3 Verfügbarkeit und Ausfallsicherheit

Bei der verteilten Speicherung muss auch die Verfügbarkeit der Disks in Betracht gezogen werden. Je mehr Hardware- und Softwarekomponenten an dem Speichernetzwerk beteiligt sind, desto größer ist die Wahrscheinlichkeit, dass eine dieser Komponenten ausfällt. Bei über ein WAN verteilten Disks können nicht nur Defekte von physischen Disks, sondern auch Ausfälle von Netzwerkknoten sowie physischer Netzwerkverbindungen die Verfügbarkeit beeinträchtigen. Diese Ausfälle müssen bei der Architektur eines Speicherpools berücksichtigt werden.

Festplattenausfälle treten bei Speichernetzwerken, die Daten im Petabyte-Größenbereich speichern, täglich auf. In [XMS+03] werden Mechanismen zur Verbesserung der

Zuverlässigkeit großer Speichersysteme vorgestellt. RAID-Technologie stellt nur bedingt eine Lösung dar, da die Wiederherstellung der Redundanz einer ausgefallenen Festplatte viel Zeit in Anspruch nimmt. Es müssen deshalb flexiblere Systeme verwendet werden, die beim Ausfall einiger Komponenten die verlorene Redundanz möglichst schnell wiederherstellen können.

Um die Ausfallsicherheit bei redundanter Speicherung zu verbessern, kann die Strategie der *Diversität* eingesetzt werden. Unter *Diversität* versteht man den bewussten Einsatz unterschiedlicher Technologien, Hardware unterschiedlicher Hersteller und die Verwendung von Software, die von unterschiedlichen Programmierern in verschiedenen Programmiersprachen umgesetzt wird. Die hinter dieser Strategie stehende Idee ist, dass die unterschiedlichen Systeme auf unerwartete Ereignisse verschieden reagieren. Sollte ein System durch eine Störung ausfallen, ist es wahrscheinlich, dass ein anderes System (welches anders umgesetzt wurde und somit vermutlich andere Schwachstellen hat) noch funktionsfähig bleibt.

4.4 Energieverbrauch

Große Speichermengen verursachen automatisch auch einen hohen Energieverbrauch. Techniken zur Verringerung des Stromverbrauchs von Festplatten existieren im Bereich von Notebooks schon seit längerer Zeit. Bei den immer größer werdenden Datenmengen in Speichernetzwerken können durch das aktive *spin-down*[1] von nicht benötigten Festplatten große Energiemengen und somit Kosten eingespart werden.

Im Gegensatz zum aktiven *spin-down* findet das *spin-up*[2] der Festplatte automatisch beim Zugriff auf die gespeicherten Daten statt. Da jedoch – gerade im hoch performanten Server-Bereich mit Festplatten-Umdrehungsgeschwindigkeiten von bis zu 15000 Umdrehungen pro Sekunde – das *spin-up* mehrere Sekunden benötigt, ist eine kontrollierte Reaktivierung der Festplatten unbedingt nötig.

Ähnlich zur dynamischen Frequenzskalierung bei mobilen Prozessoren gibt es Ansätze, die Umdrehungsgeschwindigkeiten der Festplatten zur Energieeinsparung dynamisch zu regulieren [CPB03, GSKF03]. Aufgrund der mechanischen Unterschiede zwischen Prozessoren und Festplatten konnten sich diese Techniken jedoch nicht in der Massenproduktion durchsetzen.

[1] Versetzen der Festplatte in den Ruhezustand
[2] Reaktivierung der Festplatte

In [ZDD+04, ZSZ04, ZZ05] werden weitere Lösungswege vorgestellt, in denen durch Caching- und Allokationsalgorithmen sowie Cache-Ersetzungsstrategien versucht wird, den Energieverbrauch zu senken. Da diese Ansätze aber ebenfalls auf modifizierten Festplatten basieren, die mit unterschiedlichen Geschwindigkeiten betrieben werden können, sind sie nicht auf herkömmliche Festplatten anwendbar.

RIMAC [YW06] ist ein System, das nicht auf modifizierte Festplatten angewiesen ist. Um das automatische *spin-up* der Festplatten zu verhindern, wird beim Einsatz eines RAID-5 versucht, Daten aus den Parityinformationen der anderen Laufwerke zu rekonstruieren, anstatt ein Laufwerk aus dem Ruhezustand zu holen. Sobald die Auslastung eines RAID-5 unterhalb einem festgelegten Prozentsatz liegt, versetzt *RIMAC* die Festplatte mit der geringsten Auslastung in den Ruhezustand. Anfragen an diese Festplatte werden wie oben erwähnt über XOR-Berechnungen aus den Parity-Daten rekonstruiert. Die Festplatte wird erst wieder aktiviert, wenn die Auslastung des Systems über den festgelegten Prozentsatz steigt oder aber das Verhältnis der *dirty blocks*[3] im Cache einen gewissen Schwellenwert überschreitet. Die relativ lange Dauer die sich veränderte Blocks im Cache befinden verstärkt allerdings das Risiko eines Datenverlusts. Um dieses Risiko zu minimieren, wird *RIMAC* in Verbindung mit *nicht flüchtigem (persistentem) RAM* eingesetzt.

[3] geänderte Blocks im Cache, die noch nicht auf die Festplatte geschrieben wurden

5 Implementierung

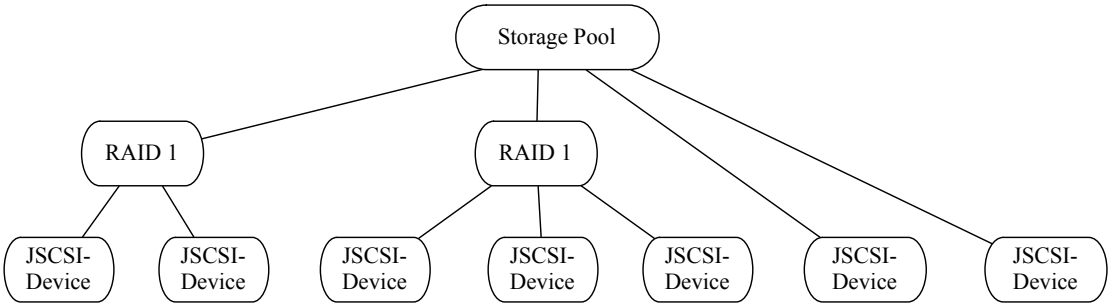


Abbildung 5.1: *Skizze eines Speicherpools*

Der implementierte Prototyp eines verteilten Speicherpools ist nach dem in Kapitel 3.2.1 beschriebenen 3-Schichten-Modell aufgebaut. Das 3-Schichten-Modell wurde aufgrund seiner höheren Flexibilität und Erweiterbarkeit gewählt.

Die unterste physische Schicht enthält die einzelnen physischen Disks. Auf der darüber gelegenen virtuellen Schicht wird die Datensicherheit durch Redundanz gewährleistet. Physische Disks können mithilfe von virtuellen Disks miteinander verbunden werden. Durch diese Disks wird in erster Linie die Datensicherheit gewährleistet. Falls auf Datensicherheit kein Wert gelegt wird, können physische Disks auch direkt in den Speicherpool integriert werden; Leseoperationen können in diesem Fall jedoch nicht mehr optimiert werden. Durch Redundanz lässt sich die Leseperformance, wie in Kapitel 3.1 beschrieben, optimieren. Die oberste Schicht stellt schließlich – ähnlich wie ein in Kapitel 2.3 beschriebener Logical Volume Manager – das Block-Mapping auf einen zusammenhängenden Speicherbereich bereit.

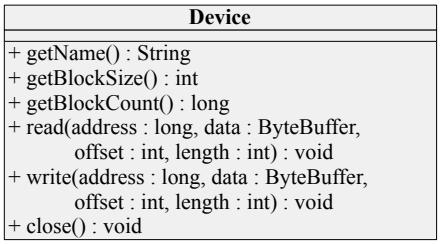


Abbildung 5.2: *Interface* `Device`

5.1 Speicherpool-Schicht

In der Speicherpool-Schicht werden die zur Verfügung stehenden physischen bzw. virtuellen Disks über Striping zu einer großen logischen Disk verbunden.

Basis des Speicherpools ist das in Abbildung 5.2 skizzierte Interface `Device`. Um eine gute Erweiterbarkeit und das Verschachteln mehrerer Implementierungen zu ermöglichen, wird dieses Interface von allen virtuellen und physischen Disks implementiert. Dies hat den positiven Nebeneffekt, dass die Verwendung eines Speicherpools identisch zu der einer physischen oder virtuellen Disk ist.

Durch die abstrakte Klasse `StoragePool` werden die grundlegenden Methoden eines Speicherpools implementiert. Über eine abstrakte Methode wird die Implementierung mehrerer Algorithmen für das Aufsplitten und Verteilen von Leseoperationen ermöglicht. Rückgabewert dieser Methode ist ein Array mit `BlockPointer`-Objekten. `BlockPointer`-Objekte stellen Verweise auf physische Adressen in einem Speicherpool dar. Dazu werden neben einem Verweis auf ein `PhysicalDevice` bzw. ein `VirtualDevice` der Offset sowie eine Anzahl Blöcke – welche die Größe der gespeicherten Dateneinheit repräsentiert – gespeichert.

Die `StoragePool`-Implementierung `SimpleStaticStoragePool` teilt jede Disk in gleich große *Chunks* ein. Die Größe dieser *Chunks* wird im Konstruktor festgelegt, die Standardgröße liegt bei 512 Blöcken. *Chunks* einer jeden Disk, mit denselben Positionen auf der Disk, bilden zusammen einen *Stripe*.

Anstatt gleich große *Chunks* für jede Disk zu wählen, wird die *Chunkgröße* beim `OptimizedStaticStoragePool` in ein Verhältnis zum Datendurchsatz beim Schreiben gesetzt. Dadurch wird beim sequenziellen Schreiben der bestmögliche Gesamtdurchsatz erzielt. Über den Konstruktor lässt sich die durchschnittliche *Chunkgröße* für die verwendeten physischen Disks festlegen.

Der `DynamicStoragePool` teilt den Speicherbereich zusätzlich in *Extents* ein. Ein *Extent* besteht aus einer über den Konstruktor festlegbaren Anzahl an *Stripes*. Die Anzahl der *Stripes* pro *Extent* ist innerhalb eines `DynamicStoragePools` einheitlich. Die letztendliche Größe kann jedoch – durch das Hinzufügen von Disks – unterschiedlich sein. Über die Anzahl der in einem *Extent* enthaltenen *Stripes* lässt sich die Flexibilität des Speicherpools steuern. Hier gilt es einen guten Kompromiss zwischen Flexibilität und Performance zu finden, da eine geringe *Extent*-Größe sowohl zu einer größeren Hauptspeicherbelegung als auch zu einem höheren Zeitaufwand zum Auflösen einer logischen in eine physische Adresse führt.

5.2 Virtuelle Schicht

Die virtuelle Schicht gewährleistet die Redundanz des Speicherpools. Sämtliche Methoden in dieser Schicht implementieren das Interface `VirtualDevice`, welches das Interface `Device` um Methoden zur Abfrage von Performancedaten, die zur Optimierung der Verteilung der Daten im Speicherpool benötigt werden, erweitert.

Zur Implementierung verschiedener RAID-1-Algorithmen wird, ähnlich zur Speicherpool-Schicht, die abstrakte Klasse `Raid1Device` vorgegeben. Das Auf- bzw. Verteilen von Leseoperationen ist ebenfalls über eine abstrakte Methode mit `BlockPointer`-Objekten als Rückgabewert realisiert. Eine Implementierung eines Lesealgorithmus für ein `Raid1Device` ist das `SimpleRaid1Device`. Dieser Algorithmus teilt die zu lesenden Blöcke je zur Hälfte auf beide unterliegende physische Disks auf.

Da das `SimpleRaid1Device` unterschiedliche Latenzen und Lesedurchsätze der physischen Disks nicht berücksichtigt, kann bei Disks mit unterschiedlichen Performanceeigenschaften das `OptimizedRaid1Device` verwendet werden. Dieses Device berechnet anhand der Latenz, des Lesedurchsatzes und der aktuellen Länge der Lesewarteschlange die für die aktuelle Leseoperation zu lesende Blockanzahl für jede der physischen Disks. Dazu werden die in Kapitel 3.1 vorgestellten Berechnungen verwendet.

5.3 Physische Schicht

Das Interface `PhysicalDevice` erweitert das `VirtualDevice` nochmals um Methoden zur Bestimmung der Leselatenz und der Länge der Lesewarteschlange. Diese Daten werden für die Optimierung der Leseoperationen von einem RAID-1 benötigt. Alle physischen Disks implementieren dieses Interface.

Das `JSCSIDevice` stellt die Schnittstelle zu einem einzelnen iSCSI-Target über *jSCSI* dar. Über den Konstruktor wird bei der Erstellung eines Objekts eine Verbindung zum gewünschten iSCSI-Target hergestellt. Um ungewünschte mehrfache Verbindungen zu demselben Target zu verhindern ist das `JSCSIDevice` als Singleton implementiert.

Zur Simulation realer Disks kann das `DummyDevice` verwendet werden. Über die vier Parameter `readLatency`, `writeLatency`, `readThroughput` und `writeThroughput` lassen sich im Konstruktor die Performancedaten der Disk festlegen, welche simuliert werden soll. Anhand der zu lesenden bzw. zu schreibenden Datenmenge wird die benötigte Lese- bzw. Schreibdauer berechnet und der aufrufende Thread für diese Dauer schlafen gelegt.

6 Benchmarks

Um den Performancegewinn zu messen, welcher durch die verschiedenen Freiheiten verteilter Disks erzielt werden kann, wurden mehrere Benchmarks durchgeführt. Die Ausführungszeiten einzelner Lese- und Schreiboperationen wurden mit dem Open-Source-Tool *Perfidix*[1] ermittelt.

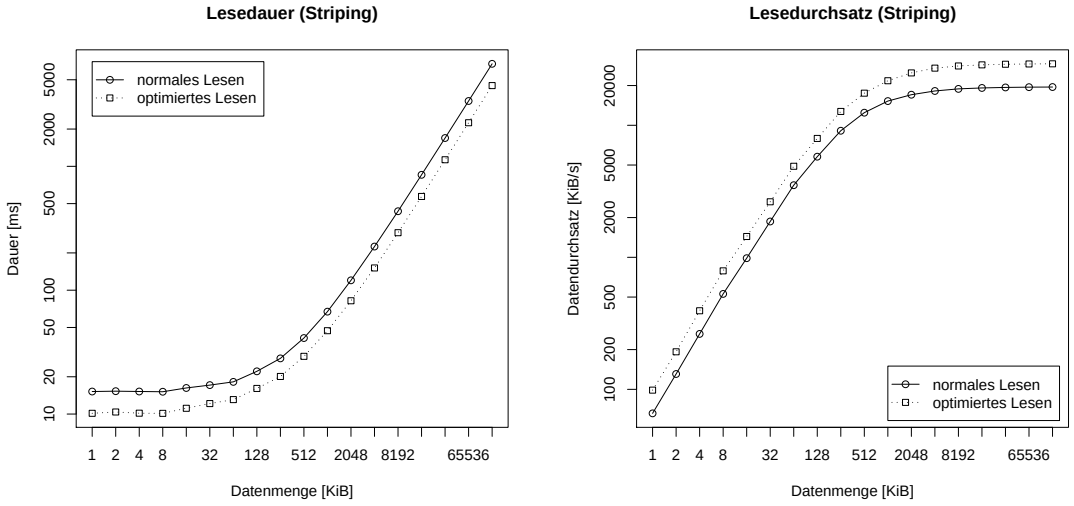
Erste Tests wurden mit *jSCSI* [2], einer Java-Implementierung des iSCSI-Standards [SMS+04], das ebenso wie *Perfidix* am Lehrstuhl für verteilte Systeme der Universität Konstanz [3] entwickelt wurde, durchgeführt. Aufgrund von Stabilitätsproblemen von *jSCSI* wurde für die Benchmarks allerdings das im Abschnitt 5.3 beschriebene `DummyDevice` für die Simulation von Lese- und Schreibverzögerungen verwendet. Durch die einheitlichen Interfaces können `DummyDevices` jedoch problemlos gegen `JSCSIDevices` ausgetauscht werden.

Sämtliche Benchmarks wurden auf einem Apple iMac mit einem Intel Core 2 Duo Prozessor (2,16 GHz) und 2 GB Arbeitsspeicher unter Mac OS X 10.5 durchgeführt. Für die Latenzen und Datendurchsätze wurden die gemessenen Werte der ursprünglich für die Tests mit *jSCSI* verwendeten iSCSI-Targets übernommen.

6.1 Leseoperationen

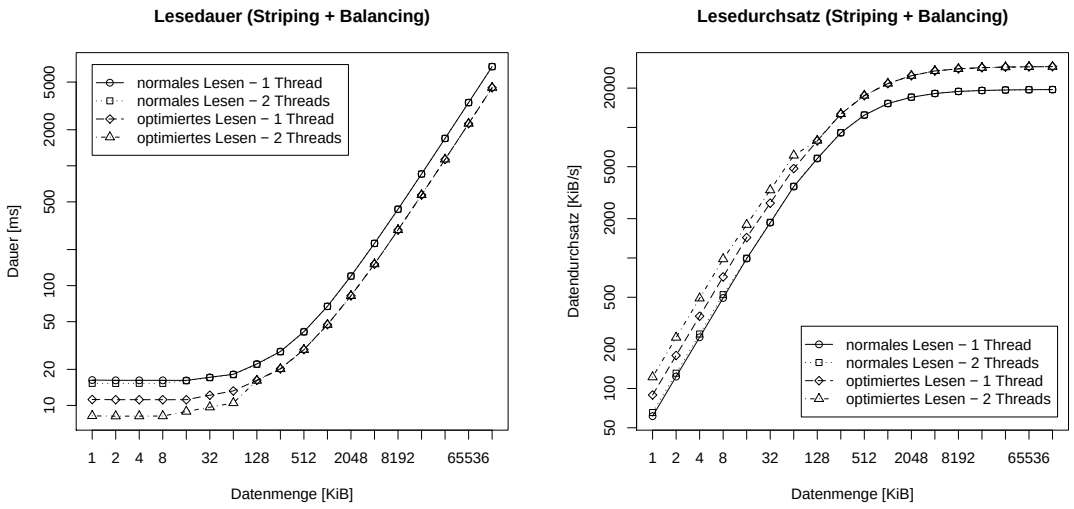
Für alle Benchmarks von Leseoperationen wurden zwei `DummyDevices` verwendet, um reale Disks zu simulieren. Ersteres simulierte eine Leseverzögerung von 10 ms und einen Lesedurchsatz von 20000 Byte/ms (ca. 19500 KiB/s), das zweite eine Leseverzögerung von 15 ms und einen Lesedurchsatz von 10000 Byte/ms (ca. 9800 KiB/s). Es wird angenommen, dass die zu lesenden Daten vollständig redundant auf den beiden Disks abgelegt worden sind.

[1] http://sourceforge.net/projects/perfidix
[2] http://sourceforge.net/projects/jscsi
[3] http://www.inf.uni-konstanz.de/disy



(a) *Lesedauer* **(b)** *Lesedurchsatz*

Abbildung 6.1: *Lesedauer/Lesedurchsatz mit und ohne Intra-Request-Parallelisierung*



(a) *Lesedauer* **(b)** *Lesedurchsatz*

Abbildung 6.2: *Lesedauer/Lesedurchsatz mit und ohne Inter-Request-Parallelisierung*

Die Leseoperationen wurden mit den in Kapitel 5.2 vorgestellten virtuellen Disks `SimpleRaid1Device` („normales Lesen“) bzw. `OptimizedRaid1Device` („optimiertes Lesen“) durchgeführt. Als kleinste Datenmenge wurde 1 KiB gewählt, was zwei physischen Blöcken entspricht. Die Zeitmessung von nur einem physischen Block ist nicht sinnvoll, da dieser nur von einer einzelnen Disk gelesen werden kann. Eine Performanceoptimierung ist somit nicht möglich.

Abbildung 6.1 beschreibt den Performancevergleich der beiden `Raid1Devices`. Es wird der Vorteil von Intra-Request-Parallelisierung bei größeren Datenmengen deutlich. Bei kleinen Datenmengen mit einer Lesezeit < 15 ms (was der Latenz der langsameren Disk entspricht) werden Lesezugriffe ausschließlich von der schnelleren Disk durchgeführt.

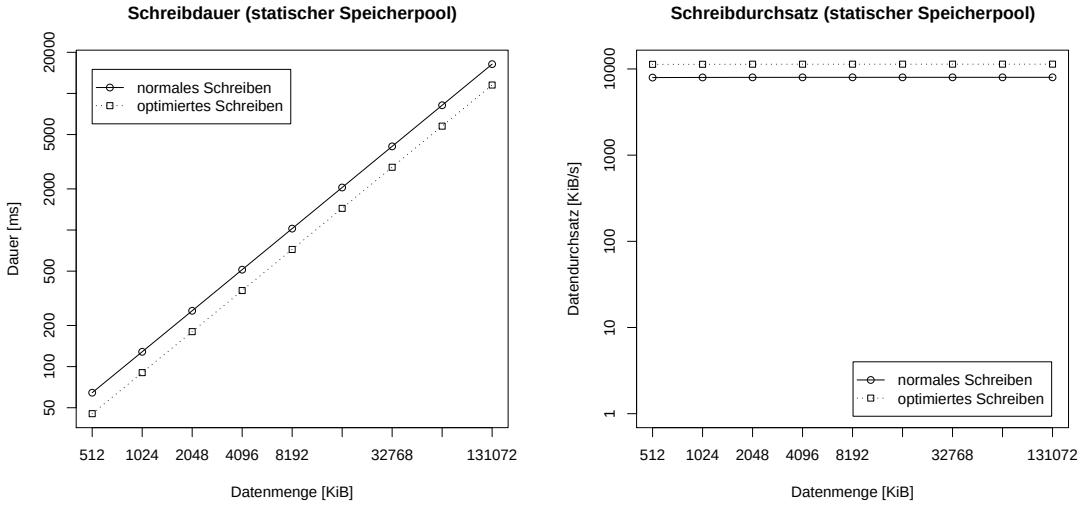
Bei der Verwendung mehrerer Threads kann zusätzlich zur Intra-Request-Parallelisierung auch Inter-Request-Parallelisierung zum Einsatz kommen. Wie aus Abbildung 6.2 ersichtlich wird, führt dies bei kleinen Datenmengen zu einer Verkürzung der Lesezeit bzw. einer Erhöhung des Lesedurchsatzes, da die verfügbare Bandbreite durch die Parallelisierung besser ausgenutzt wird. Ohne Balancing führt der Einsatz mehrerer Threads nur zu einer minimalen Verbesserung.

Die Verwendung von mehr als zwei Threads führt zu keiner Verbesserung des Datendurchsatzes, da die Leseoperationen synchron durchgeführt werden (der aufrufende Thread wird angehalten, bis die Leseoperation abgeschlossen ist). Für jedes Laufwerk ist nur eine simultane Leseoperation möglich. Dies erklärt, dass der Durchsatz ab 128 KiB (ab diesem Wert kommt Intra-Request-Parallelität zum Einsatz) durch Multithreading nicht verbessert werden kann.

6.2 Schreiboperationen

Die Benchmarks von Schreiboperationen wurden ebenfalls mit zwei `DummyDevices` durchgeführt, von denen eines eine Disk mit einer Schreibverzögerung von 15 ms und einen Schreibdurchsatz von 15000 Byte/ms (ca. 14600 KiB/s) simuliert, das andere eine mit 20 ms Schreibverzögerung und 6000 Byte/ms (ca. 5900 KiB/s) Schreibdurchsatz.

Abbildung 6.3 zeigt den Vergleich zweier Striping-Algorithmen bei sequenziellem Schreiben. Durch die Berücksichtigung der Schreibdurchsätze beider Disks kann eine geringere Schreibdauer bzw. ein höherer Schreibdurchsatz erzielt werden.



(a) *Schreibdauer* **(b)** *Schreibdurchsatz*

Abbildung 6.3: *Schreibdauer/Schreibdurchsatz, statischer Speicherpool*

Der je nach gewählter Granularität relativ geringe Berechnungs- und Speicheroverhead des dynamischen Speicherpools macht sich bei der Schreibdauer bzw. dem Schreibdurchsatz nicht bemerkbar. Für jeden Extent und jede Disk muss die Startadresse (relativ zum Extent) gespeichert werden, ab welcher die Daten gespeichert sind. Dafür werden pro Disk 4 Byte benötigt. Zusätzlich muss für jede Disk die Anzahl Blöcke, die in vorhergehenden Extents liegen, gespeichert werden, wofür pro Extent für jede Disk nochmals 8 Byte benötigt werden. Damit das richtige Extent für die jeweilige Lese-/Schreiboperation gefunden werden kann, muss zudem noch für jedes Extent die Startadresse im logischen Speicherpool-Adressbereich gespeichert werden (8 Byte). Zusätzlich fallen noch der Java-Overhead für die Verwaltung der Extent-Objekte sowie des Vector-Objekts, in welchem die Extents gespeichert sind, an.

Ein Speicherpool mit zwei Laufwerken von je 500 GiB, der Standard-Chunk-Größe von durchschnittlich 256 KiB und der Standard-Extent-Größe (100 Stripes, in diesem Fall 50 MiB) benötigt für jeden Extent 32 Byte Speicher. Bei vollständiger Ausnutzung des Speicherplatzes ist die Mappingtabelle ca. 655 KiB groß (bei Vernachlässigung des Java-Overheads). Ein Speicherpool derselben Größe mit 10 Disks hätte mit denselben Einstellungen eine Extentgröße von 250 MiB und eine Mappingtabelle mit ca. 524 KiB. Das gewünschte Extent sowie die Disk, auf welcher die Daten liegen, können über eine binäre Suche in logarithmischer Zeit gefunden werden.

6.3 Schlussfolgerung

Das Lesen von Daten kann durch einen Algorithmus, der unterschiedliche Performancedaten der Disks berücksichtigt, praktisch ohne Overhead optimiert werden. Der gewonnene Geschwindigkeitsvorteil ist von der Performancedifferenz der verwendeten Disks abhängig.

Für die Optimierung von Schreiboperationen ist der nötige Overhead ebenfalls gering (logarithmischer Aufwand für die Bestimmung der Disk). Bei sequenziellen Schreibzugriffen kann die Bandbreite aller Disks bestmöglich ausgenutzt werden. Für zufällige, gleichmäßig verteilte Schreibzugriffe trifft dies ebenso zu.

7 Zusammenfassung und Ausblick

7.1 Zusammenfassung

Die heutzutage immer größer werdenden Datenmengen verlangen nach hochgradig flexiblen und vor allem skalierbaren Speicherlösungen. Mit einem Speichernetzwerk (*SAN*), mit welchem der Speicherplatz virtualisiert wird, können diese Anforderungen erfüllt werden. In dieser Arbeit wurden Möglichkeiten aufgezeigt, wie die Performance von Speichernetzwerken, die sich aus heterogener Hardware zusammensetzen, verbessert werden kann.

Als Einstieg in die verteilte Datenspeicherung wurden in Kapitel 2 die Grundlagen der RAID-Technologie an den Beispielen eines RAID-0 und eines RAID-1 erläutert. Nach der Erklärung des Block-Mappings am Beispiel des *Logical Volume Managers* folgte eine Einführung in die unterschiedlichen Verzögerungen und Übertragungsraten, die bei verteilter Speicherung auftreten.

Auf Basis der in Kapitel 2 vorgestellten Grundlagen wurden in Kapitel 3.1 Techniken zur schnellstmöglichen Durchführung von Leseoperationen auf redundanten Daten durch *Intra-* und *Inter-Request-Parallelisierung* entwickelt. Anschließend wurden in Kapitel 3.2 Möglichkeiten zur Verbesserung von Schreiboperationen auf Blockebene aufgezeigt. Dazu wurden zwei Modelle für statisches Block-Mapping und die Erweiterung dieser Modelle zu einer dynamischen Variante vorgestellt.

In Kapitel 4 wurden weitere Freiheiten aufgezeigt. Neben der Möglichkeit zur Umverteilung von Daten über aktives Verschieben bzw. über das *copy-on-write*-Verfahren wurde der Umgang mit heterogenen Disk-Umgebungen und der Einsatz von *Solid State Disks* sowie *persistenten RAM-Disks* diskutiert. Nach einer Analyse der Verfügbarkeiten von Disks und der Verbesserung der Ausfallsicherheit wurde auf Techniken zur Minimierung des Energiebedarfs eingegangen.

Kapitel 5 beschrieb den in Java programmierten Prototyp zur Demonstrierung der in Kapitel 3 entwickelten Verbesserungen. Die Auswirkungen wurden mit den Benchmarks in Kapitel 6 dargelegt und kommentiert.

7.2 Ausblick

Um im Prototyp die gleichzeitige Ausführung vieler kleiner Leseoperationen zu beschleunigen, können asynchrone Leseoperationen umgesetzt werden. Für kleine Datenmengen kann damit der Grad der Parallelisierung und somit die Nebenläufigkeit verbessert werden. Weitere Optimierungen sind durch die Größe oder auch die Semantik der Daten auf Dateisystemebene möglich. Durch eine kompaktere Datenstruktur für die Mappingtabelle könnte der Overhead für die Java-Objekte im vorgestellten Prototyp verringert werden.

Eine Lösung um sowohl optimale Performance als auch geringe Hardwareanschaffungskosten, optimale Verfügbarkeit und Ausfallsicherheit sowie einen niedrigen Energieverbrauch zu erzielen, gibt es nicht. Deshalb muss zwischen diesen Faktoren je nach Einsatzgebiet abgewogen und ein Kompromiss gefunden werden.

A Ergebnisse der Perfidix-Benchmarks

Für sämtliche Benchmarks wurden 100 Durchläufe durchgeführt.

A.1 Leseoperationen

A.1.1 Intra-Request-Parallelisierung

	size [KiB]	sum [ms]	min [ms]	max [ms]	avg [ms]	stddev [ms]	conf95 [ms]
Simple	0,5	1019	10	12	10,19	00,42	[10,11,10,27]
Optimized	0,5	1016	10	11	10,16	00,37	[10,09,10,23]
Simple	1	1520	15	17	15,20	00,42	[15,12,15,28]
Optimized	1	1014	10	11	10,14	00,35	[10,07,10,21]
Simple	2	1530	15	22	15,30	00,81	[15,14,15,46]
Optimized	2	1039	10	17	10,39	00,99	[10,20,10,58]
Simple	4	1521	15	20	15,21	00,60	[15,09,15,33]
Optimized	4	1016	10	11	10,16	00,37	[10,09,10,23]
Simple	8	1514	15	16	15,14	00,35	[15,07,15,21]
Optimized	8	1013	10	12	10,13	00,36	[10,06,10,20]
Simple	16	1624	16	21	16,24	00,71	[16,10,16,38]
Optimized	16	1113	11	12	11,13	00,34	[11,06,11,20]
Simple	32	1713	17	18	17,13	00,34	[17,06,17,20]
Optimized	32	1215	12	13	12,15	00,36	[12,08,12,22]
Simple	64	1819	18	19	18,19	00,39	[18,11,18,27]
Optimized	64	1308	13	14	13,08	00,27	[13,03,13,13]
Simple	128	2213	22	23	22,13	00,34	[22,06,22,20]
Optimized	128	1610	16	17	16,10	00,30	[16,04,16,16]
Simple	256	2815	28	29	28,15	00,36	[28,08,28,22]
Optimized	256	2011	20	21	20,11	00,31	[20,05,20,17]
Simple	512	4109	41	42	41,09	00,29	[41,03,41,15]
Optimized	512	2924	29	40	29,24	01,13	[29,02,29,46]

Tabelle A.1: *Leseoperationen ohne (`SimpleRaid1Device`) und mit (`OptimizedRaid1Device`) Intra-Request-Parallelisierung.*

Die ersten Messewerte mit 0.5 KiB wurden in die Diagrammen 6.1 nicht aufgenommen (siehe Kapitel 6.1).

	size [MiB]	sum [ms]	min [ms]	max [ms]	avg [ms]	stddev [ms]	conf95 [ms]
Simple	1	6712	67	68	67,12	00,32	[67,06,67,18]
Optimized	1	4709	47	48	47,09	00,29	[47,03,47,15]
Simple	2	12015	120	121	120,15	00,36	[120,08,120,22]
Optimized	2	8216	82	83	82,16	00,37	[82,09,82,23]
Simple	4	22513	225	226	225,13	00,34	[225,06,225,20]
Optimized	4	15120	151	153	151,20	00,42	[151,12,151,28]
Simple	8	43411	434	435	434,11	00,31	[434,05,434,17]
Optimized	8	29116	291	292	291,16	00,37	[291,09,291,23]
Simple	16	85418	854	856	854,18	00,41	[854,10,854,26]
Optimized	16	57118	571	572	571,18	00,38	[571,10,571,26]
Simple	32	169315	1693	1694	1693,15	00,36	[1693,08,1693,22]
Optimized	32	113018	1130	1131	1130,18	00,38	[1130,10,1130,26]
Simple	64	337017	3370	3372	3370,17	00,40	[3370,09,3370,25]
Optimized	64	224914	2249	2250	2249,14	00,35	[2249,07,2249,21]
Simple	128	672613	6726	6727	6726,13	00,34	[6726,06,6726,20]
Optimized	128	448622	4486	4495	4486,22	00,94	[4486,03,4486,41]

Tabelle A.1: *Leseoperationen ohne (`SimpleRaid1Device`) und mit (`OptimizedRaid1Device`) Intra-Request-Parallelisierung (Fortsetzung).*

A.1.2 Inter-Request-Parallelisierung

#Threads		size [KiB]	sum [ms]	min [ms]	max [ms]	avg [ms]	stddev [ms]	conf95 [ms]
Simple	1	1	16291	162	176	162,91	01,89	[162,54,163,28]
Optimized	1	1	11212	111	122	112,12	01,29	[111,87,112,37]
Simple	2	1	15278	152	160	152,78	01,04	[152,58,152,98]
Optimized	2	1	8184	80	90	81,84	02,21	[81,41,82,27]
Simple	1	2	16206	161	168	162,06	00,97	[161,87,162,25]
Optimized	1	2	11168	111	114	111,68	00,55	[111,57,111,79]
Simple	2	2	15270	152	154	152,70	00,70	[152,56,152,84]
Optimized	2	2	8136	80	93	81,36	01,82	[81,00,81,72]
Simple	1	4	16205	161	179	162,05	01,76	[161,71,162,39]
Optimized	1	4	11171	111	115	111,71	00,59	[111,59,111,83]
Simple	2	4	15293	152	164	152,93	01,36	[152,66,153,20]
Optimized	2	4	8147	80	90	81,47	01,81	[81,12,81,82]
Simple	1	8	16201	161	167	162,01	00,70	[161,87,162,15]
Optimized	1	8	11165	111	114	111,65	00,54	[111,54,111,76]
Simple	2	8	15278	152	164	152,78	01,38	[152,51,153,05]
Optimized	2	8	8157	80	91	81,57	02,05	[81,17,81,97]
Simple	1	16	16183	161	164	161,83	00,49	[161,73,161,93]
Optimized	1	16	11164	111	113	111,64	00,50	[111,54,111,74]
Simple	2	16	16070	160	163	160,70	00,79	[160,54,160,86]
Optimized	2	16	8899	88	106	88,99	01,96	[88,61,89,37]
Simple	1	32	17186	171	176	171,86	00,58	[171,75,171,97]
Optimized	1	32	12174	121	125	121,74	00,63	[121,62,121,86]
Simple	2	32	17059	170	178	170,59	00,99	[170,40,170,78]
Optimized	2	32	9670	96	107	96,70	01,14	[96,48,96,92]
Simple	1	64	18188	181	183	181,88	00,43	[181,80,181,96]
Optimized	1	64	13220	131	163	132,20	03,52	[131,51,132,89]
Simple	2	64	18069	180	193	180,69	01,42	[180,41,180,97]
Optimized	2	64	10487	104	115	104,87	01,17	[104,64,105,10]
Simple	1	128	22184	221	223	221,84	00,39	[221,76,221,92]
Optimized	1	128	16233	161	181	162,33	02,39	[161,86,162,80]
Simple	2	128	22047	220	227	220,47	00,92	[220,29,220,65]
Optimized	2	128	16130	161	171	161,30	01,06	[161,09,161,51]
Simple	1	256	28202	281	292	282,02	01,11	[281,80,282,24]
Optimized	1	256	20244	201	225	202,44	03,18	[201,82,203,06]
Simple	2	256	28050	280	283	280,50	00,66	[280,37,280,63]
Optimized	2	256	20172	201	221	201,72	02,40	[201,25,202,19]

Tabelle A.2: *Leseoperationen ohne (`SimpleRaid1Device`) und mit (`OptimizedRaid1Device`) Inter-Request-Parallelisierung.*

#Threads		size [MiB]	sum [ms]	min [ms]	max [ms]	avg [ms]	stddev [ms]	conf95 [ms]
Simple	1	0,5	41243	411	436	412,43	03,09	[411,83,413,03]
Optimized	1	0,5	29255	291	313	292,55	03,25	[291,91,293,19]
Simple	2	0,5	41069	410	420	410,69	01,18	[410,46,410,92]
Optimized	2	0,5	29161	291	304	291,61	01,74	[291,27,291,95]
Simple	1	1	67204	671	682	672,04	01,07	[671,83,672,25]
Optimized	1	1	47270	471	488	472,70	03,16	[472,08,473,32]
Simple	2	1	67063	670	678	670,63	01,03	[670,43,670,83]
Optimized	2	1	47204	471	505	472,04	03,88	[471,28,472,80]
Simple	1	2	120219	1201	1213	1202,19	01,15	[1201,97,1202,41]
Optimized	1	2	82271	821	858	822,71	04,00	[821,93,823,49]
Simple	2	2	120058	1200	1204	1200,58	00,71	[1200,44,1200,72]
Optimized	2	2	82154	821	830	821,54	01,27	[821,29,821,79]
Simple	1	4	225221	2251	2263	2252,21	01,16	[2251,98,2252,44]
Optimized	1	4	151298	1511	1536	1512,98	03,89	[1512,22,1513,74]
Simple	2	4	225072	2250	2254	2250,72	00,90	[2250,54,2250,90]
Optimized	2	4	151168	1511	1520	1511,68	01,67	[1511,35,1512,01]
Simple	1	8	434222	4342	4345	4342,22	00,56	[4342,11,4342,33]
Optimized	1	8	291301	2911	2946	2913,01	04,07	[2912,21,2913,81]
Simple	2	8	434087	4340	4353	4340,87	01,59	[4340,56,4341,18]
Optimized	2	8	291670	2911	3111	2916,70	24,06	[2911,98,2921,42]
Simple	1	16	854408	8541	8602	8544,08	07,75	[8542,56,8545,60]
Optimized	1	16	571232	5711	5720	5712,32	01,18	[5712,09,5712,55]
Simple	2	16	854077	8540	8556	8540,77	01,78	[8540,42,8541,12]
Optimized	2	16	571554	5711	6098	5715,54	38,46	[5708,00,5723,08]
Simple	1	32	1693232	16932	16938	16932,32	00,86	[16932,15,16932,49]
Optimized	1	32	1130235	11302	11316	11302,35	01,57	[11302,04,11302,66]
Simple	2	32	1693067	16930	16936	16930,67	00,88	[16930,50,16930,84]
Optimized	2	32	1131438	11301	12555	11314,38	124,70	[11289,94,11338,82]
Simple	1	64	3370233	33702	33711	33702,33	01,18	[33702,10,33702,56]
Optimized	1	64	2249246	22492	22500	22492,46	01,20	[22492,22,22492,70]
Simple	2	64	3370070	33700	33707	33700,70	00,87	[33700,53,33700,87]
Optimized	2	64	2252062	22491	24987	22520,62	251,02	[22471,42,22569,82]
Simple	1	128	6726256	67262	67274	67262,56	01,86	[67262,20,67262,92]
Optimized	1	128	4486248	44862	44889	44862,48	02,86	[44861,92,44863,04]
Simple	2	128	6726077	67260	67273	67260,77	01,39	[67260,50,67261,04]
Optimized	2	128	4489199	44861	47861	44891,99	298,40	[44833,50,44950,48]

Tabelle A.2: *Leseoperationen ohne (`SimpleRaid1Device`) und mit (`OptimizedRaid1Device`) Inter-Request-Parallelisierung (Fortsetzung).*

A.2 Schreiboperationen

A.2.1 Statischer Speicherpool

	size [MiB]	sum [ms]	min [ms]	max [ms]	avg [ms]	stddev [ms]	conf95 [ms]
Simple	0,5	6443	64	84	64,43	02,07	[64,03,64,83]
Optimized	0,5	4523	45	55	45,23	01,04	[45,03,45,43]
Simple	1	12840	128	134	128,40	00,76	[128,25,128,55]
Optimized	1	9035	90	94	90,35	00,61	[90,23,90,47]
Simple	2	25631	256	259	256,31	00,58	[256,20,256,42]
Optimized	2	18030	180	182	180,30	00,48	[180,21,180,39]
Simple	4	51241	512	525	512,41	01,41	[512,13,512,69]
Optimized	4	36053	360	361	360,53	00,50	[360,43,360,63]
Simple	8	102416	1024	1025	1024,16	00,37	[1024,09,1024,23]
Optimized	8	72071	720	722	720,71	00,48	[720,62,720,80]
Simple	16	204824	2048	2049	2048,24	00,43	[2048,16,2048,32]
Optimized	16	144075	1440	1454	1440,75	01,42	[1440,47,1441,03]
Simple	32	409621	4096	4097	4096,21	00,41	[4096,13,4096,29]
Optimized	32	288066	2880	2882	2880,66	00,57	[2880,55,2880,77]
Simple	64	819236	8192	8194	8192,36	00,61	[8192,24,8192,48]
Optimized	64	576088	5760	5769	5760,88	01,13	[5760,66,5761,10]
Simple	128	1638435	16384	16388	16384,35	00,61	[16384,23,16384,47]
Optimized	128	1152079	11520	11524	11520,79	00,71	[11520,65,11520,93]

Tabelle A.3: *Einfache (`SimpleStaticStoragePool`) und optimierte (`OptimizedStaticStoragePool`) Schreiboperationen.*

Literaturverzeichnis

[CLG+94] Peter M. Chen, Edward K. Lee, Garth A. Gibson, Randy H. Katz, and David A. Patterson. Raid: high-performance, reliable secondary storage. *ACM Comput. Surv.*, 26(2):145–185, 1994.

[CP90] Peter M. Chen and David A. Patterson. Maximizing performance in a striped disk array. In *ISCA '90: Proceedings of the 17th annual international symposium on Computer Architecture*, pages 322–331, New York, NY, USA, 1990. ACM.

[CPB03] Enrique V. Carrera, Eduardo Pinheiro, and Ricardo Bianchini. Conserving disk energy in network servers. In *ICS '03: Proceedings of the 17th annual international conference on Supercomputing*, pages 86–97, New York, NY, USA, 2003. ACM.

[GSKF03] S. Gurumurthi, A. Sivasubramaniam, M. Kandemir, and H. Franke. DRPM: dynamic speed control for power management in server class disks. *Computer Architecture, 2003. Proceedings. 30th Annual International Symposium on*, pages 169–179, 2003.

[Has01] M. Hasenstein. The Logical Volume Manager (LVM). *SuSE Inc*, 2001.

[Law94] Lawrence J. Lamers. *AT Attachment Interface for Disk Drives*. ANSI, 1994.

[Leh99] Greg Lehey. The vinum volume manager. In *ATEC'99: Proceedings of the Annual Technical Conference on 1999 USENIX Annual Technical Conference*, pages 29–29, Berkeley, CA, USA, 1999. USENIX Association.

[SMS+04] J. Satran, K. Meth, C. Sapuntzakis, M. Chadalapaka, and E. Zeidner. Internet Small Computer Systems Interface (iSCSI). RFC 3720 (Proposed Standard), April 2004. Updated by RFC 3980.

[Sta] F.C. Standard. ANSI X3T11. *ANSI X*, 230.

[SWZ98] Peter Scheuermann, Gerhard Weikum, and Peter Zabback. Data partitioning and load balancing in parallel disk systems. *The VLDB Journal*, 7(1):48–66, 1998.

[TM01] D. Teigland and H. Mauelshagen. Volume Managers in Linux. *FREENIX Track: 2001 USENIX Annual Technical Conference, Boston, Massachusetts, USA*, 2001. http://www.usenix.org/event/usenix01/freenix01/full_papers/teigland/teigland_html/, Zugriff am 14.12.2007.

[WBM+06] S.A. Weil, S.A. Brandt, E.L. Miller, D.D.E. Long, and C. Maltzahn. Ceph: A Scalable, High-Performance Distributed File System. *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

[WRPK02] A.I.A. Wang, P. Reiher, G.J. Popek, and G.H. Kuenning. Conquest: Better performance through a disk/persistent-RAM hybrid file system. *Proceedings of the 2002 USENIX Annual Technical Conference*, 2002.

[WZS91] Gerhard Weikum, Peter Zabback, and Peter Scheuermann. Dynamic file allocation in disk arrays. In *SIGMOD '91: Proceedings of the 1991 ACM SIGMOD international conference on Management of data*, pages 406–415, New York, NY, USA, 1991. ACM.

[XMS+03] Q. Xin, EL Miller, T. Schwarz, DDE Long, SA Brandt, and W. Litwin. Reliability mechanisms for very large storage systems. *Mass Storage Systems and Technologies, 2003.(MSST 2003). Proceedings. 20th IEEE/11th NASA Goddard Conference on*, pages 146–156, 2003.

[YW06] Xiaoyu Yao and Jun Wang. Rimac: a novel redundancy-based hierarchical cache architecture for energy efficient, high performance storage systems. *SIGOPS Oper. Syst. Rev.*, 40(4):249–262, 2006.

[ZDD+04] Q. Zhu, FM David, CF Devaraj, Z. Li, Y. Zhou, and P. Cao. Reducing Energy Consumption of Disk Storage Using Power-Aware Cache Management. *High Performance Computer Architecture, 2004. HPCA-10. Proceedings. 10th International Symposium on*, pages 118–118, 2004.

[ZFS] ZFS. http://opensolaris.org/os/community/zfs/.

[ZSZ04] Q. Zhu, A. Shankar, and Y. Zhou. PB-LRU: a self-tuning power aware storage cache replacement algorithm for conserving disk energy. *Proceedings of the 18th annual international conference on Supercomputing*, pages 79–88, 2004.

[ZZ05] Q. Zhu and Y. Zhou. Power-Aware Storage Cache Management. *Power*, 54(5):587–602, 2005.