# Optimierungen

## Hochleistungs-Ein-/Ausgabe

Michael Kuhn

Wissenschaftliches Rechnen
Fachbereich Informatik
Universität Hamburg

2016-06-03

# E/A-Schichten

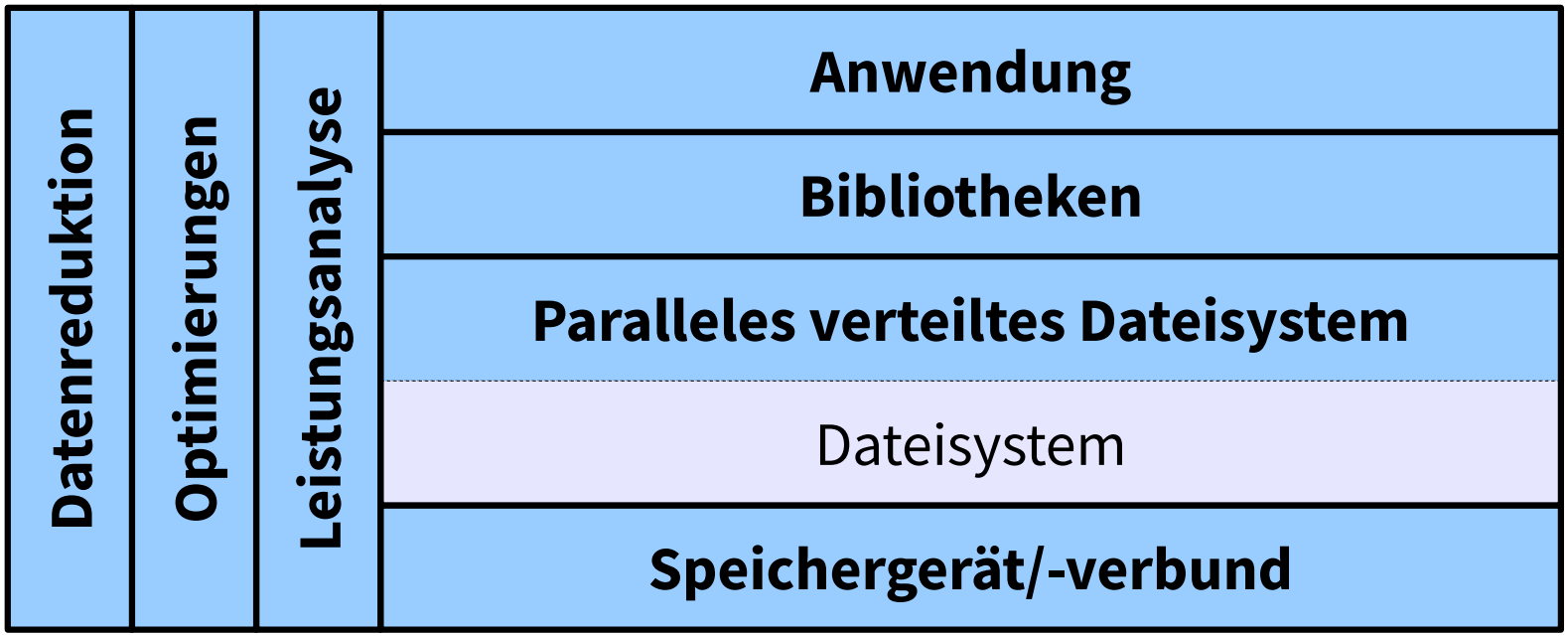



Abbildung: E/A-Schichten

# Einleitung

- Parallele verteilte Dateisysteme sind viel komplexer als traditionelle Dateisysteme
  - Diese Komplexität drückt oft auch die Leistung
  - Außerdem zusätzlicher Overhead durch das Netzwerk
- Außerdem viele zusätzliche E/A-Schichten
  - MPI-IO, HDF, NetCDF etc.
  - Komplexe Interaktion und Optimierungen auf allen Schichten
- Parallele verteilte Dateisysteme für hohe Leistung benötigt
  - Bibliotheken für wissenschaftliche Anwendungen

# Einleitung…

- Unterschiedliche Möglichkeiten zur Leistungssteigerung
  - Einige davon werden vom System selbst gesteuert
  - Andere wiederum vom Benutzer
- Vorteile und Nachteile
  - Systemgesteuerte unabhängig von Benutzerwissen
  - Häufig aber trotzdem zusätzliche Informationen notwendig
    - Z. B. Streifenbreite des parallelen verteilten Dateisystems

# Lastbalancierung

- Verteile die Last so, dass Leistung maximiert wird
  - Muss üblicherweise dynamisch geschehen
- Daten: Maximaler Datendurchsatz
  - Häufig durch Migration der eigentlichen Daten
  - Gleichmäßige Belastung aller Server
  - Beim Lesen häufig Replikation
- Metadaten: Maximaler Anfragendurchsatz
  - Intelligente Metadatenverwaltung
  - Überlastung einzelner Server vermeiden
  - Schwierig, da Metadaten häufig nicht trennbar

# Lastbalancierung…

- Lastbalancierung kann die Leistung negativ beeinflussen
  - Datenverschiebung verursacht erhöhte Last
  - Wiederholte Verschiebung auf weniger ausgelasteten Server
  - System nur noch mit Balancierung beschäftigt
- Zusätzlicher Aufwand
  - Metadaten müssen angepasst werden
  - Caches und Sperren müssen ungültig gemacht werden
- Unterschiedliche Ansätze
  - Server verschieben und replizieren Daten
  - Dynamische Einteilung der Metadatenzuständigkeit

# Lastbalancierung…

- Selten auf Systemebene anzutreffen
  - Wissen über Anwendungsverhalten notwendig etc.
  - Insbesondere nicht in parallelen verteilten Dateisystemen
  - Immer mal wieder neue Ansätze, nie produktionsreif
- Wird eher direkt in den Anwendungen implementiert
  - Kennen eigene Datenstrukturen etc.
  - Können Last dadurch besser verteilen

# Caching

- Caching auf Server-Seite relativ unproblematisch
  - Bis auf mögliche Datenverluste
- Caching auf der Client-Seite interessanter
  - Sammle Daten im Hauptspeicher, danach Versand an Server
  - Grundlage für z. B. Scheduling und Aggregation
- Zur Erhöhung der Geschwindigkeit
  - Weniger kleine Netzwerknachrichten
  - Eventuell auch weniger Daten zu verschicken
    - Z. B. wenn Daten mehrfach überschrieben werden
  - Leseanfragen können unter Umständen direkt aus dem Cache beantwortet werden

# Caching…

- Schreibvorgänge werden im lokalen Cache abgewickelt
  - Daten werden im Hintergrund geschrieben („Write Behind“)
  - Problemlos für Write-only-Zugriffsmuster möglich
    - Zumindest solange keine Überlappung vorliegt
- Neue zu schreibende Puffer werden in eine Liste eingefügt
  - Kann von einigen Threads abgearbeitet werden
  - Thread pro Schreiboperation verursacht zu viel Overhead

# Caching…

- Erhöht aber auch die Wahrscheinlichkeit von Konflikten
  - Gleichzeitiger Zugriff durch mehrere Clients
  - Lesen wegen Konsistenzproblemen problematisch
  - Z. B. Caching auf dem Server statt auf dem Client
- Trotzdem sehr nützlich für einige Szenarien
  - Home-Verzeichnisse der Benutzer
  - Prozess-lokale Dateien bzw. Daten
  - Überall wo keine oder kaum Konflikte auftreten

# Kooperatives Caching

- Globaler Cache auf allen verfügbaren Rechnern [4]
  - Zusammengenommen gigantische Kapazität
- Daten werden aus dem Hauptspeicher eines Clients gelesen
  - Schneller als von der Festplatte des Servers zu lesen
  - Im besten Fall liegen die Daten im lokalen Hauptspeicher
- Analog werden Daten in den Hauptspeicher geschrieben
  - Schreiben auf den Server dann im Hintergrund
  - Vorkehrungen, damit sie auch wirklich dort landen

# Kooperatives Caching…

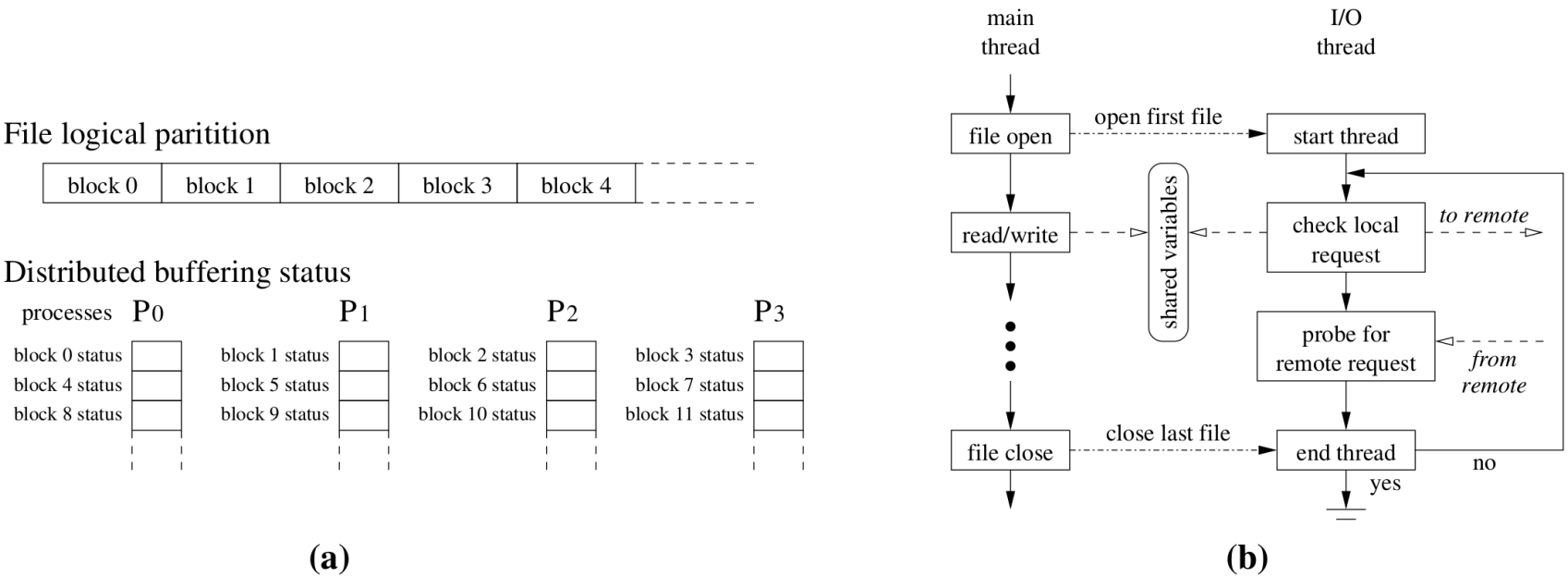



**Fig. 1.** (a) The buffering status is statically distributed among processes in a round-robin fashion. (b) Design of the I/O thread and its interactions with the main thread and remote requests.

- Fällt ein Rechner aus, sind die Daten verloren
  - Lässt sich z. B. durch redundantes Caching verhindern

# Kooperatives Caching…

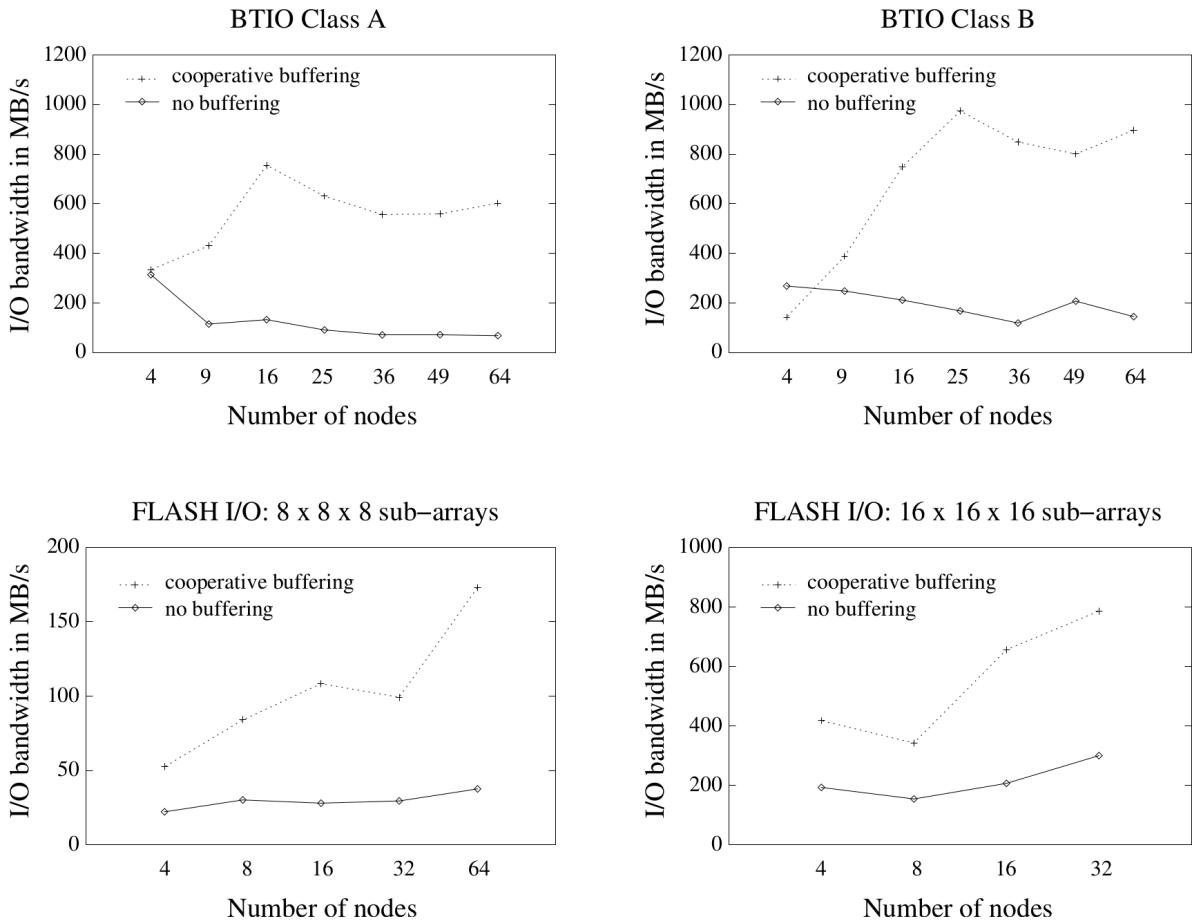



**Fig. 2.** I/O bandwidth results for BTIO and FLASH I/O benchmarks.

# Kooperatives Caching…

- Hat Vor- und Nachteile
  - Verschiebt Last vom Dateisystem in die Anwendung
- Vorteile
  - Dateisystem wird als Flaschenhals eliminiert
  - Zuordnung ist statisch, benötigt keine weitere Koordination
- Nachteile
  - Durchsatz durch den zuständigen Client beschränkt
  - Separate Kommunikations- und Speichernetzwerke erschwert
  - Negativer Einfluss auf Anwendungsleistung

# Replikation

- „Caching“ auf Server-Seite
  - Halte Daten redundant verfügbar
  - Je nach Anwendungsfall näher am Benutzer (Cloud/Grid)
- Dient auch der Lastverteilung
  - Zugriffe mehrerer Benutzer verteilen
- Kritisch bei Modifikation der Daten
  - Daten müssen auf mehreren Servern aktualisiert werden
  - Verringert die Leistung und verursacht Konsistenzprobleme
- Daher Einsatz nur bei Read-mostly-Dateien sinnvoll
  - Nachteile fallen bei Read-only-Dateien ganz weg
  - Überlicherweise eher im Big-Data-Umfeld, selten im HPC

# Metadaten

- Metadatenoperationen sind kritischer Leistungsfaktor
  - Nicht nur in verteilten Systemen
- Beispiel: Zeitstempel für den letzten Zugriff (`atime`)
  - Starte `file *` in einem Verzeichnis mit Millionen Dateien
  - Aktualisiert den Zeitstempel aller Dateien
- Problem lässt sich umgehen
  - `no[dir]atime`, `relatime`, `strictatime` und `lazytime`

"It's also perhaps the most stupid Unix design idea of all times. [...] 'For every file that is read from the disk, let's do a ... write to the disk! And, for every file that is already cached and which we read from the cache ... do a write to the disk!'"

– Ingo Molnar

- `no[dir]atime` aktualisiert die `atime` gar nicht
- `relatime` aktualisiert die `atime` nur wenn die bisherige `atime` vor der `ctime` oder `mtime` liegt (Standard in Linux)
  - Außerdem wenn sie älter als ein Tag ist
- `strictatime` aktualisiert die `atime` bei jedem Zugriff
- `lazytime` aktualisiert die `atime` nur im Hauptspeicher und schreibt sie unter bestimmten Bedingungen zurück
  - Wenn andere Metadatenänderungen geschrieben werden
  - Wenn `fsync`, `syncfs` oder `sync` aufgerufen werden
  - Wenn der Inode aus dem Speicher verworfen wird
  - Wenn der Inode seit mehr als einem Tag nicht zurückgeschrieben wurde

# Metadaten…

- Metadatenoperationen meist voneinander abhängig
  - Deshalb keine parallele Ausführung
  - Beispiele: Pfadauflösung, Datei anlegen
- Es gibt mehrere Ansätze für das Problem
  - Zusammenfassen von Metadatenoperationen
    - Sogenannte Compound Operations
  - Reduzierung der Metadatenoperationen
    - Später: Relaxierte Semantik
  - Intelligente Lastverteilung
    - Gleich: Dynamic Metadata Management

# Beispiel: hashFS

- Reduktion des Overheads bei Pfadauflösung [3]
  - Viele kleine Zugriffe für Metadaten aller Pfadkomponenten
- Hashing für direkten Zugriff auf Metadaten und Daten
  - Nutzt den vollen Pfad
  - Nur ein Lesezugriff pro Dateizugriff
- Umbenennen von Eltern ändert Hash aller Kinder
  1. Hashes werden direkt neu berechnet
     - Potentiell hoher Overhead
  2. Umbenennungen werden in Tabelle gespeichert
     - Zusätzliche Zugriffe auf Tabelle

# Beispiel: Dynamic Metadata Management

- Verteile die Metadatenverwaltung dynamisch [7]
  - Üblicherweise statische Verteilung auf Basis eines Hashes
  - Jeder Metadaten-Server ist für einen oder mehrere Teilbäume des Dateisystembaums verantwortlich
- Clients wissen a priori nicht, welcher Server zuständig ist
  - Clients fragen zufällig bei einem Server nach
  - Server leiten die Anfrage eventuell weiter

# Beispiel: Dynamic Metadata Management…

- Bäume werden zur Laufzeit aufgeteilt und verteilt
  - Erlaubt Anpassung an aktuelle Lastsituation
- Weitere Möglichkeit ist die Replikation der Metadaten
  - Replikation bei starker Nachfrage
  - Verteilung auf unterschiedliche Server
- Vorteile
  - Kann zur Lastverteilung genutzt werden
- Nachteile
  - Mehr Kommunikation, auch zwischen Servern
  - Erhöhte Latenz beim ersten Dateizugriff

# Aggregation

- Basis für diverse Optimierungen
  - Bezeichnet das Zusammenfassen von E/A-Operationen
  - Benötigt irgendeine Art von Caching
- Einzelne Operationen können nur schwer optimiert werden
  - Schreibe 100 Byte an Position x
- Ist mehr Kontext vorhanden, kann besser optimiert werden
  - Schreibe 100 Byte an Position x
  - Schreibe 100 Byte an Position x + 100
  - …

# Aggregation…

- Zusammenfassen kleinerer Operationen
  - Große Operationen sind häufig performanter
  - Eventuell mit vorheriger Umsortierung
- Allein das Zusammenfassen kann schon Vorteile bringen
  - Weniger E/A-Operationen entsprechen weniger Syscalls
  - Löst weniger Kontextwechsel aus
  - Aggregation muss dafür auf Benutzerebene stattfinden

# Scheduling

- Sortiere E/A-Operationen um
  - Entweder auf dem Client oder auf dem Server
  - Benötigt wieder Caching
- Dadurch lässt sich eventuell mehr Leistung erzielen
  - Festplatten haben positionsabhängige Suchzeiten
  - Suchen ist allgemein teuer

# Scheduling…

- Prominentes Beispiel dafür ist Native Command Queueing

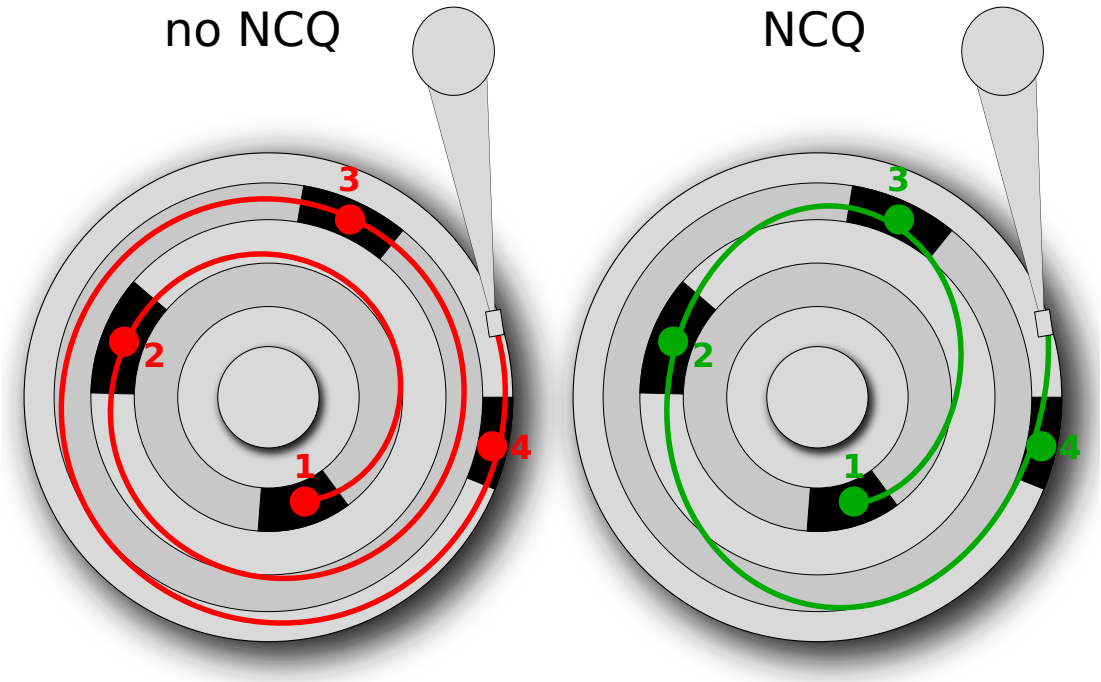



Abbildung: Native Command Queueing [8]

# Beispiel: Scheduling in ZFS

- ZFS weist jeder E/A-Operation Priorität und Deadline zu
  - Je höher die Priorität, desto kürzer die Deadline
- Leseoperationen erhalten allgemein eine höhere Priorität
  - Wichtiger für die Reaktionsrate des Dateisystems
  - Schreiboperationen können zwischengespeichert werden
  - Leseoperationen müssen auf das Speichergerät zugreifen
    - Vorausgesetzt die Daten befanden sich noch nicht im Cache

# Beispiel: Scheduling in ZFS…

| Dateisystem | Ohne Last | Mit Last |
|---|---|---|
| ZFS | 0:09 | 0:10 |
| ext3 | 0:09 | 5:27 |
| reiserfs | 0:09 | 3:50 |

Tabelle: MD5-Prüfsumme einer 512-MB-Datei unter moderater Last [1]

| Dateisystem | Ohne Last | Mit Last |
|---|---|---|
| ZFS | 0:32 | 0:36 |
| UFS | 0:50 | 5:50 |
| ext3 | 0:36 | 54:21 |
| reiserfs | 0:33 | 69:45 |

Tabelle: MD5-Prüfsumme einer 2-GB-Datei unter hoher Last [1]

- Die Leseoperationen sind jetzt viel schneller
- Schreiboperationen benötigen allerdings länger

# Transparente Komprimierung

- Daten werden nur in komprimierter Form gespeichert [9]
  - Beim Schreiben komprimiert
  - Beim Lesen temporär dekomprimiert
- Dabei entspricht der Speedup dem Komprimierungsfaktor
  - Komprimierungsalgorithmus muss die Daten schnell genug verarbeiten können
  - Darf auch den Prozessor nicht zu sehr belasten
- Für die Dekomprimierung zwei verschiedene Verfahren
  - Daten werden dekomprimiert im Speicher gehalten
  - Daten werden komprimiert im Speicher gehalten und erst auf dem Weg zum Prozessor dekomprimiert

# Transparente Komprimierung…

- Daten werden dekomprimiert im Speicher gehalten
  - Daten passieren dreimal die Speicher-Cache-Grenze
  - Beeinflusst eventuell die Leistung negativ
  - Kann bei Read-mostly-Dateien effizienter sein
- Daten werden komprimiert im Speicher gehalten
  - Müssen bei jedem Zugriff zuerst dekomprimiert werden
  - Dafür passen mehr Daten in den Speicher
- CPUs werden rasant leistungsfähiger
  - Speichergeräte verbessern sich deutlich langsamer
  - Nutze Prozessor, um die Leistung zu erhöhen

# Transparente Komprimierung…

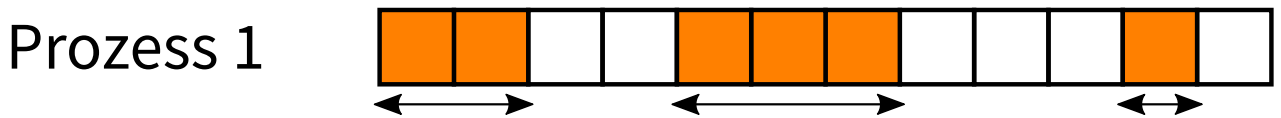
- Ein ähnlicher Ansatz kann unter Linux mit Bordmitteln umgesetzt werden
- zram erlaubt es, komprimierte Blockgeräte zu erstellen [5]
  - Diese können dann als Swap benutzt werden
- Einfluss auf Leistung schwer vorhersagbar
  - Komprimierung und Dekomprimierung nicht bei jedem Zugriff
  - Swapping je nach `swappiness`-Parameter

```
1 $ modprobe zram
2 $ zramctl --find --size 8G
3 /dev/zram0
4 $ mkswap /dev/zram0
5 $ swapon /dev/zram0
```

Listing 1: zram-Swap

# Nicht-zusammenhängende E/A

- E/A-Operationen mit „Löchern“
  - Vergleiche: Sparse Files
  - Benutzer kann z. B. Matrixdiagonale anfordern
- Bietet die Voraussetzungen für diverse Optimierungen



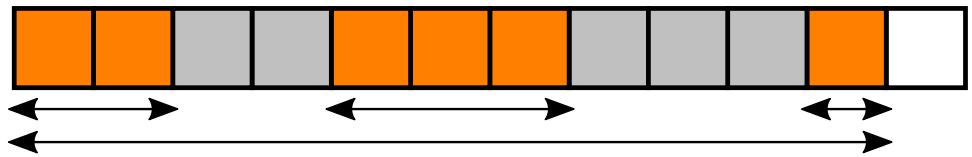
# Nicht-zusammenhängende E/A…

- Bruchstücke müssen einzeln zugegriffen werden
  - Viele kleine Zugriffe sind aber suboptimal
  - Ziel: Zusammenhängende Zugriffe
- Zusammenhängenden Block lesen bzw. schreiben
  - Enthält eventuell mehr Daten als nötig
  - Gleich: Data Sieving
- Mehrere E/A-Anfragen kombinieren
  - Danach: Kollektive E/A

# Data Sieving

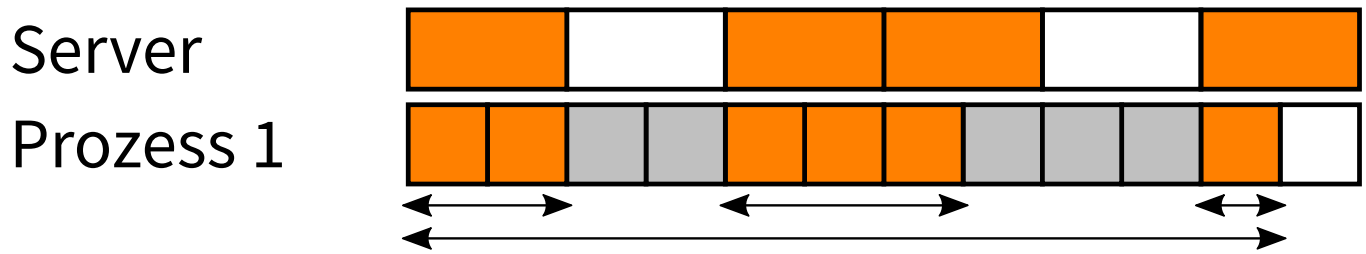
- Optimierung in Bezug auf nicht-zusammenhängende E/A
  - Z. B. in ROMIO implementiert
- Lies zusammenhängende Daten von der Platte
  - Schneller als wiederholtes Suchen
- Verwirf nicht benötigte Daten
  - Lohnt sich nicht immer

Prozess 1



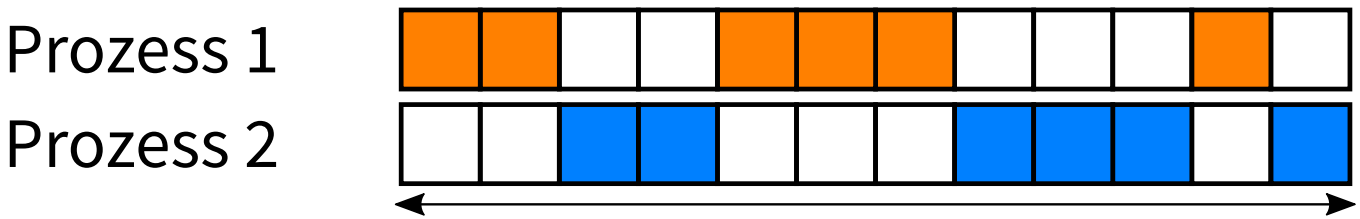
# Data Sieving…

- Schreiben komplizierter
  - Alte Daten müssen zuerst gelesen werden
  - Dadurch Read-Modify-Write
- Geht davon aus, dass zusammenhängende Bereiche einer Datei auch zusammenhängend auf Festplatte liegen
  - Kann eventuell zu Leistungsverlusten führen



# Kollektive E/A

- Clients führen E/A-Operationen koordiniert parallel aus
  - Normalerweise unkoordiniert
- Operationen mehrerer Clients können effektiver zusammengefasst werden
  - Z. B. nicht-zusammenhängende Zugriffe



# Two Phase

- Eine Optimierung für kollektive E/A [6]
  - Z. B. in ROMIO implementiert
- Die Clients koordinieren sich unabhängig vom Dateisystem
  - Clients sind für zusammenhängende Blöcke verantwortlich
  - Die Blöcke sind disjunkt und umfassen alle benötigten Daten
- Evtl. muss so jeder Client nur einen Server kontaktieren
  - Normalerweise kontaktiert jeder Client mehrere Server
  - Das kann netzwerk- und festplatten-technisch günstiger sein

# Two Phase...

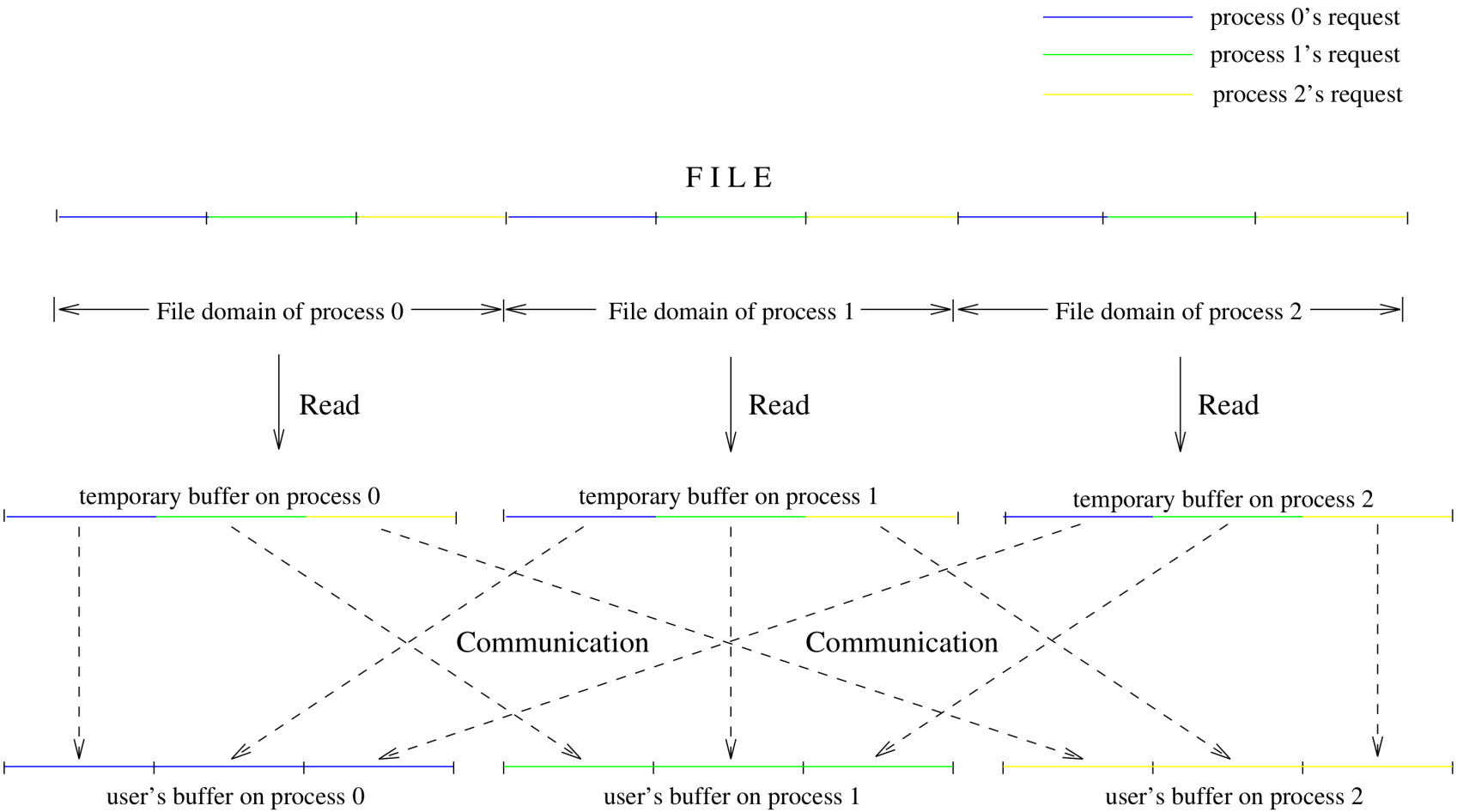


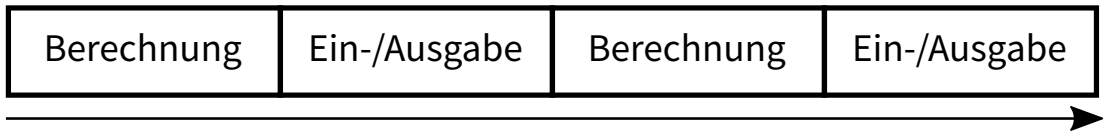

**Figure 3. A simple example illustrating how ROMIO performs a collective read**

# Asynchrone E/A

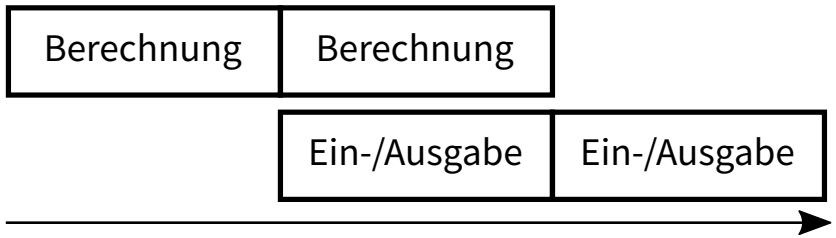
- Überlappung von E/A und Berechnung
  - Nur möglich mit genügend unabhängigen Berechnungen
- Idee: Nicht auf E/A warten und weiterrechnen
  - Spezielle asynchrone E/A-Funktionen
    - Z. B. `MPI_File_iwrite` und `aio_write`
  - Status kann mit speziellen Funktionen überprüft werden
    - Z. B. `MPI_test` und `aio_return`
- Birgt das Risiko von Race Conditions
- Der maximale Speedup ist 2

# Asynchrone E/A…

- Nach jeder Berechnungsphase werden Zwischenergebnisse geschrieben
  - Traditionell blockiert der Schreibvorgang, bis er beendet ist




- Mit asynchroner E/A wird nebenläufig geschrieben
  - Nur möglich, wenn Berechnung Daten nicht verändert
  - Alternativ zusätzlicher Puffer für asynchrone E/A



# Hints

- Idee: Stelle dem Dateisystem so viele zusätzliche Informationen wie möglich zur Verfügung
  - Zugriffe können dann (hoffentlich) optimiert werden
- Hints sind üblicherweise optional
  - Andererseits muss der Benutzer aber auch keine angeben
- Hints können für diverse Optimierungen genutzt werden
  - Informationen über die Zugriffsarten: read-only, read-mostly, append-only, non-contiguous access
  - Steuerung von Puffergrößen
  - Anzahl der an Two Phase beteiligten Prozesse

# Zugriffsmuster

- Möglichst sequentiell auf Daten zugreifen
  - Nicht hier und da ein Stückchen lesen, wenn man im Endeffekt sowieso die ganze Datei braucht
- Viele Suchvorgänge verhindern
  - Kopfbewegungen sind bei Festplatten sehr langsam
  - Kommunikation mit vielen unterschiedlichen Servern
- Viele kleine Anfragen verhindern
  - Wie beim Nachrichtenaustausch lieber einige große
  - Bei E/A zusätzlich zur Netzwerklatenz noch Suchzeiten

# Zugriffsmuster…

- Auswirkungen der E/A-Funktionen genau kennen
  - Z. B. welche Funktionen synchronisieren oder kollektiv sind
- Mitunter der wichtigste Aspekt für Leistungssteigerungen
  - Bei von Haus aus ineffizienten Programmen können Dateisystem und Bibliotheken auch nicht mehr viel retten
  - Bezieht sich nicht nur auf E/A ☺

# Relaxierte Semantik

- Anpassen der Semantik an die Anwendungsanforderungen
  - Daten: Änderungen nicht sofort sichtbar machen
  - Metadaten: Nicht alle speichern (z. B. Zeitstempel)
- Stelle dem Benutzer diese Möglichkeiten zur Verfügung
  - Der Benutzer weiß meist am besten, was er braucht
  - Gibt es leider so gut wie nirgends
- Üblicherweise Unterstützung für eine einzige Semantik
  - Für einige Anwendungsfälle geeignet, aber nie für alle

# Relaxierte Semantik…

- Optimal wäre Kontrolle durch den Benutzer
  - Z. B. ein sicherer und ein performanter Modus
- Benutze jeweils unterschiedliche Locking-Mechanismen
  - Gar kein Locking im performanten Modus
- Bewerte Datensicherheit unterschiedlich
  - Keine zusätzliche Datensicherheit im performanten Modus
- Unterschiedliche Konsistenzanforderungen
  - Intensives Caching im performanten Modus
- Performanter Modus gut für temporäre Dateien

# Beispiel: Directory-Based Metadata Optimizations

- Bezieht sich auf PVFS bzw. OrangeFS
  - Dateien aus einem Metafile und mehreren Datafiles
  - Indirektion erhöht die Anzahl der Metadatenoperationen
- Selbst kleine Dateien werden über mehrere Server verteilt
  - Standardmäßige Streifenbreite ist 64 KiB
- Idee: Eliminiere das Metafile komplett [2]
  - Vorteil: Reduzierung der Metadatenoperationen
  - Nachteil: Keine Metadaten mehr verfügbar, nur für kleine Dateien sinnvoll
  - Benutzer muss Funktionalität explizit aktivieren

# Beispiel: Directory-Based Metadata Optimizations…

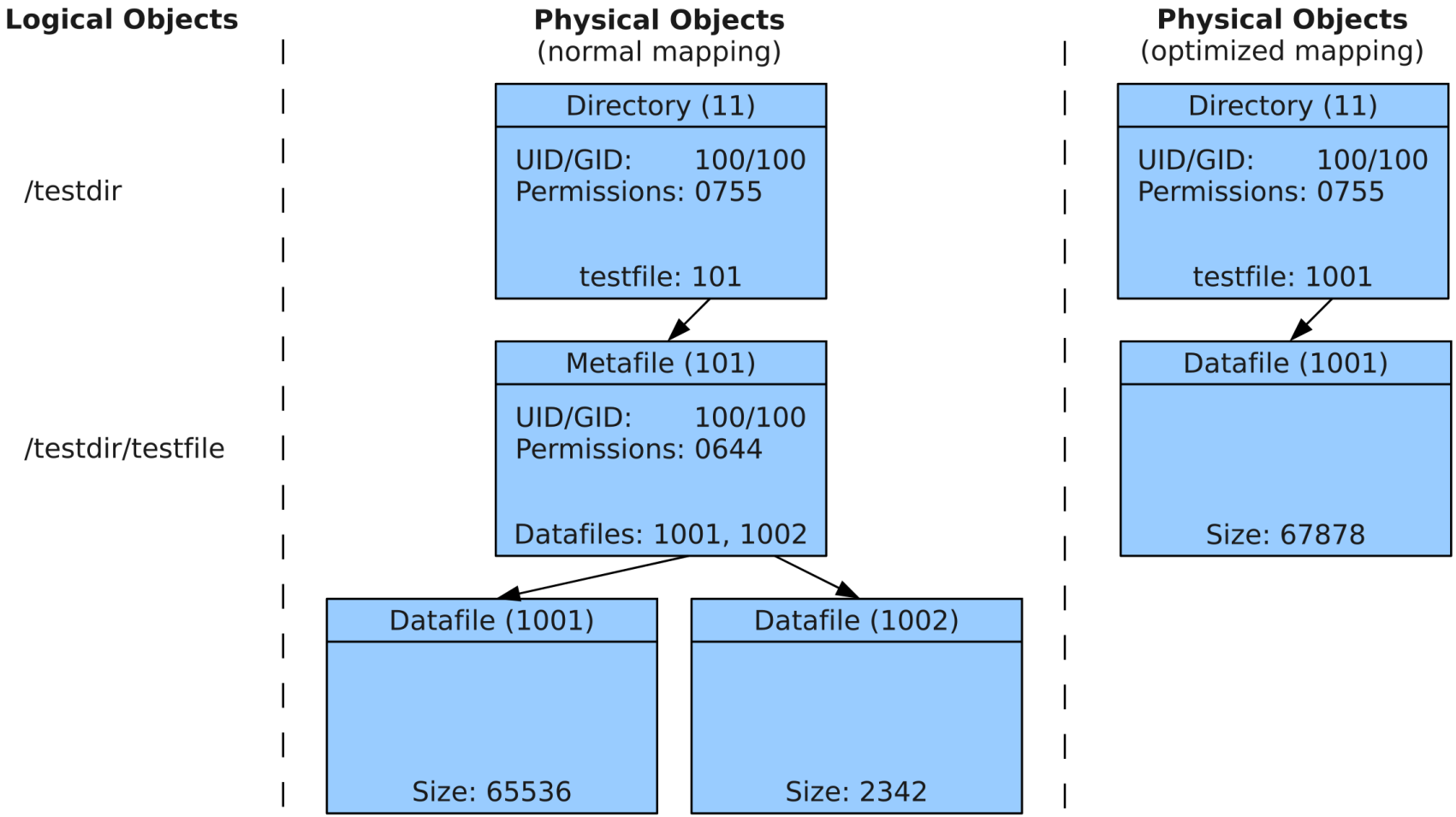



Figure 1. Directory tree.

# Beispiel: Directory-Based Metadata Optimizations…

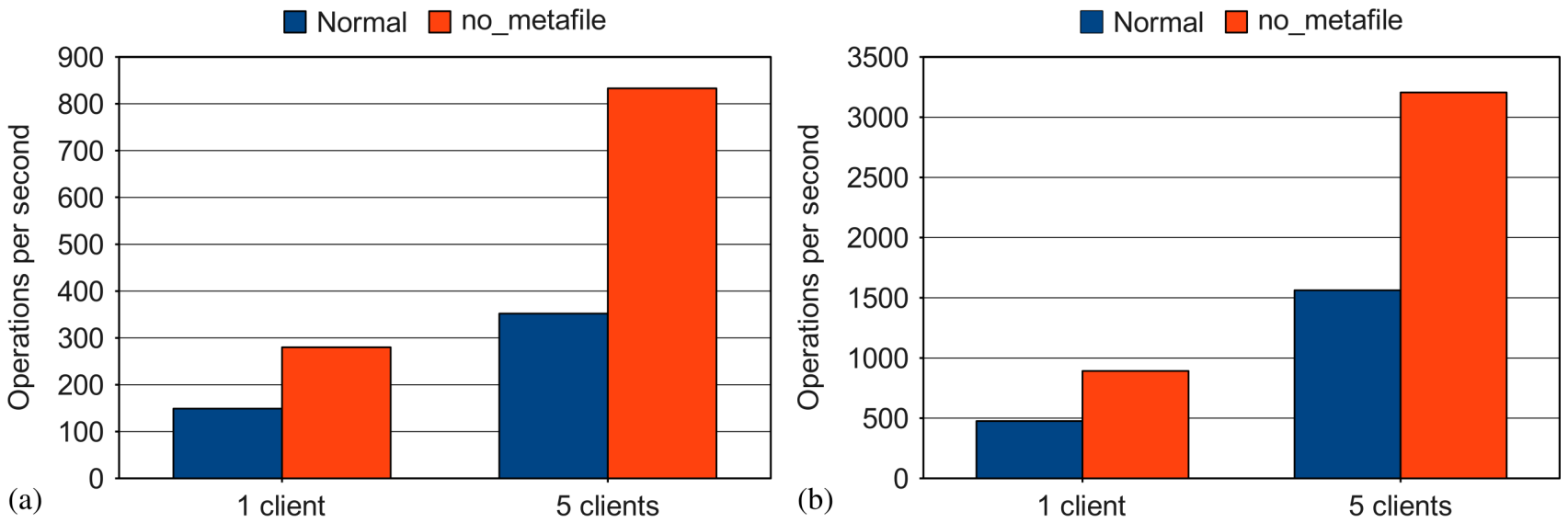



Figure 2. File creation: (a) on disk and (b) on `tmpfs`.

- 50.000 Dateien pro Client
- Höherer Leistungsgewinn bei mehreren parallelen Clients
  - Metadatenänderungen standardmäßig synchronisiert

# Beispiel: Directory-Based Metadata Optimizations…

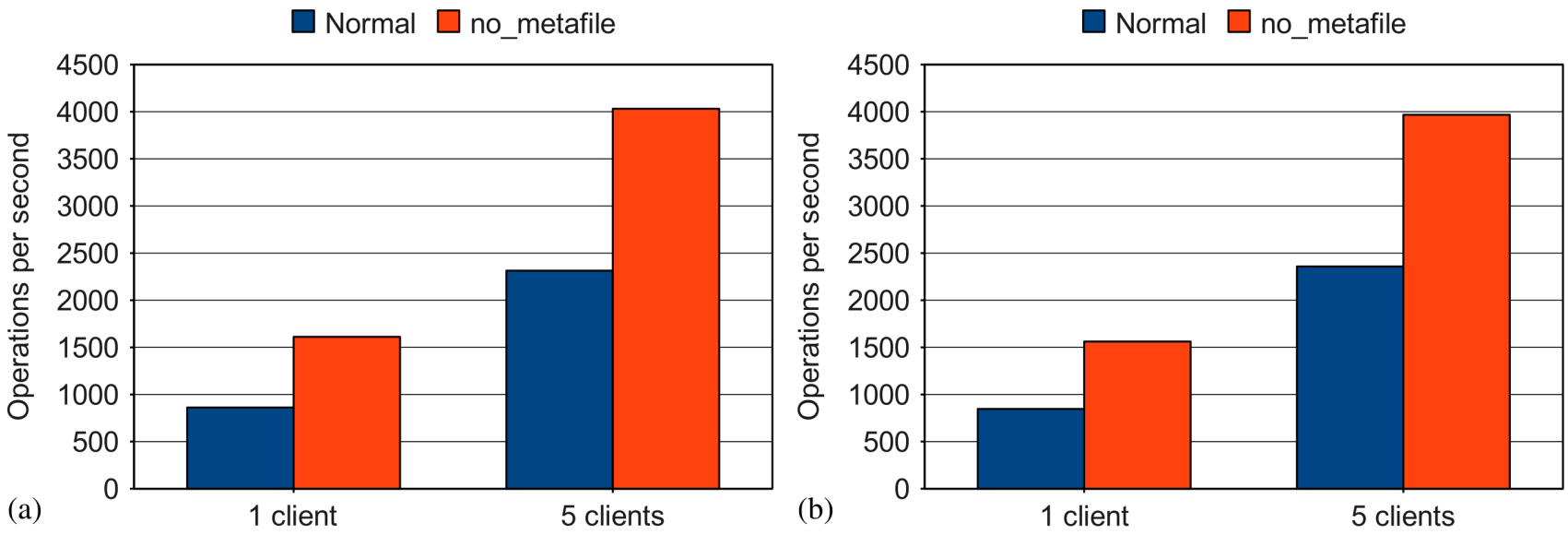



Figure 3. File listing: (a) on disk and (b) on `tmpfs`.

# Beispiel: Directory-Based Metadata Optimizations…

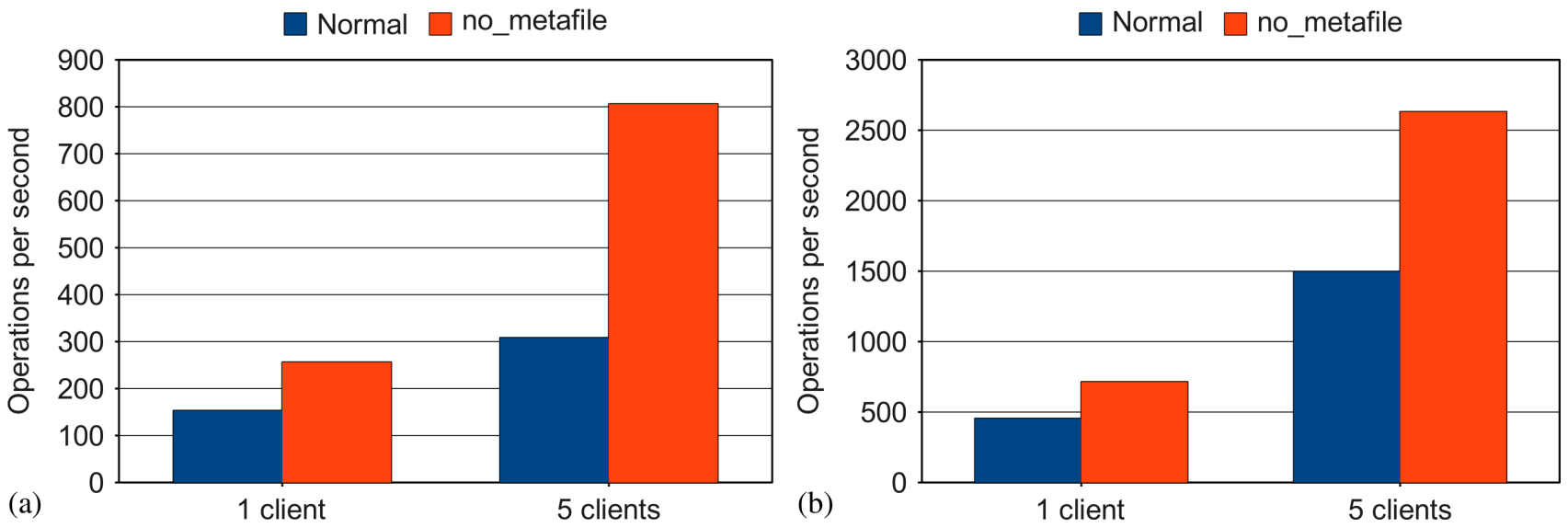



Figure 4. File removal: (a) on disk and (b) on `tmpfs`.

# Zusammenfassung

- Es gibt eine Vielzahl unterschiedlicher Optimierungen
  - Auf allen Ebenen des E/A-Stacks
  - Caching meist sehr wichtig
- Leistungsfähigkeit hängt entscheidend vom Benutzer ab
  - Muss so viele Informationen wie möglich bereitstellen
    - Zugriffsmuster, Zugriffsart, Topologie
  - Das System kann oft nur dann effektiv optimieren
- Wenn möglich sollte der Benutzer auch manuell optimieren
  - Anpassen der Semantik, Optimierung der Zugriffsmuster

# Quellen I

[1] Chad Mynhier. ZFS I/O reordering benchmark. http://cmynhier.blogspot.com/2006/05/zfs-io-reordering-benchmark.html.

[2] Michael Kuhn, Julian Kunkel, and Thomas Ludwig. Dynamic file system semantics to enable metadata optimizations in PVFS. *Concurrency and Computation: Practice and Experience*, pages 1775–1788, 2009.

[3] Paul Hermann Lensing, Toni Cortes, and André Brinkmann. Direct lookup and hash-based metadata placement for local file systems. In *Proceedings of the 6th International Systems and Storage Conference*, SYSTOR '13, pages 5:1–5:11, New York, NY, USA, 2013. ACM.

# Quellen II

[4] Wei-keng Liao, Kenin Coloma, Alok Choudhary, and Lee Ward. Cooperative Write-behind Data Buffering for MPI I/O. In *Proceedings of the 12th European PVM/MPI Users' Group Conference on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, PVM/MPI'05, pages 102–109, Berlin, Heidelberg, 2005. Springer-Verlag.

[5] Nitin Gupta. zram: Compressed RAM based block devices. `https://www.kernel.org/doc/Documentation/blockdev/zram.txt`, 11 2015. Last accessed: 2016-04.

[6] Rajeev Thakur, William Gropp, and Ewing Lusk. Data Sieving and Collective I/O in ROMIO. In *Proceedings of the The 7th Symposium on the Frontiers of Massively Parallel Computation*, FRONTIERS '99, pages 182–, Washington, DC, USA, 1999. IEEE Computer Society.

# Quellen III

[7] Sage A. Weil, Kristal T. Pollack, Scott A. Brandt, and Ethan L. Miller. Dynamic Metadata Management for Petabyte-Scale File Systems. In *Proceedings of the 2004 ACM/IEEE Conference on Supercomputing*, SC '04, pages 4–, Washington, DC, USA, 2004. IEEE Computer Society.

[8] Wikipedia. Native Command Queuing. http://de.wikipedia.org/wiki/Native_Command_Queuing.

[9] Marcin Zukowski. Improving I/O Bandwidth for Data-Intensive Applications.