Optimierungen

Hochleistungs-Ein-/Ausgabe

Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

Michael Kuhn

2019-05-14

Wissenschaftliches Rechnen
Fachbereich Informatik
Universität Hamburg

Optimierungen

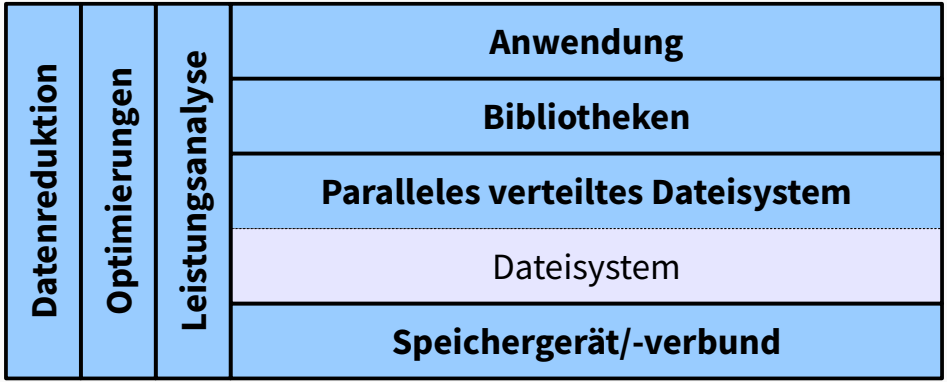


Abbildung 1: E/A-Schichten und orthogonale Themen

Einleitung

- Parallele E/A ist viel komplexer als serielle E/A
  - Zusätzliche Komplexität durch parallele verteilte Dateisysteme
  - Zugriff über mehrere zusätzliche E/A-Schichten
  - Außerdem zusätzlicher Overhead durch das Netzwerk
- Diese Komplexität beeinflusst oft auch die Leistung
  - Parallele verteilte Dateisysteme für hohe Leistung benötigt
  - Bibliotheken für wissenschaftliche Anwendungen notwendig
    - MPI-IO, HDF, NetCDF etc.
- Komplexe Interaktion und Optimierungen auf allen Schichten

Einleitung...

- Unterschiedliche Möglichkeiten zur Leistungssteigerung
  - Einige davon werden vom System selbst gesteuert, andere wiederum vom Benutzer
  - Hybride Ansätze, die Informationen vom Benutzer benötigen
- Vor- und Nachteile
  - Systemgesteuerte unabhängig von Benutzerwissen
    - Erfordern keinen Zusatzaufwand auf Benutzerseite
    - Können deswegen aber häufig nicht optimale Leistung erzielen
  - Für hohe Leistungsausbeute häufig trotzdem zusätzliche Informationen notwendig
    - Z. B. Streifenbreite des parallelen verteilten Dateisystems

- Grundlage für z. B. Aggregation und Scheduling
- Auf Server-Seite relativ unproblematisch
  - Cache existiert nur an einer Stelle, daher keine Konsistenzprobleme
  - Mögliche Datenverluste bei Absturz des Servers
- Auf der Client-Seite interessanter
  - Sammle Daten im Hauptspeicher, danach Versand an Server
    - Weniger kleine Netzwerknachrichten
  - Eventuell auch weniger Daten zu verschicken
    - Daten werden mehrfach geändert aber nur finaler Zustand ist von Interesse
- Client-seitiges Caching ist allerdings nicht immer einfach möglich
  - POSIX schreibt vor, dass Änderungen sofort global sichtbar sein müssen
    - Caching daher nur möglich, wenn Dateien nicht an anderer Stelle geöffnet sind

Caching...

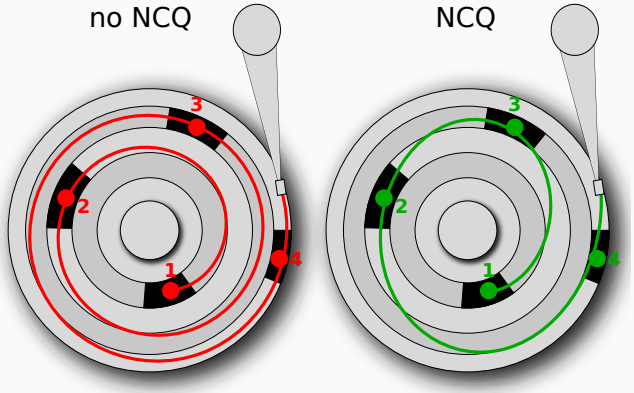
- Leseanfragen im besten Fall direkt aus dem Cache
  - Insbesondere in Kombination mit Read Ahead
  - Versteckt Latenz der Leseanfragen, da nicht auf Daten gewartet werden muss
- Schreibvorgänge werden im lokalen Cache abgewickelt
  1. Daten werden in den Cache geschrieben, später auf das Speichergerät (“write-behind”)
     - Problemlos für konfliktfreie Zugriffsmuster möglich
     - Z. B. Write-only-Zugriffsmuster ohne Überlappungen
  2. Daten werden in den Cache und auf das Speichergerät geschrieben (“write-through”)
- Cache kann durch mehrere Threads abgearbeitet werden
  - Ein Thread häufig nicht ausreichend für maximale Leistung
  - Thread pro Schreiboperation verursacht zu viel Overhead

- Caching erhöht die Wahrscheinlichkeit von Konflikten
  - Gleichzeitiger Zugriff durch mehrere Clients
  - Veraltete Inhalte führen zu Kohärenz- und Konsistenzproblemen
- Trotzdem sehr nützlich für einige Szenarien
  - Caching auf dem Server fast immer sinnvoll
  - Überall wo keine oder kaum Konflikte auftreten
    - Home-Verzeichnisse der Benutzer
    - Prozess-lokale Dateien bzw. Daten
- Später: Burst Buffer
  - Zusätzliche Cache-Stufe zur Beschleunigung des Dateisystems

Scheduling

- Sortiere E/A-Operationen um
  - Entweder auf dem Client oder auf dem Server möglich
  - Benötigt Caching, um sinnvoll funktionieren zu können
  - Häufig als Vorstufe zu Aggregation
- Durch Scheduling lässt sich eventuell mehr Leistung erzielen
  - Festplatten haben positionsabhängige Zugriffszeiten
    - Suchzeit (4–15 ms) und Rotationslatenz (2–7 ms)
  - Auch bei SSDs kann Scheduling sinnvoll sein
    - Z. B. gleichzeitiger Zugriff auf mehrere Flash-Zellen
  - Suchen ist allgemein teuer
- Linux unterstützt mehrere E/A-Scheduler
  - Unter anderem `cfq`, `deadline` und `noop`

Scheduling... [7]



- Prominentes Beispiel ist Native Command Queueing
  - Durch Anpassung der Zugriffsreihenfolge kann schnellere Abarbeitung erreicht werden

Aggregation

- Basis für diverse Optimierungen
  - Bezeichnet das Zusammenfassen von E/A-Operationen
  - Benötigt irgendeine Form von Caching
- Einzelne Operationen können nur schwer optimiert werden
  - „Schreibe 100 Bytes an Position 2342“
- Ist mehr Kontext vorhanden, kann besser optimiert werden
  - „Schreibe jeweils 100 Bytes an den Positionen 2342, 2442 und 2542“
  - Reihenfolge ist unter Umständen problematisch

Aggregation...

- Zusammenfassen kleinerer Operationen
  - Große Operationen sind häufig performanter
    - Reduziert Suchzeiten und Read-Modify-Write
  - Eventuell mit vorheriger Umsortierung
    - Gleich: Scheduling
- Allein das Zusammenfassen kann schon Vorteile bringen
  - Weniger E/A-Operationen entsprechen weniger Syscalls
    - Löst weniger Kontextwechsel aus
    - Aggregation muss dafür auf Benutzerebene stattfinden
- Aggregation ist weit verbreitet
  - Fast alle E/A-Scheduler in Linux fassen Operationen zusammen
    - Sogar noop

Replikation

- „Caching“ auf Server-Seite
  - Daten werden redundant verfügbar gehalten
  - Je nach Anwendungsfall näher am Benutzer (Cloud/Grid)
- Dient auch der Lastverteilung
  - Zugriffe mehrerer Benutzer können verteilt werden
- Kritisch bei Modifikation der Daten
  - Daten müssen auf mehreren Servern aktualisiert werden
  - Verringert die Leistung und verursacht Konsistenzprobleme
- Daher Einsatz eher bei Read-mostly-Dateien sinnvoll
  - Nachteile fallen bei Read-only-Dateien ganz weg
  - Üblicherweise eher im Big-Data-Umfeld, zunehmend auch 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
    - Außerdem müssen die ersten Bytes jeder Datei gelesen werden
- Problem lässt sich umgehen
  - `no[dir]atime`, `relatime`, `strictatime` und `lazytime`
  - Alternativ `O_NOATIME` bei `open`

> “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 `atime`, `ctime` und `mtime` nur im Hauptspeicher und schreibt sie unter folgenden 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 häufig keine parallele Ausführung möglich
  - Beispiele: Pfadauflösung, Erstellen einer Datei
- Es gibt mehrere Ansätze für das Problem
  - Zusammenfassen von Metadatenoperationen
    - Sogenannte Compound Operations
  - Reduzierung der Metadatenoperationen
    - Später: Relaxierte Semantik
  - Intelligente Lastverteilung
    - Später: Dynamic Metadata Management

Beispiel: Kooperatives Caching [4]

- Globaler Cache auf allen verfügbaren Rechnern
  - Zusammengenommen gigantische Kapazität (TB–PB)
  - Beschleunigt Zugriffe auf Dateien
- Daten werden aus dem Hauptspeicher eines Clients gelesen
  - Schneller als vom Dateisystem 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

Beispiel: Kooperatives Caching... [4]

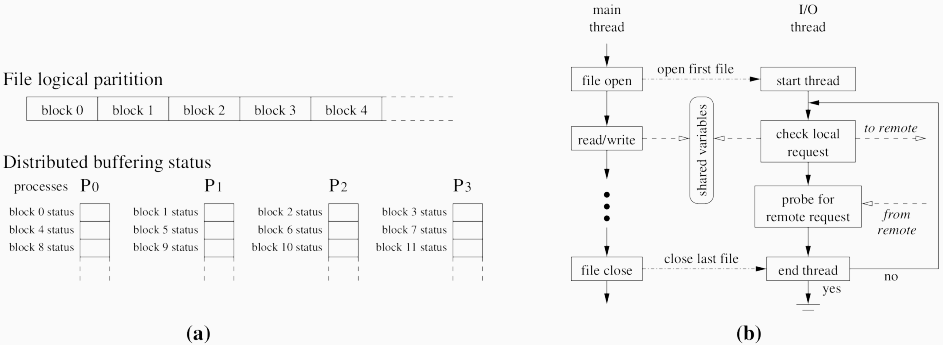


Fig. 1. (a) The buffering status is statically distributed among processes in a round-robin fashion. (b) Design of the I/O thread and its interactions with the main thread and remote requests.

- Fällt ein Rechner aus, sind die Daten verloren
  - Lässt sich z. B. durch Redundanz oder regelmäßiges Zurückschreiben verhindern

Beispiel: Kooperatives Caching... [4]

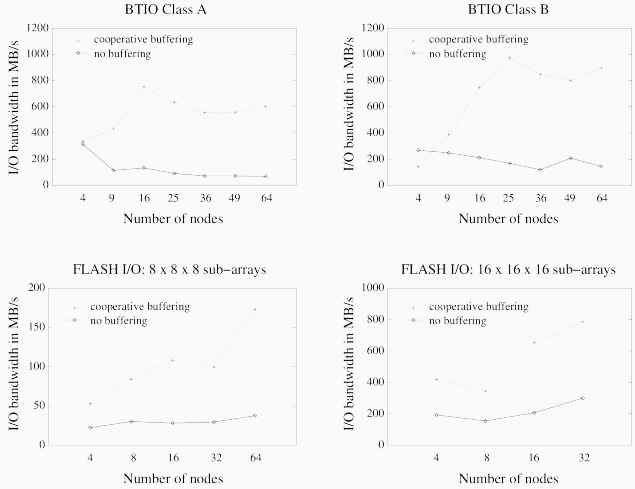


Fig. 2. I/O bandwidth results for BTIO and FLASH I/O benchmarks.

Beispiel: Kooperatives Caching... [4]

- Verschiebt Last vom Dateisystem in die Anwendung
- Vorteile
  - Dateisystem wird als Flaschenhals eliminiert
  - Zuordnung ist statisch, benötigt keine weitere Koordination
  - Optimales Zugriffsmuster durch zuständige Clients
  - Kommunikationsdurchsatz meistens höher als Dateisystemdurchsatz
- Nachteile
  - Verwendeter Hauptspeicher steht nicht für Anwendung bereit
  - Durchsatz durch den zuständigen Client beschränkt
  - Separate Kommunikations- und Speichernetzwerke erschwert
  - Negativer Einfluss auf Anwendungsleistung

Beispiel: Scheduling in ZFS [1]

- 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
- Der `deadline`-Scheduler für Linux funktioniert ähnlich

Beispiel: Scheduling in ZFS... [1]

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

(a) 512-MB-Datei mit moderater Last

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

(b) 2-GB-Datei mit hoher Last

- Die Leseoperationen sind bei ZFS unter Last viel schneller
  - Wichtig für die Interaktivität eines Systems
- Schreiboperationen benötigen allerdings länger
  - Schreibvorgänge üblicherweise besser durch Cache handhabbar

Beispiel: hashFS [3]

- Reduktion des Overheads bei Pfadauflösung
  - Üblicherweise viele kleine Zugriffe für Metadaten aller Pfadkomponenten
- Hashing für direkten Zugriff auf Metadaten und Daten
  - Nutzt den vollen Pfad
  - Erfordert nur einen Lesezugriff pro Dateizugriff
- Umbenennen von Eltern ändert Hashes aller Kinder
  1. Hashes werden direkt neu berechnet
     - Potentiell hoher Overhead
  2. Umbenennungen werden in Tabelle gespeichert
     - Zusätzliche Zugriffe auf Tabelle
- Zugriffsberechtigungen aller Eltern müssen weiterhin beachtet werden

Beispiel: Dynamic Metadata Management [6]

- Üblicherweise statische Verteilung der Metadaten auf Basis eines Hashes
  - Stattdessen Zuständigkeit auf Basis von Teilbäumen
- Verteile die Metadatenverwaltung dynamisch
  - Metadaten-Server sind für einen oder mehrere Dateisystem-Teilbäume verantwortlich
  - Zuständigkeiten können zur Laufzeit verschoben werden
- Clients wissen a priori nicht, welcher Server zuständig ist
  - Clients fragen zufällig bei einem Server nach
  - Server leiten die Anfrage wenn nötig weiter

Beispiel: Dynamic Metadata Management... [6]

- 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

Beispiel: Dynamic Metadata Management... [6]

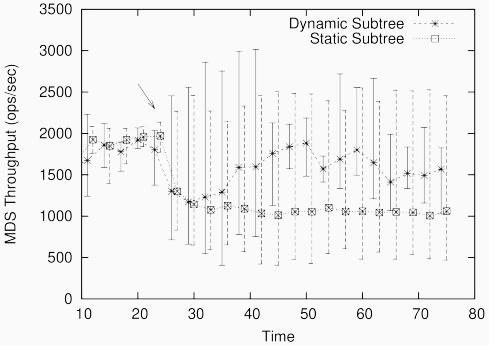


Figure 5: The range and average throughput of MDSs is shown under a dynamic workload. When clients migrate and create files in new portions of the hierarchy, a static subtree distribution remains unbalanced, while the dynamic partition re-balances load and achieves higher average performance by migrating newly popular portions of the hierarchy to non-busy nodes.

Beispiel: Dynamic Metadata Management... [6]

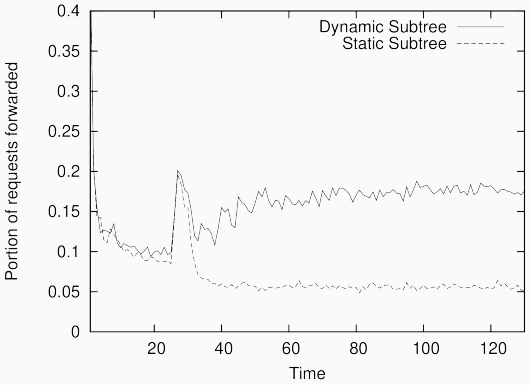


Figure 6: Forwarded requests for static and dynamic partitioning under a dynamic workload. The spike represents a shift in workload, while the difference after that point highlights overhead due to client ignorance of metadata movement from dynamic load balancing.

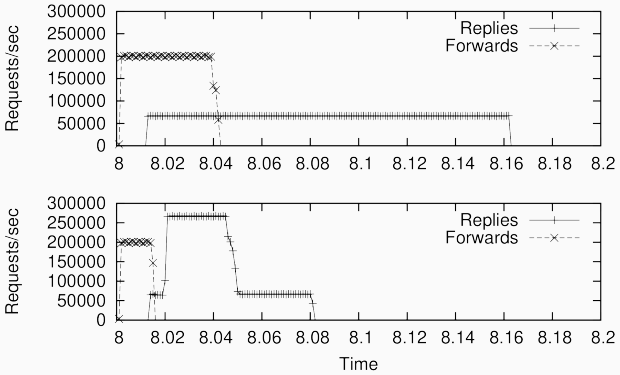


Figure 7: No traffic control (top): nodes forward all requests to the authoritative MDS who slowly responds to them in sequence. Traffic control (bottom): the authoritative node quickly replicates the popular item and all nodes respond to requests.

Nicht-zusammenhängende E/A

- Traditionell können zusammenhängende Bereiche gelesen und geschrieben werden
  - Native Unterstützung in MPI-IO
  - Bei POSIX nur über Umwege möglich
- E/A-Operationen mit „Löchern“
  - Vergleiche: Sparse Files
  - Benutzer kann z. B. Matrixdiagonale anfordern
- Bietet die Voraussetzungen für diverse Optimierungen
  - Insbesondere in Kombination mit kollektiver E/A

Nicht-zusammenhängende E/A...

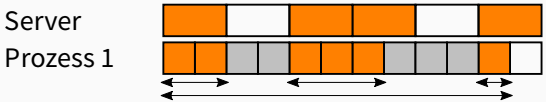
- Bruchstücke müssen einzeln zugegriffen werden
  - Speichergeräte erlauben üblicherweise nur blockweisen Zugriff
  - Viele kleine Zugriffe sind aber suboptimal
    - Ziel: Zusammenhängende Zugriffe
- Zwei Möglichkeiten
  1. Zusammenhängenden Block lesen bzw. schreiben
     - Enthält eventuell mehr Daten als nötig
     - Gleich: Data Sieving
  2. Mehrere nicht-zusammenhängende E/A-Anfragen kombinieren
     - Ergibt in Summe evtl. einen zusammenhängenden Zugriff
     - Danach: Kollektive E/A

Data Sieving

- Optimierung in Bezug auf nicht-zusammenhängende E/A
  - Z. B. in ROMIO implementiert
- Zugriff auf zusammenhängende Daten auf dem Speichersystem
  - Häufig schneller als viele kleine Zugriffe
    - Große Zugriffe üblicherweise auch auf nicht-rotierenden Medien schneller
- Nicht benötigte Daten werden verworfen
  - Lohnt sich nicht immer, daher Kostenabschätzung notwendig
  - Abschätzung in parallelen verteilten Dateisystemen erschwert

Data Sieving...

- Lesen relativ unproblematisch
- Schreiben komplizierter
  - Alte Daten müssen zuerst gelesen werden
    - Dadurch Read-Modify-Write
- Kann in beiden Fällen eventuell zu Leistungsverlusten führen
  - Logisch zusammenhängend $\neq$ physikalisch zusammenhängend
    - Dateisystem-Allokation, Sektoren-Remapping, Verteilung etc.
  - Evtl. wird auch unnötig mit zusätzlichen Servern kommuniziert



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

- Im nicht-kollektiven Fall eventuell zuerst Prozess 2, dann Prozess 1
  - Sieht für das Dateisystem wie zufällige Zugriffe aus
  - Außerdem viele kleine Anfragen durch nicht-zusammenhängenden Zugriff

Two Phase [5]

- Eine Optimierung für kollektive E/A
  - 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
- Im besten Fall muss jeder Client nur einen Server kontaktieren
  - Kann weniger Netzwerk- und Festplatten-Overhead verursachen
    - Normalerweise kontaktiert jeder Client mehrere Server
- Zusätzlicher Kommunikationsaufwand kann nachteilig sein
  - Im schlechtesten Fall wird gesamte Datenmenge noch einmal kommuniziert

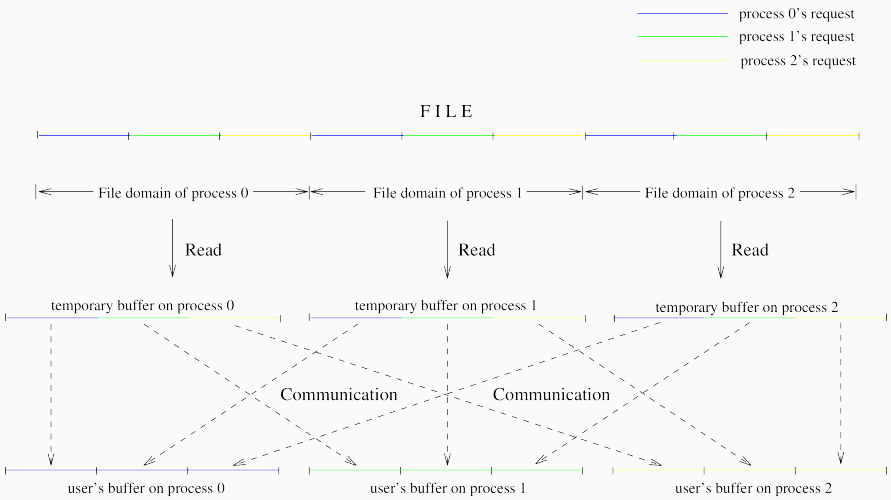
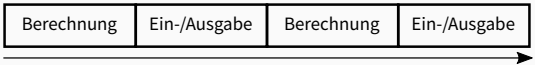


Figure 3. A simple example illustrating how ROMIO performs a collective read

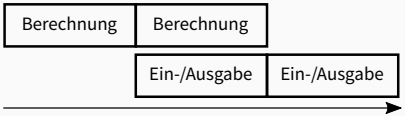
- Überlappung von E/A und Berechnung
  - Nur möglich mit genügend unabhängigen Berechnungen
    - Während E/A darf üblicherweise nicht auf Puffer zugegriffen werden
- Idee: Nicht auf E/A warten, sondern weiterrechnen
  - Spezielle asynchrone E/A-Funktionen
    - Z. B. `MPI_File_iwrite` und `aio_write`
  - Status kann mit speziellen Funktionen überprüft werden
    - Z. B. `MPI_test` und `aio_return`
- Birgt das Risiko von Race Conditions
  - Daten dürfen erst verändert werden, wenn E/A beendet ist
  - Kann durch separate Puffer vermieden werden

Asynchrone E/A...

- Nach Berechnung 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



- Einschränkung: Der maximale Speedup ist 2

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

Lastbalancierung...

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

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

Hinweise

- Idee: So viele zusätzliche Informationen wie möglich zur Verfügung stellen
  - Zugriffe können dann hoffentlich optimiert werden
- Hints sind üblicherweise optional
  - Der Benutzer muss sie für die normale Benutzung nicht angeben
  - Das System kann sie allerdings auch ignorieren
- Hints können für diverse Optimierungen genutzt werden
  - Informationen über die Zugriffsarten: read-only, read-mostly, append-only, non-contiguous access, unique, sequential
  - Steuerung von Puffergrößen
  - Anzahl der an Two Phase beteiligten Prozesse

Relaxierte Semantik

- Anpassen der Semantik an die Anwendungsanforderungen
  - Daten: Änderungen nicht sofort sichtbar machen
  - Metadaten: Nicht alle speichern (z. B. Zeitstempel)
- Der Benutzer muss die Anforderungen mitteilen können
  - Der Benutzer weiß meist am besten, was er braucht
  - Entweder nur sehr eingeschränkte oder gar keine Unterstützung
- Üblicherweise nur eine einzige Semantik verfügbar
  - Für einige Anwendungsfälle geeignet, aber nie für alle

Relaxierte Semantik...

- Optimal wäre Kontrolle durch den Benutzer
  - Z. B. ein sicherer und ein performanter Modus
- Unterschiedliche Sperren-Mechanismen je nach Anwendungsfall
  - Z. B. gar keine Sperren im performanten Modus
- Datensicherheit eventuell unterschiedlich wichtig
  - Z. B. keine zusätzliche Datensicherheit im performanten Modus
- Unterschiedliche Konsistenzanforderungen
  - Z. B. intensives Caching im performanten Modus
- Performanter Modus gut für prozesslokale temporäre Dateien

Beispiel: Directory-Based Metadata Optimizations [2]

- Bezieht sich auf PVFS bzw. OrangeFS
  - Dateien bestehen 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: Metafile kann u. U. komplett eliminiert werden
  - Vorteil: Reduzierung der Metadatenoperationen
  - Nachteil: Keine Metadaten mehr, nur für kleine Dateien sinnvoll
  - Benutzer muss Funktionalität explizit aktivieren

Beispiel: Directory-Based Metadata Optimizations...

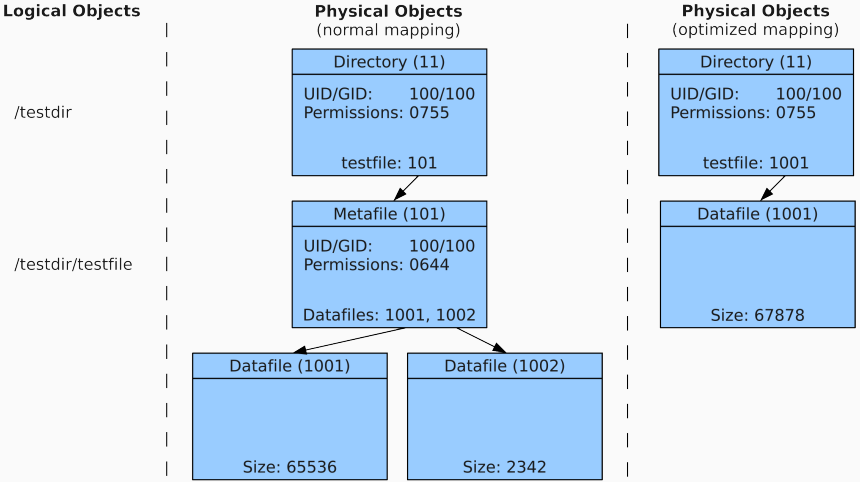


Figure 1. Directory tree.

Beispiel: Directory-Based Metadata Optimizations...

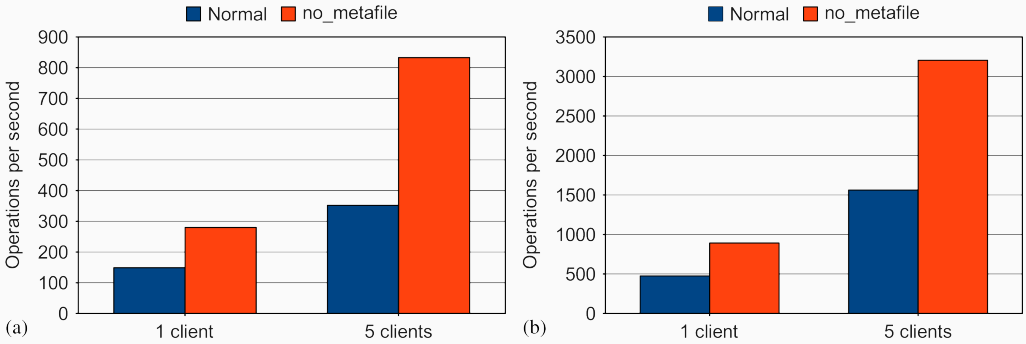


Figure 2. File creation: (a) on disk and (b) on `tmpfs`.

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

Beispiel: Directory-Based Metadata Optimizations...

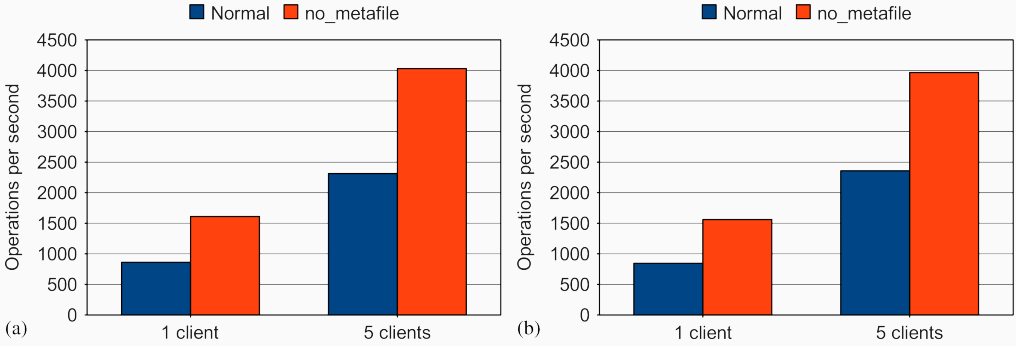


Figure 3. File listing: (a) on disk and (b) on `tmpfs`.

- Leistungsgewinn bei Leseoperationen nicht ganz so ausgeprägt

Beispiel: Directory-Based Metadata Optimizations...

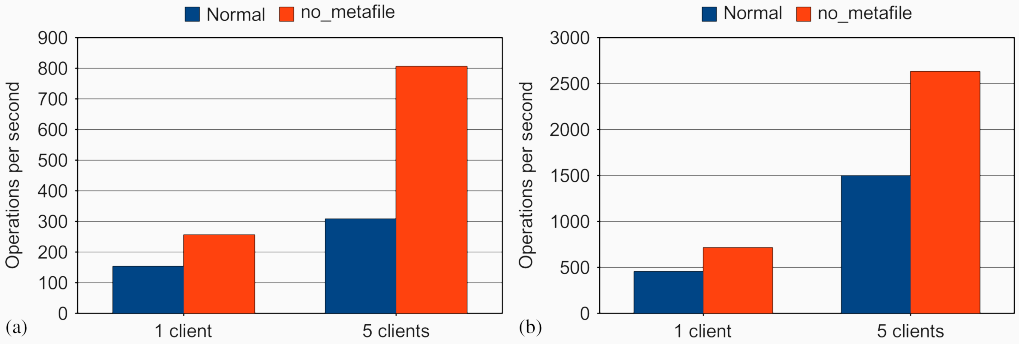


Figure 4. File removal: (a) on disk and (b) on `tmpfs`.

- Leistungsverhalten sehr ähnlich zum ersten Fall

- Möglichst sequentiell auf Daten zugreifen
  - Nicht hier und da ein Stückchen lesen
    - Häufig wird sowieso die ganze Datei gebraucht
  - Auch im Zeitalter nicht-rotierender Medien noch relevant
- Viele Suchvorgänge verhindern
  - Kopfbewegungen sind bei Festplatten sehr langsam
  - Kommunikation mit Servern erzeugt Overhead
- Viele kleine Anfragen verhindern
  - Wie beim Nachrichtenaustausch lieber einige große
  - Bei E/A zusätzlich zur Netzwerklatenz noch Suchzeiten

- Auswirkungen der E/A-Funktionen genau kennen
  - Z. B. welche Funktionen synchron oder kollektiv sind
- Zugriffsmuster sehr wichtiger Aspekt für Leistungssteigerungen
  - Dateisysteme und Bibliotheken können in einigen Fällen helfen
    - Ineffiziente Anwendungen können dadurch aber häufig nicht verbessert werden

Zusammenfassung

- Es gibt eine Vielzahl unterschiedlicher Optimierungen
  - Üblicherweise auf allen Ebenen des E/A-Stacks
    - Dadurch teilweise auch Seiteneffekte und Konflikte
  - Caching meist sehr wichtig als Grundlage
- Leistungsfähigkeit hängt entscheidend vom Benutzer ab
  - Muss so viele Informationen wie möglich bereitstellen
    - Zugriffsmuster, Zugriffsart, Topologie etc.
    - Das System kann oft nur dann effektiv optimieren
  - System muss allerdings Möglichkeiten dafür bereitstellen
- Wenn möglich sollte der Benutzer auch manuell optimieren
  - Optimierung der Zugriffsmuster, asynchrone E/A etc.

Quellen

[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 ...

[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] 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 ...

[6] 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.

[7] Wikipedia. **Native Command Queuing.** `http://de.wikipedia.org/wiki/Native_Command_Queuing`.