

Optimierungen

Hochleistungs-Ein-/Ausgabe

Michael Kuhn

Wissenschaftliches Rechnen
Fachbereich Informatik
Universität Hamburg

2018-06-01



Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

Caching...

- Leseanfragen unter Umständen direkt aus dem Cache
 - Insbesondere in Kombination mit Read Ahead
- 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
 - Ein Thread häufig nicht ausreichend für maximale Leistung
 - Thread pro Schreiboperation verursacht zu viel Overhead

Kooperatives Caching...

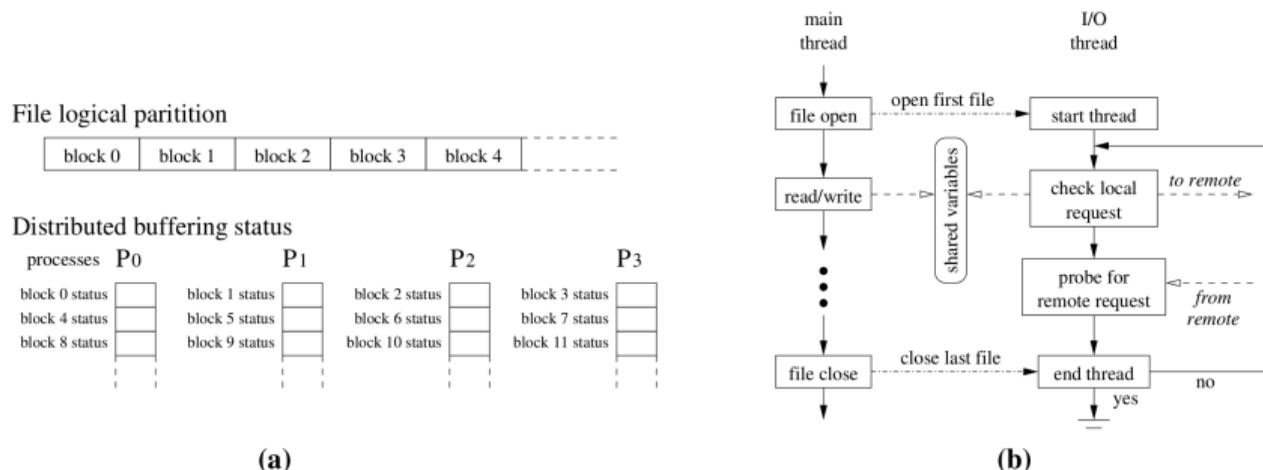


Fig. 1. (a) The buffering status is statically distributed among processes in a round-robin fashion. (b) Design of the I/O thread and its interactions with the main thread and remote requests.

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

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 eher bei Read-mostly-Dateien sinnvoll
 - Nachteile fallen bei Read-only-Dateien ganz weg
 - Üblicherweise eher im Big-Data-Umfeld, zunehmend auch im HPC

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 Bytes an Position x“
- Ist mehr Kontext vorhanden, kann besser optimiert werden
 - „Schreibe jeweils 100 Bytes an Position x, $x + 100$ und $x + 200$ “
 - 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

Scheduling

- Sortiere E/A-Operationen um
 - Entweder auf dem Client oder auf dem Server
 - Benötigt wieder Caching
 - Häufig als Vorstufe zu Aggregation
- Dadurch lässt sich eventuell mehr Leistung erzielen
 - Festplatten haben positionsabhängige Zugriffszeiten
 - Suchzeit (4–15 ms) und Rotationslatenz (2–7 ms)
 - Suchen ist allgemein teuer
- Linux unterstützt mehrere E/A-Scheduler
 - Unter anderem `cfq`, `deadline` und `noop`

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
- Der `deadline`-Scheduler für Linux funktioniert ähnlich

- `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 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
 - Stattdessen Zuständigkeit auf Basis von Teilbäumen
 - Metadaten-Server sind für einen oder mehrere Dateisystem-Teilbäume verantwortlich
- 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...

- 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

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
 - 1 Daten werden dekomprimiert im Speicher gehalten
 - 2 Daten werden komprimiert im Speicher gehalten und bei Zugriff 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
- Bei Swap-Nutzung 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

- Traditionell können zusammenhängende Bereiche gelesen und geschrieben werden
 - Nicht-zusammenhängend mit MPI-IO oder POSIX (über Umwege)
- 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

Prozess 1



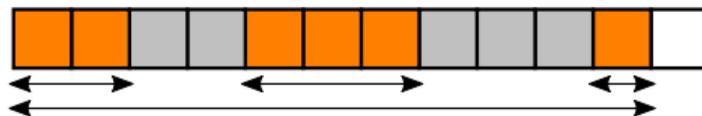
Nicht-zusammenhängende E/A...

- 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 vom Speichersystem
 - Schneller als viele kleine Zugriffe
 - Große Zugriffe auch auf nicht-rotierenden Medien schneller
- Verwirf nicht benötigte Daten
 - Lohnt sich nicht immer

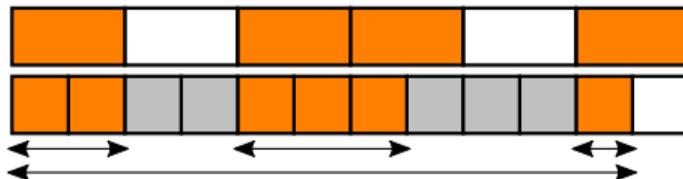
Prozess 1



Data Sieving...

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

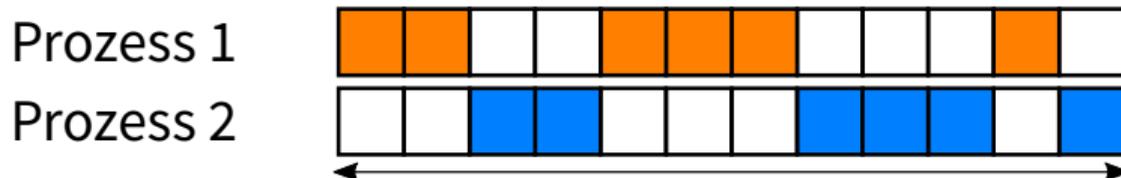
Server



Prozess 1

Kollektive E/A

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



- Im nicht-kollektiven Fall eventuell zuerst Prozess 2, dann Prozess 1
 - Sieht für das Dateisystem wie zufällige Zugriffe aus

Two Phase

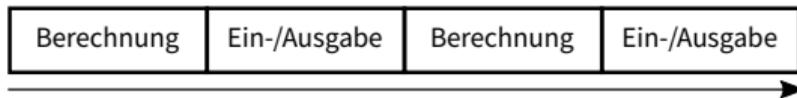
- Eine Optimierung für kollektive E/A [6]
 - Z. B. in ROMIO implementiert
- Die Clients koordinieren sich unabhängig vom Dateisystem
 - Clients sind für zusammenhängende Blöcke verantwortlich
 - Die Blöcke sind disjunkt und umfassen alle benötigten Daten
- Eventuell muss so jeder Client nur einen Server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

Asynchrone E/A

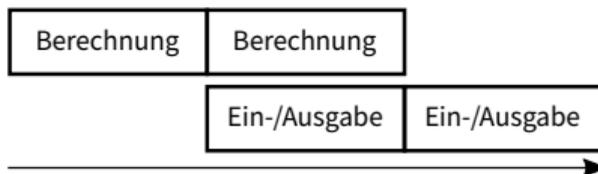
- Überlappung von E/A und Berechnung
 - Nur möglich mit genügend unabhängigen Berechnungen
- Idee: Nicht auf E/A warten sondern weiterrechnen
 - Spezielle asynchrone E/A-Funktionen
 - Z. B. `MPI_File_iread` und `aio_read`
 - Status kann mit speziellen Funktionen überprüft werden
 - Z. B. `MPI_test` und `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

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 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
- 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

Hints

- Idee: So viele zusätzliche Informationen wie möglich zur Verfügung stellen
 - Zugriffe können dann (hoffentlich) optimiert werden
- Hints sind üblicherweise optional
 - Einerseits muss der Benutzer keine angeben
 - Andererseits kann das System sie 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)
- 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 Sperren-Mechanismen
 - Gar keine Sperren im performanten Modus
- Bewerte Datensicherheit unterschiedlich
 - Keine zusätzliche Datensicherheit im performanten Modus
- Unterschiedliche Konsistenzanforderungen
 - Caching im performanten Modus
- Performanter Modus gut für temporäre Dateien

Best Practices

- 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 vielen unterschiedlichen Servern
- Viele kleine Anfragen verhindern
 - Wie beim Nachrichtenaustausch lieber einige große
 - Bei E/A zusätzlich zur Netzwerklatenz noch Suchzeiten

- 1 Optimierungen
 - Orientierung
 - Einleitung
 - Systemgesteuerte Optimierungen
 - Benutzergesteuerte Optimierungen
 - Hybride Ansätze
 - Zusammenfassung

2 Quellen

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.

Quellen III

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