Seminararbeit:

„Cooperative Caching“

Einsatz von Remote-Speicher zur Steigerung des Dateisystemperformance

Autor: Martin Pöpping

Inhaltsverzeichnis:

0. Abstract

Neue Highspeed-Netzwerke erreichen immer höhere Datendurchsatzraten. Das Performance der Prozessoren entwickelt sich deutlich schneller als die Leistungen von Speicherchips und Festplatten. Aus diesen Tatsachen wurde die Idee des Cooperative Caching entwickelt. Cooperative Caching koordiniert die Speicher der einzelnen Clients eines Netzwerkes und stellt dadurch einen gemeinsamen globalen Speichercache zur Verfügung.

Diese Seminararbeit basiert hauptsächlich auf einem Paper über eine Ablaufgesteuerte Simulation von Cooperative Caching Algorithmen der University of California at Berkeley [Dahlin94].
Dieses Paper, das begleitend zum „First Symposium on Operating Systems Design and Implementation“[OSDI94] erschien, beschreibt, dass Cooperative Caching, abhängig vom gewählten Algorithmus, die Anzahl der Speicherzugriffe halbiert und somit die Antwortzeiten von Lesezugriffen auf die Dateisysteme um 73% verbessert.
Dahlin et al. kommen zu dem Fazit, dass bereits der einfachste Cooperative Caching Algorithmus ausreicht um Leistungssteigerungen erkennen zu lassen [Dahlin94].

1. Motivation und Ziele

Zwei Entwicklungen waren für Cooperative Caching ausschlaggebend. Die Leistungen der Prozessoren stiegen deutlich im Vergleich zu den Zugriffszeiten der Festplatten. Außerdem erreichen neue High-Speed-Netze immer kürzere Latenzzeiten. So konnte man früher -vor 1994- Remote-Daten aus dem Speicher dreimal schneller abrufen als Remote-Daten von der Festplatte. Die neueren Highspeed-Netzwerke erreichten im Jahr 1994 Zugriffszeiten, die 10-20x schneller waren.

Normale Dateisysteme verwenden eine 3-Schichten-Speicherarchitektur [Ersfeld2002], welche eine reduzierte Form des Cooperative Caching praktizieren, indem ein Zwischencache eingesetzt wird, um die anderen zwei Speicher-Schichten, Hauptspeicher und Festplatte, zu versorgen.
Obwohl man, je nach Anwendung, mit der Aufrüstung des Hauptspeichers gute Leistungssteigerungen erreichen kann, gibt es 4 Argumente die für Cooperative Caching sprechen. Cooperative Caching bietet eine bessere Performance. Abhängig von der jeweiligen Implementierung wird die globale Hit-Rate verbessert und somit werden weniger Plattenzugriffe benötigt. Eine Verbesserung ist schon bei den einfachsten Cooperative Caching Algorithmen festzustellen. Durch Cooperative Caching werden außerdem clientseitig Speicherreserven eingespart. Dies erhöht ebenfalls die Hit-Rate des Lesens aus dem Speicher, da sich das Lesen auf einen kleineren Speicherbereich beschränkt. Der Server wird außerdem bei Cooperative Caching weniger belastet, da oft nur kleinere Request-Pakete weitergeleitet werden anstelle eines großen Datenflusses.
Weil nicht nur lokaler, sondern auch remote-Speicher zur Verfügung steht, lässt sich ein flexibleres Speichermanagement realisieren, je nachdem wie es die Systemanforderungen verlangen. Das letzte Argument für Cooperative Caching ist der deutliche Preisvorteil im Vergleich zu einem System mit besonders großem Hauptspeicher.

Das erste Ziel der Ausarbeitung ist die Analyse, ob Cooperative Caching unter realen Auslastungen signifikante Leistungsschübe bietet. Die Ablaufgesteuerte Simulation von Dahlin et al. unterscheidet sich von vorherigen Bemühungen von synthetischen Auslastungssimulationen um Cooperative Caching Algorithmen zu evaluieren. Das zweite Ziel ist die Evaluation einer Reihe von Algorithmen um später andere praktische Algorithmen zu finden, die Cooperative Caching effizient implementieren.

Das Hauptergebnis der Versuche war, dass Cooperative Caching die Performance für Lesezugriffe für das Testsystem mit den erzeugten Auslastungen um 73% verbessert. Als besonders geeigneter Algorithmus wird „N-Chance Forwarding“ genannt, welcher frei übersetzt als „praktischer Algorithmus der nahezu das ganze Potential der Performanceziele für die getesteten Auslastungen erreicht“ beschrieben wird.

Cooperative Caching wurde entworfen um das Cacheperformance für die Lesezugriffe auf die Dateisysteme zu verbessern. Diese Technik unterstützt nicht die Performanz für Schreibzugriffe oder für das Lesen großer Dateien. Für diese Studien wird auf das „xFS Projekt“ verwiesen [XFS93].

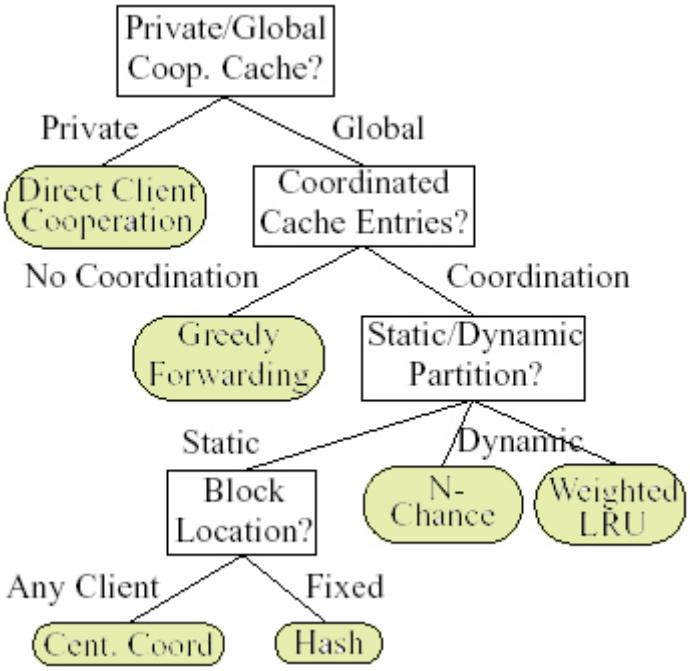
Im Folgenden werden nun die vier wichtigen Cooperative Caching Algorithmen beschrieben. Danach folgt die Beschreibung der Testsimulation und anschließen die Präsentation der Testergebnisse. Mit einer kurzen Zusammenfassung schließt diese Ausarbeitung ab.

2. Allgemeine Funktionsweise von Cooperative Caching

Cooperative Caching führt eine vierte Speicherschicht in das Speicherhierarchie des NFS ein. Die Daten können nicht nur im lokalen Hauptspeicher, Servercache oder auf der lokalen Platte gefunden werden, sondern auch auf einem remote-Speicher. Abhängig von dem verwendeten Cooperative Caching Algorithmus wird diese neue Speicherschicht entweder zwischen dem lokalen Clientspeicher und dem Serverspeicher, oder zwischen Serverspeicher und dem Plattenspeicher implementiert.
Auf die genauere Funktionsweise wird später eingegangen. Zu beachten ist jedoch, dass sich die im Folgenden beschriebenen Untersuchungen auf das Caching der Daten im Hauptspeicher des Clients konzentrieren und nicht auf das Cachen auf der lokalen Festplatte. Für zukünftige High-End-Netzwerke wird es viel schneller sein Daten aus einem remote-Speicher anstatt von einer lokalen Platte zu lesen.
Diese Seminararbeit lässt, wie die Vorlage von Dahlin et al. [Dahlin94] den Aspekt der Sicherheit unberücksichtigt. Laut Dahlin wird die Annahme gemacht, dass alle Rechner innerhalb eines LANs gleich sicher sind, da sie alle gleich administriert werden und daher das Vertrauen der Autoren darauf nicht größer ist als in aktuell populären Dateisystemen wie z.B. NFS. Es besteht jedoch die Möglichkeit, dass ein Client, dessen Betriebssystem gefährdet ist, unautorisierte File-System-Requests stellt. Das erhöhte Aufkommen von Prozessmigrationen in Netzwerken verlangt zwangsmäßig, dass auch Benutzerdaten auf anderen Clients gecacht werden, auch wenn kein Cooperative Caching angewendet wird.

3. Cooperative Caching Algorithmen

Cooperative Caching erzeugt eine neue Speicherebene, den Remote-Client-Speicher. Die verschiedenen CC-Algorithmen verwenden diese Speicherhierarchie auf unterschiedliche Art. Abbildung 1 zeigt die Einstufungen der im Folgenden beschriebenen CC-Algorithmen sortiert nach 4 elementaren Designkriterien:



[Abb. 1] Charakterisierung versch. CC-Algorithmen

Diese Algorithmen sind nur eine kleine und sollen nur Beispiele für die jeweiligen CC-Designs sein. Zu beachten ist, dass diese Algorithmen keine Rücksicht auf Datenspeicherzuverlässigkeit nehmen, da sie nicht die write-through-, write-delay, oder write-back-policies der Dateisysteme ändern. Clients schicken weiterhin die Daten an den Server, der sie dann speichert als wäre es ein herkömmliches System ohne Cooperative Caching.

a) Direct Client Cooperation

Direct Client Cooperation ist ein einfacher CC-Algorithmus, der es einem aktiven Client erlaubt Daten im Speicher eines inaktiven Clients zu speichern. Der aktive Client leitet die Daten, welche nicht mehr in seinen lokalen Speicher passen, direkt an den inaktiven Client weiter.
Danach kann der Client jederzeit auf seinen eigenen Remote-Cache in dem entfernten inaktiven Rechner zugreifen, bis der inaktive Client aktiv wird und den Cooperative Cache auflöst. Das System muss Kriterien festlegen, wonach Clients als aktiv oder inaktive eingestuft werden. Direct Client Cooperation ist interessant wegen seines einfachen Aufbaus. Er kann implementiert werden ohne dass der Server selbst modifiziert werden muss.
Ein Nachteil dieses Algorithmus ist, das aktive Clients nicht auf freien Speicher anderer aktiver Clients zugreifen können. Dies hat zur Folge, dass der vorhandene Speicher nicht optimal ausgenutzt wird und somit der Performancegewinn sehr eingeschränkt ist.

b) Greedy Forwarding

Greedy-Forwarding behandelt alle Caches eines einzelnen Clients als eine globale Cache-Ressource, auf die jeder Client zugreifen kann. Der Algorithmus nimmt jedoch keine Koordination des Caches vor. Wie herkömmliche Dateisysteme verwaltet jeder Client seinen lokalen Cache selbst nach einem Greedy-Algorithmus ohne Rücksicht auf fremde Daten oder das potentielle Nutzen der Daten für den jeweiligen Remote-Client. Zur Erinnerung: Ein Greedy-Algorithmus ist

ein Algorithmus, der zu jedem Zeitpunkt die „lokal“ beste Entscheidung trifft, ohne diese rückgängig zu machen [Blum92].
Findet ein Client einen bestimmten Datenblock nicht in seinem lokalen Cache, stellt er eine Anfrage an den Server. Hat der Server die erforderlichen Daten in seinem Speichercache übermittelt er diese.
Findet er die Daten nicht, schaut er in einer Liste nach, der Forwarding-Liste. Diese listet die Inhalte aller Client-Caches auf. Findet der Server das gesuchte Datenpaket in seiner Liste, forwardet er den Request an den entsprechenden Client. Der Client sendet die angeforderten Daten aus seinem Speicher direkt an den Client, der die Daten angefordert hat, ohne wieder den Umweg über den Server zu machen.
Sind die Daten in keinem Cache zu finden, werden sie, wie herkömmlich, von der Serverplatte gelesen.

Die enstehende forwarding-Tabelle ist nicht viel größer als die traditionelle per-file-callback-Liste. Dahlin nennt ein Beispiel eines CC-Systems, dessen Server einen 2 GB verteilten Cache (64 Clients x 32 MB Cache) mit der 6 MB großen Index-Liste einer Hashtable verwaltet.
Greedy-Forwarding ist interessant, weil es ein sehr fairer Algorithmus ist. Clients können ihre lokalen Ressourcen zu ihrem eigenen Vorteil managen und bringen trotzdem noch einen Nutzen für andere Remote-Clients.
Andererseits führt diese fehlende Koordination auch zu duplizierten Daten und unnötigen Plattenzugriffen, da nicht für alle Clients im Netz die optimalen Möglichkeiten ausgenutzt werden und jeder Client sozusagen „egoistisch“ denkt.
Die zwei folgenden Algorithmen sollen dieses Problem beheben.

c) *Centrally Coordinated Caching*

Centrally Coordinated Caching erweitert den Greedy-Forwarding-Algorithmus um eine Koordination, in dem jeder Client-Cache statisch in lokale Partitionen geteilt wird, welche von dem jeweiligen Client nach einem Greedy-Algorithmus verwaltet wird. Zusätzlich gibt es einen vom Server global gemanagten Bereich, welcher vom Server als Erweiterung seines zentralen Caches verwaltet wird. Findet ein Client einen Block nicht in seinem lokal verwalteten Cache, setzt er sich mit dem Server in Verbindung. Hat der Server die benötigten Daten in seinem Speicher so sendet er sie dem Client. Findet er den Block nicht, prüft er, ob er ihn in seinem zentral verwalteten Speicher hat. Findet er den Block dort, forwardet er ihn an den Client, der diesen Block gespeichert hat. Anderenfalls lädt er ihn von seiner lokalen Platte.
Centrally Coordinated Caching ist mit dem physischen Verschieben von Speicherblöcken vom Client zum Server zu vergleichen. Der Server verwaltet den global zu verwaltenden Teil jedes Client-Cache, indem er einen globalen Ersetzungsalgorithmus verwendet.
Wenn der Server einen Block aus seinem lokalen Cache entfernt um Platz für einen anderen Block zu machen, wird der entfernte Block in den Centrally Coordinated Distributed Cache geschrieben und ersetzt dort den in jüngster Zeit am seltensten verwendeten Block.
Wenn der Server eine Clientanfrage zu einem verteilten Cache-Eintrag weiterleitet, erneuert er den Eintrag auf seiner LRU-List für den globalen verteilten Cache.
Der Hauptvorteil von Centrally Coordinated Caching ist die hohe globale Hit-Rate, welche durch die globale Verwaltung der Speicherreserven erreicht wird.
Die Hauptnachteile dieses Algorithmus sind, dass die lokale Hit-Rate niedriger ist aufgrund des reduzierten lokalen Caches.
Außerdem erzeugt die zentrale Koordination des Caches deutlich mehr Serverlast.

d) N-Chance Forwarding

N-Chance Forwarding gleicht dynamisch den Anteil jedes Clients im Cooperative Cache, abhängig von den Client Aktivitäten, ab. Der N-Chance-Algorithmus modifiziert Greedy-Forwarding durch die Tatsache, dass Clients kooperieren und Singlets bevorzugen. Singlets sind Blöcke, welche nur in einem Client Cache gespeichert sind. Abgesehen von den Singlets arbeitet N-Chance-Forwarding wie Greedy-Forwarding.

N-Chance-Forwarding versucht zu vermeiden, dass Singlets aus dem Client-Speicher entfernt werden. Wenn ein Client einen Block aus seinem Speicher entfernt, prüft er, ob der Block die letzte gecachte Kopie im gesamten Cooperative Cache ist. Dies wird durch ein zusätzliches Flag für jeden Block resultiert.
Ist der Block, den ein Client entfernen möchte, ein Singlet, setzt er, bevor er ihn entfernt, dessen „recirculation count“ (Umlauf-Zähler) auf n, forwardet den Block an einen Random-Peer und schickt eine Message an den Server, dass die Daten verschoben wurden.
Der Peer, welcher die Daten erhält, setzt den Block auf seine LRU-Liste, als wäre er erst kürzlich referenziert worden. Erreicht ein recirculating Block das Ende der LRU-Liste, wird der Zähler dekrementiert und der Block wird erneut geforwardet. Erreicht der recirculating Block den Wert 0, so wird er endgültig entfernt.
Referenziert ein anderer Client den Singlet, setzt er den recirculation-Zähler neu und cacht den Block in seinem eigenen Cache. Der Client, der den Block bisher gecacht hat, entfernt den Block aus seinem Cache.
Der Parameter n bestimmt wie oft ein Singlet durch verschiedene Client LRU-Listen wandern darf ohne neu referenziert zu werden.
Greedy Forwarding ist im Grunde genommen eine einfache Version dieses Algorithmus für n=0.
Eine Erweiterung dieses Algorithmus könnte laut Dahlin sein, dass Singlets nur an Clients mit dem Status idle geschickt werden um beschäftigte Clients nicht zu stören.
Eine Implementierung des Algorithmus sollte einen Welleneffekt verhindern. Erhält ein Client einen neuen Block, so darf er keinen anderen Block forwarden um diesen Block frei zu machen, damit eine tiefere Rekursion und ein möglicher Deadlock verhindert wird.
Erhält ein Client einen solchen Block, verwendet er einen Ersetzungsalgorithmus um den ältesten Block der mehr als einmal vorkommt zu ersetzen. Enthält der Cache nur Blöcke die alle ein einziges Mal vorkommen, überschreibt er den ältesten recirculating Block.
Verschiedene Optimierungen für diesen Algorithmus reduzieren das Kommunikationsaufkommen mit dem Server. Eine Optimierung ist, dass der Client bei einem Cache Miss die Nachricht an den Server zur Anforderung des Blockes kombiniert mit dem Update des Server Directories. Dieses Update gibt an welchen Block der Client verworfen hat und wohin, wenn überhaupt, der Block geforwardet wurde.
Das zweite Set an Optimierungen reduziert die Anzahl der Anfragen an den Server die fragen, ob ein Block die letzte gecachte Kopie besitzt, wenn ein Client entscheiden muss, ob ein Block weitergeschickt oder verworfen wird.
Jeder Block, dessen recirculation Zähler gesetzt ist, muss ein Singlet sein, so dass keine Servermessage notwendig ist um zu bestimmten was mit diesem Block passiert.
Für non-recirculating-Blocks muss der Client normalerweise eine Message an den Server senden. Sobald der Block allerdings als Singlet deklariert wird, forwarded oder verwirft der Client den Block, ohne mit dem Server zu kommunizieren. Daher ist nur eine Message an den Server in der Lebenszeit eines Blockes im Cache notwendig.
Der Hauptvorteil von N-Chance-Forwarding ist, dass er einen einfachen dynamischen Kompromiss zwischen privatem Client Cache und bereitgestellten globalen Cache bietet. Das Bevorzugen von Singlets bietet ein besseres Performance als der einfache Greedy-Algorithmus, weil das Verwerfen eines Singlets kostenintensiver als das Verwerfen eines mehrfach auftretenden Blocks ist.
Ein Nachteil dieses Algorithmus ist, dass ein Block möglicherweise zwischen verschiedenen Caches hin- und hergeschickt wird und somit eine unnötige Systemlast erzeugt.

e) *andere Algorithmen*

Es gibt noch andere Algorithmen, auf die hier allerdings nicht eingegangen wird, da sie sich nur gering von den o.g. Algorithmen unterscheiden.
Hashing-Distributed Caching ist beispielsweise eine Abwandlung des Centrally Coordinated Caching. Im Gegensatz zum Centrally Coordinated Caching verwaltet Hashing-Distributed Caching den zentral zu verwaltenden Cache basierend auf „block identifiers“. Jeder Client verwaltet hier eine Partition des Caches.
Der zentrale Server sendet Blocks die aus seinem lokalen Cache entfernt werden an einen Client, der mittels eines Hashings auf den „block identifier“ (des zu verschiebenden Blocks) ausgesucht wird.
Bei einem „local miss“ d.h. wenn ein Client den gesuchten Block nicht in seinem lokalen Cache findet, schickt er direkt einen Request an den passenden Client anhand des „block identifiers“. Der betreffende Client übermittelt die Daten, wenn er sie besitzt, oder forwardet den Request an den Server.
Simulationen von Dahlin haben ergeben, dass Hashing-Distributed Caching im Vergleich zum Centrally-Coordinated Caching nahezu identische Hit-Rates hat.
Vorteil von Hashing-Distributed Caching ist, dass die Serverlast deutlich verringert wird, da viele Requests direkt von den Clients abgearbeitet werden.
Weighted LRU ist der letzte zu erwähnende Algorithmus. Dieser dynamische Algorithmus versucht Objekte mit der niedrigsten Kosten-/ Nutzen-Rate zuerst zu ersetzen.
Wie bei den N-Chance-Algorithmus sind duplizierte Objekte in mehreren verteilten Caches nicht sehr nützlich. Andererseits sparen duplizierte Blöcke eventuell einen Remote-Zugriff über das Netzwerk auf einen remote-Client.
Weighted LRU balanciert daher das Verwahren von oft genutzten duplizierten Blöcken um Netzwerkzugriffe zu vermeiden gegen das Behalten von weniger gesuchten Singlets um Plattenzugriffe zu vermeiden.
Die Antwortzeiten dieses Algorithmus sind allerdings leicht schlechter als die des einfachen N-Chance Forwarding.

4. *Beschreibung der Simulationsmethode*

Dahlin et al. verwenden eine Ablaufgesteuerte Simulation zur Evaluation der vorgestellten Cooperative Caching Algorithmen.
Diese Simulation verfolgt den jeweiligen Status aller Caches im System und zeichnet die Requests und Hit-Rates jedes Clients auf. Es wird von einer 8kB-Blockgröße ausgegangen, wobei Teilblöcke mit weniger als 8kB nicht erlaubt sind.
Die Antwortzeiten werden errechnet, indem die Zugriffszeiten vom lokalen Speicher, dem Remote-Speicher, Serverspeicher und Platten-Hit-Rates miteinander multipliziert werden.
Die Annahmen basieren auf 155 Mbit/s ATM, wobei bei einer 8kB-Blockgröße Zugriffszeiten von 250 µs für lokalen Speicher, 400 µs für Remote-Speicher, 200 µs pro Netzwerk-Hop und 14,800 µs durchschnittlicher Festplattenzugriffszeit angenommen werden. Abbildung 2 summiert die Zugriffszeiten der Algorithmen für die verschiedenen Ressourcen. Da keiner der Algorithmen die Serverlast beachtlich erhöht und moderne Netzwerke über Switch-Topologien verfügen wird angenommen, dass keine Pausen durch Warteschlagen (für Serverzugriffe) entstehen.
Im System selbst besitzt jeder Client 16 MB und der Server 128 MB Speicher.

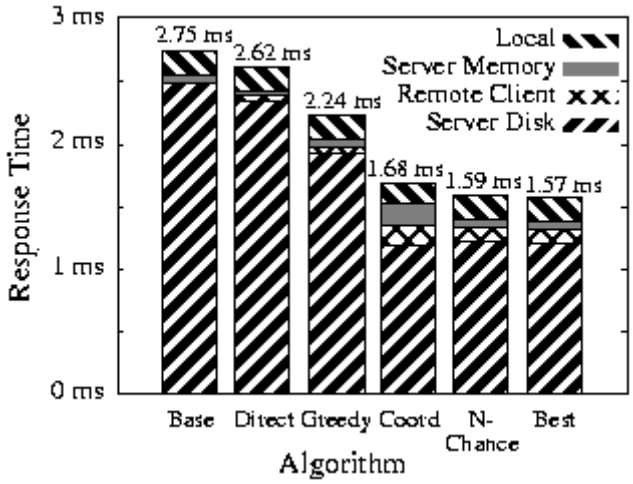
	Local Memory	Remote Client Memory	Server Memory	Server Disk
Direct	250 µs	1050 µs	1050 µs	15,850 µs
Greedy	250 µs	1250 µs	1050 µs	15,850 µs
Central	250 µs	1250 µs	1050 µs	15,850 µs
N-Chance	250 µs	1250 µs	1050 µs	15,850 µs

[Abb. 2] Zugriffszeiten für verschiedene Ebenen in der Speicherhierarchie für verschiedene Cooperative Caching Algorithmen. Die Unterschiede der Remote-Client-Zeiten für die verschiedenen Algorithmen sind abhängig von der Anzahl der Netzwerk-Hops.

Die Ergebnisse werden zum einen gegen ein Base-Case-Modell und zum anderen gegen ein unrealistisches Best-Case-Modell verglichen. Das Base-Case-Modell geht davon aus, dass jeder Client einen Cache besitzt und der zentrale Server ebenfalls einen Cachespeicher hat. Es wird von keinem Cooperative Cache Algorithmus gebrauch gemacht.
Der nicht realisierbare Best-Case nimmt an, dass der Cooperative Cache Algorithmus eine globale Hit-Rate erreicht als wenn alle Client-Caches als ein gemeinsamer globaler Cache gemanagt würden. Weiterhin wird angenommen, dass die lokalen Hit-Rates so hoch sind als würden alle Client-Speicher als private lokale Speicher verwaltet.
Dieser Best-Case bietet eine Untergrenze für die Antwortzeiten von Cooperative Caching Algorithmen die physikalsch Clientspeicher aufteilen und LRU-Ersetzungen verwenden.
Der Best-Case-Algorithmus wird simuliert, indem der lokale Cache jedes Clients verdoppelt wird, den Clients erlaubt wird die Hälfte ihres Caches lokal zu verwalten und dem Server erlaubt wird die Hälfte des Caches global zu verwalten(vgl. Centrally Coordinated Algorithmus).
Für den Best-Case wurde weiterhin angenommen, das in einem Speicher gefundene Daten über drei Netzwerk-Hops, Request, Forward und Reply, 1250 µs pro Remote-Speicher-Hit benötigen.

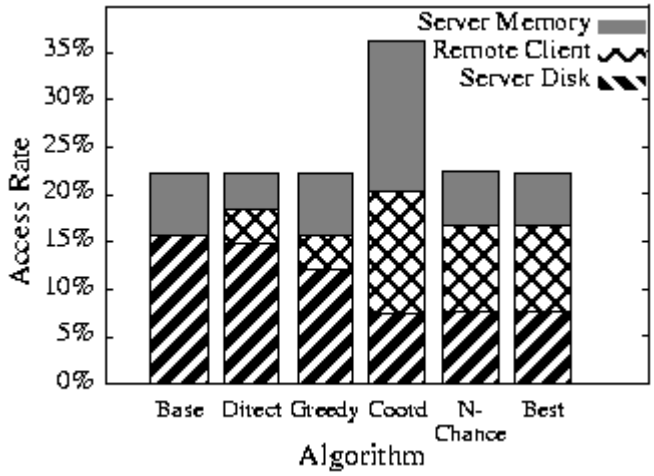
Für den Direct Client Algorithmus wurde bei den Simulationen die optimistische Annahme gemacht, dass die einzelnen Clients nicht untereinander interferieren.
Dies wurde dadurch simuliert, dass jeder Client einen permanenten Remote-Cache von derselben Größe wie sein lokaler Cache beansprucht. Effektiv wurden die Client-Caches dadurch verdoppelt. Für den Central Coordination Algorithmus wurde angenommen, dass 80% des Client-Caches als Cooperative Cache und 20% als lokaler Speicher verwaltet werden.
Für N-Chance wurde ein recirculation count von 2 festgelegt.

Vergleich der Algorithmen



[Abb. 3] Durchschnittliche Blocklesezeit: Jeder Balken repräsentiert einen Algorithmus. Der Balken ist weiterhin aufgeteilt in Local Memory, Server Memory, Remote Client Memory und Server Disk. Der gesamte Balken repräsentiert die durchschnittliche Gesamtlesezeit eines Blocks

Abbildung 3 zeigt die Antwortzeiten der einzelnen Algorithmen, sowie den Base Case (links) – ohne Cooperative Caching- und den als optimal anzunehmenden Best Case (rechts). Hier zeigt sich, dass Direct Client Cooperation den kleinesten Speedup von 1,05 hat. Greedy-Forwarding ist etwas leistungsfähiger mit einem Speedup von 1,22. Die anderen zwei Algorithmen, die eine Koordination der Caches implementieren, zeigen deutlich bessere Ergebnisse. Centrally Coordinated Caching hat einen Speedup von 1,64, N-Chance Forwarding einen Speedup von 1,73. So ist die gemessene Antwortzeit des N-Chance-Algorithmus mit 1,59 ms nur 0,02 ms langsamer als der optimistische Best Case.
Zwei Folgerungen lassen sich aus Abbildung 3 schließen. Zum einen zeigt sich, dass Plattenzugriffe maßgeblich an der Latenz des Base Case beteiligt sind und das Cooperative Caching die gesamte Hit Rate deutlich reduzieren kann.
Zum anderen lässt sich erkennen, dass die Algorithmen mit Koordination ein deutlich besseres Performance bieten, da diese Algorithmen versuchen die Mehrfacheinträge zu reduzieren um die Gesamt-Hit-Rate zu verbessern.

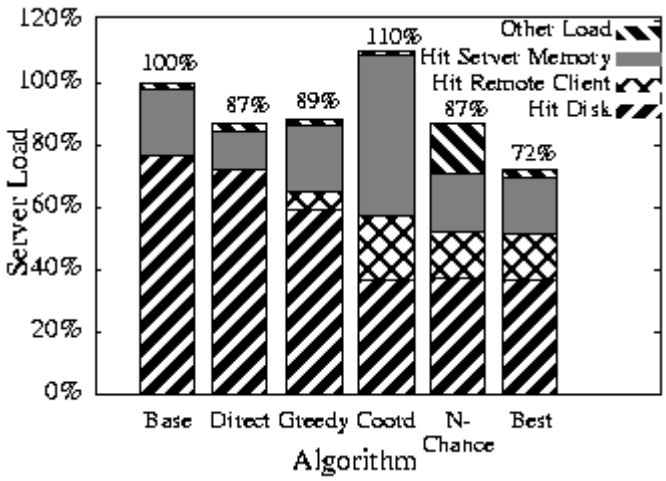


[Abb. 4] Anteil eines Requests zu jeder Stufe der Speicherhierarchie. Die Gesamthöhe jedes Balken zeigt die lokale Miss-Rate für jeden Algorithmus.

Abbildung 4 zeigt eine weitere Performance-Analyse die Zugriffsrate der einzelnen Speicherhierarchien. Daraus ergibt sich, dass die Gesamthöhe jedes Balken die Miss Rate der

Algorithmen im lokalen Cache zeigt, da auf die anderen Speicherbereiche erst zugegriffen wird, wenn die Daten im lokalen Cache nicht gefunden wurden. Base Case, Direct Client Cooperation, Greedy-Forwarding und der Best Case verwalten ihre lokalen Caches nach einem Greedy-Algorithmus und haben daher alle eine identische lokale Miss-Rate von 22%. Central Coordination hat mit 36% eine deutlich höhere Miss-Rate. Diese Ineffizienz kommt zustande durch die aggressive Koordination. Die Zugriffsrate auf die langsame Serverplatte ist jedoch mit 7,6% identisch mit dem Best Case und halb so hoch wie der langsame Base Case von 15,2%.
N-Chance zeigt überraschend, obwohl kein Greedy-Algorithmus angewandt wurde, eine lokale Miss-Rate von 22-23% mit zum Best Case identischen Zugriffszeiten und einer sehr niedrigen Festplatten-Zugriffsrate von 7,7%.
Ein Vergleich zwischen Algorithmen mit statischer Speicheraufteilung, Centralized Coordination, und dynamischer Speicheraufteilung, N-Chance, zeigt, dass beide Raten, die lokale und die globale Miss-Rate, verglichen werden müssen um diese Algorithmen zu evaluieren. Obwohl der statische Algorithmus geringere Plattenzugriffe aufweist, kostet diese geringe Zugriffsrate deutlich viel Performance im lokalen Cache.
N-Chance verbessert die Hit-Rate im lokalen Cache, hat jedoch eine schlechtere Werte Hit-Rate im globalen Cache.

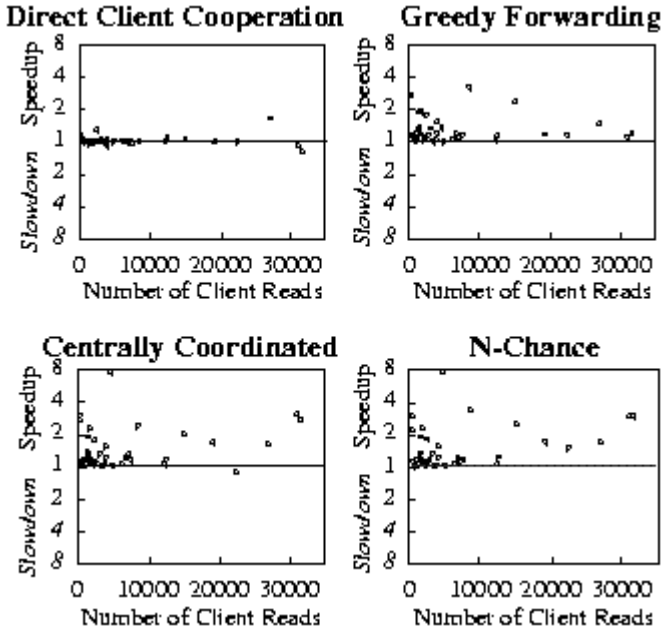
Weitere interessante Eigenschaften weist der Vergleich der Serverbelastung auf. Wenn einzelne Cooperative Caching Algorithmen die Serverlast erhöhen, so verringern Warteschlangen des Servers deutlich die Performance.



[Abb. 5] Serverlast der Algorithmen im Vergleich zum Base Case. Hit Disk beinhaltet die Netzwerk- und die Plattenlast. Hit-Remote-Client zeigt die Serverlast die entsteht um Requests und Forwards an die Remote-Clients weiterzuleiten. Hit-Server-Memory beinhaltet die Kosten für den Erhalt von Requests und das Übermitteln der Daten aus dem Serverspeicher. Other Load stellt hier den zusätzlichen adminstrativen Aufwand für die Validierung und Verwaltung von Client Caches und die Antworten auf Client-Anfragen dar.

Ein Hauptziel für Cooperative Caching ist, dass der Server nicht zu sehr beansprucht wird. Dahlin et al. machten für die Untersuchungen einige Vereinfachungen. Die Last für write-backs, deletes, file-attribute-requests wurden beispielsweise nicht berücksichtigt, da diese für alle Algorithmen gleich sind.
Die Ergebnisse der Serverlast-Untersuchung zeigen, dass kein Cooperative-Caching-Algorithmus die Serverlast deutlich erhöht. Die höhere Last des Centrally-Coordinated-Caches kommt dadurch zustande, dass eine sehr hohe lokale Miss-Rate hat, und alle Misses an den Server weitergeleitet werden.

Der letzte Vergleich untersucht die Performance von einzelnen Clients. Abbildung 6 zeigt die relative Performance der individuellen Clients für jeden einzelnen Algorithmus. Jeder Punkt repräsentiert einen Client, wobei besonders aktive Clients rechts und weniger aktive Clients mehr links zu finden sind. Die Speedups und Slowdowns für inaktive Clients sind hier nicht von besonderer Bedeutung.
Ein wichtiger Aspekt dieser individuellen Analyse ist die Fairness. Die Untersuchung zeigt, ob das Performance einzelner Clients unter der Performancesteigerung von anderen Clients leidet, wenn die Caches nicht nach einem Greedy-Algorithmus gemanagt werden.
Laut Abbildung 6 wird kein Clients bedeutsam benachteiligt, da Slowdowns nur in zwei Fällen im bei Direct Client Cooperation und in einem Fall beim Centrally Coordinated Cache zu erkennen sind.



[Abb. 6] Performance jedes einzelnen Clients: Jeder Punkt repräsentiert den Speedup oder den Slowdown (Inverse eines Speedups) eines Clients, gemessen aus der Sicht eines Clients und verglichen zum Base Case dieses Clients. Die x-Achse repräsentiert die Anzahl der Clients-Reads.

Obwohl Centrally Coordinated Caching und N-Chance Forwarding nicht hauptsächlich auf einem Greedy-Algorithmus basieren, profitieren so gut alle Clients von dem Cooperative Caching N-Chance Forwarding behindert keinen einzelnen Client, Centrally Coordinated Caching behindert nur einen einzelnen Client zu 19%.

Zusammenfassung

Die Untersuchungen haben ergeben, dass N-Chance Forwarding sich als ein relativ einfacher Algorithmus mit sehr guter Performance ist. Centrally Coordinated Caching und der nebenläufig erwähnte Hash Distributed Caching Algorithmus bieten ein sehr gutes Performance, dieses jedoch nicht sehr gleichmäßig verteilt auf alle Clients im Cooperative Cache. Des Weiteren sind sie sehr von dem Performance des Netzwerkes abhängig wegen der reduzierten lokalen Hit-Rates.
Weighted LRU, der nicht ausführlich getestete Algorithmus, verhält sich ähnlich wie N-Chance, ist jedoch komplizierter und belastet den Server mit Requests für Informationen über den globalen Status des Systems.
Greedy-Forwarding ist ein geeigneter Algorithmus für Systeme die einfach gehalten werden sollen. Direct Client Cooperation ist ebenfalls ein einfacher Algorithmus, bietet jedoch nicht das entsprechende Performance.

Literaturverzeichnis:

[Dahlin et al.94] Dahlin, Wang, Anderson und Patterson: „Cooperative Caching: Using Remote Client Memory to Improve File System Performance“;
http://www.cs.utexas.edu/users/dahlin/papers/osdiAbstract.html

[OSDI94] „First Symposium on Operating Systems Design and Implementation (OSDI ´94)“;
http://www.cs.utah.edu/~lepreau/osdi94/

[Ersfeld2002] Ersfeld: „Organisation von Caches“; http://www2.inf.fh-bonn-rhein-sieg.de/~rberre2m/lehre/ws0203/vups2/Seminar/01_Ersfeld_Caches.pdf

[XFS93] xFS: Serverless Network File Service; http://now.cs.berkeley.edu/Xfs/xfs.html

[BLUM02] Prof. Dr. N. Blum: “Greedy Algorithmen”: http://www.informatik.uni-bonn.de/kvv/ws0203/ver29.html