PACA

A Cooperative File System Cache for Parallel Machines

Seminararbeit

Verteilte und Parallele Systeme II

WS 2003 / 2003

Fachhochschule Bonn-Rhein-Sieg
Fachbereich Angewandte Informatik
Reto Kortas
reto.kortas@smail.inf.fh-brs.de

Inhaltsverzeichnis

Inhaltsverzeichnis 2
1 Einleitung 3
2 Hardware und Software Voraussetzungen 4
3 Der PACA Mechanismus 5
4 Design 6
4.1 Globaler Cache 6
4.2 Caching Algorithmen 7
4.2.1 Ermitteln des Speicherplatzes für neue Blöcke 7
4.2.2 Auffinden eines Blockes im Cache 7
4.2.3 LRU Algorithmus 8
4.2.4 N-Chance Forwarding Algorithmus 8
4.2.5 Vergleich zwischen N-Chance und LRU-Interleaved 10
4.3 Prefetching Algorithmen 11
4.3.1 One-Block-Ahead Algorithmus 11
4.3.2 Erweiterung zum Full-Fill-On-Open in PACA 11
5 Simulation 12
5.1.1 Parameter zur Simulation 12
6 Performance Analyse 13
6.1 Lese und Schreib Performance 13
6.2 Prefetching Einfluss 13
6.3 Bandbreite des Verbindungsnetzwerkes 14
6.4 Größe des Caches 14
6.5 Anzahl der Links pro Node zum Verbindungsnetzwerk 15
6.6 Performance mit nur einem zentralen File-Server 15
7 Geplante Erweiterungen 16
8 Résume 16
Quellenverzeichnis 17

1 Einleitung

Massiv-Parallele-Architekturen verfügen heutzutage über eine immense Rechenleistung, wodurch immer komplexer werdende Aufgaben oder Problemstellungen meist in konstanter oder kürzerer Zeit gelöst werden können.

Bedarf die Lösung des Problems jedoch dem Zugriff auf externe Daten, also solche die sich nicht im Cache oder Hauptspeicher des jeweiligen Systems befinden, so kann dieser Zugriff schnell zur Schwachstelle des Systems werden und dieses „ausbremsen“. Über Jahre hinweg wurde die Rechenleistung der Systeme stetig verbessert, wobei jedoch die I/O-Performance dieser meist außer Acht gelassen wurde. Erst in der jüngeren Vergangenheit machte man sich daran, die Zugriffes-Performance auf das zugrunde liegende File System zu verbessern.

Um dieses zu erreichen gibt es etliche Ansätze, wie z.B. die Entwicklung neuer Schnittstellen, Run-Time Libraries oder File Systeme. Da bekannt ist, dass die Leistung von High-Performance I/O-Systemen entscheidend vom Prefetching und Caching abhängt, wurde genau hier der Schwerpunkt bei der Entwicklung von PACA gesetzt. Es galt neue, besonders einfache, gut skalierende und hoch performante Caching- und Prefetching-Strategien zu entwickeln. Beide Techniken könne sich jedoch in ihrer Leistung gegenseitig stark beeinflussen, wodurch steht’s Kompromisse zwischen dem verwendeten Prefetching und optimalem Caching zu finden sind.

Ziel dieser Ausarbeitung soll es nicht sein, die verwendeten Algorithmen im Detail zu beschreiben, was auch mangels der erhältlichen Detail-Informationen nicht möglich ist, sondern viel mehr die zugrunde liegenden Strategien und Sachverhalte zu erläutern. Auch soll ein Gefühl dafür vermittelt werden, dass die Leistung eines Systems nicht nur von der reinen CPU Leistung abhängig ist, sondern das eine Reihe von weiteren Faktoren existieren, von denen die Performance des Systems maßgeblich mit beeinflusst wird. Des weiteren ist die Basis für diese Ausarbeitung, das Paper[1] von Toni Cortes et al. aus dem Jahre 1996 und es war nicht zu erfahren, in wie weit die Analysen und Erkenntnisse von damals weiter untersucht wurden, oder in wie weit sie in heutigen Systemen Anwendung fanden.

[1] PACA: a Cooperative File System Cache for Parallel Machines

2 Hardware und Software Voraussetzungen

PACA wurde speziell für Parallele Maschinen, welche aus mehreren Nodes bestehen, entwickelt. Die einzelnen Nodes verfügen dabei jeweils über eigenen lokalen Speicher und sind über ein breitbandiges Interconnection Network miteinander verbunden. Jeder einzelne Node kann dabei über einen oder mehrer Links zu dem Netzwerk verfügen.

Ein Node kann die einzelnen Links für die Kommunikation mit einem anderen Node auch parallel benutzen um so die Bandbreite für den erforderlichen Datentransfer zu erhöhen.

Die Parallele Maschine sollte über mehr als eine Festplatte pro Node verfügen, auf welche dann die Daten verteilt werden.

Auf dem parallelen System läuft ein, auf einem Microkernel basierendes Betriebssystem. Hierbei wird benötigte Funktionalität, welche nicht durch den Kernel des Betriebssystems gegeben ist, durch sogenannte „*user-level*“ Server implementiert. Speziell für Operationen auf dem File System wird ein eigener Server, der File Server, eingesetzt, welcher alle Operationen überwacht.

PACA wurde zu Beginn als ein File System Cache Prototyp für den PAROS Microkernel entwickelt, welcher die folgenden notwendigen Eigenschaften bereitstellt:

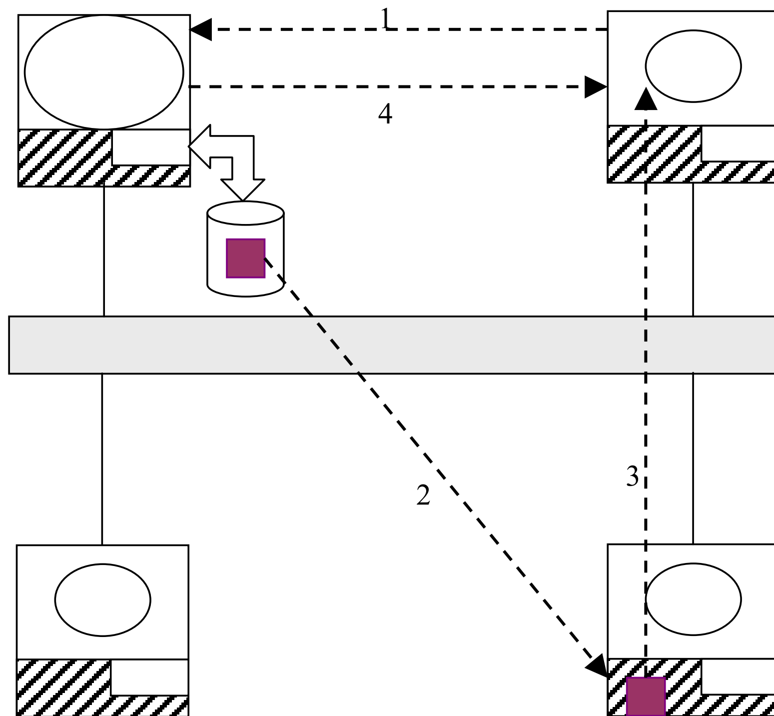
- Damit der File Server in der Lage ist, mehrere verschiedene Anfragen parallel zu bearbeiten, müssen „*multithreaded applications*“ unterstütz werden, wobei es möglich sein muss, das die einzelnen Threads auf verschiedenen Nodes laufen können.
- Die Kommunikation zwischen den einzelnen Applikationen müssen über Ports realisierbar sein, denn User Requests und Bestätigungsmeldungen werden ausschließlich über Ports gesendet. Zum Datentransfer wird allerdings eine schnellere Technik eingesetzt.
- Eine sogenannte „*memory_copy*“ Funktion wird gefordert. Diese wird zum Datentransfer zwischen dem Cache und dem User benötigt. Hierbei muss es möglich sein, das ein Prozessor einen Datenaustausch zwischen zwei anderen Prozessoren beauftragen kann, wobei der ausführende Prozessor dann mit dem ganzen Overhead der Steuerung und Verwaltung des Datentransfers belastet wird.

3 Der PACA Mechanismus

Anhand eines kleinen Beispiels soll hier zunächst das grobe Funktionsprinzip von PACA erläutert werden. Genauere Beschreibungen der zugrundeliegenden Mechanismen und Algorithmen sind im anschließenden Kapitel 4 zu finden.

Eine Applikation (Client) fordert einen benötigten Block des File Systems an, in dem sie über einen Port einen r*equest* (1) an den File Server schickt. Stellt der File Server fest, dass sich der angeforderte Block noch nicht im globalen Cache befindet, so wird er von der Festplatte geladen und mittels *memory_copy* Funktion an eine passende Stelle im globalen Cache kopiert (2), welche sich auf einem beliebigen Node im System befinden kann. Ist der Block nun gecached, wird er in den Puffer des Users kopiert (3). Der File Server informiert (4) den anfordernden Client über einen neuen Port über den Abschluss der Aktion. Alle erforderlichen Kopieraktionen von Speicher zu Speicher werden mittels des *memory_copy* Mechanismussees durchgeführt.

Eine genaue Beschreibung der Strategien zur Auffindung von Blöcken im Cache und zur Bestimmung des Speicherplatzes neu zu ladender Blöcke ist wie schon oben beschrieben im Kapitel 4 nachzulesen.



4 Design

Beim verwendeten File Server handelt es sich um eine *multi-threaded* Anwendung, welche auf einem Node der parallelen Maschine läuft, sich allerdings die CPU mit weiteren Anwendungen teilen muss, welche auf dem selben Node laufen. Jedoch besitzt der File Server eine höhere Priorität als alle anderen Prozesse des Nodes.

4.1 Globaler Cache

PACA wurde mit dem Ziel entwickelt, das die Größe des Caches mit der Anzahl der Nodes im System skalieren soll. Um dies zu erreichen, existieren mehrere Möglichkeiten.

Zum einem ist es möglich, den lokalen Speicher desjenigen Node zu erhöhen, welcher in der parallelen Maschine die Funktion des File Servers übernimmt. Dies führt aber dazu, das man schnell zu einem Node mit einem immensen Bedarf an lokalem und sehr teurem Speicher gelangt.

Ein weiterer Ansatz ist es, jeden Node seine eigenen Daten cachen zu lassen, ungeachtet dessen, welche Daten die anderen Nodes des Systems bereits in ihrem Cache halten. Dieses Verfahren skaliert zwar ebenfalls mit der Anzahl der Nodes, führt aber zu einer unausgewogenen und uneffizienten Ausnutzung des Speichers. So kann z.B. der frei Cache eines Nodes nicht von einem anderen, welcher einen höheren Bedarf an Cache hat genutzt werden.

Bei PACA handelt es sich nun um einen speziellen, kooperativen Cache. Die lokalen Caches eines jeden Nodes der Parallelen Maschine werden zu einem globalen parallelen Cache zusammengefasst. Diese Art der Cache Organisation führt durch eine hohe globale Trefferrate zu einer guten Cache Performance und verbessert somit die gesamt Leistung des Systems.

Mit diesem Caching Mechanismus lassen sich zwei Arten von Cache-Hits beobachten.

- „*local hit*“, der anfragte Block befindet sich im lokalen Cache des Nodes
- „*remote hit*“, der angefragte Block befindet sich in dem Cache eines anderen Notes

Erste Implementierungen von PACA arbeiten mit einem einzelnen zentralen Server (File Server) zur Kontrolle des globalen Caches, welcher in der Lage war, Anfragen von einer angemessene Anzahl von Prozessoren zu bewältigen. Nach Untersuchungen führt der zentrale Server bei einem parallelen System mit bis zu 50 Nodes zu keinem Performance Einbruch, da die zu erledigenden Aktionen nur sehr wenig Server Last verursachen und es sich um einen verteilten Datentransfer handelt, d.h. der Server muss nur den angeforderten Cache Block im globalen Cache ermitteln und den Datentransfer beauftragen. Da der Datentransfer direkt vom lokalen Cache eines Nodes in den Speicher des anfragenden Clients oder umgekehrt erfolgt und somit nicht über das Netzwerk Interface des Servers läuft, ist dessen Anbindung an das Interconnection Network ebenfalls nicht als Schwachstelle zu sehen.

Als Cache Strategie wird das „*delayed-write-back*“ Verfahren eingesetzt. Änderungen an Cache Zeilen werden nicht gleichzeitig auch auf die Festplatten geschrieben, sondern statt dessen wird bei der betroffenen Cache Zeile ein Bit gesetzt, das signalisiert, dass die Daten im Cache aktuelle sind, nicht aber der zugehörige Blocke auf der Festplatte. Die so markierten Blöcke werden auch als „*dirty-blocks*“ bezeichnet. Ein spezieller Thread, der sogenannte „*syncer*“, schreibt alle 30 Sekunden sämtliche „*dirty-blocks*“ des gesamten Caches auf die Festplatten.

4.2 Caching Algorithmen

Bei der Entwicklung von PACA wurde besonders auf gute Performance Wert gelegt, welche mittels einfacher und gut skalierender Algorithmen erreicht wird. Die Algorithmen wurden dahingehend entwickelt das es keiner Mechanismen zur Gewährleistung der Kohärenz zwischen Daten im Cache und Daten auf den Festplatten im System bedarf, ohne dabei die Performance zu verschlechtern.

Ein weiterer Performance Gewinn wird durch die Möglichkeit der parallelen Kommunikation erreicht, welche die zugrundeliegende Architektur bietet. Fordert ein Client mehrere Blöcke beim File Server an und befinden sich diese bereits im Cache, so können die Blöcke parallel zum Client geschickt werden, was den Overhead für den Datentransfer von einem entfernten Node reduziert. Hier hängt die Performance allerdings auch von der Anzahl der Links ab, über welche die beteiligten Nodes verfügen, sowie von der im Interconnection Netzwerk zur Verfügung stehenden Bandbreite.

In der aktuellen Version von PACA können bis zu 5 Blocks parallel zum Client gesendet werden.

Als Organisationsform für den Cache wurde der Typ des assoziative abbildenden Caches (*set-associative*) gewählt. Die Anzahl der Sets entspricht dabei der Anzahl der in der Parallelen Maschine vorhandenen Nodes. Die Größe eines einzelnen Sets ist dabei gleich der Größe des lokalen Caches eines Nodes, gemessen in Blocks. Bei den *set-assoziativen* Caches wird jedem Speicherblock eine Menge (Set) von Cacheblöcken, den sogenannten *cache lines* zugeordnet. Ein Speicherblock darf dabei in einen beliebige Cacheblock dieser Menge geschrieben werden.

4.2.1 Ermitteln des Speicherplatzes für neue Blöcke

Wenn der Server einen neuen Block in den Cache zu laden hat, muss er zunächst den Node ermitteln, in dessen Cache er den Block zu speichern hat. Dies erreicht er, in dem er auf den Namen der Datei und den aktuellen Block eine *Hash-Funktion* anwendet. Alternative zu dem Namen der Datei kann auch der *I-Node* der Datei verwendet werden. Anhand es errechneten Hashwertes erkennt der File Server, bei welchem Node der Maschine der Block zu speichern ist. Über die verwendete Hash-Funktion lassen sich auf Grund mangelnder Informationen keine weiteren Aussagen manchen.

Nach dem der Node ermittelt ist, muss nun noch eine *cache-line* im lokalen Cache des Nodes bestimmt werden, in welcher der Block gespeichert werden soll. Falls die ermittelte *cache-line* bereits durch einen Block belegt ist, muss ermittelt werden, welcher bereits im Cache vorhandene Block für den neuen Block den Cache verlassen muss. Hierzu wird ein *LRU-Algorithmus* verwendet, welcher unter 4.2.3 näher beschrieben wird.

4.2.2 Auffinden eines Blockes im Cache

Muss der File Server ermitteln, ob sich ein Block bereits im Cache befindet, so braucht er lediglich die Hash-Funktion auf den Namen des Files anzuwenden und den geforderten Block in der Liste der aus der *Hash-Funktion* resultierenden Liste von Blocks zu suchen. Um die Suche des Blockes in der Cache-Liste des Nodes zu beschleunigen, werden die sich im Cache des Nodes befindlichen Blöcke in einer *Hash-Tabelle* gespeichert.

Das Auffinden und Speichern eines Blockes im globalen Cache ist somit durch das einfache Berechnen zweier Hash-Funktionen möglich und bedarf nur minimaler Rechenzeit. Die Last, welche hierdurch auf dem File Server erzeugt wird ist nahezu zu vernachlässigen.

4.2.3 LRU Algorithmus

Ist die ermittelte *cache-line* bereits belegt, so muss das System eine *cache-line* ermitteln, die aus dem Speicher entfernt wird, um Platz für den neu einzulagernden Block zu schaffen.

Obwohl es möglich ist, zufällig einen Block aus dem Cache zu löschen, ist die Performance des Systems doch erheblich besser, wenn man nicht einen zufällig ermittelten Block aus dem Cache löscht, sondern denjenigen, der am wenigsten benutzt wird. Falls nämlich ein viel benutzter Block aus dem Cache gelöscht wurde, so ist es relativ wahrscheinlich, dass dieser schnell wieder eingelagert werden muss, was dann zu extra Kosten führt.

Eine gute Annäherung an den optimalen Algorithmus basiert auf der folgenden Beobachtung. Blöcke, welche durch die letzte Instruktion referenziert wurden, werden auch bei den folgenden Instruktionen erneut referenziert. Dagegen werden Blöcke, welche schon länger nicht mehr benutzt wurden, auch weiterhin für eine längere Zeit nicht mehr referenziert. Diese Überlegung führt zu der Idee, Blöcke aus dem Cache zu löschen, die am längsten unbenutzt sind. Ein Algorithmus, welcher diese Strategie benutzt, wird LRU (*least-recently-used)* genannte und zählt zu der Familie der Marking Algorithmen. Obwohl LRU theoretisch realisierbar ist, ist dessen Implementierung nicht einfach.

Um LRU komplett zu implementieren, wäre es erforderlich, eine verkettete Liste aller sich im Cache befindlichen Blöcke zu verwalten. Der am letzten benutzte Block müsste sich am Anfang der Liste befinden und der am wenigsten benutzte am Ende. Die Schwierigkeit dieser Umsetzung liegt jedoch in der Tatsache, dass diese Liste bei jedem Zugriff auf den Cache aktualisiert werden müsste. Die hierfür erforderlichen Operationen auf der Datenstruktur sind jedoch sehr zeitintensiv. Hierfür kann entweder spezielle Hardware benutzt werden, oder man versucht das Verhalten in Software zu implementieren und so eine möglichst optimale Annäherung zu erreichen.

Auf die einzelnen Verfahren, welche mittels Software realisiert wurden, möchte ich hier jedoch nicht eingehen. Es ließen sich keine genauen Informationen darüber ermitteln, wie der Algorithmus in PACA softwaremäßig implementiert wurde. Es ist nur bekannt, das die Blöcke, welche lokal im Cache eines Nodes gespeichert sind, in einer Hash-Tabelle hinterlegt werden, wodurch sich auch die gute Performance des verwendeten Algorithmus erklären lässt, denn eine Suche in einer Hash-Tabelle hat im Durchschnitt die Komplexität O(1), d.h. deren Laufzeit ist von der Größe der Tabelle und dessen Belegungsfaktor unabhängig.

Mit einer „guten“ Hash-Funktion würde so immer der älteste Block im Cache ersetzt werden, was näherungsweise an die Genauigkeit eines globalen LRU Algorithmus heran reicht und die gestellten Anforderungen erfüllt.

4.2.4 N-Chance Forwarding Algorithmus

Beim N-Chance Forwarding liegt die Idee zu Grunde, das der Cache eines Client in zwei separate Teile getrennt wird. Der erste Teil wird vom lokalen Client verwaltet und enthält Blöcke die von ihm referenziert wurden. Der zweite Teil wird zum Aufbau eines global gemanagten Caches verwendet. Hier werden die Blöcke gespeichert, die nicht eigens von einem Node benutzt werden, aber durchaus die Leistungsfähigkeit des Systems verbessern können, wenn sie später erneut referenziert werden und dann schon im Cache vorhanden sind. Das heißt aber nicht, dass ein Node den lokalen Cache eines anderen nicht verwenden kann, sondern er darf nur nicht dessen Inhalt manipulieren.

Die Aufteilung des Caches ist nicht fest, sondern wird dynamisch in Abhängigkeit der I/O-Aktivität des Client eingestellt. N-Chance berücksichtigt, dass sogenannte *singlets* wertvoller sind als Duplikate, so dass vorzugsweise *singlets* gecached werden.

Ein *singlet* ist definiert als die einzige, sich im Cache befindliche Kopie eines File Blocks.

Muss ein Node nun einen Block aus dem Cache verwerfen um einen neuen einzulagern, so wird immer der am wenigsten benutzte gewählt. Handelt es sich hierbei jedoch um ein *singlet*, so leitet er den Block an den Cache eines anderen, zufällig bestimmten Nodes weiter (*forwarding*), mit der Folge das dieser nicht aus dem kooperativen Cache verworfen werden muss. Der Node, welcher den Block empfängt, fügt ihn zu seiner LRU-Liste hinzu, so als sei er kürzlich von ihm referenziert worden.

Um den durch alte *singlets* verwendeten Speicher zu begrenzen, hat jeder Block einen *recirculation_count*, welcher beim erstmaligen Weiterleiten auf den Wert *N* gesetzt wird und danach von jedem Client, der den Block weiterleitet um eins erniedrigt wird. Blöcke, die das Ende der LRU-Liste erreicht haben und deren *recirculation_count* null ist werden nicht mehr weitergeleitet, sondern aus dem Cache verworfen.

Referenziert ein Node ein lokales *singlet*, so wird dessen *recirculation_count* zurückgesetzt. Wird ein *recirculating singlet* in einem remote Cache referenziert, so wird dessen *recirculation_count* vom remote Node zurück gesetzt und es wird zu dem Node weitergeleitet, welcher den Block referenziert hat. Danach wird der Block aus dem Cache des remote Node verworfen.

Der Parameter N gibt also an wie, oft ein unreferenziertes *singlet* weitergeleitet werden darf, bevor es endgültig verworfen wird.

Durch den *recirculation_count* ist der Algorithmus in der Lage einen dynamischen Kompromiss zwischen der Verteilung des Client Caches auf den lokalen und globalen Bereich zu gewährleisten. Besonders aktive Clients tendieren dazu, die ihnen geschickten globalen Cache-Daten wieder besonders schnell aus ihrem Cache zu verdrängen, da sie sonst durch lokale Referenzierungen eines anderen Blocks aus ihrem Cache verdrängt würden. Nicht so aktive Clients tendieren dagegen dazu, die globalen Blöcke zu sammeln und sie möglichst lange im Speicher zu halten.

Bei der Implementierung dieses Algorithmus muss jedoch darauf geachtet werden, das ein weitergeleiteter Block eines Clients bei dem Empfänger einen Block verwirft und so weiter. In den meisten Fällen handelt es sich jedoch bei den verworfenen Blöcken nicht um *singlets*. Um jedoch die so mögliche Rekursion zu vermeiden, ist es einem Client verboten, Platz für einen *recirculating* Block durch das Weiterleiten eines anderen zu gewinnen. Tritt dieser Fall ein, so verwendet der Client einen modifizierten Ersetzungsalgorithmus, der die älteste Kopie des Blocks verwirft. Enthält der cache jedoch keine Duplikate, so wählt der Client das älteste *recirculating* singlet mit dem kleinsten *recirculation_count.*

Da es sich bei PACA aber um ein System für parallele Maschinen mit einem Microkernel basierenden Betriebssystem handelt und nicht um ein Netzwerk aus WS, wofür N-Chance ursprünglich gedacht war, musste der Algorithmus zunächst an die Gegebenheiten des neuen Systems angepasst werden.

Im originalem N-Chance wird bei einem *local-hit* der angeforderte Block direkt vom Betriebssystem der WS an die lokale Applikation übergeben, ohne das hier eine andere WS

informiert werden muss. Jedoch müssen im PACA System, bedingt durch die Zentrale Rolle des File Servers, alle Anfragen zu einem möglicherweise entfernten Server geschickt werden. Aufgrund dieses Ablaufes wird die Zugriffszeit bei einem *local-hit* erhöht und die eines *remote-hit* verringert.

Die Zugriffszeit eines *local-hit* wird um die Transferdauer eines Request zum Server nach folgender Formel erhöht: *(startup + req_transfer_time) / bks_in_request* .

Die Zugriffszeit bei einem *remote-hit* kann ebenfalls mit dieser Formel angegeben werden, jedoch handelt es sich hier im Vergleich zu der alten Bedingung *(startup+ req_transfer_time)* um eine Verbesserung, denn es wird nur immer eine Nachricht pro User Anfrage zum Server geschickt und nicht eine Nachricht pro *remote-hit*.

4.2.5 Vergleich zwischen N-Chance und LRU-Interleaved

Beide Algorithmen implementieren einen Kooperativen Cache, bei dem sich alle beteiligten Nodes ihren File System Cache teilen. Der Hauptunterschied liegt darin, wie die Cache Daten auf die beteiligten Nodes verteilt werden. Dadurch das der LRU-Algorithmus keine Replikaktions-Mechanismen besitz, wird das Problem der Cache Kohärenz vermieden und es bedarf keiner Mechanismen zu dessen Gewährleistung. Die Strategie der Verteilung der Cache Daten auf die Nodes macht das Auffinden der Cache Daten beim verwenden des LRU-Algorithmuses besonders schnell und effektive.

Beim *N-Chance Forwarding* bedarf es eines extra Aufwandes für das Verwalten der Informationen, wo die Blöcke gespeichert wurden. Auch muss nachgehalten werden, welche Blöcke zu anderen Nodes repliziert wurden um festzustellen, ob es sich bei einem Block um ein *singlet* handelt oder nicht. Das Verwalten all dieser weiteren Informationen verursacht zusätzliche Kosten, welche beim *LRU-Interleaved* nicht anfallen. Diese extra Kosten werden jedoch in der Simulation zur Performance Analyse nicht mit eingerechnet, da hier nur eine System mit einem Server untersucht wurde.

Beim *LRU-Interleaved* wurde besonders auf eine gute Auslastung des Caches und auf die Vermeidung von Replikaktionen als auf die Verringerung der zwischen den Nodes zu transferierenden Datenmenge Wert gelegt. Deshalb hängt die Performance des *LRU-Interleaved* maßgeblich von der den Nodes zur Verfügung stehenden Kommunikations-Bandbreite ab, welche aber heutzutage kein Problem mehr darstellt. Im Gegensatz hierzu wurde der *N-Chance Forwarding* Algorithmus mit dem Ziel entwickelt, eine Möglichst hohe *local-hit* Rate zu erreichen und somit den Block-Transfer zwischen den Nodes zu vermeiden. Die Performance hängt deshalb nicht so stark von der Bandbreite ab.

Auch unterscheiden sich die Kosten für einen *remote-hit* bei den beiden Algorithmen. Beim *N-Chance Algorithmus* wird der Block zunächst vom remote Cache in den lokalen Cache des anfordernden Nodes kopiert, danach wird er vom lokalen Cache erst zum User kopiert. Bei *LRU-Interleaved* wird der entsprechende Block direkt aus dem remote Cache zum User kopiert.

Da der LRU Mechanismus in der zur Zeit implementierten Version, als Single-Server Variante, keine Cache Blöcken repliziert, bedarf es keiner Kohärenz Mechanismen. Beim N-Chance sind Kohärenz Mechanismen erforderlich, da hier durchaus mehrere Kopien eines Blocks im Cache vorhanden sein können.

In der Simulation zur Bewertung der Algorithmen sind folgende Punkte zu beobachten:

- Den Einfluss des lokalen Cachens auf die Anzahl der *dirty* und *forwarded* Blocks.
- Ob die Kosten eines *remote-hits* bei *LRU* evtl. schwerer wiegen als die des Block Forwarding bei *N-Chance*.
- Den Einfluss der unterschiedlichen Kosten eines *remote-hits*.
- Die Leistungsfähigkeit der Parallelisierung des Datentransfers.

4.3 Prefetching Algorithmen

Ein weitere Steigerung der Performance einer Applikation ließe sich erreichen, wenn man feststellen könnte, welche Daten als nächstes von ihr benötigt werden. Diese Daten könnten dann schon vom File System aus gezielt in den Cache vorgeladen werden, bevor sie vom File Server im Auftrag der Applikation angefordert werden. Somit lässt sich die Verzögerungszeit, die sich durch das Kommunikations-Netzwerk und durch die doch immer noch relative langsamen I/O-Systeme ergibt verbergen. Diesen Mechanismus, Daten im Voraus vom File System zu laden, bezeichnet man als *prefetching*.

Durch die Vereinigung aller lokalen Node-Caches zu einem globalen Cache entsteht ein Cache von immenser Größe. Durch diese Größe würde es Stunden dauern, bis der Cache voll gefüllt wäre und die gecachten Daten könnten dann mehrere Stunden alt sein. In der Simulation des PACA Systems mit 50 Nodes und je 16 MB lokalem Cache hat es annähernd 13 Stunden gedauert, bis der Cache gefüllt war. Dieser Umstand führt zu der Annahme, dass so große Caches mit einer sehr aggressiven Prefetching Strategie arbeiten sollten um so den Cache schnellstmöglich mit den wahrscheinlich von den Applikationen benötigten Daten zu füllen. Es ist jedoch auch bekannt, das Prefetching die Applikationen auch ausbremsen kann und zwar genau dann, wenn zu viele falsche Vorhersagen getroffen werden. Bei einem entsprechend großen Cache, wie dem bei PACA, würden zuvor fälschlicher Weise in den Cache geladene Daten jedoch nur extrem alte Daten verdrängen, was dann vernachlässigbar ist. Somit lässt sich allgemein sagen, je größer der Cache des Systems ist, umso aggressiver kann der Prefetching Mechanismus sein. Bei der Entwicklung von PACA wurden zwei Prefetching Strategien untersucht, *One-Block-Ahead* und *Full-Fill-On-Open*.

4.3.1 One-Block-Ahead Algorithmus

Beim *One-Block-Ahead* Algorithmus, wird bei jedem Lese- und Schreibvorgang der nächste Block in die Prefetching Queue eingestellt. Sobald das betroffene I/O-System (Disk) im Leerlauf ist, wird der erste Block aus der Queue vom I/O-System geladen. Sollte der Block jedoch bereits zuvor benötigt werden und es befindet sich noch ein Auftrag in der Queue, so wird dieser aus der Queue entfernt.

4.3.2 Erweiterung zum Full-Fill-On-Open in PACA

Neben dem *One-Block-Ahead* Algorithmus wurden auch einige aggressive Prefetching Strategien untersucht und es wurden durchweg gute Ergebnisse erzielt, die besten mit dem *Full-Fill-On-Open* Algorithmus. Hier wird nicht nur der nächste Block in die Prefetching Queue aufgenommen, sondern das ganze File, sobald es erstmalig geöffnet wird. Mit diesem Mechanismus befinden sich die meisten Blöcke bereits im Cache, wenn sie von der Applikation benötigt werden. Jedoch kann sich dieser Algorithmus auch als zu aggressive erweisen, wenn die Files zu groß sind.

Noch bessere Resultate in Bezug auf die Performance könnten erzielt werden, wenn der Algorithmus Informationen von der Applikation beziehen könnte und damit in der Lage wäre genau vorher zu bestimmen, welche Blöcke als nächstes zu laden sind.

5 Simulation

Die hier verwendeten Systeme zur Simulation des File Systems und des Caches sind Teil von DIMEMAS[2], womit sich das Verhalten eines parallelen Systems mit verteiltem Speicher in Abhängigkeit von diversen Parametern simulieren lässt. Der Simulator ist *Trace* basiert, wobei ein *Trace* die CPU-, Kommunikations- und I/O-Kommando-Sequenzen eines jeden Prozesses enthält und nicht die absolute Zeit jedes Ereignisses.

Jede Art von Kommunikation in dem System ist fest in zwei Teile separiert, eine *startup*- und eine *data-transfer* Phase. Der *startup* ist für jede Art von Kommunikation konstant und es wird CPU Bedarf angenommen. Der *data-transfer* ist proportional zu den zu versendenden Daten und der zur Verfügung stehenden Bandbreite des Interconnection Netzwerks.

Der File System Simulator ist in der Lage folgende Algorithmen und Strategien zu simulieren:

- Kein Caching – alle Anfragen gehen direkt zu den Festplatten
- N-Chance Forwarding – Algorithmus zum Vergleich der Performance
- LRU-Interleaved – Caching Algorithmus in PACA
- Kein Prefetching
- Full-Fill-On-Open Prefetching
- One-Block-Ahead Prefetching

5.1.1 Parameter zur Simulation

Bei allen Messungen verfügte der File Server über nur eine Festplatte. Bei den Untersuchungen wurden Messungen mit einem, zwei und vier Links pro Node zum Netzwerk vorgenommen, um deren Einfluss beim parallelen Transfer von mehreren Blöcken zu ermitteln.

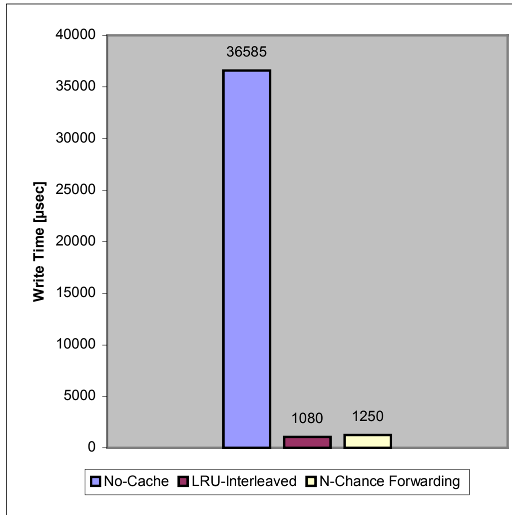
Parameter	Wert / Beschreibung
Größe eines Disk Blocks	8 KB
Größe eines Cache Blocks	8 KB
Dauer eines Block Schreibvorgangs	14,7 Millisekunden
Dauer eines Block Lesevorgangs	18,3 Millisekunden
Bandbreite vom Node zum ICN	155 Mbits/s
Bandbreite bei lokalen Kopien	320 Mbits/s
Dauer des Port startup	100 Mikrosekunden
Dauer einer memory_copy Operation	50 Mikrosekunden
Größe des lokalen Node Caches	16 MB

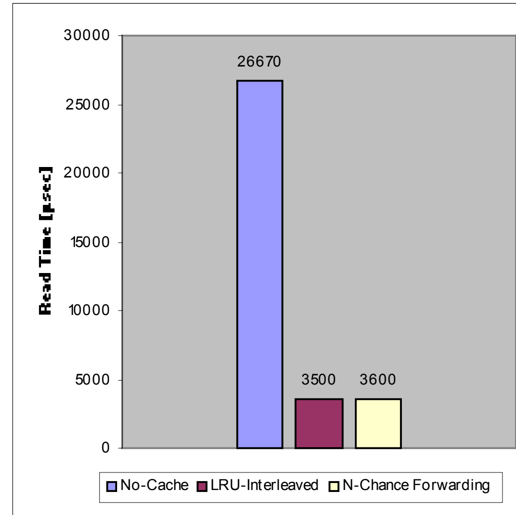
[2] DIMEMAS ist ein Simulator zur Performance Analyse bei Parallelen Systemen

6 Performance Analyse

6.1 Lese und Schreib Performance

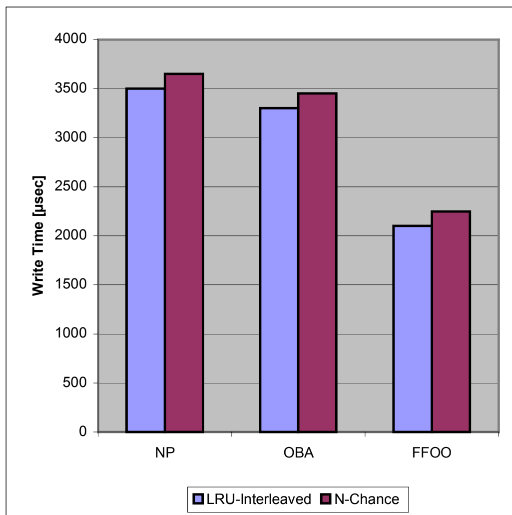
Die Ergebnisse zeigen, das sich die Lese Performance deutlich durch den Einsatz von Caching Algorithmen verbessern lässt. Der *LRU-Interleaved* bringt zudem noch eine Steigerung von ca. 3,1% gegenüber dem N-*Chance Forwarding* Algorithmus. Obwohl N-Chance eine bessere *local-hit* Rate hat, kann er dadurch keinen Vorteil gewinnen.

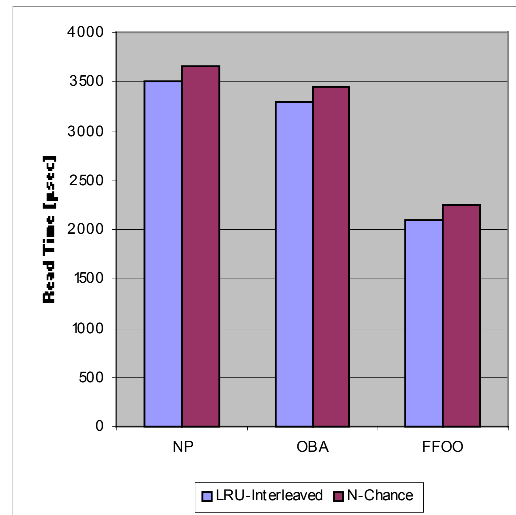




6.2 Prefetching Einfluss

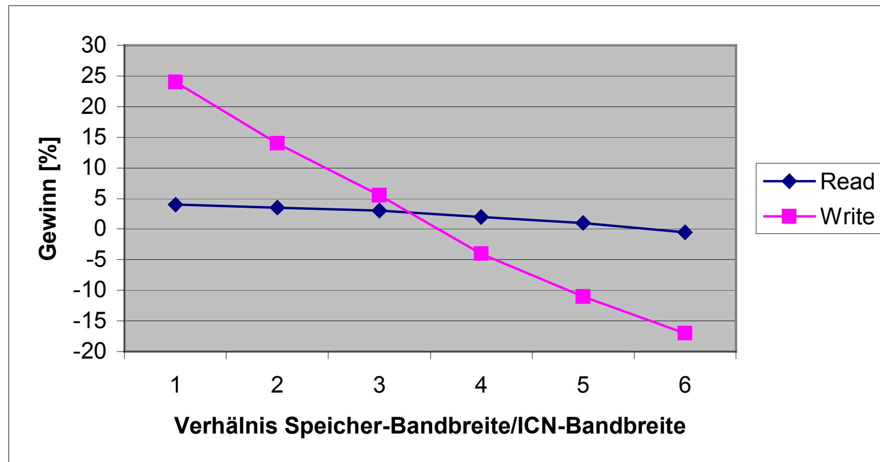
Mittels *One-Block-Ahead* Strategie lies sich nur eine unwesentliche Verbesserung erreichen, da das Prefetching des nächsten Blockes unmittelbar nach dem Zugriff auf den vorhergehenden schon zu spät ist. Auch wenn die Applikation die Daten in einer zufälligen Reihenfolge anfordert, sind die mittels *One-Block-Ahead* vorgeladenen Blöcke meist falsch. Mittels dem vollständigen Laden des Files, wenn es geöffnet wird, erhöht sich die Wahrscheinlichkeit, das der benötigte Block schon in den Cache vorgeladen ist. Auch bei einem zufälligem Zugriff sind die meisten Blocks des Files so bereits geladen.





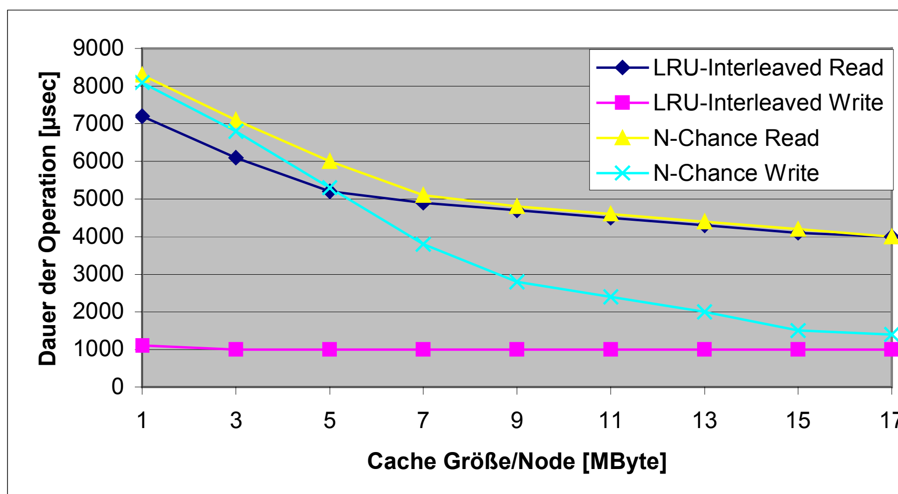
6.3 Bandbreite des Verbindungsnetzwerkes

Die Ergebnisse zeigen, dass die Vorteile von *LRU-Interleaved* gegen über *N-Chance* maßgeblich vom Verhältnis der Speicher-Bandbreite zur ICN-Bandbreite abhängen. *LRU-Interleaved* erreicht seine besten Resultate, wenn sich die beiden Bandbreiten nicht unterscheiden. So ist er bei Read-Operationen um ca. 5% schneller und bei Write-Operationen um bis zu 25%. Bis zu einem Bandbreitenverhältnis von 6 ist *LRU-Interleaved* beim Lesen und einem Verhältnis von 4 beim Schreiben schneller. Dies zeigt, das *LRU-Interleaved* bei dem zugrunde liegendem System die bessere Wahl ist. Erst ab einem Bandbreitenverhältnis von 5 ist der *N-Chance* Algorithmus vorzuziehen.



6.4 Größe des Caches

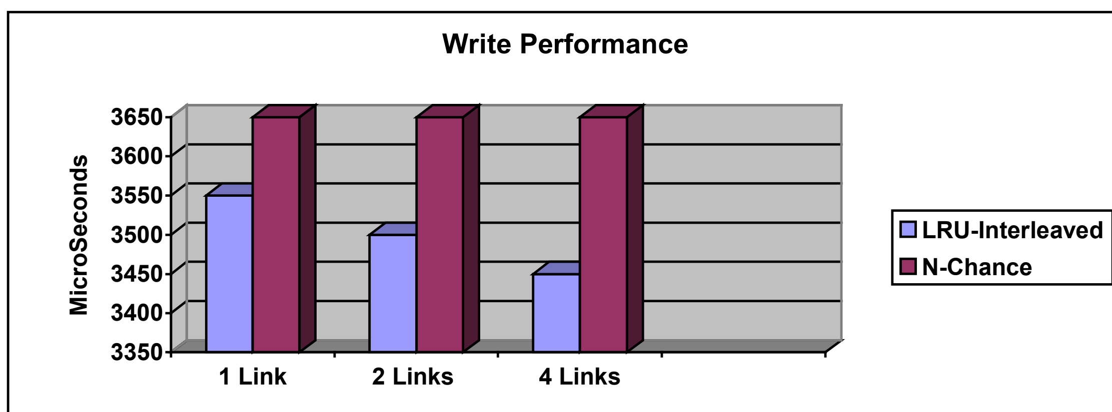
Die Untersuchungen zeigen, das *LRU-Interleaved* besser mit kleinen lokalen Caches arbeitet als *N-Chance*. Durch die bessere Ausnutzung des Caches erreicht *LRU-Interleaved* eine bessere *global-hit* Rate, denn dadurch, dass die Daten nicht repliziert werden, enthält der gesamte Cache nur „nützlich" Daten und keine mehrfachen Kopien durch replizierte Blöcke. Auch der Vorteil von *N-Chance* durch eine bessere *local-hit* Rate kann das nicht ausgleichen. Wenn *N-Chance* mit kleinen Caches benutzt wird, wächst die Wahrscheinlichkeit das ein *dirty-Block* weitergeleitet werden muss. Dieser Block muss zuvor gelöscht werden, bevor er zu einem anderen Node geschickt wird, was den Zeitbedarf bei Schreib Operationen erhöht.

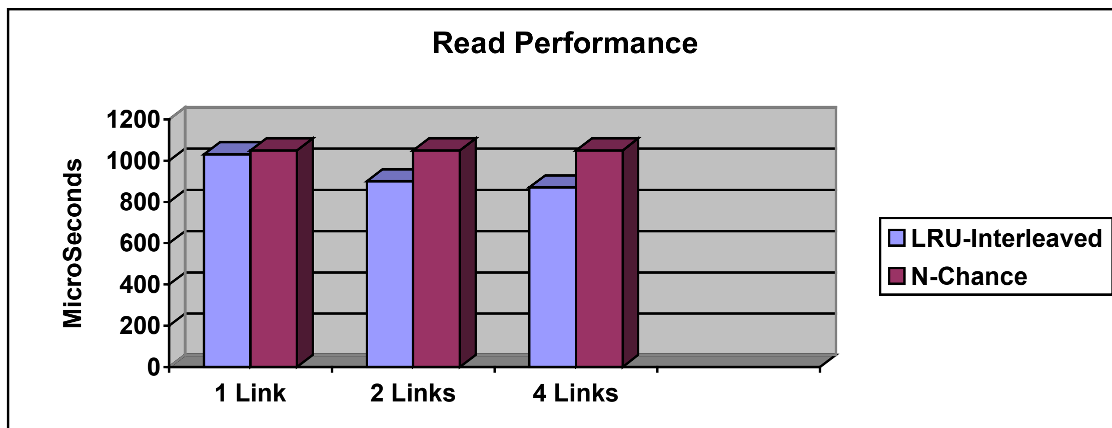


6.5 Anzahl der Links pro Node zum Verbindungsnetzwerk

Ein Node des Systems kann über mehrere Links zum Interconnection Netzwerk (ICN) verfügen und so mehrere Daten parallel senden und empfangen. Die Test wurden mit 1, 2 und 4 Links durchgeführt, wobei jedoch 4 Links unnötig sind, da das Netzwerk nur halb so langsam ist wie der Speicher. Bei Lese Vorgängen konnten durch mehrere parallele Links keine besonderen Verbesserungen erreicht werden, da das Netzwerk schnell genug ist um Lese Anforderungen zu übertragen und diese auch selten parallel erfolgen.

Schreib Vorgänge werden jedoch beim *LRU-Interleaved* Algorithmus wesentlich von der Anzahl der Links beeinflusst, da das Schreiben im Wesentlichen aus *memory_copy* Operationen besteht. Außerdem sind Schreib Vorgänge umfangreicher und demnach können mehrere Kopien parallel gestartet werden. Der *N-Chance* Algorithmus ist hiervon weniger betroffen da er eine höhere *local-hit* Rate hat und somit das ICN weniger benutzen muss.





6.6 Performance mit nur einem zentralen File-Server

Die Messungen aller Zeiten für einen *local-hit, remote-hit, cache-miss, miss-on-dirty und miss-on-clean* haben ergeben, das ein einzelner Server nicht zur Schwachstelle in einem System mit bis zu 50 Nodes werden kann. Auch die Tatsache, dass neben dem File Server noch weitere User Prozesse auf dem selben Node laufen können ist zu vernachlässigen, solange es sich nicht um zeitkritische Anwendungen handelt.

7 Geplante Erweiterungen

Der bisher verwendete File Server lieferte in den Simulationsläufen recht gute Resultate. Hier wäre der nächste Schritt, diesen Server so zu implementieren, das er sich auf mehrere Nodes des Systems verteilen ließe. Hierbei ist jedoch dann darauf zu achten, dass der Kommunikationsbedarf zwischen den einzelnen Servern so gering wie möglich zu halten ist.

Auch soll versucht werden, das von Miller und Katz entwickelte RAMA[3] File System mit dem hier entwickelten Cache Modell zu kombinieren und so ein extrem schnelles und gut skalierendes Files System zu entwickeln.

Ebenso lässt sich die Performance der Applikationen mittels einer besseren Prefetching Strategie verbessern, welche eine bessere Vorhersage ermöglicht und somit auch die Effizienz des Caches steigert. Die hierfür nötigen Informationen könnten mittels bereits entwickelter Systeme, wie *Disk-Directed I/O*[4], gewonnen werden.

8 Résume

An das zu entwickelnde System wurden zu Beginn drei maßgeblich Anforderungen gestellt, die es umzusetzen galt. Dieses waren Einfachheit, Skalierbarkeit und Performance. Zufriedenstellend sind zur Zeit die Anforderung an die Performance und die Einfachheit des Systems gelöst worden. Die Skalierbarkeit beschränkt sich zur Zeit noch auf eine maximale Anzahl von 50 Nodes im System.

Die Untersuchungen haben auch gezeigt, das der implementierte, relative einfache Caching Algorithmus die gleiche *read-Performance* und sogar eine bessere *write-Performance* erzielt, wie der komplexere *N-Chance* Algorithmus.

Auch ist das Verhältnis der Speicherbandbreite zur Bandbreite des Interconnection Netzwerkes entscheiden, ob nun der *LRU*- oder der *N-Chance* Algorithmus bessere Ergebnisse liefert. Erst wenn die Speicherbandbreite 5-mal so hoch ist wie die des Netzwerkes, lässt sich mit *N-Chance* eine bessere Performance erreichen.

Die Simulation hat gezeigt, das es einige entscheidende Faktoren gibt, die man bei dem Entwurf eines verteilten Caches berücksichtigen muss. Steht ein Interconnection Netzwerk mit hoher Bandbreite zur Verfügung, so können nicht optimale *remote*- und *local-hit* Raten durch die Reduzierung des Bedarfs an *block-forwardings* und das Reduzieren der nötigen *cache-cleans* kompensiert werden. Auch kann das erhöhte Transfervolumen eines *remote-hits* durch die Parallelisierung der Kommunikation, ausgeglichen werden.

Es hat sich gezeigt, dass eine aggressive Prefetching-Strategie die Performance des Systems erheblich beeinflussen kann. So ist hierdurch die *cache-hit* Rate verbessert worden, was zur Folge hat, das die durchschnittliche *read-time* verringert wird.

[3] Ethan L. Mille, Randy H. Katz: RAMA - An Easy-To-Use, High-Performance Parallel File System
[4] D. Kotz: Disk-Directed I/O for MIMD Multiprocessors

Quellenverzeichnis

Tannenbaum A. S.: Moderne Betriebssysteme, 2.Auflage, Hanser Verlag, München, 1995

Tannenbaum A. S., Goodman J.(2001): Computerarchitektur, Pearson Studium, München, 2001

Cortes T., Girona S., Labarta J.(1996): PACA: a Cooperative File System Cache for Parallel Machines, Universitat Politècnia de Catalunya, 1996

Ottman T., Widmayer P.: Algorithmen und Datenstrukturen, Spektrum Verlag, Heidelberg, Berlin, Oxford, 1996

Stockinger H.: Dictionary on Parallel Input / Output, Master's Thesis, Institute for Applied Computer Science and Information Systems, University of Vienna, 1998