Adaptives Caching in verteilten Informationssystemen

Dissertation

zur Erlangung des Grades

Doktor der Ingenieurwissenschaften (Dr.-Ing.)

der Technischen Fakultät

der Universität des Saarlandes

von

Dipl.-Inform.

Markus Sinnwell

Saarbrücken

Februar, 1998

Inhaltsverzeichnis

Danksagung

Diese Arbeit entstand während meiner Zeit als wissenschaftlicher Mitarbeiter am Lehrstuhl für Datenbanken und Informationssysteme an der Universität des Saarlandes. Für die Gelegenheit hierzu möchte ich mich bei Herrn Prof. Gerhard Weikum besonders bedanken. Darüber hinaus hat sein Interesse an automatischem *Performance-Tuning* die Entstehung dieser Arbeit erst ermöglicht und seine Anregungen und Diskussionen ihren Inhalt entscheidend geprägt.

Bedanken möchte ich mich auch bei Herrn Prof. Erhard Rahm, der sich bereit erklärt hat, das Koreferat für diese Arbeit zu übernehmen.

Für das sehr angenehme Arbeitsklima möchte ich auch meinen Kollegen am Lehrstuhl danken. Besonders herausstellen möchte ich dabei Achim Kraiß. Während der gemeinsamen Zeit am Lehrstuhl war er der beste Kollege und Zimmergenosse, den man sich wünschen kann, und ich werde mich immer gerne an diese Zeit erinnern.

Mein ganz besonderer Dank gilt meinen Eltern Toni und Linde Sinnwell. Nur durch ihre Unterstützung und ihren Ansporn ist diese Arbeit möglicht geworden. Besonderer Dank gebührt meinem Vater auch für das sehr genaue Korrekturlesen der Arbeit und für die vielen Verbesserungsvorschläge, die zur besseren Lesbarkeit der Arbeit beigetragen haben.

für meine Mutter
Linde Sinnwell

Kurzfassung

In dieser Arbeit stellen wir eine neuartige Methode für verteiltes Caching vor, die das Ausnutzen des aggregierten Hauptspeichers eines lokalen Rechnernetzwerkes für datenintensive Anwendungen erlaubt. Ein detailliertes Kostenmodell dient als Basis für die Bestimmung einer guten Cache-Allokation. Da im allgemeinen die Eingabegrößen für dieses Kostenmodell im Online-Fall nicht a priori bekannt sind und darüber hinaus dynamisch schwanken können, werden speziell angepaßte Protokolle zur Bestimmung und Verteilung dieser Informationen benutzt. Durch eine approximative Online-Auswertung des Kostenmodells paßt sich unsere Caching-Strategie adaptiv an die aktuelle Last an und kann dadurch Engpäße auf den Systemressourcen verhindern. Zur Vermeidung von Lastungleichgewichten wird aus dem Kostenmodell zusätzlich eine Lastverteilungsheuristik abgeleitet, die im Hintergrund zielgerichtete Migrationen von Objekten zur Balancierung des Systems durchführen kann. Sowohl die Anpassung an die aktuelle Last als auch die Vermeidung von Ungleichgewichten führt zu einer – verglichen mit bisherigen Methoden – deutlich besseren Leistungsfähigkeit unserer Heuristik. Eine Erweiterung des entwickelten Caching-Verfahrens berechnet eine Partitionierung des aggregierten Hauptspeichers, so daß Antwortzeitziele für unterschiedliche Klassen von Operationen sichergestellt werden.

Abstract

This dissertation presents a new method for distributed caching to exploit the aggregate memory of networks of workstations in data-intensive applications. The approach is based on a detailed cost model to compute a good cache allocation. As the input parameters for the cost model are in general a-priori unkown and possibly evolve, we use specifically designed protocols to estimate and disseminate this information. Using an approximative online evaluation of the cost model, our caching heuristics adapts automatically to evolving workloads and thus avoids bottlenecks on system resources. To prevent load imbalances we further derive a load distribution method from the cost model. This load distribution method asynchronously migrates objects from highly loaded nodes onto lightly loaded nodes. The adaptation to the current workload and the prevention of imbalances result in significant performance improvements compared to prior methods. We further present a goal-oriented extension of the developed caching method, which determines a partitioning of the aggregate cache to satisfy given response-time goals for different classes of operations.

Zusammenfassung

Netzwerke mit leistungsfähigen Workstation-Rechnern, auch als NOWs (*Network of Workstation*) bezeichnet, bieten eine hervorragende Basis für kosteneffiziente und skalierbare verteilte Computeranwendungen. Solche Rechnerverbunde werden bereits in einer Vielzahl von Projekten benutzt, wobei jedoch der Schwerpunkt auf der Ausnutzung der aggregierten CPU-Leistung liegt. In datenintensiven Anwendungen, wie z.B. Datenbanksystemen, bilden jedoch häufig die Plattenzugriffe den Engpaß. Für den Einsatz von NOWs in datenintensiven Anwendungen ist es daher entscheidend, die aggregierten Speicherressourcen (Hauptspeicher und Festplatte) ebenfalls zu nutzen. Da bei schnellen Netzwerken ein Datenobjekt im allgemeinen schneller aus dem Hauptspeicher eines anderen Rechners gelesen werden kann als von der lokalen Platte, wird in dieser Arbeit eine heuristische verteilte Caching-Strategie entwickelt mit dem Ziel, die mittlere Antwortzeit von Objektzugriffen zu minimieren. Die Zugriffe können dabei auf jedem Rechner des NOWs – unabhängig von der Datenplazierung und dem Zugriffsverhalten anderer Rechner – initiiert werden.

Um die Leistungsfähigkeit unseres neuen Verfahrens beurteilen zu können, betrachten wir zusätzlich zwei einfache Heuristiken, die jeweils einen Grenzfall bezüglich des Grads an Kooperation darstellen. Die erste Heuristik agiert völlig egoistisch auf jedem Rechner in dem Sinne, daß jeder Rechner versucht, die lokal wertvollsten Objekte in seinem lokalen Cache zu halten. Die zweite Heuristik stellt im Gegensatz dazu die globale Kooperation zwischen den Rechnern in den Vordergrund, indem sie ohne Rücksicht auf die lokalen Auswirkungen versucht, die global wertvollsten Objekte im aggregierten Cache zu halten. Um möglichst viele Objekte im aggregierten Speicher halten zu können, verbietet diese Heuristik zusätzlich die Replikation von Objekten.

Beide Heuristiken versuchen jeweils, die Auslastung einer Ressource zu minimieren, ohne jedoch die aktuellen Last-, Daten- und Systemparameter zu berücksichtigen. Während die egoistische Heuristik durch die Maximierung der lokalen Cache-Trefferrate die Netzauslastung reduziert, erreicht die altruistische Heuristik durch die Maximierung der globalen Cache-Trefferrate eine Verringerung der Plattenauslastung. Da jedoch die Engpaßressource (Netzwerk oder Festplatte) von den aktuellen Parametern abhängig ist, die sich darüber hinaus dynamisch ändern können, kann keine dieser Methoden als universell geeignet angesehen werden.

Unsere neue Heuristik versucht nun durch approximative Kostenformeln, den Nutzen eines Objekts dynamisch zu bestimmen, um so flexibel und für jedes Objekt getrennt den Replikationsgrad festzulegen. Den Nutzen eines Objekts bestimmen wir dabei als die Differenz in den mittleren Zugriffskosten, wenn wir entweder das betreffende Objekt löschen oder im Cache halten. Die Kostenformeln für die Berechnung des Nutzens leiten wir aus einem detaillierten, analytischen Modell ab, das auch die Auslastung der Ressourcen berücksichtigt. Nach einer Reihe von Approximationen, die wir durchführen, um den Berechnungs-Overhead gering zu halten, sind für die Bestimmung des Nutzens eines Objekts nur noch die folgenden Informationen notwendig:

- die lokale Hitze (d.h. die lokale Zugriffshäufigkeit) des Objekts,
- die globale Hitze des Objekts,
- der Replikationsgrad des Objekts und
- die Auslastung der Festplatten und des Netzwerks.

Um diese Informationen lokal verfügbar zu machen, benutzen wir speziell angepaßte Protokolle, die eine approximate Bestimmung und Verteilung dieser Größen mit geringem Overhead ermöglichen.

Zwar spiegelt der berechnete Nutzen den Wert eines Objekts innerhalb des Gesamtsystems wieder, jedoch führen wir die Bestimmung eines Opfers bei einer Cache-Ersetzung immer lokal durch. Dies verhindert, daß die Antwortzeit eines Zugriffes durch die ansonsten notwendige Migration von Opfern zwischen verschiedenen Rechnern zu sehr verzögert wird.

Um trotzdem einen Ausgleich zwischen unterschiedlich stark belasteten Rechnern zu erreichen, haben wir – basierend auf dem oben angesprochenen Kostenmodell – eine Erweiterung entwickelt, die im Hintergrund und abhängig von einem eventuell vorhandenen Ungleichgewicht Objektmigrationen durchführt und dadurch eine Balancierung innerhalb des Gesamtsystems erreicht.

Neben der Minimierung der mittleren Antwortzeit über alle Zugriffe, die wir durch unsere kostenbasierte Heuristik erzielen können, ist in wachsendem Maße auch eine Berücksichtigung von vorgegebenen Antwortzeitzielen für unterschiedliche Anfrageklassen wichtig. Eine Möglichkeit solche Ziele zu gewährleisten besteht darin, den insgesamt vorhandenen Hauptspeicher in dedizierte Bereiche aufzuspalten, die dann jeweils nur von den Anfragen einer Klasse genutzt werden dürfen.

Ausgehend von dieser Vorgehensweise stellen wir eine Methode vor, die eine zielorientierte Cache-Partitionierung des aggregierten Hauptspeichers eines NOWs berechnet. Die Methode führt während des laufenden Systems Beobachtungen durch, aus denen eine lineare Approximation der Abhängigkeit zwischen den Größen der dedizierten Caches und der daraus resultierenden mittleren Antwortzeit abgeleitet wird. Ausgehend von dieser Information kann nun durch Extrapolation für eine Klasse die Cache-Partitionierung bestimmt werden, bei der diese Klasse die gewünschte Zielantwortzeit erreicht. Um die Fehler durch die Approximation zu kompensieren und um eine adaptive Anpassung zu ermöglichen, betten wir diesen Algorithmus in einen Regelkreis ein. Führt eine Laständerung zu einer Verletzung der Zielantwortzeit einer Klasse, so wird eine dynamische Neuberechnung der Cache-Partitionierung durchgeführt. Bei einer neuen Partitionierung verändert sich jeweils nur die Aufteilung der Caches, während die aggregierte Größe des zur Verfügung stehenden Hauptspeichers konstant bleibt. Um die Vorteile unserer kostenbasierten verteilten Cache-Heuristik auch bei der Verwaltung der dedizierten Caches ausnutzen zu können, verallgemeinern wir die Heuristik durch das Berücksichtigen klassenspezifischer Hitzen.

In dieser Arbeit haben wir eine neue Cache-Heuristik für verteiltes Caching vorgestellt. Durch die Berücksichtigung eines Kostenmodells kann sie adaptiv auf Lastschwankungen reagieren und Ungleichgewichte im System vermeiden. Diese Eigenschaften konnten wir durch eine umfangreiche Simulationsstudie beweisen, in der wir die Last-, Daten- und Systemparameter über ein breites Spektrum variiert haben. Innerhalb dieser Studie hat sich außerdem gezeigt, daß unser neues kostenbasiertes Verfahren – trotz des zusätzlichen Verwaltungs-Overheads – den einfachen Heuristiken durchgängig überlegen ist. Zusätzlich haben wir zur Berücksichtigung vorgegebener Zielantwortzeiten für Anfrageklassen erstmals einen Partitionierungsalgorithmus für den aggregierten Cache eines NOWs vorgestellt.

Summary

Networks of workstations, also known as NOWs, are an intriguing architectural paradigm for cost-efficient and scalable distributed computer applications. Using such architectures for computing-intensive problems has been investigated in a variety of projects with emphasis on exploiting the aggregated CPU power of a NOW. In data-intensive applications such as database systems, on the other hand, the bottleneck is usually in the disk I/O load. Here the key factor rather is to exploit the performance capacity of distributed disks and, for caching benefits, the aggregate memory of a NOW. As the costs for accessing a data object from a remote cache are often lower than accessing the same object from a local disk (assuming a high-speed network), we have developed a heuristic distributed-caching strategy that aims to minimize the mean response time of accesses, where accesses can be initiated on every node of the NOW independently of the data placement and the access behavior of other nodes.

In addition to this new heuristics we consider two simple heuristics, which represent two extremes with regard to the coordination of the caches of different nodes. The first heuristics acts totally egoistically on every node by keeping the locally most valuable objects in the local cache. In contrast, the second heuristics favors the cooperation between different nodes by storing globally valuable objects in the caches, so as to keep as many objects as possible in the aggregate cache by disallowing the replication of objects among nodes.

Both heuristics aim to minimize the utilization of a single resource independently of the current workload, data, and system characteristics. While the egoistic heuristics reduces the network utilization by maximizing the local cache hit rate, the altruistic heuristics achieves a reduction of the disk utilization by maximizing the global cache hit rate. However, as the bottleneck resource (network or disk) depends on the current parameters and these parameters can even dynamically evolve, none of these simple heuristics can be considered as acceptably good.

Our own heuristics overcomes this problem by using approximative cost formulas to compute the caching benefit of an object dynamically. This information can then be used to determine the degree of replication on a per object basis. We consider the benefit of an object to be the difference in the access cost between keeping the object in the local cache versus dropping it. To derive the cost formulas, we use a detailed analytical model which takes the utilization of the hardware resources into account. Using some simplifications, which are justified by the high complexity of the exact cost formulas, we merely have to determine the following information to compute the benefit of an object:

- the local heat (i.e., local access frequency) of the object,
- the global heat of the object,
- the degree of replication of the object, and
- the utilization of the disks and the network.

To collect and disseminate this information, we have developed specifically designed low-overhead protocols.

The approximative cost model is exploited for intelligent decisions about local cache replacements. Ideally, the object with the lowest benefit in the entire system would be the best choice as a

cache replacement victim. However, to bound the overhead, we always determine the victim locally.

To further improve performance, we have developed a load balancing method that is also based on the analytical cost model. This method is demand-driven in the sense that it becomes active only upon load imbalances. In this case it asynchronously migrates objects between nodes with the goal of keeping the objects with the highest benefit in the aggregate cache while striving for a global load balance.

In addition to minimizing the mean response time over all accesses, which is the objective of our cost-model-based heuristics, the consideration of given response time goals for various workload classes becomes increasingly important. A possible approach to satisfy such response time goals is the partitioning of the available memory into dedicated areas which are then used solely for the caching of objects accessed by the corresponding workload classes.

We extend our distributed-caching approach to compute a goal-oriented cache partitioning for the aggregate main memory of a NOW. Based on online observations, this method approximates the dependency between the sizes of the dedicated cache areas and the resulting class-specific response times. Extrapolating this information, we can compute appropriate sizes of the dedicated caches for a class so that the response time goals are satisfied. To compensate the faults caused by the approximation and for adaptivity, we embed this computation in a feedback loop. When a change in the observed response time signals a violation of the goal, a recomputation of the cache allocation is initiated and the sizes of the caches are adjusted accordingly. The change caused by the recomputation only affects the partitioning of the aggregate cache, not the overall size. To exploit the advantages of our cost-model-based distributed caching heuristics for the management of the dedicated cache partitions, we generalize the method by incorporating workload-class-specific heat estimation and dissemination procedures.

To summarize our contribution, we have introduced a novel heuristics for distributed caching. Based on a cost model, this method adapts dynamically to evolving workloads and avoids imbalances in the system. We have substantiated these properties in an extensive simulation study in which we have varied the workload, data, and system parameters over a wide range. Despite its higher overhead the newly developed method has consistently shown better results than the two simpler heuristics. To satisfy response time goals for workload classes, we have introduced a method for computing an appropriate cache partitioning of the aggregate memory of a NOW.

Kapitel 1

Einleitung

In diesem Kapitel beschreiben wir in Abschnitt 1.1 zunächst die technologischen Trends, die dazu geführt haben, daß Workstation-Rechner zum Aufbau skalierbarer Super-Server benutzt werden können. In Abschnitt 1.2 betrachten wir dann die Probleme, die zur effizienten Verwaltung des aggregierten Hauptspeichers innerhalb eines Netzwerks gelöst werden müssen und stellen unseren Beitrag in diesem Bereich vor. Anschließend diskutieren wir in Abschnitt 1.3 verwandte Arbeiten auf diesem Gebiet und geben schließlich in Abschnitt 1.4 einen Überblick über den Aufbau dieser Arbeit.

1.1 Technologietrends

Zunehmende Leistungsanforderungen, wachsende Datenvolumina und komplexere Anwendungen erfordern eine immmer leistungsfähigere Hardware bei der Realisierung von Informationssystemen. Während man in früheren Jahren versucht hat, diese hohe Leistungsfähigkeit durch speziell konstruierte, zentrale Datenbank-Server [HMP89, KePr92] zu erreichen, bietet heutzutage die verteilte Bearbeitung auf einem Netzwerk von Rechnern – auch NOW (*Network of Workstation*) [ACP*95] genannt – eine kostengünstige Alternative. Im Gegensatz zu spezialisierten Datenbankmaschinen, die wegen ihrer langen Entwicklungszeit oft von neuen Technologieentwicklungen überholt werden und wegen ihrer geringen Stückzahl zu sehr hohen Preisen verkauft werden müssen [BoDe83, DeGr92], kann bei einem NOW auf günstige und der aktuellen Technologie entsprechende Workstation-Rechner zurückgegriffen werden [Gray95, Micr97]. Obwohl die CPU-Leistung einer solchen Architektur durch das Hinzufügen weiterer Rechner voll skalierbar ist, hat die begrenzte Bandbreite der Verbindungsnetzwerke bei datenintensiven Anwendungen in der Vergangenheit eine entsprechende Gesamtskalierung verhindert. Daher werden NOWs bisher hauptsächlich für rechenintensive Anwendungen [LLM88, Bjor92, RoSe92, SGDM94, ACP*95] eingesetzt.

Durch neue Entwicklungen wurden in den letzten Jahren jedoch enorme Fortschritte in der Netzwerktechnologie erreicht. In Tabelle 1.1 haben wir die technischen Daten und die Preise einiger Netzwerke aufgetragen. Deutlich zu erkennen ist das enorme Anwachsen der maximalen Bandbreite, die durch neue Hardwarekomponenten (z.B. schnellere Netzwerkkarten und Übergang zu Glasfaserverbindungen) möglich wurde. Die Bandbreite bestimmt die Menge an Daten, die pro

Netzwerk-typ	Computer-typ	Software-protokoll	max. Band-breite [MBit/s]	Latenzzeit [µs]	Preis[1] [DM]
Ethernet	SPARC 20	TCP/IP	10	6700	200+8×700 [Tran97]
Fast-Ethernet	Pentium	U-Net	100	800	1000+8×100 [Tran97]
ATM AN2	Alpha	DEC-ATM	155	870	–
Myrinet	SPARC 20	UIUC FM	140	551	4300+8×2500 [Myri97]
SP/2	RS/6000 SP	MPI	360	480	–

Tabelle 1.1: Leistungsfähigkeit verschiedener Netzwerke [Venk96,WBE97]

Zeiteinheit über das Netzwerk transportiert werden kann und ist daher ein Maß für den Durchsatz des Netzwerks. Neben der Erhöhung des Durchsatzes erkennen wir jedoch auch eine signifikante Verringerung der Latenzzeit. Als Maß für die Latenzzeit betrachten wir in Tabelle 1.1 die minimale Zeit, die für das Lesen eines 8 kByte großen Datenobjekts aus dem Hauptspeicher eines anderen Rechners benötigt wird. Hierin ist sowohl die Zeit für das Versenden der Anfrage als auch das Zurücksenden der eigentlichen Daten mit eingeschlossen. Nicht berücksichtigt werden jedoch die eventuell auftretenden Verzögerungen durch Warteschlangeneffekte. Während die Gewinne bei dem Durchsatz hauptsächlich durch Hardwarekomponenten erzielt werden, trägt die Optimierung der Kommunikationsprotokolle und die daraus resultierende Reduzierung des CPU-Overheads entscheidend zur Verbesserung der Latenzzeit bei. Neben dem technisch Machbaren ist jedoch auch der Preis ein wichtiges Argument bei der Anschaffung eines neuen Netzwerks. In Tabelle 1.1 können wir erkennen, daß sehr leistungsstarke Netzwerke heute bereits zu einem günstigen Preis kommerziell vertrieben werden, und somit steht der Nutzung eines NOWs auch für datenintensive Anwendungen nichts mehr im Weg. Neben der aggregierten CPU-Leistung eines NOWs können datenintensive Anwendungen vor allem von den im Gesamtsystem vorhandenen, jedoch über die Rechner verteilten, Speichermedien (lokale Hauptspeicher und lokale Festplatten) profitieren. Insbesondere die Ausnutzung des aggregierten Hauptspeichers als Daten-Cache verspricht bei intelligenter Nutzung eine Reduktion der Festplattenzugriffe und damit eine bessere Antwortzeit für Datenzugriffe.

Das Potential für diese Verbesserung können wir in der Abbildung 1.1 erkennen. In dieser Abbildung haben wir die Entwicklung der lokalen Zugriffszeiten auf Hauptspeicher und Festplatte über die letzten 20 Jahre aufgetragen. Deutlich zu erkennen ist, daß sich der bereits 1980 vorhandene Abstand zwischen den Zugriffszeiten bis 1995 auf den Faktor 10^4 vergrößert hat. Der Grund für den geringen Gewinn bei den Festplatten liegt darin, daß hier – im Gegensatz zum Hauptspeicher – mechanische Bewegungen durchgeführt werden müssen. Durch die Massenträgheit der entsprechenden Bauteile und der für die Beschleunigung notwendigen Energie sind jedoch vergleichbar große Gewinne wie bei der Halbleitertechnologie nicht möglich und auch in der Zukunft nicht zu erwarten. Innerhalb eines NOWs kann diese Lücke in den Zugriffszeiten durch den nichtlokalen Hauptspeicher geschlossen werden. So haben wir in Tabelle 1.1 gesehen, daß in modernen Netzwerken die Zeit für das Lesen eines Objekts aus dem nichtlokalen Speicher eines anderen Rechners weniger als eine Millisekunde beträgt. Somit ist die Zugriffszeit auf den nichtlokalen Speicher um ungefähr eine Zehnerpotenz besser als die für einen lokalen Festplattenzu-

1. Der Preis entspricht jeweils den Kosten für einem *Hub* bzw. *Switch* mit mindestens 8 Eingängen zusammen mit 8 Netzwerkadaptern für die entsprechende Computerhardware.

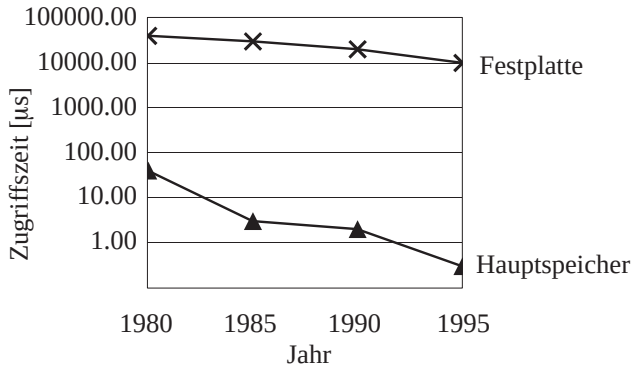


Abbildung 1.1: Entwicklung der Zugriffszeiten nach [HePa96]

griff. Da Zugriffe auf den nichtlokalen Hauptspeicher darüber hinaus sowohl von zukünftigen Entwicklungen in der Halbleiter- als auch in der Netzwerktechnologie profitieren, ist zu erwarten, daß sich die Ausnutzung des aggregierten Hauptspeichers auch in Zukunft lohnt.

Entsprechend dieser Beobachtungen führen wir für ein NOW die in Abbildung 1.2 gezeigte Speicherhierarchie ein. Wird ein Zugriff initiiert, so wird zuerst kontrolliert, ob sich das entsprechende Objekt im lokalen Hauptspeicher befindet. Wird es dort nicht gefunden, so wird als nächstes überprüft, ob sich das Objekt im Hauptspeicher eines anderen Rechners befindet. Erst wenn auch dies erfolglos ist, wird das Objekt von Platte gelesen. Bei den Platten unterscheiden wir dabei nach Möglichkeit zwischen den lokalen und den nichtlokalen Platten.

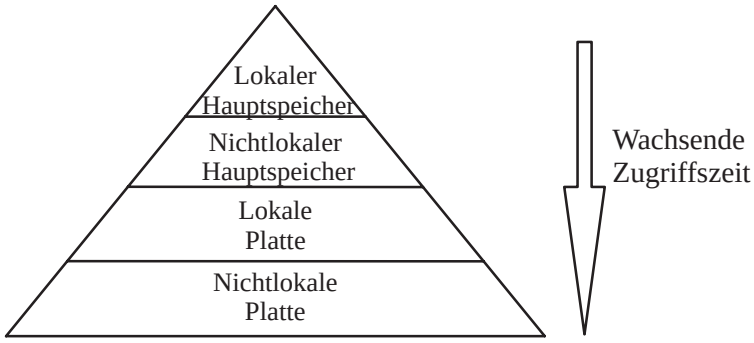


Abbildung 1.2: Zugriffshierarchie für verteiltes Caching

1.2 Problemstellung und Beitrag der Arbeit

Bevor wir auf die Probleme bei der effizienten Ausnutzung des aggregierten Hauptspeichers eines NOWs zu sprechen kommen, wollen wir zunächst die Systemumgebung genauer beschreiben. Wir nehmen an, daß das NOW aus mehreren Rechnern besteht, die durch ein schnelles Netzwerk miteinander verbunden sind. Jeder Rechner besitzt eine eigene CPU und einen Hauptspeicherbereich, der als Daten-Cache genutzt wird. Die Größen der Daten-Caches verschiedenen Rechner können sich unterscheiden, jedoch bleiben die Größen jeweils konstant. Zusätzlich kann lokal an jeden Rechner eine Festplatte angeschlossen sein. Zur Sicherstellung der Persistenz ist jedes Datenobjekt auf mindestens einer Platte gespeichert. Unabhängig von der Datenallokation auf Platte und von den Zugriffsmustern anderer Rechner können auf jedem Rechner lokal Zugriffe auf beliebige Objekte initiiert werden. Bei der Abarbeitung eines Zugriffes wird das Objekt

von der höchsten Stufe der Hierarchie entsprechend der Abbildung 1.2 gelesen, auf den Initiatorrechner transferiert und dort in den Daten-Cache eingefügt.

Da der lokale Daten-Cache nur eine begrenzte Größe besitzt, muß im allgemeinen bei einer Einfügeoperation ein anderes Objekt aus dem Cache entfernt werden. Eine erste, naheliegende Heuristik zur Bestimmung eines solchen Objekts besteht darin, das Objekt mit dem lokal geringsten Wert (z.B. entsprechend einer lokalen LRU-Strategie) auszuwählen. Ist das Zugriffsmuster jedoch auf allen Rechnern gleich, so führt diese Strategie dazu, daß auf jedem Rechner die gleichen Objekte im lokalen Hauptspeicher gehalten werden. Dadurch ist die Anzahl der unterschiedlichen Objekte im aggregierten Hauptspeicher sehr gering, und es müssen viele Objekte von Platte gelesen werden. Die schnellere Zugriffszeit von nichtlokalem Hauptspeicher – im Vergleich zu Festplatten – wird bei dieser Heuristik daher sehr schlecht ausgenutzt.

Aus diesem Grund sollten die Rechner bei der Verwaltung des aggregierten Hauptspeichers kooperieren. Dazu müssen neben lokalen Entscheidungskriterien auch die Hauptspeicherinhalte anderer Rechner berücksichtigt werden. Eine mögliche Strategie könnte darin bestehen, den aggregierten Hauptspeicher analog zu einem lokalen Hauptspeicher zu verwalten, d.h. jedes Objekt kommt höchstens einmal im aggregierten Hauptspeicher vor, und es wird jeweils das Objekt gelöscht, das den geringsten globalen Wert besitzt (z.B. entsprechend einer globalen LRU-Strategie). Trotz der großen Bandbreite und der geringen Latenzzeit von modernen Netzwerken sind jedoch die Zugriffszeiten auf den lokalen Hauptspeicher immer noch wesentlich geringer als die auf den nichtlokalen Hauptspeicher. Daher kann die Vermeidung von Replikation dazu führen, daß mehr Objekte aus dem nichtlokalen Hauptspeicher gelesen werden müssen und somit der Vorteil von lokalen Hauptspeicherzugriffen nicht entsprechend ausgenutzt wird.

Ein optimales Verfahren muß daher zwischen diesen beiden Extremen abwägen. Auf der einen Seite müssen Objekte repliziert gespeichert werden, damit diese im lokalen Hauptspeicher schnell zugreifbar sind, aber auf der anderen Seite muß auch berücksichtigt werden, daß nicht zu viele Replikate entstehen, damit der Vorteil von nichtlokalen Hauptspeicherzugriffen – verglichen mit Plattenzugriffen – ausgenutzt werden kann. Die Bestimmung des optimalen Replikationsgrads ist dabei abhängig von den Zugriffskosten auf die verschiedenen Stufen der Speicherhierarchie. Da diese Zugriffskosten von den aktuellen Auslastungen des Netzwerks und der Festplatte abhängen, muß ein optimales Verfahren für die Verwaltung des aggregierten Hauptspeichers adaptiv auf Schwankungen reagieren können. Ein weiteres Problem kann auftreten, wenn die Daten oder die Last ungleichmäßig über die Rechner verteilt ist. Um auch in dieser Situation eine optimale Leistungsfähigkeit des Systems zu erreichen, muß durch eine dynamische Datenmigration eine geschickte Verteilung erreicht werden, so daß eine Überlastung einzelner Rechner vermieden werden kann.

Fassen wir diese Beobachtungen zusammen, so erkennen wir, daß ein optimales Verfahren für die Verwaltung des aggregierten Hauptspeichers die folgenden Anforderungen erfüllen muß:

- eine objektbezogene Replikationskontrolle in Abhängigkeit vom Systemzustand,
- eine adaptive Anpassung an Laständerungen und
- eine Vermeidung von Engpäßen, die durch ungleichmäßige Verteilung der Daten und der Last entstehen können.

Diese Anforderungen haben uns als Entwurfsprinzipien für die Herleitung einer neuen adaptiven Heuristik für verteiltes Caching gedient. Ausgehend von einem detaillierten quantitativen Modell leiten wir Kostenformeln für den Wert eines Objekts her. Diese Kostenformeln berücksichtigen neben den Zugriffskosten auf die verschiedenen Stufen der Speicherhierarchie auch das Referenzverhalten und den Replikationsgrad des betreffenden Objekts. Dadurch ist es möglich, für jedes Objekt getrennt zu entscheiden, ob und in wievielen Kopien es in dem aggregierten Cache gehalten werden soll. Um adaptiv auf Laständerungen reagieren zu können, muß die Auswertung der Kostenformeln online erfolgen. Dazu ist es notwendig, die Komplexität für die Berechnung der Kostenformeln zu reduzieren und die Eingabegrößen für die Kostenformeln während des laufenden Betriebs zu sammeln. Während wir eine Reduktion des Overheads durch eine Reihe von Vereinfachungen und Approximationen erreichen, leiten wir zur Berechnung und Verteilung der notwendigen Informationen speziell angepaßte Protokolle her. Eine Vermeidung von Lastungleichgewichten erzielen wir durch eine Lastverteilungsheuristik, die auf dem gleichen Kostenmodell wie die Replikationskontrolle basiert. Diese Heuristik führt im Hintergrund Objektmigrationen von den stark belasteten zu den unterlasteten Rechnern durch und erreicht dadurch eine Lastbalancierung auf den Systemressourcen.

1.3 Verwandte Arbeiten

1.3.1 Verteiltes Paging durch das Betriebssystem

Eine sehr einfache Methode, die Leistung eines verteilten Systems durch die Benutzung des insgesamt vorhandenen Speichers zu verbessern, wird in [CoGr90] beschrieben. Bei dieser Methode werden zusätzlich zu den im Netzwerk vorhandenen Rechnern und einem File-Server sogenannte Hauptspeicher-Server hinzugefügt. Diese Server sind jeweils mit einem großen Hauptspeicherbereich ausgestattet, der von anderen Rechnern zur Auslagerung von Seiten benutzt werden kann. Durch das Verschieben in den Hauptspeicher anderer Rechner können Auslagerungen auf langsamere Festplatten vermieden werden, und daher wird eine Leistungssteigerung erzielt. Zur besseren Ausnutzung des Server-Speichers ist zwar ein gleichzeitiges und gemeinsames Benutzen einer Seite von verschiedenen Prozessen vorgesehen, jedoch erfolgt keine Koordination zwischen den verschiedenen Caches. Ein weiterer Nachteil dieses Ansatzes ist, daß ein dedizierter Speicher-Server genutzt wird, der leicht zu einem Systemengpaß werden kann.

Analog zu den Methoden, die versuchen Nutzen aus der aggregierten CPU-Leistung eines Rechnernetzes zu ziehen (z.B. [LLM88]), wird in [FeZa91, MDP95] der aggregierte Hauptspeicher eines Rechnernetzes ausgenutzt. An Stelle einer statischen Partitionierung in normale Rechner und Speicher-Server – wie in [CoGr90] – erfolgt dabei eine dynamische Aufteilung in *aktive* und *untätige* Rechner. Aktive Rechner können nun auf den untätigen Rechnern Speicherplatz allozieren und, anstatt verdrängte Seiten auf Platte auszulagern, können diese in den Speicher der untätigen Rechner transferiert werden. Ähnlich wie bei [LLM88] tritt jedoch auch hier das Problem auf, daß entschieden werden muß, welche Rechner aktiv und welche untätig sind. Ein weiteres Problem betrifft den Wechsel des Status eines Rechners. So müssen bei einem Übergang von *untätig* nach *aktiv* zuerst alle Seiten, die von anderen Rechnern stammen, entweder auf Platte ausgelagert oder aber in den Speicher von anderen untätigen Rechnern transferiert werden. Da darüber

hinaus die lokalen Prozesse auch noch die von ihnen benötigten Seiten wieder in den Speicher einlagern müssen, kann dies zu einer sehr großen und damit für die Benutzer unerwünschten Latenzzeit führen. Eine weitere entscheidende Einschränkung dieses Verfahrens ist, daß weder *Sharing*, d.h. ein gemeinsames Nutzen einer Seite durch verschiedene Prozesse, möglich ist, noch eine Koordination der Cache-Inhalte durchgeführt wird.

1.3.2 Verteiltes Caching in Client-Server-Datenbanksysteme, Web-Diensten und Filesystemen

Verteiltes Caching in Client-Server-Datenbanksystemen

Auch in Client-Server Datenbanksystemen wird verteiltes Caching zur Steigerung der Leistungsfähigkeit eingesetzt. In [FCL92, Fran96] werden Methoden vorgestellt, die ein bereits vorhandenes Kohärenzprotokoll erweitern. Erhält ein Server von einem Client eine Anfrage nach einer Seite, so überprüft der Server zuerst ob ein anderer Client diese Seite im lokalen Cache hält, bevor er die Seite von Platte liest. Existiert ein solcher Client, so wird die Anfrage an diesen Rechner weitergereicht, und dieser sendet die Seite direkt an den anfragenden Client. Neben dieser Erweiterung werden noch einfache Methoden vorgestellt, die das Ziel haben, den Anteil der verschiedenen Seiten im aggregierten Cache zu erhöhen. So kann der Server jeweils seine Kopie einer Seite als unwichtig markieren (*Hate-Hint*), indem er sie an das niedrig priorisierte Ende seiner LRU-Ersetzungsstruktur hängt, nachdem er diese Seite an einen Client gesendet hat. Dies soll verhindern, daß eine Seite sowohl im Cache des Clients als auch im Cache des Servers vorkommt. Eine Vermeidung der Replikation von Seiten in den verschiedenen Client-Caches wird jedoch nicht durchgeführt. Auf dem Client kann man andererseits verhindern, daß eine Seite komplett aus dem aggregierten Speicher verdrängt wird, indem man die jeweils letzte Kopie einer Seite im Gesamtsystem an den Server zurücksendet und dort in den Cache einfügt. Während experimentell nachgewiesen wurde, daß das Weitersenden von Anfragen stets zu Verbesserungen führt, hat sich auch gezeigt, daß der Einsatz der einfachen Heuristiken von den Lastparametern abhängig gemacht werden sollte. Unklar ist jedoch, wie das System die aktuelle Lastsituation feststellen und automatisch die korrekte Heuristik wählen kann. Ein weiterer Nachteil besteht darin, daß eine strikte Unterscheidung zwischen den Clients und dem Server existiert. Dies kann dazu führen, daß das System nicht skalierbar ist, da der Server zu einem Engpaß wird.

Eine Heuristik zur Koordination der Client-Caches wird in [DWAP94] vorgestellt. Dieser Algorithmus, der auch *N-Chance*-Algorithmus genannt wird, versucht ebenfalls die Anzahl verschiedenen Seiten im aggregierten Cache hoch zu halten, wobei im Gegensatz zu [FCL92] jedoch auch die Replikation von Seiten in den verschiedenen Client-Caches vermieden wird. Jedesmal, wenn ein Objekt, das nur noch einmal im aggregierten Cache vorhanden ist, aus dem Cache entfernt werden soll, wird ein Zielrechner zufällig ausgewählt, an den die Seite gesendet wird. Auf diesem Rechner wird die Seite dann in den Cache und in die Ersetzungsstruktur so eingefügt, als ob ein aktueller Zugriff auf diese Seite erfolgt wäre. Hat eine Seite eine vordefinierte Anzahl solcher Migrationen durchgeführt, ohne daß ein wirklicher Zugriff auf diese Seite erfolgt ist, wird die Seite bei der nächsten lokalen Cache-Ersetzung aus dem aggregierten Cache entfernt. Ziel dieser Methode ist, daß Seiten von stark belasteten Rechnern auf nicht so stark belastete Clients ausgela-

gert werden. Nachteilig ist jedoch, daß bei der Migration der Objekte die Zugriffshäufigkeit und damit der Wert der Objekte nicht berücksichtigt wird. Dies führt – zusammen mit der zufälligen Bestimmung eines Zielrechners – zu einer Bandbreitenverschwendung des Netzwerks. Des weiteren kann diese zufällige Bestimmung auch zu einer zusätzlichen Belastung bereits ausgelasteter Rechner führen, da mit einer gewissen Wahrscheinlichkeit auch diese Rechner als Ziel einer Migration ausgewählt werden.

In [FMP*95] wird eine noch stärkere Kooperation der vorhandenen Rechner eingeführt. Jeder Rechner teilt nun seinen lokalen Cache in einen lokalen und einen globalen Bereich auf. Die Größen der Bereiche werden dynamisch und in Abhängigkeit von der aktuellen Last bestimmt. Im lokalen Bereich werden jeweils die Seiten gespeichert, die lokal die größte Zugriffshäufigkeit besitzen, während im globalen Bereich Seiten gespeichert sind, die, lokal unwichtig, jedoch wegen ihrer häufigen Referenz von anderen Rechnern im aggregierten Cache gehalten werden sollten. Innerhalb eines Rundenmodells werden jeweils Statistiken über das Alter der Seiten in den Caches der verschiedenen Rechner gesammelt und an einen Koordinator gesendet. Basierend auf dieser Information kann der Koordinator einen Schwellwert für das Alter der Objekte bestimmen, die für die nächsten M Schritte im aggregierten Cache gehalten werden sollen. Die Größe von M ist ein Tuning-Parameter, der wiederum die Länge einer Runde bestimmt. Zusätzlich zu dem Schwellwert berechnet der Koordinator für jeden Rechner ein Gewicht, das dem Anteil der Seiten mit einem größeren Alter als dem berechneten Schwellwert entspricht. Nachdem diese Informationen wieder an alle Rechner verteilt sind, kann bei einer Cache-Ersetzung jeweils lokal entschieden werden, ob die entsprechende Seite direkt gelöscht (Alter > Schwellwert) oder aber an einen anderen Rechner gesendet werden soll (Alter $\leq$ Schwellwert). Im Gegensatz zu [DWAP94] wird hier der Zielrechner nicht zufällig ausgewählt, sondern es können mit Hilfe der Gewichte weniger belastete Rechner bevorzugt werden. Dies führt zwar zu einer geringeren Netzwerkbelastung, jedoch wird nach wie vor die Zugriffshäufigkeit der Objekte bei einer Migration nicht berücksichtigt. Da bei einer Cache-Ersetzung immer nur solche Seiten aus dem Cache entfernt werden dürfen, deren Alter größer als der berechnete Schwellwert ist, kann dieser Verlust an Autonomie eines einzelnen Rechners zu beträchtlichen Verzögerungen führen. Soll z.B. eine Seite auf einem Rechner ersetzt werden, der selbst keine Seiten mit entsprechend großem Alter besitzt, so müssen Seitenmigrationen im kritischen Pfad der Ersetzung erfolgen.

In [SaHa96] werden die Methoden aus [FMP*95], die für das Finden eines Objekts, für die Vermeidung von Replikationen und für die Bestimmung des Zielrechners bei einer Migration zuständig sind, durch entsprechende approximative Methoden ersetzt. Dies soll dazu führen, daß trotz eines wesentlich geringeren Overheads eine vergleichbar gute Performance erzielt wird. Zur Lokalisierung von Seiten im aggregierten Cache werden sogenannte *Hinweise* benutzt. Hinweise werden jeweils lokal verwaltet und bestimmen den Rechner, der die gesuchte Seite mit großer Wahrscheinlichkeit in seinem Cache hält. Mit einer gewissen Restwahrscheinlichkeit, die abhängig von der aktuellen Last ist, befindet sich die Seite jedoch nicht mehr dort, und ein *Forwarding*-Mechanismus muß benutzt werden, um die Seite zu lokalisieren. Zur Erhöhung der Anzahl der Seiten im aggregierten Cache wird ebenfalls eine approximative Methode angewendet. Dazu wird das Konzept der *Master-Seite* eingeführt. Als Master-Seite wird diejenige Kopie bezeichnet, die von Platte in den aggregierten Hauptspeicher geladen wird. Da bei Zugriffen jeweils die im Cache vorhandenen Kopien gegenüber den Plattenkopien bevorzugt werden, existiert für jede

Seite immer nur eine Master-Seite im aggregierten Cache. Wird nun durch die Ersetzungsstrategie eine Master-Seite als Opfer ausgewählt, so wird diese Seite an einen anderen Rechner gesendet und in dessen Cache eingefügt. Zur Bestimmung des Zielrechners einer solchen Migration wird wiederum ein approximativer Algorithmus benutzt. Jeder Rechner verwaltet dazu lokal eine Liste mit dem Alter der ältesten Seite eines jeden Rechners, die mit geringem Overhead durch einen *Piggybacking*-Mechanismus aktualisiert wird. Als Zielrechner wird nun jeweils der Rechner bestimmt, der die älteste Seite besitzt, und die migrierte Seite ersetzt auf dem Zielrechner diese älteste Seite. Zwar reduzieren diese Approximationen den Overhead für die Verwaltung der Informationen, jedoch können sie gleichzeitig durch die stark vereinfachenden Annahmen Zusatzaufwand verursachen. So werden z.B. Seiten migriert, obwohl noch mehrere Kopien im aggregierten Cache existieren.

Der Schwerpunkt von [VJV97] liegt in der Bestimmung eines guten Zielrechners für eine Seitenmigration. Im Gegensatz zu [FMP*95] und [SaHa96] wird hier der Rechner nicht nur nach dem Alter seiner Seiten ausgewählt, sondern es wird auch die Last des Rechners berücksichtigt. Um einen Rechners entsprechend des Alters seiner Seiten zu charakterisieren können die Methoden aus [FMP*95] oder [SaHa96] benutzt werden, und um die Last der Rechner festzulegen, wird die Anzahl der lokalen Seiten bestimmt, die entfernt werden können, ohne daß “zu große” lokale Leistungseinbußen auftreten. Durch die gewichtete Betrachtung dieser beiden Größen wird ein Maß für die Eignung eines Rechners als Zielrechner hergeleitet. Bei diesem Ansatz bleibt jedoch völlig unklar, wie der Parameter zur Charakterisierung der Last gewählt werden soll und wie die Anzahl der Seiten, die ohne Leistungseinbuße verdrängt werden können, bestimmt werden kann.

Der Einfluß der Datenplazierung auf verschiedene Heuristiken für verteiltes Caching wird in [VLN95] untersucht. In dieser Arbeit wird angenommen, daß jeder Rechner innerhalb eines lokalen Netzwerks eine eigene Festplatte besitzt. Die Daten können nun entweder *clustered*, d.h. alle Seiten eines Objekts befinden sich auf der gleichen Platte, oder *unclustered*, d.h. die Seiten eines Objekts befinden sich auf Platten unterschiedlicher Rechner, gespeichert sein. In einer Simulationsstudie hat sich herausgestellt, daß bereits sehr einfache Strategien (vgl. [FCL92]) bei einer *Unclustered*-Allokation zu einer sehr guten Ausnutzung von verteiltem Speicher führen. Dies kommt daher, daß sich die einzelnen Seiten der Objekte, auf die sehr häufig zugegriffen wird, automatisch in den Caches von unterlasteten Rechnern ansammeln, da diese Seiten beim Laden von den Platten der verschiedenen Rechner in die entsprechenden Caches eingefügt werden. Nachteilig an einer solchen Allokation ist jedoch die Reduzierung des Durchsatzes, die durch die zusätzlichen Suchvorgänge beim Lesen der verschiedenen Seiten eines Objekts auftritt. Das gleichmäßige Verteilen der Seiten über alle Rechner führt darüber hinaus nicht notwendigerweise zu einer guten Balancierung, beispielsweise wenn die Rechner unterschiedlich große Caches besitzen. Schließlich leidet auch die Verfügbarkeit der Daten unter der Verteilung, da bereits der Ausfall eines einzigen Rechners den Zugriff auf ein beliebiges Objekt unmöglich macht.

Eine Erweiterung von [VLN95] mit einem besseren Ersetzungsalgorithmus wird in [VNL98] vorgestellt. Hier werden auf jedem Rechner mehrere lokale LRU-Stacks verwaltet, die jeweils Objekte mit unterschiedlichen Eigenschaften enthalten. Diese Eigenschaften umfassen den Replikationsgrad der Objekte und eventuelle *Hate-Hints* [FCL92]. Bei einer Cache-Ersetzung werden diese Stacks in einer fest vorgegebenen Reihenfolge durchsucht, bis ein Opfer gefunden wird. Nachteilig an diesem Verfahren ist, daß durch die fest vorgegebene Reihenfolge nur eine

Maximierung der im aggregierten Cache vorhandenen Seiten erfolgt, ohne daß eine wirkliche Replikationskontrolle in Abhängigkeit von der aktuellen Last durchgeführt wird.

Verteiltes Caching in Web-Diensten

Ein Bereich, in dem in wachsendem Maße die Leistungsfähigkeit lokaler Rechnernetze ausgenutzt wird, betrifft die Implementierung skalierbarer HTTP-Server [KBM94, DKMT96, CYD97, GoHu97, VLN97]. Während sich jedoch die meisten Arbeiten nur mit der Balancierung der Last durch eine geeignete Verteilung der ankommenden Aufträge an die verschiedenen Rechner beschäftigen, wird die koordinierte Ausnutzung des zur Verfügung stehenden verteilten Caches nur wenig beachtet.

Eine Ausnahme hiervon bildet der Ansatz von [Venk96, VLN97]. Bei dieser Methode erfolgt eine explizite Replikationskontrolle, indem in Abhängigkeit von der aktuellen Last und der Zugriffshäufigkeit auf eine Seite entschieden wird, ob diese Seite repliziert, einfach oder gar nicht in dem aggregierten Cache gehalten werden soll. Auf jedem Rechner werden dazu zwei getrennte LRU-Ersetzungsstrukturen verwaltet: eine für die lokalen Seiten, die noch mindestens in einem anderen Cache vorkommen, und eine für die Seiten, die nur noch in dem lokalen Cache vorhanden sind. Zur Bestimmung und kostengünstigen Verbreitung der für diese Unterscheidung notwendigen Information wird das benutzte Kohärenzprotokoll erweitert. Bei der Bestimmung eines Ersetzungsopfers müssen nur die jeweils ältesten Seiten in den beiden lokalen LRU-Strukturen miteinander verglichen werden. Als Opfer wird dann diejenige ausgewählt, deren erwartete Zugriffskosten am geringsten sind. Diese erwarteten Kosten werden durch das Produkt aus der Zugriffswahrscheinlichkeit und den Kosten für das Lesen dieser Seite berechnet. Zur Approximation der Zugriffswahrscheinlichkeit wird der Kehrwert der Zeit seit dem letzten lokalen Zugriff betrachtet, und zur Bestimmung der Lesekosten werden die gemessenen Zeiten von bereits durchgeführten Zugriffen auf die entsprechende Stufe der Speicherhierarchie benutzt. Nachteilig an dieser Heuristik ist, daß hier keine explizite Methode zur Ausnutzung des Speichers von leicht belasteten Rechnern vorgesehen ist. So wird das lokal bestimmte Opfer, das im allgemeinen nicht mit dem global günstigsten Ersetzungsopfer übereinstimmt, nicht zu einem anderen Rechner migriert, sondern es wird immer lokal gelöscht. Diese Vorgehensweise verhindert auch eine Lastbalancierung zwischen den Rechnern. Während dies in der vorgestellten Anwendung – nämlich der Beschleunigung eines HTTP-Servers – noch tolerabel ist, da durch einen vorgeschalteten Router ankommende Anfragen gleichmäßig auf alle Rechner verteilt werden, kann dies in einer allgemeineren Umgebung zu sehr starken Leistungseinbußen führen.

Neben der Beschleunigung eines einzelnen HTTP-Servers können durch den Einsatz sogenannter Proxy-Server auch Objekte in den Caches von Rechnern aus unterschiedlichen (Teil-)Netzwerken des Web gehalten werden. Durch intelligente Ersetzungsstrategien soll auch hier das Caching zu einer Reduktion der Zugriffszeit führen [LuAl94, ASA95*, Best95, Best96, CDN*95, GwSe95]. Im Gegensatz zu den von uns betrachteten LANs (*Local Area Network*) sind hier jedoch die am Caching beteiligten Rechner durch ein WAN (*Wide Area Network*) miteinander verbunden. Durch die sehr viel größere Verzögerung bei der Übertragung in einem WAN, die sowohl durch die höhere Auslastung als auch durch die größere Entfernung verursacht wird, verändert sich die Zielfunktion für eine Optimierung der Cache-Inhalte entscheidend. Während wir bei ver-

teiltem Caching im lokalen Netzwerk versuchen, die Latenzzeit eines Festplattenzugriffes durch das Caching des Objekts in dem aggregierten Hauptspeicher zu umgehen, ist bei einem WAN das Netzwerk in der Regel der Engpaß [MaSe97]. Daher steht bei derartigen Caching-Strategien die Verringerung der Netzauslastung im Vordergrund. Um dies zu erreichen, werden unter Umständen selbst Einschränkungen bezüglich der Kohärenz der in den verschiedenen Caches vorhandenen Daten in Kauf genommen [GwSe96].

Verteiltes Caching in Filesystemen

Auch im Umfeld von Netzwerk-Filesystemen wird Client-Caching zur Verbesserung der Performance benutzt [HKM*88, NWO88, Sun88, ADN*95]. Obwohl Client-Rechner in allen diesen Verfahren Daten in ihren lokalen Caches halten können, wird der aggregierte Cache bei den meisten Verfahren nicht effizient genutzt. So können bei *AFS* [HKM*88], *Sprite* [NWO88] und *NFS* [Sun88] Dateien nur aus dem eigenen Cache, aus dem Server-Cache oder von der Server-Platte gelesen werden; ein Zugriff auf andere Client-Caches ist jedoch nicht möglich. Ein wesentlicher Unterschied zwischen diesen Verfahren betrifft die Protokolle, die zur Sicherstellung der Kohärenz in den Client-Caches benutzt werden. Während in *NFS* sogar Inkonsistenzen durch eine *Timer*-basierte Kohärenzkontrolle auftreten können, werden diese bei *Sprite* durch ein Nachfragen beim Server vermieden. Dies führt jedoch dazu, daß selbst bei einem lokalen Cache-Treffer ein Nachrichtenaustausch notwendig ist. Eine völlig lokale Abarbeitung von lokalen Cache-Treffern wird in *AFS* durch eine Vergabe von lokalen Sperren zusammen mit einem Server-initiierten Invalidierungsprotokoll erreicht. Als einziges Netzwerk-Filesystem erlaubt *xFs* [ADN*95] ein Weiterleiten von Zugriffen auf die Caches von anderen Clients. Zur Koordination der Caches werden jedoch auch hier keine eigenen Methoden hergeleitet, sondern es wird auf bereits bekannte Heuristiken [FCL92, DWAP94] zurückgegriffen.

1.3.3 Verteiltes Caching als Optimierungsproblem

Offline-Optimierung für statische Lasten

Im Gegensatz zu den bisherigen Ansätzen wird in [LYW91, LWY93] ein Kostenmodell definiert, aus dem eine Methode zur Berechnung einer optimalen Cache-Allokation abgeleitet wird. Wegen der großen Komplexität dieser Methode wird zusätzlich eine Heuristik beschrieben, die durch eine Koordination der Cache-Inhalte eine vergleichbare Leistungsfähigkeit erzielen soll. Dies wird durch eine spezielle Ersetzungsstrategie erreicht, bei der der Wert einer Seite davon abhängt, wieviele Kopien von dieser Seite im aggregierten Cache vorhanden sind. Existieren mehrere Kopien, so reduziert sich der lokale Wert, da im Falle einer lokalen Ersetzung diese Seite immer noch aus dem Cache eines anderen Rechners gelesen werden kann. Existiert hingegen nur noch eine einzige Kopie, so muß nach deren Löschen die Seite durch einen teureren Plattenzugriff wieder in den Speicher geladen werden. Während dieses Verfahren innerhalb des betrachteten Modells gute Ergebnisse liefert, muß doch kritisiert werden, daß die Zugriffskosten auf die verschiedenen Stufen der Speicherhierarchie als konstant angenommen werden und daher eine Berücksichtigung von dynamischen Lastvariationen unmöglich ist. Weiterhin liegt sowohl dem exakten als auch dem heuristischen Verfahren die unrealistische Annahme zugrunde, daß die Zugriffshäufigkeiten im vorhinein bekannt und konstant sind.

Online-Optimierung für dynamische Lasten

Auch im Bereich der Online-Algorithmen existiert eine Reihe von theoretischen Ansätzen für die Migration und Replikation von Daten in Netzwerken [BFR92, ABF93a, ABF93b, HuWo94]. Bei diesen Ansätzen werden jedoch gravierende Einschränkungen bezüglich der Modellannahmen gemacht. So wird in [ABF93b] zwar ein Online-Algorithmus für die Replikationskontrolle in einem verteilten Cache vorgestellt, jedoch liegt dem Algorithmus die Annahme zugrunde, daß der komplette Datenbestand permanent im aggregierten Cache gehalten wird. Dadurch reduziert sich die Tiefe der Speicherhierarchie, da Zugriffe auf Festplatte in diesem Modell nicht vorkommen. In [HuWo94] wird für jedes Objekt individuell eine Replikationskontrolle in Abhängigkeit von der Anzahl der Lese- und Schreibzugriffe mit dem Ziel durchgeführt, die Kommunikationskosten zu minimieren. Da die Kommunikation zwischen den Rechnern im Vordergrund steht, wird nur zwischen lokalen und nichtlokalen Zugriffen unterschieden, während der Unterschied zwischen Hauptspeicher und Festplatte nicht betrachtet wird. Eine weitere starke Einschränkung betrifft die als unendlich groß angenommene lokale Speicherkapazität. Dies bewirkt, daß nur die Schreibzugriffe – und nicht die lokalen Speicherbeschränkungen – einer kompletten Replikation über alle Rechner entgegenwirken.

1.3.4 Einordung des eigenen Ansatzes

Unser neuer Ansatz für verteiltes Caching, der in seinen Grundzügen auch in [SiWe97] beschrieben wird, ist am nächsten mit dem zeitgleich entstandenen Verfahren von [VLN97] verwandt. Beide Verfahren führen in Abhängigkeit von der aktuellen Last eine objektbezogene Replikationskontrolle durch. Dadurch grenzen sich beide Ansätze deutlich von den anderen Heuristiken aus den Abschnitten 1.3.1 und 1.3.2 ab, die feste und lastunabhängige Optimierungsziele benutzen. Während jedoch das Verfahren von [VLN97] keine Ungleichgewichte bezüglich der Last und der Datenplazierung ausgleichen kann und daher strikte Vorgaben bezüglich der Datenplazierung fordert, ermöglicht unser Verfahren durch die Benutzung eines gemeinsamen Kostenmodells für die Replikationskontrolle und für die Lastbalancierung eine integrierte Betrachtung dieser beiden Probleme. Durch die Berücksichtigung einer mächtigeren Metrik erlaubt unser Verfahren darüber hinaus sowohl variabel große Objekte als auch heterogene Hauptspeichergrößen der Rechner.

Obwohl unser Kostenmodell eng mit den entsprechenden Modellen in [LYW91, LWY93] verwandt ist, geht unsere Heuristik doch weit über die dort vorgestellten Methoden hinaus, da wir – im Gegensatz zu der statischen Optimierung – das Kostenmodell als Ausgangspunkt für die Herleitung einer adaptiven Online-Heuristik benutzen. Auch gegenüber den sehr theoretischen Online-Heuristiken aus Abschnitt 1.3.3 grenzt sich unser Ansatz deutlich ab, da unsere Heuristik nicht auf die dort beschriebenen unrealistischen Annahmen angewiesen ist.

1.4 Aufbau der Arbeit

In *Kapitel 2* werden wir das Problem des verteilten Cachings als ein mathematisches Optimierungsproblem definieren. Hierbei beschränken wir uns zunächst auf ein statisches Modell, bei

dem die Zugriffszeiten auf die verschiedenen Stufen der Speicherhierarchie und die Lastparameter konstant sind. Für das Problem des verteilten Cachings leiten wir innerhalb dieses Modells sowohl exakte Algorithmen als auch einfache Heuristiken her, die wir bezüglich ihrer Güte und ihrer Laufzeit analytisch vergleichen.

In *Kapitel 3* verallgemeinern wir das Problem, indem wir nun die Berechnungen online durchführen und dynamische Variationen der Last und der Zugriffszeiten (z.B. durch Warteschlangeneffekte) berücksichtigen. Da wir in einer dynamischen Umgebung nicht mehr davon ausgehen können, daß die Informationen über die lokalen Belastungen global bekannt sind, leiten wir in diesem Kapitel zunächst Protokolle her, mit deren Hilfe die notwendigen Informationen bestimmt und zwischen den verschiedenen Rechnern verteilt werden können. Anschließend beschreiben wir mögliche Online-Erweiterungen der jeweils korrespondierenden einfachen Heuristiken aus Kapitel 2. Wegen der sehr hohen Komplexität der exakten Algorithmen können wir diese nicht in einer dynamischen Umgebung benutzen. Daher leiten wir aus einem Kostenmodell eine neue Heuristik ab, die bei einem geringen Overhead das Verhalten der optimalen Algorithmen dynamisch approximiert. Um ungleichmäßige Auslastungen innerhalb des NOWs zu vermeiden, erweitern wir die vorgestellten Heuristiken – soweit dies möglich ist – um eine Lastverteilungsmethode.

Eine detaillierte Beschreibung des Prototyps folgt in *Kapitel 4*. Zuerst beschreiben wir die für die verschiedenen Heuristiken notwendigen Softwarekomponenten. Da wir für die Durchführung von Experimenten auf keine entsprechende Hardware zurückgreifen konnten, stellen wir anschließend die Modellierung und Implementierung der Hardwarekomponenten innerhalb einer Simulationsumgebung vor. Zur systematischen Evaluation benutzen wir ein synthetisches Last- und Datenmodell.

In *Kapitel 5* präsentieren wir die Resultate einer umfangreichen Simulationsstudie der verschiedenen Heuristiken auf der Basis des entwickelten Prototyps. Dabei variieren wir systematisch die Last-, Daten- und System-Parameter. Außerdem untersuchen wir die Adaptivität der Heuristiken bezüglich sich dynamisch ändernder Lasten.

In *Kapitel 6* betrachten wir eine Erweiterung des verteilten Caching-Problems im Hinblick auf vorgegebene Antwortzeitziele für verschiedene Lastklassen. Nachdem wir die Notwendigkeit eines solchen zielorientierten Ansatzes motiviert und verwandte Arbeiten vorgestellt haben, leiten wir einen Algorithmus zur Partitionierung des aggregierten Caches entsprechend der vorgegebenen Ziele her. Anschließend beschreiben wir eine mögliche Einbettung dieses Verfahrens in das Prototypsystem und die Verallgemeinerungen, die notwendig sind, um unsere kostenbasierte Heuristik auch für die Verwaltung der klassenspezifischen Caches benutzen zu können.

Abschließend geben wir in *Kapitel 7* einen Ausblick auf Verallgemeinerungen und Erweiterungsmöglichkeiten.

Kapitel 2

Verteiltes Caching als statisches Optimierungsproblem

In diesem Kapitel untersuchen wir das Problem des verteilten Cachings als statisches Optimierungsproblem. Dazu führen wir in Abschnitt 2.1 zunächst eine formale Definition ein. Ausgehend von dieser Definition leiten wir in Abschnitt 2.2 Algorithmen her, die dieses Problem exakt lösen, und in Abschnitt 2.3 stellen wir einfache Heuristiken vor. Diese verschiedenen Methoden vergleichen wir analytisch in Abschnitt 2.4, und abschließend fassen wir die Ergebnisse dieses Kapitels in Abschnitt 2.5 zusammen.

2.1 Eine formale Definition

Wir betrachten ein Netzwerk, das aus N Rechnern besteht. Jeder Rechner i stellt einen Hauptspeicherbereich der Größe B_i für das Speichern von Objekten zur Verfügung. Diesen Bereich bezeichnen wir im folgenden als *Cache*. Zusätzlich besitzt jeder Rechner eine eigene Festplatte der Größe D_i. Einen Rechner ohne Festplatte können wir modellieren, indem wir D_i gleich Null setzen. Der komplette Datenbestand besteht aus M Objekten, und die Größe des Objekts p ist durch den p-ten Eintrag des Vektors $s = (s_1,s_2,...,s_M)$ gegeben. Um das Problem des verteilten Cachings von dem Problem der Datenplazierung zu trennen, nehmen wir weiterhin an, daß bereits eine feste Zuordnung der Objekte zu den lokalen Platten existiert, unter der Nebenbedingung, daß die Gesamtgröße aller auf einer Platte gespeicherten Objekte kleiner sein muß als die Kapazität der Festplatte. Diese Zuordnung, die prinzipiell auch die Replikation von Objekten über mehrere Festplatten beinhalten kann, beschreiben wir durch die Matrix Y in der folgenden Weise:

$$Y_{ip} = \begin{cases} 1 & \text{falls Objekt } p \text{ auf der Platte von Rechner } i \text{ gespeichert ist} \\ 0 & \text{sonst} \end{cases}$$

Analog zur Plattenplazierung beschreiben wir eine Cache-Plazierung durch die Matrix X:

$$X_{ip} = \begin{cases} 1 & \text{falls Objekt } p \text{ im Cache von Rechner } i \text{ liegt} \\ 0 & \text{sonst} \end{cases}$$

Zur einfachen Beschreibung der Zugriffscharakteristik der Rechner auf die Objekte übernehmen wir aus [CABK88] die Begriffe der Hitze und der Temperatur.

Definition 2.1: (Hitze, Temperatur)

Die **lokale Hitze** eines Objekts p auf dem Rechner i ist definiert durch die Zugriffshäufigkeit von Rechner i auf das Objekt p über einen hinreichend großen Beobachtungszeitraum. Unter der **globalen Hitze** eines Objekts p verstehen wir die Summe der lokalen Hitzen von p über alle Rechner.

Die **lokale Temperatur** eines Objekts p ist definiert durch den Quotient aus seiner lokalen Hitze und seiner Größe s_p. Entsprechend definieren wir die **globale Temperatur** eines Objekts p durch den Quotient aus der globalen Hitze und der Objektgröße s_p.

■

In diesem Kapitel nehmen wir an, daß die Matrix f der lokalen Hitzen bekannt ist, wobei der Eintrag f_{ip} die lokale Hitze des Objekts p auf dem Rechner i enthält. Des weiteren gehen wir in diesem Kapitel davon aus, daß die Zugriffskosten auf eine bestimmte Stufe der Speicherhierarchie pro Größeneinheit konstant und von den beteiligten Rechnern unabhängig sind. Für einen lokalen Speicherzugriff nehmen wir die Kosten c_{lc}, für einen nichtlokalen Speicherzugriff c_{rc}, für einen lokalen Plattenzugriff c_{ld} und für einen nichtlokalen Plattenzugriff c_{rd} an. Die Werte dieser Größen sind ebenfalls a priori bekannt.

Mit Hilfe dieser Bezeichnungen sind wir nun in der Lage, das Problem des verteilten Cachings formal zu definieren.

Definition 2.2: (Problem des verteilten Cachings)

Sei *cost_fkt* eine noch nicht näher spezifizierte Kostenfunktion und c eine $N \times M$ Matrix, deren Einträge folgendermaßen definiert sind:

$$
\begin{aligned}
c_{ip} \quad &= X_{ip} \times c_{lc} && \text{(a)}\\
&+ \left(1 - X_{ip}\right) \times \left[1 - \prod_{k=1}^{N}\left(1 - X_{kp}\right)\right] \times c_{rc} && \text{(b)}\\
&+ \prod_{k=1}^{N}\left(1 - X_{kp}\right) \times Y_{ip} \times c_{ld} && \text{(c)}\\
&+ \prod_{k=1}^{N}\left(1 - X_{kp}\right) \times \left(1 - Y_{ip}\right) \times c_{rd}\,. && \text{(d)}
\end{aligned}
$$

Dann besteht das **Problem des verteilten Cachings** darin, die Kosten $C = \text{cost\_fkt}\,(f, c, S)$ unter der Nebenbedingung $\sum_{q=1}^{M} X_{iq} \times s_q \leq B_i$ für alle Rechner i zu minimieren.

■

In der obigen Definition berücksichtigt der Term (a) die Kosten für Zugriffe auf Objekte, die im lokalen Cache gehalten werden, der Term (b) die Kosten für Zugriffe auf Objekte im nichtlokalen Cache, der Term (c) Kosten für lokale Plattenzugriffe und der Term (d) schließlich die Kosten für Zugriffe auf nichtlokale Platten. Die weiter oben definierten Allokationsmatrizen X und Y ermöglichen uns innerhalb der Formel sicherzustellen, daß jeweils nur einer dieser Terme für die Zugriffe auf ein Objekt p von einem Knoten i aus, betrachtet wird.

Bisher sind wir noch nicht näher auf die Berechnungsvorschrift der Kostenfunktion eingegangen. Zwei mögliche Realisierungen sind:

- $\text{cost\_fkt}_1(f, c, S) = \sum_{j=1}^{N} \sum_{q=1}^{M} f_{jq} \times c_{jq} \times s_q$ oder
- $\text{cost\_fkt}_2(f, c, S) = \max_j \sum_{q=1}^{M} f_{jq} \times c_{jq} \times s_q$

Bei der ersten Implementierung werden die Zugriffskosten über alle Rechner und alle Objekte gemittelt. Minimiert man daher diese Funktion, so erhält man eine Allokation, bei der die mittlere Zugriffszeit innerhalb des Systems minimal ist; es sind jedoch keine Aussagen bezüglich der "Fairneß" zwischen den einzelnen Rechnern möglich. Dies kann dazu führen, daß bei einer optimalen Allokation die Kosten der verschiedenen Rechner sehr stark voneinander abweichen.

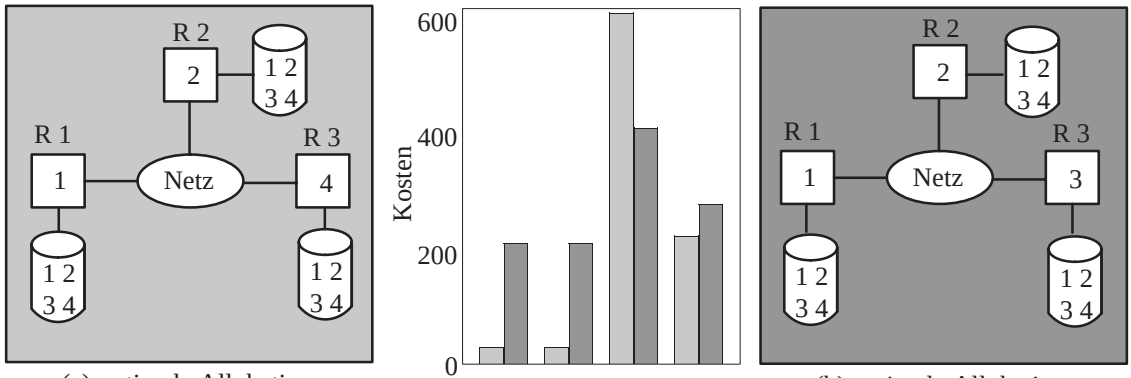
Im Gegensatz dazu geht die zweite Implementierung davon aus, daß die Leistungsfähigkeit des Gesamtsystems nicht besser sein kann als die Leistungsfähigkeit des Rechners mit den höchsten Kosten. Dies bewirkt, daß bei der Minimierung die Varianz zwischen den Rechnern ebenfalls günstig beeinflußt wird. Gleichzeitig schränkt dies jedoch den Suchraum für die optimale Lösung stärker ein, wie wir an dem folgenden Beispiel zeigen werden.

Beispiel 2.1:

Die Zugriffskosten seien nun $c_{lc} = 1$, $c_{rc} = 10$ und $c_{ld} = 100$. Aus Gründen der Vereinfachung nehmen wir an, daß alle Objekte über alle Festplatten repliziert gespeichert sind. Da sich die lokale Platte in der Zugriffsreihenfolge vor den nichtlokalen Platten befindet, bewirkt die Replikation, daß keine Zugriffe auf nichtlokale Platten erfolgen. Daher brauchen wir den Wert von c_{rd} nicht zu spezifizieren. Das betrachtete System bestehe aus 3 Rechnern mit jeweils einer Cache-Kapazität der Größe 1, und die Gesamtheit der Daten umfasse 4 Objekte jeweils der Größe 1. Berechnen wir nun das Optimum bezüglich der oben angegebenen Kostenfunktionen für die Zugriffsverteilung

$$f = \frac{1}{30}\begin{pmatrix} 8 & 0 & 0 \\ 0 & 8 & 0 \\ 0 & 0 & 6 \\ 2 & 2 & 4 \end{pmatrix},$$

so erhalten wir als Optimum für die verschiedenen Kostenfunktionen die in Abbildung 2.3 gezeigten Allokationen.



(a) optimale Allokation für *cost_fkt*$_1$ (b) optimale Allokation für *cost_fkt*$_2$

Abbildung 2.3: Auswirkung der unterschiedlichen Kostenfunktionen auf die optimale Allokation

An der Darstellung der mittleren Kosten für die einzelnen Rechner und der mittleren Gesamtkosten erkennen wir, daß die Varianz zwischen den Rechnern durch die Kostenfunk-

tion cost_fkt2 reduziert wird, die mittlere Gesamtantwortzeit jedoch schlechter wird.

■

Wir werden im folgenden nur noch die Kostenfunktion *cost_fkt1* betrachten, da die Minimierung der Gesamtantwortzeit für viele Anwendungen ein Hauptoptimierungsziel ist. Die Verfahren zur Lösung des Problems des verteilten Cachings, die wir in diesem Kapitel noch vorstellen werden, können jedoch auch analog mit der zweiten Kostenfunktion durchgeführt werden.

Nachdem wir das Problem formal definiert haben, wollen wir nun die Komplexität untersuchen. Durch eine einfache Reduktion des PARTITION-Problems [GaJo79] können wir zeigen, daß das Problem, eine optimale Allokation für verteiltes Caching zu finden, $\mathcal{NP}$-vollständig ist (Anhang B). Hierbei nutzen wir jedoch nur die Komplexität, die aus der variablen Größe der Objekte resultiert, und nicht die des eigentlichen Caching-Problems. Daher könnte man annehmen, daß das Problem leichter wird, wenn wir uns auf konstant große Objekte beschränken.

Obwohl uns kein Beweis für die $\mathcal{NP}$-Vollständigkeit dieses vereinfachten Problems bekannt ist, können wir doch an Hand des Beispiels 2.2 zeigen, daß die folgende, naheliegende Hypothese nicht für alle optimalen Lösungen erfüllt ist.

Hypothese:

Die Menge der Cacheinhalte jeder optimalen Allokation kann als eine Vereinigung von Präfixen der folgenden N+1 Listen beschrieben werden:

- LOCAL$_i$ ist die nach lokaler Hitze f_{ip} auf Rechner i absteigend sortierte Liste der Objekte. Für jeden Rechner existiert eine solche Liste.
- GLOBAL ist die nach globaler Hitze absteigend sortierte Liste der Objekte.

■

Diese Hypothese erscheint sinnvoll, da man erwarten würde, daß ein Objekt im Cache eines Rechners gehalten wird, weil entweder ein einzelner oder die Gesamtheit aller Rechner sehr häufig darauf zugreifen. Bevor wir jedoch diese Hypothese widerlegen, führen wir zur Vereinfachung der Argumentation den Begriff der Trefferrate ein.

Definition 2.3: (Trefferrate)

Die **lokale** (bzw. **globale**) **Cache-Trefferrate** ist der Anteil der Anfragen, die aus dem lokalen bzw. globalen Cache beantwortet werden können, und entsprechend ist die **lokale** (bzw. **globale**) **Plattentrefferrate** der Anteil der Anfragen, die von der lokalen (bzw. globalen) Platte befriedigt werden müssen.

■

Beispiel 2.2:

Wir nehmen wiederum an, daß das verteilte System aus 3 Rechnern mit jeweils der Cache-Kapazität 1 besteht. Die Anzahl der Objekte sei nun 5, jedoch gelte immer noch, daß alle Objekte die Größe 1 haben und daß sie auf allen Festplatten repliziert gespeichert sind. Die lokalen Hitzen seien durch die folgende Matrix gegeben:

$$f = \frac{1}{30 + 9\epsilon} \begin{pmatrix} 8 & 0 & 1+\epsilon \\ 0 & 8 & 1+\epsilon \\ 0 & 0 & 4+\epsilon \\ \epsilon & \epsilon & 4 \\ 2(1+\epsilon) & 2(1+\epsilon) & 0 \end{pmatrix}$$

Besonders zu betonen ist, daß die Einträge der Matrix so konstruiert wurden, daß die Gesamtlast auf jedem Rechner identisch ist und sich nur die Zugriffsmuster unterscheiden. Nehmen wir außerdem an, daß $c_{lc} + c_{ld} > 2c_{rc}$ und daß ϵ entsprechend der Ungleichung

$$\epsilon < \frac{2(c_{rc} - c_{lc})}{c_{ld} - c_{rc}}$$

gewählt wurde, so können wir zeigen, daß die kostengünstigste Allokation die oben beschriebene Hypothese nicht erfüllt.

Unter diesen Annahmen hält die optimale Allokation das Objekt 1 im Cache von Rechner 1, das Objekt 2 im Cache von Rechner 2 und das Objekt 4 im Cache von Rechner 3. Betrachten wir jedoch die Listen LOCAL$_i$ und GLOBAL, so erkennen wir, daß sich diese optimale Lösung nicht als Vereinigung von Präfixen dieser Listen darstellen läßt:

LOKAL$_1$ = 1, 5, 4, {2, 3}
LOKAL$_2$ = 2, 5, 4, {1, 3}
LOKAL$_3$ = 3, 4, {1, 2}, 5
GLOBAL = {1, 2}, 5, 4, 3

(Durch Mengenklammern zusammengefaßte Zahlen bezeichnen Objekte gleicher Hitze, und daher können diese innerhalb der Klammern in beliebiger Reihenfolge auftreten.)

Die besten Allokationen, die die oben beschriebene Hypothese erfüllen, sind:

(1) Objekt 1 auf Rechner 1, Objekt 2 auf Rechner 2 und Objekt 3 auf Rechner 3

(2) Objekt 1 auf Rechner 1, Objekt 2 auf Rechner 2 und Objekt 5 auf Rechner 3

Der Grund für die höheren Kosten der Lösung (1) im Vergleich zur optimalen Lösung besteht darin, daß das Objekt 4 fast genauso heiß ist wie das Objekt 3, jedoch die Rechner 1 und 2 zusätzlich von dem Caching des Objekts 4 auf dem Rechner 3 profitieren können, während auf Objekt 3 von diesen Rechnern aus nicht zugegriffen wird. Dadurch, daß wir Einbußen in der lokalen Trefferrate zugunsten der globalen Trefferrate in Kauf nehmen, verbessert sich in dieser Situation die Gesamtleistung des Systems. Daß diese Überlegung jedoch nicht generell anwendbar ist, zeigt der Vergleich der Lösung (2) mit der optimalen Lösung. Zwar ist nun die globale Trefferrate maximiert, jedoch kann die daraus resultierende Reduktion der Kosten den Anstieg der Kosten durch die wesentlich schlechtere lokale Trefferrate auf Rechner 3 nicht ausgleichen.

■

Obwohl wir nicht bewiesen haben, daß das eingeschränkte verteilte Caching Problem $\mathcal{NP}$-vollständig ist, hat Beispiel 2.2 doch gezeigt, daß weder die beiden Extremlösungen (Maximieren der lokalen bzw. globalen Trefferrate) noch eine einfache Kombination dieser beiden Verfahren zu einer garantiert optimalen Lösung führen.

Optimale Lösungen sowohl für das eingeschränkte als auch für das allgemeine Problem des verteilten Cachings können natürlich mit Standardverfahren der kombinatorischen Optimierung be-

rechnet werden, wobei jedoch der Aufwand im allgemeinen exponentiell in der Problemgröße sein wird. In dem nächsten Abschnitt werden wir die *Branch-and-Bound*-Methodik benutzen, um eine optimale Allokation für ein gegebenes verteiltes Caching-Problem zu finden.

2.2 Exakte Algorithmen

Prinzipiell kann das Problem, eine optimale Allokation für verteiltes Caching zu finden, durch einen Algorithmus gelöst werden, der alle erlaubten Allokationen aufzählt, jeweils die Kosten der aktuellen Allokation bestimmt und sich die Allokation mit den minimalen Kosten merkt. Die Komplexität des Problems macht diese Vorgehensweise jedoch praktisch unmöglich.

Zur einfacheren Abschätzung der Komplexität gehen wir von einer konstanten Objektgröße s aus und nehmen an, daß jeder Rechner die gleiche Cache-Kapazität B besitzt. Daraus folgt, daß jeder Rechner genau $b = \lfloor B/s \rfloor$ Objekte im Cache halten kann. Besteht der gesamte Datenbestand aus M Objekten, so kann jeder Rechner eine beliebige Teilmenge von b oder weniger Objekten auswählen. Da sich jedoch die Kosten jeder Allokation, bei der ein Rechner weniger als b Objekte im Cache hält, durch Hinzunahme beliebiger Objekte verringern, betrachten wir nur Allokationen, bei denen jeder Rechner seinen Cache maximal ausnutzt. Somit erhalten wir $\binom{M}{b}$ verschiedene Kombinationen für jeden Rechner, und da jeder Rechner seine Auswahl unabhängig von allen anderen Rechnern durchführen kann, erhalten wir insgesamt $\binom{M}{b}^N$ mögliche Allokationen.

Wegen dieser großen Komplexität ist eine erschöpfende Suche somit völlig unmöglich, und wir werden daher einen *Branch-and-Bound*-Algorithmus vorstellen, der versucht, den Suchraum – ohne Verlust der Optimalität – zu beschränken.

Bei einem *Branch-and-Bound*-Algorithmus stellt man den Suchraum als Baum dar, dessen innere Knoten Teillösungen und dessen Blätter komplette Lösungen des Problems repräsentieren. Durch das Berechnen einer unteren Schranke für die Kosten aller in einem Teilbaum vorhandenen Blätter, versucht man die Größe des Suchraums zu beschränken. Dies ist möglich, da ein Teilbaum, dessen untere Schranke schlechter als das bisherige Minimum ist, garantiert nur schlechtere Lösungen beinhaltet und daher nicht durchsucht werden muß. In Abbildung 2.4 haben wir zwei Bäume skizziert, die jeweils den gleichen Suchraum auf unterschiedliche Art darstellen. Jeder einzelne Knoten dieser Bäume beinhaltet die in Abschnitt 2.1 eingeführte Allokationsmatrix X. Im Gegensatz zu der dortigen Definition erlauben wir jetzt jedoch neben den Werten ’1’ (Objekt p liegt im Cache von Rechner i) und ’0’ (Objekt p liegt nicht im Cache von Rechner i), auch den Wert ’?’. Ein Eintrag mit dem Wert ’?’ besagt, daß auf dieser Stufe im Baum noch nicht festgelegt ist, ob das entsprechende Objekt im Cache gehalten werden soll oder nicht. Die beiden Varianten in der Abbildung 2.4 untscheiden sich nun dahingehend, daß bei der Variante (a) auf jeder Stufe des Baums die Plazierung eines weiteren Objekts spezifiziert wird, während bei der Variante (b) auf jeder Stufe die Allokation eines zusätzlichen Rechners festgelegt wird.

Obwohl jeder der in Abbildung 2.4 dargestellten Bäume $2^{M \times N}$ Blätter hat, fallen doch auf Grund der Speicherplatznebenbedingung bereits eine Vielzahl dieser Blätter als mögliche Lösungen aus. Zur Abschätzung der maximalen Anzahl von erlaubten Knoten nehmen wir wiederum ver-

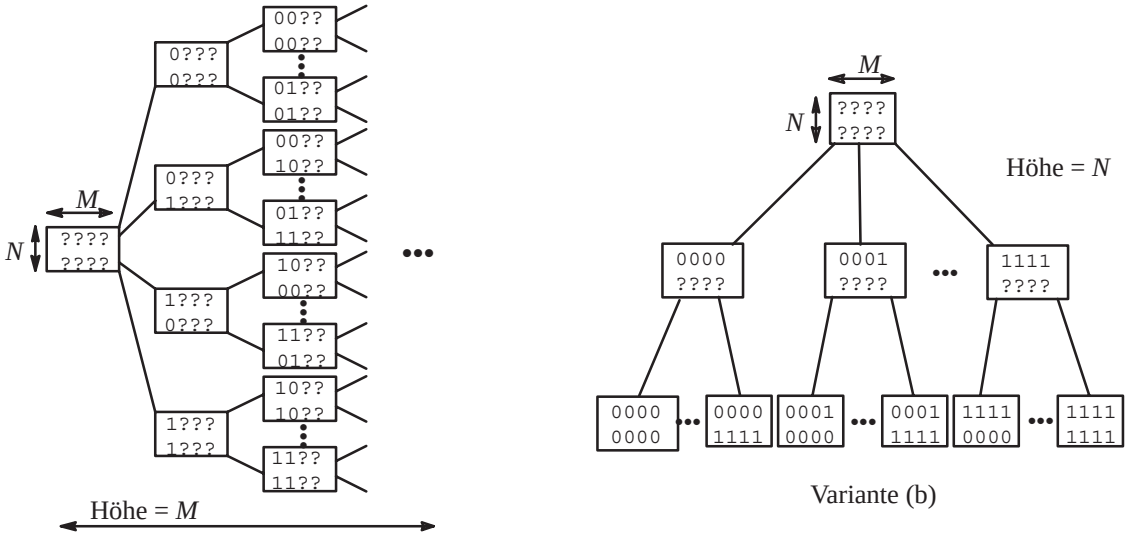


Abbildung 2.4: Zwei unterschiedliche Möglichkeiten, den Suchraum als Baum darzustellen. Eine Beschränkung der Größe des Suchraums durch eine Berücksichtigung der Speicherplatznebenbedingung erfolgt in der Darstellung jedoch nicht.

einfachend an, daß alle Objekte gleich groß sind. Betrachten wir die Variante (a) aus der Abbildung 2.4, so besitzt der Baum auf der ersten Ebene einen Knoten (die Wurzel) und auf der zweiten Ebene bereits 2^N Knoten, da wir unabhängig auf jedem Rechner entscheiden können, ob wir das Objekt mit dem Index 1 im Cache halten oder nicht. Verallgemeinern wir diese Vorgehensweise, so erhalten wir auf der *i*-te Ebene des Baums 2^{iN} Knoten. Dies ist jedoch nur dann korrekt, wenn jeder Rechner auch mindestens *i* Objekte speichern kann. Betrachten wir jedoch eine Ebene *k* des Baums, wobei *k* größer ist als die Größe des Caches, so befinden sich wegen der Speicherplatzbeschränkung auf dieser Ebene nur

$$\left(\binom{i}{0} + \cdots + \binom{i}{B}\right)^N$$

Knoten. Die Gesamtanzahl der Knoten erhalten wir nun, indem wir diesen Wert über alle Ebenen des Baums aufsummieren.

Algorithmus 2.1 zeigt einen generischen *Branch-and-Bound*-Algorithmus, den wir für die beiden möglichen Darstellungen aus Abbildung 2.4 benutzen können. Die Funktion *dfs* implementiert eine *Depth-First-Search*-Strategie auf dem betrachteten Suchbaum, wobei vor jedem Tiefersteigen überprüft wird, ob sich in dem entsprechenden Teilbaum günstigere Allokationen befinden können. Kann dies auf Grund der Berechnung der unteren Schranke (*compute_lower_bound*) ausgeschlossen werden, so wird die Suche für diesen Teilbaum abgebrochen. Erreicht die Suche ein Blatt, so berechnet die Funktion *compute_cost* die exakten Kosten für diese Allokation. Das eigentliche Problem eines jeden *Branch-and-Bound*-Algorithmus ist jedoch die Implementierung der Funktion *compute_lower_bound*, da diese die Selektivität des Algorithmus bestimmt.

```
void branch_and_bound ()
begin
    min = ∞;
    Initialisiere Matrix X mit '?';
    Initialisiere Matrix MIN mit '0';
    dfs (X, 1);
    output (MIN, min);
end;

void dfs (matix X, int z)
begin
    forall (mögliche 0-1-Kombination der Zeile z der Matrix X) do
        if (Speicherplatz-Nebenbedingung erfüllt) then
            if (z < MAX_ROW) then
                // Betrachten eines inneren Knotens
                if (compute_lower_bound (X, z) < min) then
                    dfs (X, z+1);
                fi
            else
                // Betrachten eines Blattes
                c = compute_cost (X)
                if (c < min) then
                    MIN = X;
                    min = c;
                fi;
            fi;
        fi;
    od;
end;

real compute_cost (X)
begin
    cost = 0.0;
    forall (Rechner i) do
        forall (Objekte p) do
            if (X_ip == 1) then          cost += f_ip × c_lc × s_p;
            elsif (∃k: X_kp == 1) then   cost += f_ip × c_rc × s_p;
            elsif (Y_ip == 1) then       cost += f_ip × c_ld × s_p;
            else                         cost += f_ip × c_rd × s_p;
            fi;
        od;
    od;
    return cost;
end;
```

Algorithmus 2.1: Ein generischer *Branch-and-Bound*-Algorithmus

Definition 2.4: (Selektivität)

Die **Selektivität** eines Branch-and-Bound-Algorithmus definieren wir als den Quotienten aus der Anzahl der von dem Algorithmus überprüften Knoten zu der Gesamtanzahl der in dem erlaubten Suchraum vorhandenen Knoten.

■

Im folgenden werden wir nun für jede der Baumvarianten aus Abbildung 2.4 eine untere Schranke herleiten.

Betrachten wir zunächst einen Knoten der Variante (a), der sich auf der Ebene p ($p < M$) befindet:

- Für die Objekte q ($q \leq p$) ist die Allokation auf allen Rechnern bereits festgelegt. Daher wissen wir bereits, ob diese Objekte aus dem lokalen, dem nichtlokalen Cache oder von Festplatte gelesen werden müssen.
- Betrachten wir nun einen Rechner i, so können wir annehmen, daß sein noch freier Cache mit seinen lokal heißesten Objekten aufgefüllt wird. Eine minimale mittlere Antwortzeit für diesen Rechner erhalten wir, wenn wir außerdem annehmen, daß der restliche freie Cache aller anderen Rechner benutzt wird, um die lokal nächst heißesten Objekte zu speichern, sofern diese nicht schon in dem Cache eines anderen Rechners gehalten werden.

Da die Zugriffskosten konstant sind, beeinflussen sich Entscheidungen bezüglich der Speicherung zweier Objekte j_1 und j_2 nicht gegenseitig, und daher führt wegen der folgenden allgemeingültigen Ungleichung

$$\min C = \min \sum_i \sum_j f_{ij} \times c_{ij} \times s_j \geq \sum_i \min \sum_j f_{ij} \times c_{ij} \times s_j = \sum_i \min C_i$$

eine Minimierung der mittleren Kosten für jeden einzelnen Rechner – in den durch die bisherige Allokation vorgegebenen Grenzen – auch zu eine Minimierung der mittleren Gesamtkosten.

Es muß jedoch besonders betont werden, daß diese – für die Berechnung der unteren Schranke benutzte Allokation – im allgemeinen nicht die gesuchte optimale Allokation darstellt, da wir für jeden Rechner unabhängig von allen anderen Rechnern das Minimum bestimmen. Dabei berücksichtigen wir nicht, daß sich diese Minima wegen der Speicherplatznebenbedingung gegenseitig ausschließen können.

Für die Variante (b) aus Abbildung 2.4 können wir analog eine untere Schranke herleiten. Auch hier versuchen wir wiederum, die mittlere Antwortzeit jedes einzelnen Rechners in den Grenzen der bereits festgelegten Allokation zu minimieren. Betrachten wir dazu einen Knoten des Baums auf der Ebene i ($i < N$):

- Für jeden Eintrag $X_{kp} = 1$ wissen wir, daß das Objekt p im Cache des Rechners k lokal gehalten wird. Dies gilt auch für alle Knoten und Blätter des Unterbaums, und daher steht fest, daß die Zugriffskosten von Rechner k auf das Objekt p gleich den Kosten eines lokalen Cache-Zugriffs sind.
- Ist $X_{kp} = 0$ so müssen wir zwei Fälle unterscheiden. Falls ein Rechner l existiert mit $X_{lp} = 1$, dann ist das Objekt p auf jedem Fall über einen Zugriff auf den nichtlokalen Cache erreichbar. Existiert jedoch kein solcher Rechner, so dürfen wir nicht generell davon ausgehen, daß wir das Objekt von Festplatte lesen müssen, da es noch Rechner gibt, deren Speicherbelegung bis jetzt noch nicht festgelegt ist. Die mittlere lokale Antwortzeit des Rechners k minimieren wir, wenn wir annehmen, daß der gesamte noch nicht belegte Cache des Systems (Cache aller Rechner mit Index > i) Rechner k zur Verfügung gestellt wird, um die lokal heißesten Objekte von Rechner k zu speichern, die nicht bereits im Cache des Rechners k oder eines anderen Rechners gehalten werden. Für Zugriffe von Rechner k auf diese Objekte berechnen wir die Kosten eines Zugriffs auf den nichtlokalen Cache und für alle anderen Zugriffe von Rechner k aus berechnen wir die Kosten eines Festplattenzugriffs.
- Für alle Rechner k, denen noch keine Objekte zugeordnet sind ($k > i$) nehmen wir an, daß maximal viele der auf diesem Rechner heißesten Objekte im lokalen Cache gehalten werden

und daß der komplette restliche Cache des verteilten Systems für die lokal nächst heißesten Objekte zur Verfügung steht. Durch diese Vorgehensweise minimieren wir auch hier die mittlere Antwortzeit des Rechners *k*.

Auch hier ist wiederum anzumerken, daß die Allokationen, die wir zur Berechnung der unteren Schranke konstruieren, im allgemeinen keine – bezüglich der Nebenbedingung – gültigen Allokationen darstellen.

2.3 Einfache Heuristiken

In diesen Abschnitt werden wir zwei einfache *Greedy*-Heuristiken vorstellen, die im Gegensatz zu der exakten Methode in polynomieller *Worst-Case*-Laufzeit eine Allokation berechnen. Dazu gehen beide von einer positiven Korrelation zwischen der Cache-Trefferrate für eine gegebene Allokation und den entsprechenden Kosten dieser Allokation aus. Da jedoch in unserem verteilten System – je nach Sichtweise – entweder jeder Rechner einen lokalen Cache oder aber das Gesamtsystem einen globalen verteilten Cache besitzt, unterscheiden sich die beiden Heuristiken hinsichtlich ihres Optimierungsziels dahingehend, daß sie entweder die lokale oder die globale Cache-Trefferrate zu maximieren versuchen.

2.3.1 Egoistische Heuristik

Die erste einfache Heuristik versucht die lokale Cache-Trefferrate jedes einzelnen Rechners – unabhängig von allen anderen Rechnern – zu maximieren. Da wir variabel große Objekte betrachten, ist es nicht ausreichend, die Objekte mit der größten Hitze im Cache zu halten, sondern wir müssen die Größe der Objekte mit berücksichtigen. Dies können wir durch die Betrachtung der Temperatur (vgl. Definition 2.1) an Stelle der Hitze erreichen. Bei der egoistischen Heuristik hält daher jeder Rechner die Objekte mit der größten lokalen Temperatur in seinem lokalen Cache, ohne dabei jedoch zu berücksichtigen, daß sich diese Objekte bereits im Cache eines anderen Rechners befinden können. Der Algorithmus 2.2 zeigt den Pseudocode für diese Heuristik.

```
void egoistic ()
begin
    forall (Rechner i) do
        free_space = B_i;
        forall (Objekte p) sorted desc by lokaler Temperatur do
            if (free_space > s_p) then
                Rechner i alloziert Objekt p im lokalen Cache;
                free_space = free_space - s_p;
            fi;
        od;
    od;
end;
```

Algorithmus 2.2: Eine egoistisch agierende Heuristik

Die *Worst-Case*-Laufzeit dieses Algorithmus setzt sich aus einem Sortiervorgang der M Objekte ($\mathcal{O}(M \log M)$) auf jedem der N Rechner und der eigentlichen Zuordnung der Objekte zu den Rech-

nern zusammen. Da diese Zuordnung in jedem Fall in der Zeit $\mathcal{O}(MN)$ durchgeführt werden kann, erhalten wir insgesamt die Laufzeit $\mathcal{O}(NM \log M)$ für diese Heuristik.

Da das Optimierungsziel dieser Heuristik die Maximierung der lokalen Cache-Trefferrate ist, erwarten wir, daß sie vor allem in Umgebungen mit hohen Netzzugriffskosten gute Ergebnisse liefert. Nachteilig an der rein-lokalen Sicht dieser Heuristik kann sich jedoch die unter Umständen große Anzahl von Replikaten in den verschiedenen Caches auswirken. Diese Replikate reduzieren die globale Cache-Trefferrate und können dadurch zu einer schlechteren Performance führen.

2.3.2 Altruistische Heuristik

Im Gegensatz zu der egoistischen Heuristik versucht die altruistische Heuristik, die globale Cache-Trefferrate zu maximieren. Dazu bestimmen wir für jedes Objekt die globale Temperatur und versuchen wiederum möglichst viele der global heißesten Objekte im aggregierten Cache des Gesamtsystems zu halten. Wegen des Fragmentierungsproblems, das durch die unterschiedliche Objektgröße hervorgerufen wird, ist es – ohne erheblichen Aufwand – nicht möglich, das Optimum bezüglich der globalen Trefferrate sicherzustellen. Heuristische Verfahren, die eine gute, jedoch im allgemeinen nicht optimale Lösung generieren, sind *First-Fit* oder *Best-Fit*. Da wir jedoch auch keine zu starken Einbußen bei der lokalen Trefferrate in Kauf nehmen wollen, haben wir eine *Greedy*-Heuristik gewählt, die die lokalen Zugriffshäufigkeiten mit berücksichtigt. So versuchen wir ein Objekt zuerst im Cache des Rechners zu allozieren, auf dem es die größte lokale Temperatur besitzt. Bei mehreren Rechnern mit der gleichen lokalen Temperatur wählen wir einen zufällig aus. Eine andere Möglichkeit wäre, den Rechner mit der bislang geringsten akkumulierten Temperatur als Ziel auszuwählen. Dies würde zwar zu einer gleichmäßigeren Belastung aller Rechner führen. Da wir jedoch konstante Zugriffskosten haben, wirkt sich eine Balancierung der Ressourcen in diesem Modell nicht aus. Den Pseudocode zur Berechnung einer altruistischen Cache-Allokation zeigt Algorithmus 2.3.

```
void altruistic ()
begin
    forall (Rechner i) do
        free_space_i = B_i;
    od;
    forall (Objekte p) sorted desc by globaler Temperatur do
        i = Rechner mit maximaler lokaler Temperatur auf Objekt p und free_space_i > s_p;
        Rechner i alloziert Objekt p im lokalen Cache;
        free_space_i = free_space_i - s_p;
    od;
end;
```

Algorithmus 2.3: Eine altruistisch agierende Heuristik

Im Gegensatz zu der egoistischen Heuristik müssen wir bei diesem Algorithmus die Objekte nur einmal in Zeit $\mathcal{O}(M \log M)$ sortieren. Innerhalb jedes Schleifendurchlaufs müssen wir jedoch den Rechner bestimmen, auf dem das entsprechende Objekt die maximale Temperatur besitzt und der gleichzeitig noch Platz für das Caching des betreffenden Objekts hat. Dies können wir durch das einmalige Betrachten jedes Rechners und damit in Zeit $\mathcal{O}(N)$ erreichen. Somit erhalten wir insgesamt eine *Worst-Case*-Laufzeit von $\mathcal{O}(M \log M + MN)$.

Wegen der Priorisierung der globalen Trefferrate erwarten wir, daß diese Heuristik besonders gut geeignet ist, wenn die Differenz zwischen den Zugriffskosten auf den lokalen und den nichtlokalen Cache gering ist. Da jedoch jede Replikation von Objekten explizit verhindert wird, ist zu erwarten, daß diese Heuristik schlecht abschneidet, wenn mehrere Rechner eine sehr große Temperatur auf den gleichen Objekten besitzen.

2.4 Meßergebnisse

In diesem Abschnitt stellen wir einige analytische Ergebnisse der beschriebenen Algorithmen vor. Bei den *Branch-and-Bound*-Algorithmen betrachten wir nur den Algorithmus, der auf der Variante (a) aus Abbildung 2.4 basiert, da dieser in allen unseren Messungen wesentlich geringere Laufzeiten erzielt hat als der auf Variante (b) basierende Algorithmus und natürlich beide die gleichen Resultate liefern. In einer ersten Meßreihe werden wir untersuchen, wie sich die Güte der heuristischen Lösungen – beim Variieren der Lastparameter – zu der optimalen Lösung verhalten. Anschließend werden wir in einer weiteren Meßreihe den Einfluß der Systemparameter auf die Rechenzeit des *Branch-and-Bound*-Algorithmus untersuchen.

2.4.1 System- und Lastbeschreibung

Die Parameter, die unser System beschreiben, sind: die Anzahl der Rechner, die Anzahl der Objekte und die Cache-Größe der Rechner. Bei unseren Messungen gehen wir von einem homogenen System aus, bei dem die Cache-Größen auf allen Rechnern gleich sind. Da wir uns ebenfalls auf Objekte konstanter Größe beschränken, können wir die Cache-Größe durch die Anzahl der Objekte angeben, die im Cache gehalten werden können. Die Werte für die Kosten eines Zugriffs auf eine bestimmte Hierarchiestufe übernehmen wir aus [DWA*94]. Um den Einfluß der initialen Objektallokation auf Festplatte herauszufaktorisieren, nehmen wir an, daß jeder Rechner eine eigene Festplatte besitzt, auf der alle Daten repliziert gespeichert sind. Die konkreten Werte für die Zugriffskosten bei zwei unterschiedliche Netzwerktechnologien sind in der Tabelle 2.1 aufgeführt.

10 Mbit/s Ethernet			**155 Mbit/s ATM**		
Lokaler Cache c_{lc}	Nichtlokaler Cache c_{rc}	Lokale Festplatte c_{ld}	Lokaler Cache c_{lc}	Nichtlokaler Cache c_{rc}	Lokale Festplatte c_{ld}
0.25 ms	6.9 ms	15.05ms	0.25 ms	1.05 ms	15.05 ms

Tabelle 2.1: Zugriffskosten bei unterschiedlicher Netzwerktechnologie

In unseren Experimenten nehmen wir an, daß das Zugriffsverhalten auf jedem Rechner gleich ist und daß die Zugriffsverteilung auf die Objekte einer Zipf-Verteilung [Zipf49, Knut73] entspricht. Ohne Beschränkung der Allgemeinheit können wir weiterhin davon ausgehen, daß die Objekte entsprechend ihrer Hitze sortiert sind. Unter diesen Voraussetzungen bestimmt die folgende Gleichung die Hitze des Objekts mit dem Index p:

$$heat(p) = \frac{1}{C \times p^{\theta}} \text{ mit der Normierungskonstante } C = \sum_{q=1}^{M} 1/q^{\theta}$$

Der Parameter θ charakterisiert die Schräge der Verteilung. Während die Zipf-Verteilung für θ gleich Null der Gleichverteilung entspricht, nimmt die Schräge des Zugriffsverhaltens mit wachsenden Werten für θ an Schräge zu.

In den folgenden Experimenten werden wir jeweils einen Parameter der Last oder des Systems variieren und die Auswirkungen dieser Variation auf die Güte des berechneten Ergebnisses und auf die Rechenzeit betrachten. Die Parameter, die wir nicht variieren, erhalten jeweils die Standardwerte, die wir in der Tabelle 2.2 aufgeführt haben.

Parameter	Standardwert
Anzahl der Rechner N	3
Anzahl der Objekte M	20
Cache-Größe B	5
Schräge der Verteilung θ	1.5
Netzwerkkosten c_{rc}	6.9 ms

Tabelle 2.2: Standardwerte der Parameter

2.4.2 Variation der Last und der Zugriffskosten

In dem ersten Experiment untersuchen wir den Einfluß der Schräge in der Verteilung auf die Zugriffskosten der berechneten Allokationen. In Abbildung 2.5 haben wir die mittlere Antwortzeit aufgetragen, wenn wir die Schräge θ von 0 bis 3 variieren.[2] Deutlich zu erkennen ist, daß die Rangfolge der Heuristiken von der gewählten Schräge abhängt. So liefert die altruistische Methode bei geringer Schräge bessere und bei großer Schräge schlechtere Ergebnisse als die egoistische Methode. Des weiteren können wir sehen, daß zu den Grenzen des beobachteten Bereichs der Schräge jeweils eine der einfachen Heuristiken gegen das durch die *Branch-and-Bound*-Methode berechnete Optimum konvergiert. Zwischen diesen Extremen liegt jedoch ein weiter Bereich, in dem das Optimum durch keine der Heuristiken erreicht wird.

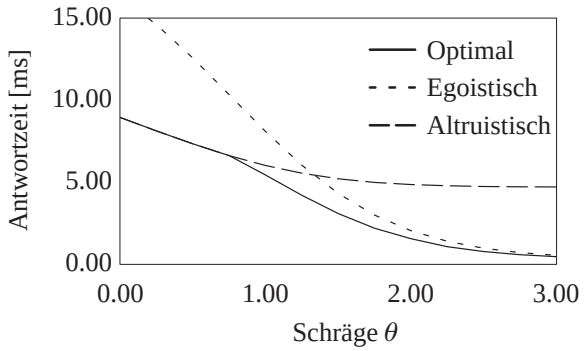


Abbildung 2.5: Variation der Schräge der Verteilung

Dieses Verhalten wird klar, wenn wir uns in der Abbildung 2.6 die optimalen Allokationen für die verschiedenen Schrägen anschauen. Bei sehr geringer Schräge (Gleichverteilung) entspricht die optimale Lösung gerade der altruistischen Allokation. Mit zunehmender Schräge nähert sich die optimale Allokation immer mehr einer egoistischen Verteilung an. Die Ursache hierfür ist, daß bei hoher Schräge einige wenige Objekte existieren, die auf allen Rechnern sehr heiß sind. Vermeiden man nun explizit, daß diese Objekte repliziert in den Caches gehalten würden, so müssen

2. Bei der egoistischen Heuristik betrachten wir nur den Grenzwert für θ gegen 0, da die Antwortzeitkurve bei θ=0 unstetig ist. Der Grund hierfür liegt darin, daß bei einer Gleichverteilung (θ=0) keine Rangfolge bezüglich der Hitze zwischen den Objekten besteht und daher der Replikationsgrad der Objekte sich sprunghaft ändert. Dieses Problem betrachten wir genauer in Kapitel 5.

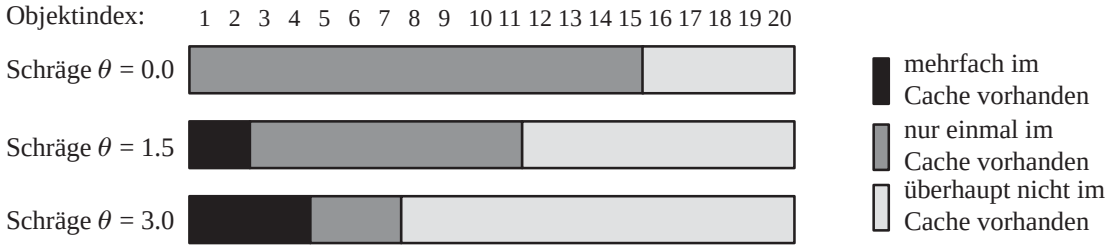


Abbildung 2.6: Optimale Allokationen für unterschiedliche Schrägen der Verteilung

sehr viele Rechner diese Objekte aus dem nichtlokalen Cache lesen, während bei einem replizierten Caching jeder Rechner diese heißen Objekte lokal vorfindet.

Im nächsten Experiment variieren wir die Kosten eines Netzzugriffs von 1 ms (entspricht ungefähr den Kosten bei 155 MBit/s *ATM*) bis zu 7 ms (entspricht ungefähr den Kosten bei 10 MBit/s *Ethernet*). Abbildung 2.7 zeigt die Ergebnisse dieser Meßreihe. Auch hier erkennen wir, daß die Rangfolge der Heuristiken bezüglich der Kosten der berechneten Allokation von den aktuellen Parametern – in diesem Fall von der Netzgeschwindigkeit – abhängt. Bei geringen Kosten für Zugriffe auf den nichtlokalen Cache ist es günstig, möglichst viele verschiedene Objekte im Speicher zu halten, da dies die globale Trefferrate erhöht. Daß hierdurch die lokale Trefferrate reduziert wird, spielt eine untergeordnete Rolle, da der Unterschied zwischen den Kosten für einen lokalen und einen nichtlokalen Zugriff sehr gering ist. Daher ist für eine solche Umgebung die altruistische Heuristik besser geeignet. Dies ändert sich jedoch mit ansteigenden Zugriffskosten auf den nichtlokalen Cache, da hierdurch die Differenz der Zugriffskosten zwischen nichtlokalem Cache und Festplatte geringer wird. Bei sehr langsamen Netzwerken führt daher eine Erhöhung der lokalen Trefferrate zu einer Leistungssteigerung des Systems, und dies bewirkt die Umkehrung in der Rangfolge zwischen der altruistischen und der egoistischen Heuristik, die wir in der Abbildung 2.7 beobachten können.

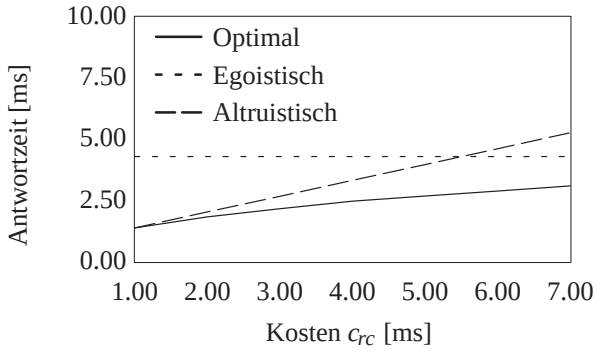


Abbildung 2.7: Variieren der Netzzugriffskosten

2.4.3 Untersuchung des Einflusses der Systemparameter auf die Laufzeit

Nachdem wir nun die Güte der berechneten Allokationen betrachtet haben, möchten wir in diesem Abschnitt untersuchen, wie sich die Rechenzeiten des *Branch-and-Bound*-Algorithmus in Abhängigkeit von den Systemparametern verändern. Der Grund für die Beschränkung auf die Systemparameter liegt darin, daß alleine diese Parameter die Größe des Suchraums bestimmen.

Zwar haben auch die Lastparameter einen Einfluß auf die Laufzeit, indem sie die Verteilung der Kosten und somit die Anzahl der zu besuchenden Knoten verändern, jedoch lassen sie die maximale Größe des Suchraums unbeeinflußt. In den Tabellen 2.3, 2.4 und 2.5 haben wir die für die Laufzeit des *Branch-and-Bound*-Algorithmus relevanten Statistiken aufgeführt. Die maximale Größe des Suchraums haben wir entsprechend der in Abschnitt 2.2 vorgestellten Methode ermittelt, und die Effektivität unseres *Branch-and-Bound*-Algorithmus beurteilen wir an Hand der in Definitioin 2.4 eingeführten Selektivität. Zur Bestimmung der Rechenzeit haben wir die Algorithmen auf einer *SUN*-Workstation vom Typ *Sparc 4* ausgeführt. Da die Rechenzeit der einfachen Heuristiken in allen von uns untersuchten Szenarien unter 0.01 Sekunden lag, haben wir diese in den Tabellen nicht aufgeführt.

In der ersten Meßreihe variieren wir die Anzahl der Objekte (Tabelle 2.3). Sehr deutlich ist das explosionsartige Anwachsen des gesamten Suchraums in der Tabelle an der Knotenanzahl zu erkennen. Zwar ist unser Algorithmus durch das *Branch-and-Bound*-Paradigma in der Lage den wirklich betrachteten Suchraum sehr stark einzuschränken, jedoch kann selbst die gute Selektivität bei der Erhöhung der Objektanzahl ein schnelles Anwachsen des Suchraums und damit der Rechenzeit nicht verhindern.

Objekt-anzahl	Knoten-anzahl	Betrachtete Knoten	Selektivität	Rechenzeit [s]
20	$1.82\ 10^{13}$	181979	$1.00\ 10^{-08}$	10
30	$1.26\ 10^{16}$	246271	$1.96\ 10^{-11}$	21
40	$1.31\ 10^{18}$	183374	$1.40\ 10^{-13}$	18
50	$4.79\ 10^{19}$	765663	$1.60\ 10^{-14}$	106
60	$9.04\ 10^{20}$	2091722	$2.31\ 10^{-15}$	349
70	$1.07\ 10^{22}$	2896231	$2.68\ 10^{-16}$	558
80	$9.25\ 10^{22}$	3348810	$3.61\ 10^{-17}$	729
90	$6.16\ 10^{23}$	2269784	$2.71\ 10^{-17}$	548
100	$3.35\ 10^{24}$	4254421	$1.27\ 10^{-18}$	1140

Tabelle 2.3: Selektivität des *Branch-and-Bound*-Algorithmus bei Variation der Objektanzahl

Ein noch stärkeres Anwachsen der Rechenzeit können wir beobachten, wenn wir die Anzahl der Rechner erhöhen (Tabelle 2.4). Auch hier liegt der Grund wiederum darin, daß die Selektivität des Algorithmus nicht die exponentielle Vergrößerung des Suchraums ausgleichen kann.

Rechner-anzahl	Knoten-anzahl	Betrachtete Knoten	Selektivität	Rechenzeit [s]
2	$1.09\ 10^{9}$	543	$1.61\ 10^{-9}$	0.02
3	$1.82\ 10^{13}$	181970	$1.00\ 10^{-8}$	10
4	$3.35\ 10^{17}$	$3.19\ 10^{8}$	$9.52\ 10^{-10}$	24200
5	$6.51\ 10^{21}$			> 70 h

Tabelle 2.4: Selektivität des *Branch-and-Bound*-Algorithmus bei Variation der Rechneranzahl

In der letzten Messung variieren wir die Größe des Caches (Tabelle 2.5). Bis zu der Cache-Größe 8 beobachten wir auch hier einen starken Anstieg in der Rechenzeit. Erhöhen wir die Cache-Größe jedoch noch weiter, so sinkt die Rechenzeit wieder. Die Ursache für das Fallen der Rechenzeit bei einem großen Cache liegt darin, daß bereits auf den hohen Ebenen des Baums sehr große Be-

reiche des Suchraums ausgeschlossen werden können. Betrachten wir zum Beispiel den Fall, in dem der Cache die Größe 18 hat und der Datenbestand aus insgesamt 20 Objekten besteht. Da wir in Abschnitt 2.2 bereits argumentiert haben, daß jede optimale Allokation den zur Verfügung stehenden Cache voll ausnutzt, muß in diesem Fall bei einer optimalen Allokation jeder Rechner 18 Objekte im Cache halten. Aus diesem Grund können wir bereits auf der vierten Ebene des Baums alle die Unterbäume aus dem Suchraum entfernen, bei denen ein Rechner mehr als zwei Nullen in der Allokationsmatrix besitzt. Besteht unser System aus 3 Rechnern, so können wir durch diese Methode auf der vierten Ebene des Baums schon über 80 % des Suchraums ausschließen. Entsprechend erlaubt uns diese notwendigen Bedingung für eine optimale Allokation auch auf tieferen Ebenen eine Vielzahl von Unterbäumen auszusparen. Insgesamt erreichen wir dadurch für Umgebungen, in denen die Cachegröße vergleichbar mit der Objektanzahl ist, eine sehr gute Selektivität. Dies ist jedoch ein in der Realität äußerst selten auftretender Fall, da im allgemeinen selbst der aggregierte Hauptspeicher des Gesamtsystems deutlich kleiner als die Gesamtdatenmenge ist. Für diesen interessanteren Bereich haben wir in unserer Messung jedoch auch einen sehr starken Anstieg der Rechenzeit mit wachsender Cache-Größe beobachtet.

Cache-größe	Knoten-anzahl	Betrachtete Knoten	Selektivität	Rechenzeit [s]
2	$3.31\ 10^{7}$	83	$2.55\ 10^{-6}$	0.01
4	$4.94\ 10^{11}$	12467	$2.52\ 10^{-8}$	0.71
6	$3.50\ 10^{14}$	$3.94\ 10^{6}$	$1.13\ 10^{-8}$	234
8	$2.49\ 10^{16}$	$8.89\ 10^{7}$	$3.57\ 10^{-9}$	5180
10	$2.89\ 10^{17}$	$5.13\ 10^{6}$	$1.78\ 10^{-11}$	282
12	$8.84\ 10^{17}$	629664	$7.12\ 10^{-13}$	33
14	$1.24\ 10^{18}$	63053	$5.08\ 10^{-14}$	3.2
16	$1.31\ 10^{18}$	3282	$2.49\ 10^{-15}$	0.16
18	$1.31\ 10^{18}$	187	$1.42\ 10^{-16}$	0.01
20	$2.47\ 10^{18}$	8	$3.24\ 10^{-18}$	<0.01

Tabelle 2.5: Selektivität des *Branch-and-Bound*-Algorithmus bei Variation der Cache-Größe

2.5 Folgerungen

Die Auswertung unserer Experimente zeigt, daß jeweils in den Extremsituationen die Ergebnisse einer der einfachen Heuristiken nahe am bzw. gleich dem Optimum sind, während die der jeweils anderen eine deutlich schlechtere Performance liefert. Eine erste Verbesserung gegenüber den einfachen Heuristiken könnte daher darin bestehen, in Abhängigkeit von den Systemparametern und der aktuellen Last, jeweils die Heuristik auszuwählen, die in dieser Situation die besseren Ergebnisse liefert. Da wir jedoch in den Abbildungen 2.6 und 2.7 sehen können, daß selbst eine solche Kombination der beiden einfachen Heuristiken über einen weiten Bereich das Optimum deutlich verfehlen würde, verbleibt noch ein großer Spielraum für intelligentere Heuristiken.

Um eine solche bessere Heuristik herzuleiten, können wir das Verhalten der optimalen Algorithmen untersuchen. In unseren Experimenten haben wir gesehen, daß es vorteilhaft ist, wenn die Entscheidung über den Replikationsgrad auf Objekt- und nicht auf Systemebene getroffen werden kann. Dadurch ist es möglich für jedes Objekt separat zu entscheiden, ob es nur einmal (altruistisch) oder repliziert über alle Rechner (egoistisch) im Cache gehalten werden soll.

Die in diesem Kapitel durchgeführte Annahme einer konstanten Last beschränkt den Einsatzbereich dieser Heuristiken, da sich in realen Anwendungen die Lastparameter über die Zeit ändern. Eine Möglichkeit diese Änderungen zu berücksichtigen besteht darin, in einem festen Zeitraster den entsprechenden Algorithmus mit den jeweils aktuellen Parametern auszuwerten. Hierzu ist es jedoch notwendig, diese Parameter online bestimmen zu können. Jedoch selbst unter der Annahme, daß diese Parameter bekannt sind, ist dieses Verfahren der wiederholten Auswertung nur für die einfachen Heuristiken einsetzbar, da unsere Untersuchung der Laufzeit gezeigt hat, daß selbst die einmalige Auswertung des optimalen Algorithmus für realistisch große Systeme unmöglich ist.

Alle diese Beobachtungen motivieren die Entwicklung einer Heuristik, die in Abhängigkeit von den aktuellen Last- und Systemparametern, mit geringem Overhead, eine objektbezogene Replikationskontrolle durchführen kann. Eine solche Heuristik werden wir im folgenden Kapitel herleiten.

Kapitel 3

Online-Heuristiken für verteiltes Caching

Im Gegensatz zu den Verfahren aus Kapitel 2, die wir zur Berechnung einer statischen Cache-Allokation benutzen können, wollen wir in diesem Kapitel Heuristiken vorstellen, die in der Lage sind, mit dynamisch schwankenden Lasten und Ressourcenauslastungen umzugehen. Beim Übergang zu Online-Heuristiken sind jedoch einige Probleme zu lösen, die wir zunächst genauer beschreiben wollen:

- Im Gegensatz zu den statischen Methoden können wir nun nicht länger davon ausgehen, daß wir ein *globales Wissen* besitzen, auf das wir *ohne Zusatzkosten* zugreifen können. Dies trifft sowohl auf die Informationen zur Abarbeitung von Zugriffen (wie z.B. die Plazierung der Objekte in den Caches und auf den Platten) als auch auf die von den Heuristiken benötigten Daten (wie z.B. die lokalen und globalen Hitzen und die Zugriffskosten) zu.

  Zur Verwaltung der ersten Art von Information werden wir eine Kombination von globalem und lokalem Katalog benutzen. Da die Cache-Heuristiken jedoch weitestgehend von der konkreten Implementierung dieser Kataloge unabhängig sind, werden wir erst in Kapitel 4 die von uns benutzte Implementierung vorstellen. Wichtiger sind die Methoden zur Bestimmung der für die verschiedenen Heuristiken notwendigen Informationen. Um Reaktionen auf dynamische Schwankungen zu ermöglichen, müssen wir diese Informationen während des laufenden Systems sammeln. Gleichzeitig verursachen diese Aktionen aber selbst eine Last, die sogar zu einer Verschlechterung der Gesamtleistung führen kann. Um dies zu vermeiden, stellen wir in Abschnitt 3.1 speziell auf unsere Anforderungen zugeschnittene Protokolle zum Sammeln und Verteilen der entsprechenden Informationen vor.

- Während bei den statischen Heuristiken aus Kapitel 2 nur *eine einmalige Optimierung* durchgeführt wurde und anschließend das System konstant mit dieser berechneten Allokation lief, müssen bei den Online-Heuristiken die Berechnungen während des laufenden Systems erfolgen. Ähnlich wie bei der Sammlung und Verteilung der Informationen kann auch hier der Zusatzaufwand für die Berechnungen einer komplexen Heuristik sogar zu einer Verschlechterung der Leistungsfähigkeit gegenüber einer einfacherer Heuristik führen.

  Aus diesem Grund ist es wichtig, daß die Optimierungsschritte nur an besonders geeigneten Stellen vorgenommen werden. Hier bieten sich besonders die Cache-Ersetzungen an, da an diesen Stellen zwangsläufig der Cache-Inhalt geändert werden muß. Bei der Opferauswahl können wir daher mit geringem Aufwand den Cache-Inhalt entsprechend des Optimie-

rungsziels beeinflussen. In den Abschnitten 3.2 und 3.3 werden wir jeweils Ersetzungsstrategien vorstellen, die den Cache-Inhalt – analog zu den entsprechenden einfachen statischen Heuristiken – beeinflussen. Eine Heuristik, die versucht – ähnlich den exakten Algorithmen aus Kapitel 2 – zwischen einem altruistischen und einem egoistischen Verhalten abzuwägen, leiten wir in Abschnitt 3.4 her. Da aus Performance-Gründen die Opferauswahl bei allen diesen Methoden immer lokal durchgeführt wird, werden wir in Abschnitt 3.5 für die Heuristiken, die eine Optimierung des globalen Systemzustandes anstreben, eine Methode vorstellen, die einen Lastausgleich zwischen verschiedenen Rechnern erlaubt.

3.1 Methoden zum Sammeln und Verteilen statistischer Informationen

In diesem Abschnitt werden wir einige einfache Protokolle vorstellen, die es uns erlauben, die notwendigen Informationen für die verschiedenen Caching-Strategien zu berechnen und an die unterschiedlichen Rechner zu verteilen. Zuerst beschreiben wir eine Methode, die die lokale Hitze eines Objekts bestimmt. Diese Information, die jeder Rechner selbständig berechnen kann, benutzen wir anschließend, um die globale Hitze eines Objekts zu approximieren. Hierbei müssen die lokal berechneten Werte koordiniert zusammengefaßt werden. Abschließend stellen wir in diesem Abschnitt ein Verfahren vor, das eine approximierte Berechnung der zu erwartenden Kosten für einen Zugriff auf ein gegebenes Objekt bestimmt.

Bei der Beschreibung dieser Protokolle werden wir sehen, daß wir für ein Objekt häufig einen ausgezeichneten Rechner benötigen, der die Buchhaltung über die Existenz, über die Lokalisierung und über eventuell weitere Eigenschaften der Cache- und Plattenkopien dieses Objekts durchführt. Diesen ausgezeichneten Rechner werden wir im folgenden *Heimatrechner* des entsprechenden Objekts nennen. Obwohl wir zu diesem Zeitpunkt noch nicht näher auf die Beschreibung des Heimatrechners eingehen werden, wollen wir doch hier schon betonen, daß die Zuordnung eines Heimatrechners für jedes Objekt separat erfolgen kann. Dies soll die Engpaßproblematik, die durch die Benutzung eines zentralen Servers verursacht würde, vermeiden.

3.1.1 Approximation der lokalen Hitze

Zur Approximation der lokalen Hitze benutzen wir den LRU-k Algorithmus [OOW93], der eine Verallgemeinerung der bekannten Least-Recently-Used (LRU) Ersetzungsstrategie darstellt. Im Gegensatz zu LRU merken wir uns bei LRU-k für jedes Objekt jeweils die Zeitpunkte der letzten k Zugriffe. Nehmen wir an, daß wir die letzten k Zugriffe auf das Objekt p in dem Feld $last\_access\_times_p$ gespeichert haben (wobei größere Indizes zu älteren Zugriffen korrespondieren) und daß now der aktuelle Zeitpunkt ist, so können wir die Hitze des Objekts p durch die mittlere Zwischenankunftszeit der letzten k Zugriffe entsprechend der Formel 3.1 approximieren:

$$loc\_heat_p = \frac{k}{now - last\_access\_times_p[k]}. \tag{3.1}$$

Problematisch bei dieser Formel ist die Abhängigkeit von der aktuellen Zeit. Dies würde bedeuten, daß wir die Formel jedesmal neu auswerten müßten, wenn wir die lokale Hitze innerhalb ei-

ner Berechnung benutzen würden. Da wir die lokale Hitze als ein Kriterium für die Cache-Ersetzung benutzen wollen, hätte dies zur Folge, daß bei jeder lokalen Ersetzung die Formel 3.1 für alle Objekte ausgewertet werden müßte. Dies stellt jedoch einen viel zu großen Overhead dar, und daher benutzen wir die folgende Formel aus [SWZ94] zur Approximation der lokalen Hitze:

$$loc\_heat_p = \frac{k - 1}{last\_accesses_p[1] - last\_accesses_p[k]}. \quad (3.2)$$

Auch in der Formel 3.2 berechnen wir die Zwischenankunftszeit zwischen den letzten *k* Zugriffen, wobei wir nun aber die Zeit seit dem letzten Zugriff nicht mit berücksichtigen. Dies führt dazu, daß eine Änderung der berechneten lokalen Hitze nur dann erfolgt, wenn ein neuer Zugriff auf das entsprechende Objekt durchgeführt wird.

Der Algorithmus 3.1 zeigt den Pseudocode für die Berechnung der lokalen Hitze eines Objekts *p*. Sind auf das Objekt *p* weniger als *k* Zugriffe erfolgt, so approximieren wir die Hitze durch die mittlere Zwischenankunftszeit dieser $no\_of\_accesses_p$ vielen Zugriffe. Eine leichte Reduzierung der Overheads durch die Protokollierung der Zugriffszeiten können wir erreichen, wenn wir berücksichtigen, daß eine Neuberechnung der Hitze jeweils nur zu dem Zeitpunkt eines Zugriffs erfolgt. Daher ist es ausreichend wenn das Feld $last\_access\_times_p$ die *k*-1 letzten Zeitpunkte von bereits beendeten Zugriffen enthält, da wir zusammen mit dem aktuellen Zugriff, die benötigten *k* letzten Zugriffszeitpunkte kennen, und somit die Hitze berechnen können.

```
void compute_local_heat (Objekt p)
begin
    if (no_of_accesses_p < k) then
        no_of_accesses_p++;
    fi;

    if (no_of_accesses_p < k) then
        loc_heat_p = no_of_accesses_p / now;
    else
        loc_heat_p = (k-1) / (now - last_access_times_p[k-1]);
    fi;

    for (i=k-1; i>1; i--) do
        last_access_times_p[i] = last_access_times_p[i-1];
    od;
    last_access_times_p[1] = now;
end;
```

Algorithmus 3.1: Berechnung der lokalen Hitze

Der Parameter *k* ist ein Tuning-Parameter, wobei größere Werte für *k* zu einer größeren Genauigkeit und Stabilität der Approximation führen, jedoch gleichzeitig auch einen größeren Speicher- und Berechnungsoverhead verursachen. Zusätzlich verhindert ein großer Wert für *k* eine schnelle Adaption der lokalen Hitze auf sich ändernde Lastcharakteristika.

Zwar werden Hitzeänderungen von vorher kalten Objekten zu heißen Objekten von der Formel 3.2 und dem Algorithmus 3.1 schnell bemerkt, jedoch können umgekehrt Änderungen von vorher heißen zu kalten Objekten unbemerkt bleiben. Dies liegt daran, daß eine Neuberechnung der Hitze nur bei einem erneuten Zugriff stattfindet. Erfolgt nach einer Laständerung jedoch kein Zu-

griff mehr auf ein vorher heißes Objekt, so behält dieses Objekt fälschlicherweise seine hohe Hitze.

Um dies zu verhindern, werden in [SWZ94] die sogenannten *Pseudozugriffe* vorgeschlagen. Bei diesem Verfahren wird in einem regelmäßigen Zeitraster überprüft, ob ein jetzt stattfindender, fiktiver Zugriff die Hitze des Objekts verringern würde. Ist dies der Fall, so wird die Hitze berechnet, als ob ein Zugriff zu diesem Zeitpunkt erfolgt wäre, und ansonsten behält das Objekt seine alte Hitze. Um zu verhindern, daß Pseudozugriffe zu einer fälschlichen Erhöhung der berechneten Hitze führen, dürfen sie jedoch selbst dann nicht mit ihrem Initiierungszeitpunkt in dem Feld $last\_access\_times_p$ vermerkt werden, wenn sie eine Neuberechnung der Hitze veranlassen.

Diese Vorgehensweise können wir an Hand der Abbildung 3.1 verdeutlichen. Seien a_i reguläre Zugriffe auf das entsprechende Objekt und p_i die Zeitpunkte, zu denen Pseudozugriffe auf dieses Objekt gestartet werden. Nehmen wir ferner an, daß wir LRU-2 betrachten, so ist die lokale Hitze nach Zugriff a_3 – entsprechend der Formel 3.2 – gleich $1/(a_3\text{-}a_1)$. Zum Zeitpunkt p_4 wird nun überprüft, ob dieser Pseudozugriff zu einer Verringerung der Hitze führen würde. Da aber $1/(p_4\text{-}a_3)$ größer als $1/(a_3\text{-}a_1)$ ist, wird dieser Pseudozugriff komplett ignoriert. Zum Zeitpunkt p_5 überprüfen wir erneut den Einfluß eines Pseudozugriffs auf die Hitze. Dieses Mal stellen wir fest, daß er zu einer Verringerung der Hitze führen würde und daher führen wir eine Neuberechnung der Hitze durch $(1/(p_5\text{-}a_3))$. Der Zeitpunkt von p_5 wird jedoch nicht als Zugriffszeitpunkt vermerkt, so daß a_3 immer noch der letzte registrierte Zeitpunkt eines Zugriffs ist. Daher erhalten wir nach dem Zugriff a_6 die korrekte – nur auf den regulären Zugriffen basierende – Hitze $1/(a_6\text{-}a_3)$.

Abbildung 3.1: Veranschaulichung von Pseudozugriffen

Der Pseudocode für diese Methode ist im Algorithmus 3.2 skizziert. Der Dämon wird jeweils in einem festen Zeitraster aufgeweckt und initiiert entsprechend des oben skizzierten Verfahrens für jedes Objekt, für das lokale Hitzeinformationen gespeichert sind, einen Pseudozugriff. Die Grö-

```
void pseudo_demon ()
begin
    loop
        forall (Objekte p, für die lokale Hitzeinformationen existieren) do
            if (no_of_accesses_p < k) then
                loc_heat_p = no_of_accesses_p / now;
            else
                loc_heat_p = min (loc_heat_p, (k-1) / (now - last_access_times_p[k-1]));
            fi;
        od;
        sleep (pseudo_time);
    pool;
end;
```

Algorithmus 3.2: Hitzedämon

ße des Zeitrasters wird durch den Wert der Variablen *Pseudo-Time* bestimmt. Um Hitzeänderungen schnell zu erkennen, sollte der Werte für *Pseudo-Time* möglichst klein gewählt werden, jedoch verbietet der zusätzliche Berechnungsoverhead eine zu extreme Wahl dieses Parameters.

Um eine korrekte LRU-k Approximation durchführen zu können, muß das Feld *last_access_times*$_p$ auch dann lokal alloziert bleiben, wenn das Objekt *p* selbst nicht mehr im lokalen Cache oder auf der lokalen Platte gespeichert ist. Dies führt jedoch dazu, daß für jedes Objekt, auf das irgendwann einmal auf einem Rechner zugegriffen worden ist, Hitzeinformationen gespeichert werden müssen. Um dies zu verhindern, wird in [OOW93] die *Retained-Information-Period* eingeführt. Diese besagt, daß jedes Objekt, auf das innerhalb dieses Zeitraums nicht mehr zugegriffen worden ist, nicht mehr *wichtig* ist. Für solche Objekte können die Hitzeinformationen gelöscht werden, und das Objekt wird bei einem erneuten Zugriff so behandelt, als ob noch nie ein Zugriff erfolgt wäre.

Ähnlich wie bei den Pseudozugriffen, benutzen wir auch hier für die Durchführung dieses Verfahrens einem Hintergrundprozeß, den sogenannten Aufräumdämon, dessen Pseudocode wir in Algorithmus 3.3 skizziert haben. In einem Zeitraster der Größe *Cleanup-Time* werden alle Objekte mit Hitzeinformation überprüft, ob innerhalb der *Retained-Information-Period* ein Zugriff erfolgt ist und falls keiner registriert wurde, wird die entsprechenden Hitzeinformationen gelöscht. Würde das Löschen der Hitzeinformation keine weiteren Aktionen verursachen, so könnten wir die *Cleanup-Time* gleich der *Retained-Information-Period* wählen. Da dieser Zeitraum jedoch im allgemeinen sehr groß ist, sammeln sich bei jedem Durchlauf des Dämons sehr viele Änderungen an. Bei der Berechnung der globalen Hitze in Abschnitt 3.1.2 werden wir sehen, daß jedes Löschen der lokalen Hitzeinformationen potentiell zu einer Nachricht führen kann. Daher ist es in diesem Fall günstiger, einen kleineren Wert für die *Cleanup-Time* zu wählen, um somit die durch den Aufräumdämon verursachte CPU- und Netzbelastungen über die Zeit zu verteilen und größere Pulkankünfte von Nachrichten zu vermeiden. Eine Möglichkeit die Anzahl der Objekte, die innerhalb der Schleife untersucht werden müssen, zu verringern, besteht darin, die Hitzeinformation in einem LRU-Stack zu verwalten. Bei jedem Zugriff wird die entsprechende Hitzeinformation an das höher priorisierte Ende des Stacks gesetzt. Innerhalb der Schleife des Dämons wird nun der Stack in umgekehrter Reihenfolge durchlaufen. Nach dem ersten Objekt, dessen Hitzeinformation nicht gelöscht wird, kann die Schleife bereits abgebrochen werden, da alle anderen Objekte auf jedem Fall einen jüngeren Zugriffszeitpunkt besitzen.

```
void cleanup_demon ()
begin
    loop
        forall (Objekte p, für die lokale Hitzeinformationen existieren) do
            if (now - last_accesses_p[1] > retained_information_period) then
                delete last_accesses_p;
            fi;
        od;
        sleep (cleanup_time);
    pool;
end;
```

Algorithmus 3.3: Aufräumdämon

Den Einfluß, den der Aufräumdämon auf die berechnete Hitze der Objekte nimmt, können wir folgendermaßen verdeutlichen. Alle Objekte mit einer approximierten Hitze größer als (*k*-1)/*Retained-Information-Period* haben mindestens einen Zugriff innerhalb der Retained-Information-Period und werden deshalb von dem Aufräumdämon nicht beeinflußt. Bei Objekten mit einer geringeren Hitze hängt der Einfluß von dem speziellen Referenzmuster ab (siehe Abbildung 3.2). Obwohl beide Objekte unter LRU-3 zum aktuellen Zeitpunkt die gleiche approximierte Hitze besitzen (falls die Pseudozugriffe genügend häufig initiiert werden), wird die Hitzeinformation des Objekts *p* von dem Aufräumdämon gelöscht, und damit wird seine Hitze auf Null gesetzt, während die Hitze des Objekts *q* unverändert bleibt. Da bei einem erneuten Zugriff auf das Objekt *p* die Hitze so berechnet wird, als sei noch nie ein Zugriff auf *p* erfolgt, kann der Hitzedämon nur dazu führen, daß die Hitze von selten zugegriffenen Objekte unterschätzt wird.

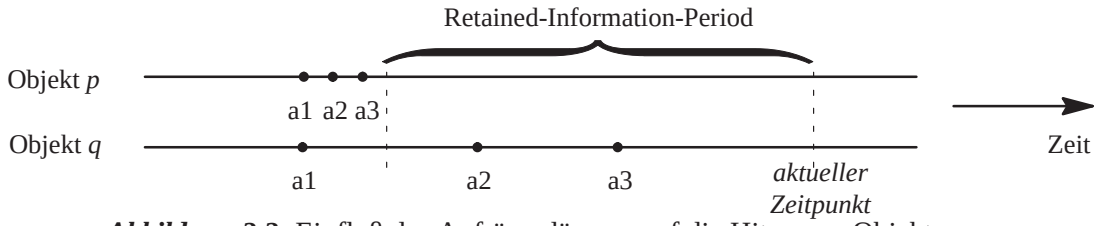


Abbildung 3.2: Einfluß des Aufräumdämons auf die Hitze von Objekten

3.1.2 Verteilte Berechnung der globalen Hitze

Im vorangegangenen Abschnitt haben wir ein Verfahren vorgestellt, das es uns erlaubt, die lokale Hitze eines Objekts auf einem Rechner zu approximieren. Ausgehend von dieser Information wollen wir nun eine Methode herleiten, die uns die Berechnung der globalen Hitze auf dem jeweiligen Heimatrechner erlaubt. Obwohl uns eine solche Methode bereits ermöglichen würde die globale Hitze jeweils bei Bedarf auf dem Heimatrechner zu erfragen, stellt der dafür im allgemeinen notwendige Netzzugriff einen sehr großen Nachteil dar. Daher erweitern wir das Protokoll so, daß die globale Hitze an alle diejenigen Rechner propagiert wird, die diese Information für ihre Cache-Ersetzungsentscheidungen brauchen.

Um einen möglichst allgemeinen Einsatz dieses Verfahrens garantieren zu können, gehen wir von sehr schwachen Annahmen bezüglich des Netzwerks und der Rechner aus. In [Mull93a] wird der Begriff des *asynchronen Systems* definiert, der besagt, daß keine Beschränkung der relativen Geschwindigkeiten der verschiedenen Rechner und auch keine Beschränkung bezüglich der Zeit für das Versenden einer Nachricht existiert. Diese Annahmen erlauben uns heterogene, nicht synchronisierte Rechner zu betrachten, die durch ein Netzwerk verbunden sind, an dem Nachrichten durch Warteschlangeneffekte beliebig verzögert werden können.

Die verteilte Berechnung einer globalen Eigenschaft innerhalb eines solch asynchronen Systems ist ein bekanntes Problem. Eine häufig benutzte Methode zum Lösen solcher Probleme sind Algorithmen, bei denen ein Koordinator alle beteiligten Rechner nach ihren lokalen Systemzuständen fragt, danach wartet bis er von allen eine Antwort bekommen hat, und abschließend den globalen Systemzustand auf Grund der empfangenen lokalen Informationen berechnet. Angewandt auf unser Problem und erweitert um eine Verteilungsphase, erhalten wir das folgende Protokoll, das wir in der Abbildung 3.3 skizziert haben. In dieser Abbildung entsprechen die Pfeile jeweils

einem Nachrichtenaustausch zwischen zwei Rechnern und die Kreuze markieren jeweils eine Änderung der lokalen Hitze des betrachteten Objekts auf dem entsprechenden Rechner, die entweder durch einen regulären oder durch einen Pseudozugriff verursacht wurde.

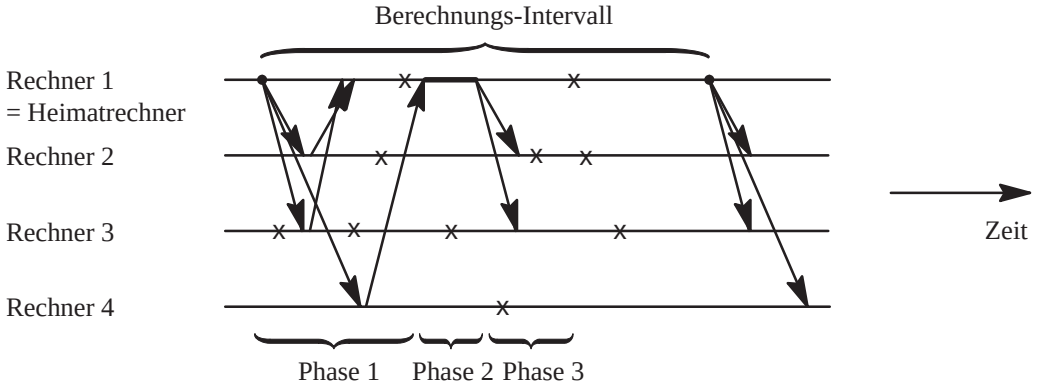


Abbildung 3.3: Synchrones Protokoll zur Berechnung der globalen Hitze. Wir nehmen an, daß die Rechner 1, 2 und 3 das Objekt im Cache besitzen, während das Objekt auf dem Rechner 4 nicht lokal verfügbar ist. Die Änderungen auf Rechner 4 resultieren nur aus Pseudozugriffen.

Ein synchrones Protokoll:

(1) **Sammelphase:** In dieser Phase sendet der Heimatrechner, der die Koordination des Verfahrens übernimmt, eine Anfrage an alle Rechner, und nach dem Empfang der Nachricht antworten die einzelnen Rechner mit der jeweils aktuellen lokalen Hitze des entsprechenden Objekts.

(2) **Berechnungsphase:** Der Heimatrechner kennt nun die lokalen Hitzen auf den verschiedenen Rechnern und kann daher die globale Hitze als die Summe der einzelnen lokalen Hitzen berechnen.

Durch die endliche Übertragungsgeschwindigkeit des Netzwerks kann es jedoch vorkommen, daß die aktuelle lokale Hitze auf den anderen Rechnern von der auf dem Heimatrechner bekannten Hitze abweicht. Obwohl somit die berechnete globale Hitze nicht unbedingt der Summe der aktuellen lokalen Hitzen entspricht, ist jedoch immer gewährleistet, daß die berechnete globale Hitze zumindest zu einem konsistenten Systemzustand [Mull93a], d.h. zu einem Zustand, der die Kausalität berücksichtigt, korrespondiert.

Damit ist das Protokoll zur Bestimmung der globalen Hitze bereits beendet, jedoch können wir aus Gründen der schnelleren Zugreifbarkeit dieser Information in einer dritten Phase noch den auf dem Heimatrechner berechneten Wert an andere Rechner propagieren.

(3) **Verteilungsphase:** Der Heimatrechner versendet die berechnete globale Hitze nur an diejenigen Rechner, die diese Information benötigen. In Abschnitt 3.3 und 3.4 werden wir sehen, daß dies nur die Rechner sind, die das Objekt im lokalen Cache halten.

Eine analoge Einschränkung der beteiligten Rechner wie bei der Verteilungsphase können wir bei der Sammelphase nicht durchführen, da sich – wir in Abschnitt 3.1.1 gesehen haben – die lokale Hitze eines Objekts auch dann noch durch Pseudozugriffe ändern kann, wenn das Objekt selbst nicht mehr im lokalen Cache gehalten wird. Aus diesem Grund müssen wir in der Sammelphase die lokale Hitze des Objekts auf allen Rechnern abfragen.

Obwohl dieses Verfahren in der beschriebenen Art korrekt funktioniert, besitzt es doch einen entscheidenden Nachteil: Das Protokoll berücksichtigt nicht die Anzahl der lokalen Hitzeänderungen auf den verschiedenen Rechnern. Um trotzdem genügend schnell auf Hitzeänderungen reagieren zu können, darf der Abstand zwischen zwei Runden nicht zu groß gewählt werden. Dies führt jedoch zu einem beträchtlichen Mehraufwand an Nachrichten insbsondere bei Objekten, die sehr selten referenziert werden. Während man dieses Problem noch durch separate Berechnungsintervalle für jedes Objekt zumindest prinzipiell in den Griff bekommen könnte, ist dies nicht möglich, wenn sich die Anzahl der lokalen Hitzeänderungen für das selbe Objekt auf unterschiedlichen Rechnern stark unterscheidet. Aus diesem Grund wollen wir nun ein asynchrones Protokoll vorstellen, das jeweils durch lokale Hitzeänderungen aktiviert wird. Auch hier können wir – analog zu dem synchronen Protokoll – drei verschiedene Teile identifizieren, die jedoch nicht mehr strikt nacheinander, sondern verschränkt ablaufen können. Eine exemplarischer Ablauf dieses Verfahrens ist in Abbildung 3.4 skizziert.

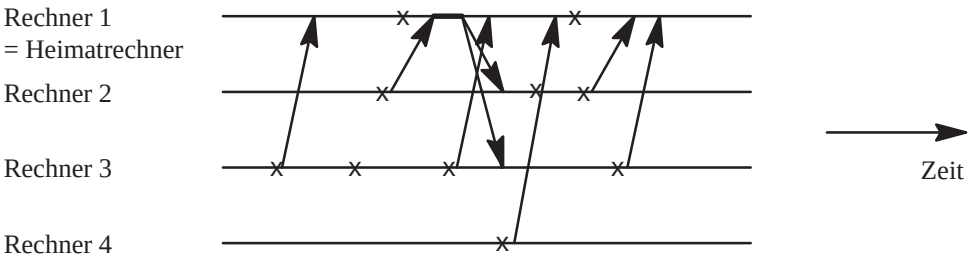


Abbildung 3.4: Asynchrones Protokoll zur Berechnung der globalen Hitze. Auch hier gehen wir wieder davon aus, daß Rechner 4 das Objekt werde im Cache noch auf Platte besitzt.

Ein asynchrones Protokoll:

(1) **Propagierung der lokalen Hitze:** Nachdem ein lokaler Zugriff (entweder ein regulärer oder ein Pseudozugriff) auf das Objekt *p* erfolgt ist, wird die lokale Hitze auf dem Rechner neu bestimmt. Ist die gesamte relative Hitzeänderung seit der letzten Benachrichtigung an den Heimatrechner größer als ein spezifizierter Schwellwert Θ_{local}, so sendet der Rechner eine Nachricht mit der lokalen Änderung der Hitze an den Heimatrechner. Ist die Änderung jedoch kleiner, so wird keine weitere Aktion ausgelöst.

(2) **Berechnung der globalen Hitze:** Empfängt der Heimatrechner eine Änderungsnachricht, so kann er inkrementell die neue globale Hitze bestimmen, indem er einfach den Wert der Änderung auf die globale Hitze aufaddiert.

Auch hier dient die dritte Phase wiederum nur der Verteilung der berechneten globalen Hitze auf andere Rechner. Im Gegensatz zu dem vorherigen Algorithmus benutzen wir auch hier einen Schwellwert, um die Anzahl der Nachrichten gering zu halten. Dies ist bei diesem Protokoll wichtig, da im Gegensatz zu dem synchronen Protokoll wesentlich mehr Neuberechnungen der globalen Hitze durchgeführen werden müssen.

(3) **Verteilung der globalen Hitze:** Der Heimatrechner merkt sich den Wert der globalen Hitze, über den er als letztes alle Rechner, die das Objekt im Cache halten, benachrichtigt hat. Weicht der aktuelle Wert der globalen Hitze relativ um mehr als den Schwellwert Θ_{global}

von dieser gemerkten Hitze ab, so werden diese Rechner über die neue globale Hitze informiert.

Der Nachteil dieses Protokolls ist, daß im Gegensatz zu dem synchronen Protokoll, selbst in dem Fall eines stabilen Systems, die berechnete globale Hitze von der exakten Summe der lokalen Hitzen abweichen kann. Zur Abschätzung des maximalen Fehlers nehmen wir an, daß $glob\_heat^{exact}$ die Summe der zum aktuell betrachteten Zeitpunkt durch LRU-k approximierten lokalen Hitzen ($loc\_heat_i{}^{curr}$) und $glob\_heat$ die auf dem Heimatrechner berechnete globale Hitze ist. Nehmen wir weiterhin an, daß $loc\_heat_i{}^{old}$ die lokale Hitze auf dem Rechner i ist, über die der Heimatrechner zuletzt informiert wurde, so können wir den Fehler auf dem Heimatrechner in Abhängigkeit von den Schwellwerten folgendermaßen bestimmen:

$$\begin{aligned}
|glob_heat^{exact} - glob_heat| &= \left| \sum_{i=1}^{N} loc_heat_i^{curr} - \sum_{i=1}^{N} loc_heat_i^{old} \right| \\
&= \left| \sum_{i=1}^{N} \left(loc_heat_i^{curr} - loc_heat_i^{old} \right) \right| \\
&\leq \sum_{i=1}^{N} |loc_heat_i^{curr} - loc_heat_i^{old}| \\
&\leq \Theta_{local} \times \sum_{i=1}^{N} loc_heat_i^{curr} \\
&= \Theta_{local} \times glob_heat^{exact} \qquad (3.3)
\end{aligned}$$

Während sich auf dem Heimatrechner nur die Fehler beim Sammeln der lokalen Hitzen auswirken, führt auf den anderen Rechnern die schwellwertgesteuerte Verteilung der globalen Hitze zu einem weiteren Fehler. Die Berechnung des Gesamtfehlers auf diesen Rechnern verläuft analog zur Herleitung der Formel 3.3, und wir erhalten als Abschätzung für den maximalen Fehler in diesem Fall:

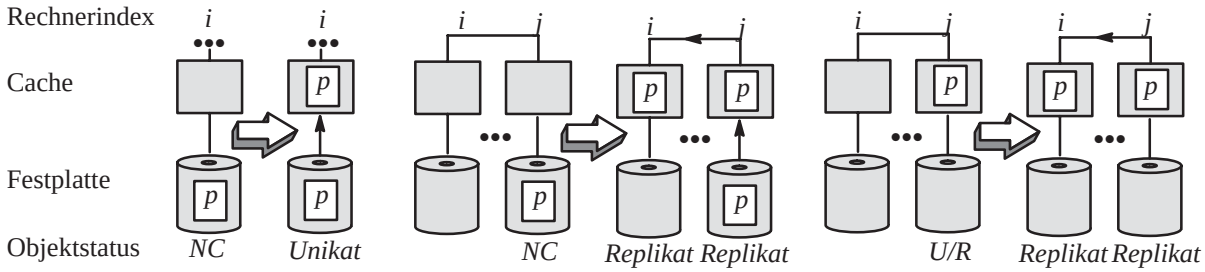
$$|glob\_heat^{exact} - glob\_heat_i| \leq \left(\Theta_{local} + \Theta_{global} + \Theta_{local} \times \Theta_{global} \right) \times glob\_heat^{exact}$$

3.1.3 Ein Protokoll zur Bestimmung und Verteilung des Objektstatus

Einige Heuristiken , die wir später beschreiben werden, benötigen den Objektstatus. Dieser beschreibt die Häufigkeit eines Objekts im aggregierten Cache. Wie wir noch sehen werden, ist jedoch nicht das Wissen über die genaue Anzahl der Replikate erforderlich, sondern es genügt die Charakterisierung des Objektstatus durch die folgenden drei Werte:

- **Nicht-im-Cache:** Das Objekt befindet sich in keinem der Caches des verteilten Systems.
- **Unikat:** Das Objekt befindet sich in dem Cache genau eines Rechners.
- **Replikat:** Das Objekt befindet sich in den Caches mehrerer Rechner.

Bei der Einführung des Begriffs “Heimatrechner”, haben wir bereits gesagt, daß dieser Rechner über die Anzahl und Plazierung der Cache-Kopien des betreffenden Objekts informiert ist. Daher



(a) Zugriff auf die lokale Platte

(b) Zugriff auf eine nichtlokale Platte

(c) Zugriff auf einen nichtlokalen Cache

Abbildung 3.5: Bestimmung des Objektstatus bei einem Lesezugriff

können wir auf diesem Rechner den Status eines Objekts sehr einfach aus dieser Information ableiten. Ähnlich wie die globale Hitze wird jedoch auch der Status eines Objekts von einigen unserer Heuristiken benutzt, um ein Ersetzungsopfer zu bestimmen. Daher wollen wir auch hier ein Protokoll vorstellen, das mit geringem Zusatzaufwand die Information über den Status eines Objekts an alle relevanten Rechner verteilt. Diese replizierte Information verhindert, daß sich nichtlokale Anfragen an den Heimatrechner im kritischen Pfad einer Cache-Ersetzung befinden und dadurch die Anwortzeit von Zugriffen verschlechtern.

Der Status eines Objekts kann sich nur ändern, wenn das Objekt entweder in den Cache neu eingefügt wird oder wenn es aus einem Cache entfernt wird. Betrachten wir zuerst die Fälle, in denen ein Objekt *p* in den Cache des Rechners *i* eingefügt wird. Die verschiedenen möglichen Aktionen, die zu einer solchen Einfügung führen können, sind in Abbildung 3.5 skizziert. Hierbei ist jeweils links von dem Pfeil der Ausgangszustand und rechts der Zustand nach dem Zugriff von Rechner *i* auf das Objekt *p* gezeigt. Die Fälle (a) und (b), in denen das Objekt von Platte gelesen wird, können jeweils nur auftreten, wenn das Objekt in keinem der Caches vorhanden ist, da ansonsten entsprechend der Zugriffshierarchie ein Cache-Zugriff dem Festplattenzugriff vorgezogen würde. Daher muß der Status des Objekts vor dem Zugriff in diesen beiden Fällen jeweils *Nicht-im-Cache* (*NC*) sein. Im Fall (a) weiß der Rechner *i*, daß er nach dem Einfügen des Objekts in den Cache die einzige Cache-Kopie von *p* besitzt und kann daher den Status auf *Unikat* setzten.

Im Fall (b) weiß der Rechner *j*, daß er nach dem Einfügen des Objekts in seinen lokalen Cache, das Objekt *p* zusätzlich auch noch an den Rechner *i* sendet, der es seinerseits in seinen Cache einfügt. Daher setzt er lokal den Status auf *Replikat*. Rechner *i* empfängt in diesem Fall das Objekt über das Netzwerk und weiß, daß zumindest noch eine Cache-Kopie bei dem Sender des Objekts existiert, und entsprechend setzt Rechner *i* den Status dieses Objekts ebenfalls lokal auf *Replikat*.

Ähnlich ist die Situation im Fall (c). Bereits vor dem Zugriff des Rechners *i* existiert mindestens eine Kopie des Objekts im Cache, wobei dieses Objekt entweder ein *Unikat* oder *Replikat* sein kann. Unabhängig von dem aktuellen Status auf dem Rechner *j*, existieren auf jeden Fall nach dem Versenden des Objekts *p* an den Rechner *i* mindestens zwei Cache-Kopien, und damit ist der Status auf dem Rechner *j* ein *Replikat*. Analog zu dem Fall (b) empfängt auch hier Rechner *i* wiederum das Objekt über das Netzwerk und weiß damit, daß das Objekt mindestens zweimal im aggregierten Cache existieren muß; daher setzt Rechner *i* den lokalen Status des Objekts *p* ebenfalls auf *Replikat*.

Bei dieser Beschreibung haben wir gesehen, daß jeder Rechner lokal den Status eines neu gelesenen Objekts bestimmen kann, ohne daß eine zusätzliche Nachricht versendet werden muß. Um jedoch die Buchhaltungsinformation auf dem Heimatrechner über die neue Cache-Kopie und den eventuell damit verbundenen Statuswechsel zu informieren, muß jede Einfügeoperation eine Benachrichtigung des Heimatrechners durchführen.

Ebenso wie bei den Einfügeoperationen muß auch bei Löschoperationen die Buchhaltungsinformation auf dem Heimatrechner aktualisiert werden. Auch dies erreichen wir wiederum durch eine Nachricht an den Heimatrechner. Empfängt der Heimatrechner eine solche Nachricht, so wird die Liste der Rechner, die eine Cache-Kopie halten, entsprechend geändert. Ist die Anzahl der Rechner mit einer Cache-Kopie nach der Änderung immer noch größer als eins, so ist das Objekt auch weiterhin ein *Replikat*, und wir brauchen keine weiteren Aktionen durchzuführen. Ähnlich verhält es sich, wenn nach der Aktualisierung kein Rechner mehr eine Kopie besitzt. Zwar ändert sich der Status in diesem Fall von *Unikat* zu *Nicht-im-Cache*, da jedoch der einzige Rechner, der diese Änderung durchführen muß, diese bereits beim Löschen des Objekts erkennt, brauchen wir auch in diesem Fall keine zusätzliche Nachricht zu versenden. Somit bleibt nur noch der Fall, in dem nach der Aktualisierung der Liste noch exakt ein Rechner übrig bleibt. In diesem Fall sendet der Heimatrechner eine entsprechende Nachricht an den Rechner, der die letzte Cache-Kopie besitzt, und dieser ändert beim Empfang der Nachricht den Status des entsprechenden Objekts von *Replikat* auf *Unikat*.

In Abbildung 3.6 ist ein möglicher Ablauf dieses Protokolls zur Bestimmung des Objektstatus aufgezeigt. Da wir in diesem Beispiel davon ausgehen, daß der Heimatrechner die einzige Plattenkopie des Objekts besitzt, kann der Zugriff (a) lokal erfolgen. Bei Zugriff (b) muß der Rechner 2 zuerst bei dem Heimatrechner nachfragen (*Req*), da er selbst weder eine Cache- noch eine Plattenkopie besitzt. Da die Cache-Kopie auf dem Heimatrechner bereits gelöscht wurde, muß sie erneut von Platte gelesen werden. Die Daten werden nun an den Initiator zurückgesendet (*Obj*), und da zwei Cache-Kopien im System existieren, setzen beide den lokalen Status des Objekts auf Replikat. Wird auf dem Rechner 1 nun wiederum die Cache-Kopie gelöscht, so wird erkannt, daß nur noch ein anderer Rechner eine Cache-Kopie besitzt. Diesem wird daher eine Unikat-Nachricht (*Uni*) gesendet, damit er lokal den Status des Objekts aktualisieren kann. Schließlich illustriert den Zugriff (c) den Fall, in dem eine Anfrage an einen anderen Rechner weitergeleitet wird, damit das Objekt dort aus dem Cache gelesen werden kann. Bevor der Heimatrechner den Zugriff

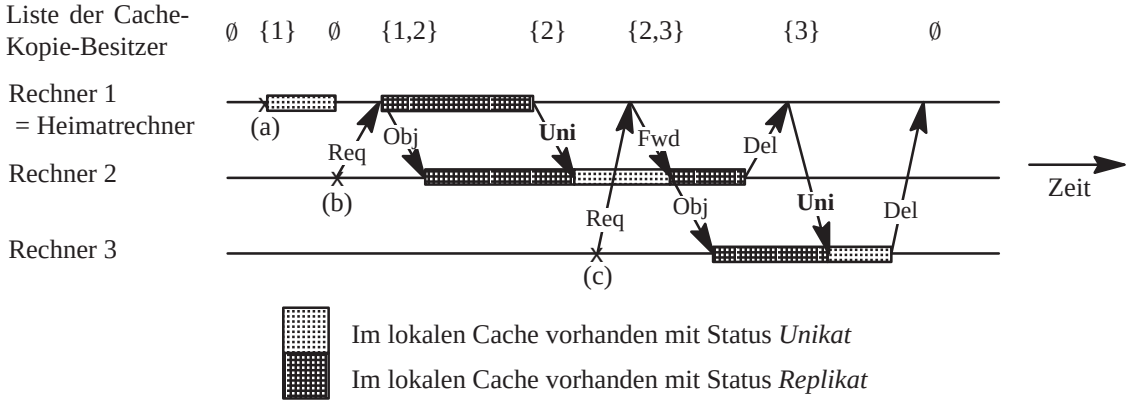


Abbildung 3.6: Protokollierung des Objektstatus

an den Rechner weiterleitet, der das Objekt im Cache besitzt (*Fwd*), wird die Liste der Cache-Kopie-Besitzer aktualisiert. Wie wir bereits weiter oben argumentiert haben ist es auch notwendig den Heimatrechner über Löschaktionen zu informieren. Dies wird in unserem Beispiel jeweils durch die Nachricht (*Del*) erreicht.

3.1.4 Dynamische Schätzung der Zugriffskosten

In Abschnitt 2.2 haben wir gesehen, daß der Algorithmus zur exakten Berechnung einer optimalen statischen Allokation die Information über die Zugriffskosten benötigt. Während wir jedoch dort ein globales Wissen voraussetzen konnten, müssen wir bei der dynamischen Heuristik diese Werte mit Hilfe der aktuell verfügbaren Informationen approximieren. In diesem Abschnitt werden wir zunächst beschreiben, wie wir die Zugriffskosten einer Speicherstufe für bereits beendete Zugriffe bestimmen können. Anschließend stellen wir zwei Methoden vor, die es uns erlauben, aus dieser Information die zu erwartenden Kosten für einen zukünftigen Zugriff abzuleiten. Betrachten wir ein System, bei dem jeder Rechner eine eigene Platte besitzt, so müssen wir im allgemeinsten Fall die Kosten für die folgenden Terme schätzen können:

- $c_{lc,s}(p)$: Die Zugriffskosten auf das Objekt p im lokalen Cache des Rechners s.
- $c_{rc,s,z}(p)$: Die Zugriffskosten des Rechners s auf das Objekt p im Cache des Rechners z. Hier muß gelten, daß die Rechner s und z verschieden sind.
- $c_{ld,s}(p)$: Die Zugriffskosten auf das Objekt p auf der lokalen Festplatte des Rechners s.
- $c_{rd,s,z}(p)$: Die Zugriffskosten des Rechners s auf das Objekt p auf der Platte des Rechners z. Auch hier fordern wir wiederum, daß Rechner s und z verschieden sind.

Da jedoch die Bestimmung, Speicherung und Verteilung all dieser Werte einen zu großen Aufwand verursachen würde, werden wir in diesem Abschnitt einige Vereinfachen einführen. Bevor wir jedoch auf diese genauer eingehen stellen wir eine Methode zur Protokollierung der Kosten von gerade beendeten Zugriffen vor.

Zur Bestimmung der Zugriffskosten auf die verschiedenen Stufen der Speicherhierarchie, speichern wir für jeden lokal initiierten Zugriff die Antwortzeit und die Hierarchiestufe, auf welcher der Zugriff bedient wurde. Dazu merken wir uns den Startzeitpunkt des Zugriffs, und können dann nach der Lieferung des Objekts die Antwortzeit als die Differenz zwischen der aktuellen und der Startzeit bestimmen. Die Bestimmung der Hierarchiestufe, auf welcher der Zugriff bedient wurde, ist ebenfalls sehr einfach möglich. Der Rechner, der den Zugriff durchgeführt hat, weiß von welcher Stufe er das Objekt gelesen hat und kann daher diese Information zusammen mit dem Objekt an den Initiator zurücksenden.

Dieser Methode erlaubt jedoch nur Kosten für solche Zugriffe zu bestimmen, die lokal initiiert worden sind. Zwar wäre es möglich die lokalen Werte auch explizit zwischen den Rechnern zu verteilen und damit überall verfügbar zu machen, jedoch werden wir bei der Beschreibung der Heuristiken argumentieren, daß dieser Mehraufwand nicht benötigt wird. Somit reicht es aus, wenn wir auf jedem Rechner $\boldsymbol{i}$ die Kosten $c_{lc,\boldsymbol{i}}(p)$, $c_{rc,\boldsymbol{i},z}(p)$, $c_{ld,\boldsymbol{i}}(p)$ und $c_{rd,\boldsymbol{i},z}(p)$ bestimmen können. Unter der Annahme, daß sich die Kosten innerhalb des Systems nicht sprunghaft ändern,

können wir die zu bestimmenden Kosten aus den Antwortzeiten früherer Zugriffe ableiten. Um den Einfluß von stochastischen Schwankungen auf unsere Schätzung zu reduzieren, werden wir nach der Beschreibung weiterer Vereinfachungen zwei Methoden vorstellen, die eine Historie von antwortzeiten auf die entsprechende Hierarchiestufe berücksichtigen.

Vereinfachungen

Bis jetzt sind wir davon ausgegangen, daß für jedes Objekt die Bestimmung der Zugriffskosten separat erfolgt. Gehen wir davon aus, daß nur die Größe und nicht der Inhalt eines Objekts Einfluß auf die Zugriffskosten nimmt, so können wir die sehr kostspielige Buchhaltung auf Objektebene auf eine sehr viel günstigere und damit skalierbare rechnerorientierte Buchhaltung umstellen. Nehmen wir an, daß die Kosten jeweils proportional zu der Größe des Objekts sind, so können wir die Kosten für einen Zugriff des Rechners s auf das Objekt p der Größe s_p auf der Hierarchiestufe m des Rechners z durch die folgende Gleichung approximieren:

$$c_{m,s,z}(p) = var_{m,s,z} \times s_p + fix_{m,s,z} \tag{3.4}$$

Bei dieser Gleichung gehen wir davon aus, daß für m, s und z nur solche Kombinationen zugelassen werden, die wir bereits oben beschrieben haben. Durch diese Vereinfachung braucht jetzt jeder Rechner nur noch Schätzungen für die verschiedenen Paare der objektunabhängigen Koeffizienten $var_{m,s,z}$ und $fix_{m,s,z}$ herzuleiten.

Eine zusätzliche Vereinfachung und eine Reduktion des Overheads können wir dadurch erreichen, daß wir die Anzahl der Kombinationen aus Speicherhierarchie und beteiligten Rechnern reduzieren indem wir alle nichtlokalen Caches zu einer einzigen Stufe zusammenfassen. Dies ist gerechtfertigt, wenn wir annehmen, daß die Übertragungskosten zwischen zwei Rechnern für alle Paare von Rechnern gleich sind und wenn zusätzlich gilt, daß die lokalen CPUs, die für die Abarbeitung des Kommunikationsprotokolls zuständig sind, keine Engpäße bilden. Die erste Annahme wird näherungsweise von allen Bus-Netzen erfüllt, da hier alle Rechner durch das gleiche Medium miteinander verbunden sind. Durch die Entwicklung neuer, effizienter Netzwerkprotokolle [PLC95, RAC97, WBE97] und die steigende Leistungsfähigkeit der CPUs [HePa96] wird der Einfluß der CPUs auf die Kommunikationskosten immer geringer. Somit können wir im folgenden annehmen, daß die Kosten für nichtlokale Cachezugriffe unabhängig von den beteiligten Rechnern sind und nur von der Objektgröße abhängen:

$c_{rc}(p) = c_{rc,i,z}(p)$ für alle Rechner i und z.

Analog zu der eben angeführten Vereinfachung könnten wir nun auch versuchen die Kosten für die unterschiedlichen nichtlokalen Festplatten zusammenzufassen. Dies ist jedoch nicht zu rechtfertigen, da sich die mittleren Kosten für die Zugriffe auf die verschiedenen Festplatten sehr stark unterscheiden können. Der Grund hierfür liegt darin, daß wir in unserem Modell keine Annahmen bezüglich der Datenallokation auf Platte machen. Daher können sich die Zugriffshäufigkeiten und damit auch die Auslastungen der verschiedenen Platten sehr stark unterscheiden. Betrachten wir jedoch die obige Argumentation für das Zusammenfassen der nichtlokalen Caches, so können wir hier mit der selben Begründung rechtfertigen, daß die Kosten für einen nichtlokalen Festplattenzugriff unabhängig von dem Initiatorrechner sind. Somit gilt:

$c_{rd,z}(p) = c_{rd,i,z}(p)$ für alle Rechner i.

Nachdem wir diese Vereinfachungen besprochen haben, wollen wir nun auf das Problem zurückkommen, wie wir die Koeffizienten für die lineare Interpolation der Gleichung (3.4) aus bereits vorhandenen Meßwerten ableiten können. Da dieses Verfahren für alle benötigten lokalen Kostenschätzungen identisch ist, werden wir die Vorgehensweise für den allgemeinen Fall beschreiben. Es sollte jedoch berücksichtigt werden, daß – gemäß der soeben eingeführten Vereinfachungen – nicht alle mögliche Kombinationen auch wirklich gebraucht werden.

Schätzung durch exponentielle Glättung

Bei diesem ersten Verfahren werden nach der Beendigung eines Zugriffs auf ein Objekt q, welches sich auf der Stufe m der Speicherhierarchie befunden hat, die Koeffizienten zur Kostenschätzung aus Formel (3.4) entsprechend der folgenden Vorschrift berechnet:

$$\begin{aligned} var_{m,s,z} &= \gamma \times \frac{t_q}{s_q} + (1-\gamma) \times var_{m,s,z} \quad \text{mit } \gamma \in (0,1) \\ fix_{m,s,z} &= 0 \end{aligned} \tag{3.5}$$

Hierbei sind t_q die Antwortzeit des gerad beendeten Zugriffs und s_q die Größe des Objekts q. Die Variable γ ist ein Gewichtungsfaktor, der bestimmt, wie stark die Kosten früherer Zugriffe bei der Berechnung des neuen Wertes mit berücksichtigt werden sollen. Große Werte von γ bewirken, daß die Historie nur einen sehr geringen Einfluß auf die neue Berechnung hat, und somit erlauben solche Werte eine schnelle Adaption auf sich ändernde Ressourcenauslastungen. Nachteilig bei zu großen Werten ist jedoch, daß die Glättung von stochastischen Schwankungen relativ gering ist. Entsprechend umgekehrt verhält sich die Situation für kleine Werte von γ.

Die Vorteile dieses Verfahrens liegen in seinem sehr geringen Berechnungsaufwand und in dem geringen Speicherplatzbedarf. So muß auf jedem Rechner für jede Stufe der Speicherhierarchie nur eine einzige Variable reserviert werden.

Nachteilig ist jedoch, daß dieses Verfahren es nicht ermöglicht, die Kosten in einen konstanten und in einen variablen Anteil aufzuspalten, sondern daß implizit angenommen wird, daß der konstante Anteil immer gleich Null ist. Hierdurch ist lediglich eine exakte Approximiation der Kosten für Ressourcen möglich, deren Zugriffszeit nur aus einer größenabhängigen Komponente besteht. Bei Ressourcen mit einer größenunabhängigen Komponente in der Zugriffszeit in Kombination mit variabel großen Objekten kann diese Beschränkung zu sehr großen Fehlern in der Berechnung führen. Daher sollte dieses Verfahren nur in solchen Umgebungen eingesetzt werden, in denen alle Objekte die gleiche Größe haben oder der Einfluß von größenunabhängigen Kosten vernachlässigbar ist.

Schätzung durch Minimierung des Fehlerquadrates

Im Gegensatz zu der vorangegangenen Methode merken wir uns jetzt für jede Stufe der Speicherhierarchie die Antwortzeit und die Objektgröße für die letzten w Zugriffe auf die entsprechende Stufe. Mit Hilfe dieser Information können wir die Koeffizienten der Geraden bestimmen, die das Fehlerquadrat zwischen den gegebenen Werten und einer Geraden minimiert.

Benutzen wir die Gaußsche Summenschreibweise:

$$[f(x)] \stackrel{def}{=} \sum_{i=0}^{w} f(x_i),$$

so können wir die Koeffizienten der Geraden nach den folgenden Formeln bestimmen [BrSe91]:

$$var_{m,s,z} = \frac{w \times [s_{m,s,z}t_{m,s,z}] - [s_{m,s,z}] \times [t_{m,s,z}]}{w \times [s^2_{m,s,z}] - [s_{m,s,z}]^2}$$
$$fix_{m,s,z} = \frac{[t_{m,s,z}] \times [s^2_{m,s,z}] - [s_{m,s,z}t_{m,s,z}] \times [s_{m,s,z}]}{w \times [s^2_{m,s,z}] - [s_{m,s,z}]^2} \quad (3.6)$$

Hierbei sind $s_{m,s,z}$ und $t_{m,s,z}$ jeweils Felder, die die Größen bzw. die Kosten der w letzten Zugriffe des Rechners s auf die Stufe m des Rechners z enthalten. Um ein definiertes Ergebnis für die Formeln zu erhalten, müssen wir fordern, daß nicht alle Einträge innerhalb des Feldes $s_{m,s,z}$ gleich sind. Im Gegensatz zu der oben vorgestellen Methode der exponentiellen Glättung ist diese Methode daher nur für Umgebungen mit variabel großen Objekten geeignet. Da alle Einträge innerhalb der Felder bei der Berechnung gleich behandelt werden – also insbesondere keine Gewichtung bezüglich ihrer Aktualität erfolgt – können wir die Felder als Ringpuffer implementieren, so daß die alten Werte jeweils zyklisch durch neue Werte überschrieben werden.

Ähnlich wie der Parameter γ bei der Methode des exponentiellen Glättens, bestimmt die Wahl der Fenstergröße w die Adaptivität und das Glättungsvermögen des Algorithmus. Kleine Werte für w bewirken eine schnelle Adaptivität, reagieren jedoch auch stärker auf stochastisches Rauschen.

Der Parameter w bestimmt auch den Overhead, da er die Anzahl der Werte festlegt, die sich der Algorithmus merken muß. Bei einer simplen Berechnung der Formel 3.6 ist zusätzlich die Laufzeit von der Fenstergröße w linear abhängig. Berücksichtigen wir jedoch, daß bei jeder Neuauswertung nur die Werte für den ältesten Zugriff durch die Werte für den gerade beendeten Zugriff ersetzt werden, so können wir die Berechnung jeweils inkrementell und in konstanter Zeit durchführen.

3.2 Egoistische Heuristiken

Nachdem wir in den vorangegangenen Abschnitten die Protokolle zum Berechnen und Verteilen der notwendigen Informationen erklärt haben, können wir jetzt die unterschiedlichen heuristischen Online-Algorithmen für verteiltes Caching vorstellen. In diesem Abschnitt beschreiben wir zwei unterschiedliche Varianten der egoistischen Heuristik. Analog zu dem egoistischen Verfahren aus Abschnitt 2.3.1 ist auch hier das Optimierungsziel die Maximierung der lokalen Cache-Trefferrate. Die Entscheidung, welche Objekte im lokalen Cache gehalten werden sollen und welche als Ersetzungsopfer in Frage kommen, wird also nur auf Grund ihrer lokalen Wichtigkeit getroffen. Die beiden Heuristiken unterscheiden sich in den Methoden zur Abschätzung dieser lokalen Wichtigkeit.

3.2.1 Eine LRU-basierte egoistische Heuristik

Bei dieser Methode verwaltet jeder Rechner die im lokalen Cache vorhandenen Objekte in einem Stack. Jedes Mal, wenn ein Zugriff auf ein vorhandenen Objekt erfolgt, wird der korrespondierende Eintrag an das höher priorisierte Ende des Stacks bewegt. Ebenso wird bei einem neu ein-

gefügten Objekt ein entsprechender Eintrag generiert und an das wertvollere Ende angehängt. Da jedoch die Größe des Caches begrenzt ist, müssen beim Einfügen eines neuen Objekts im allgemeinen ein odere mehrere Objekte aus dem lokalen Cache entfernt werden. Entsprechend der LRU-Strategie werden dabei die Objekte als Ersetzungsopfer ausgesucht, die sich am wertloseren Ende des Stacks befinden. Nach dem Löschen der Ersetzungsopfer werden auch die korrespondierenden Einträge aus dem Stack entfernt.

Ein Problem besteht darin, daß ein Rechner auch Objekte auf Platte besitzen kann, die kein anderer Rechner im System besitzt. Fordert nun ein anderer Rechner ein solches Objekt an, so können wir zwei Fälle unterscheiden. Befindet sich das angeforderte Objekt schon im Cache, so kann dieser nichtlokale Zugriff aus dem Cache heraus beantwortet werden, wobei wir – im Gegensatz zu lokalen Zugriffen – die Position des Objekts im LRU-Stack nicht verändern. Dies verhindert, daß nichtlokale Zugriffe auf Cache-Kopien die lokale Rangfolge des LRU-Stacks verändern. Anders ist jedoch die Situation, wenn sich das Objekt nicht im Cache befindet, sondern zuerst noch von Platte geladen werden muß. Da das Versenden eines Objekts nur möglich ist, wenn es vorher in den Cache geladen wurde, müssen wir im allgemeinen Objekte aus dem lokalen Cache entfernen, um Platz für dieses neue Objekt zu schaffen. Dadurch nehmen jedoch nichtlokale Zugriffe Einfluß auf die lokalen Cache-Entscheidungen. Um diesen Einfluß möglichst gering zu halten, haben wir das Verfahren der sogenannten *Hate-Hints* aus [FCL92, Fran96] übernommen. Bei diesem Verfahren werden Objekte, die von anderen Rechnern angefordert werden, am unteren Ende des Stacks positioniert und werden somit als Opfer für nachfolgende Ersetzungen bevorzugt betrachtet.

Für die Benutzung von LRU sprechen die einfache Implementierung und die konstante Berechnungskomplexität pro Operation. Bei der Bestimmung der Position innerhalb des Stacks und damit der Wichtigkeit eines Objekts wird jedoch die Größe der Objekte nicht mit berücksichtigt. Dies macht bei Umgebungen mit konstant großen Objekten nichts aus, da hier jedes Objekt die gleichen Speicherkosten hat; in Umgebungen mit variabel großen Objekten kann dies jedoch zu einer suboptimalen lokalen Cache-Trefferrate führen.

3.2.2 Eine temperaturbasierte egoistische Heuristik

Bei dieser zweiten Methode benutzen wir an Stelle der LRU-Ersetzungsstrategie die in Abschnitt 3.1.1 vorgestellte Methode, um die lokale Wichtigkeit eines Objekts zu schätzen. Analog zu dem oben skizzierten Verfahren verwaltet auch hier jeder Rechner seinen Cache unabhängig von allen anderen Rechner. Im Gegensatz zu der Heuristik aus Abschnitt 3.2.1, in der wir nur qualitative Aussagen über die Wichtigkeit der Objekte machen können, liefert uns LRU-k mit der lokalen Hitze ein quantitatives Maß für die Berechnung der Wichtigkeit. Um neben dem Objektwert auch die Speicherkosten für ein Objekt zu berücksichtigen, entfernen wir bei einer Cache-Ersetzung jeweils die Objekte mir der geringsten Temperatur. Eine Prioritätswarteschlange ermöglicht uns die Objekte jeweils entsprechend ihrer Temperatur sortiert zu halten und dadurch Ersetzungsopfer sehr schnell zu bestimmen.

Auch bei diesem Verfahren tritt – ähnlich wie bei dem LRU-basierten Verfahren – das Problem auf, daß nichtlokale Zugriffe die lokale Cache-Ersetzung beeinflußen können. Um auch hier den

Einfluß dieser Zugriffe möglichst gering zu halten, benutzen wir die folgende Methode. Während die Zeitpunkte von lokalen Zugriffen gemerkt werden, um daraus die lokale Hitze abzuleiten, werden die Zugriffszeitpunkte von nichtlokalen Zugriffen ignoriert. Werden nun Objekte von nichtlokalen Rechner angefordert, die erst von Platte geladen werden müssen, so werden diese Objekte entsprechend ihrer lokalen Temperaturzwar in den Cache eingefügt. Da jedoch diese Temperatur geringer sein muß als die Temperatur aller anderen Objekte im lokalen Cache (ansonsten befände sich das Objekt bereits darin), wird dieses Objekt bei der nächsten Einfügeoperation wieder gelöscht.

Vorteilhaft an dieser Methode ist, daß wir nun ein quantitatives Maß für die Wichtigkeit eines Objekts haben, welches auch die lokalen Speicherkosten mit berücksichtigt. Daher ist dieses Verfahren auch geeignet, um variabel große Objekte zu verwalten. Die Benutzung einer Prioritätswarteschlange als Verwaltungsstruktur führt jedoch zu einer komplexeren Implementierung, und die *Worst-Case*-Laufzeit einer Operation ist nun logarithmisch in der Anzahl der im Cache befindlichen Objekte.

3.3 Altruistische Heuristiken

Bei diesen Heuristiken versuchen wir, die globale Cache-Trefferrate zu maximieren. Bei den statischen Heuristiken haben wir gesehen, daß wir die globale Cache-Trefferrate maximieren können, indem wir die Objekte mit maximaler globaler Temperatur in den Caches halten. Die Replikation eines Objekts über mehrere Caches führt zu keiner Erhöhung der globalen Cache-Trefferrate, bewirkt auf der anderen Seite jedoch, daß die Anzahl der verschiedenen, im aggregierten Cache gehaltenen Objekte reduziert wird. Daher fordern wir bei den altruistischen Heuristiken, daß jedes Objekt höchstens einmal im aggregierten, verteilten Cache vorkommen darf. In diesem Abschnitt stellen wir zwei mögliche Varianten vor, die mit Hilfe unterschiedlicher Information versuchen, diese Bedingungen für ein altruistische Verhalten zu erzielen.

3.3.1 Eine LRU-basierte altruistische Heuristik

Bei dieser Methode liegt der Schwerpunkt auf der Vermeidung von Replikationen, während eine explizite Berücksichtigung der globalen Temperatur nicht erfolgt. Zur Bestimmung des Objektstatus benutzen wir das in Abschnitt 3.1.3 vorgestellte Protokoll. Dies erlaubt uns die Partitionierung der Objekte im lokalen Cache in Unikate und Replikate. Für jede dieser Klassen benutzen wir einen separaten LRU-Stack. Muß nun ein Opfer bestimmt werden, so wird zuerst der Stack mit den Replikaten durchsucht und erst wenn dort nicht genügend Opfer gefunden werden, wird der Stack mit den Unikaten überprüft. Dieses Verfahren stellt dadurch auf sehr einfache Weise sicher, daß Objekte, die in mehreren Caches vorkommen, bei der Cache-Ersetzung bevorzugt als Opfer ausgewählt werden. Die globale Wichtigkeit wird bei diesem Verfahren jedoch nur in soweit berücksichtigt, als jeder Zugriff, egal ob lokal oder global, zu einer Neupositionierung an den Anfang des entsprechenden Stacks führt. Da die Objekte innerhalb eines LRU-Stacks jeweils nur eine relative Wichtigkeit durch ihre Position im Stack besitzen, kann bei dem Wechsel eines Objekts zwischen den zwei lokalen Stacks keine Information über ihre Wichtigkeit übernommen werden. Um daher zu vermeiden, daß ein Objekt bei dem Wechsel zu stark benachteiligt wird, fügen wir die Objekte jeweils am wertvolleren Ende des entsprechenden Stacks ein.

Ein Problem bei dieser Heuristik entsteht dadurch, daß wir zwar ein globales Optimierungsziel verfolgen, jedoch die Ersetzungsopfer lokal bestimmten. So kann es z.B. vorkommen, daß wir auf einem Rechner ein Unikat entfernen, obwohl auf einem anderen Rechner noch Replikate existieren. Entsprechend des Optimierungsziels müßten diese Replikate jedoch dem Unikat vorgezogen werden. Um dieses Ziel zu erreichen, führen wir in Abschnitt 3.5 eine Erweiterung ein, die asynchron, d.h. ohne ein Verlangsamung der akutell laufenden Cache-Einfügung, einen Ausgleich zwischen den Rechnern herbeiführt.

Ähnlich wie in Abschnit 3.2.1 profitieren wir auch hier von der einfachen Implementierung der Cache-Verwaltung durch Stacks und dem konstanten Berechnungsaufwand für die Operationen auf diesen Strukturen. Ein weiterer Vorteil dieser Heuristik liegt in dem sehr geringen Mehraufwand, der durch das Protokoll zur Propagierung des Objektstatus hervorgerufen wird (siehe Abschnitt 3.1.3). Nachteilig müssen wir jedoch bemerken, daß dieses Verfahren keine explizite Berücksichtigung der Objektgröße erlaubt.

3.3.2 Eine temperaturbasierte altruistische Heuristik

Nachdem wir in Abschnitt 3.3.1 ein Verfahren beschrieben haben, das sich hauptsächlich auf die Vermeidung von Replikation beschränkt, wollen wir nun noch eine Heuristik präsentieren, die auch die globale Temperatur der Objekte berücksichtigt. Dazu bestimmen wir für jedes Objekt zunächst die lokale Hitze entsprechend der LRU-k Methode aus Abschnitt 3.1.1. Anschließend führen wir – basierend auf dieser Information – eine verteilte Berechnung der globalen Hitze entsprechend des in Abschnitt 3.1.2 skizzierten Protokolls durch. Zusätzlich zu den Hitzeinformationen, verteilen wir auch den Replikationsstatus für jedes Objekt entsprechend der Vorgehensweise aus Abschnitt 3.1.3.

Mit Hilfe dieser Information bestimmen wir bei einer Cache-Ersetzung das Opfer entsprechend der folgenden Vorgehensweise: Wir teilen die im lokalen Cache gespeicherten Objekte wiederum in Unikate und Replikate auf. Für jede dieser Klassen benutzen wir eine eigene Prioritätswarteschlange, in der die Objekte entsprechend ihrer globalen Temperatur sortiert gehalten werden. Müssen wir nun ein Opfer bestimmen, so durchsuchen wir zuerst die Struktur, die die Replikate enthält und erst danach wird bei Bedarf die Struktur mit den Unikaten betrachtet. Da die globale Temperatur eine quantitative Metrik darstellt, können wir diese Information auch bei dem Wechsel des Objektstatus benutzen, um das Objekt an der *korrekten* Stelle in die andere Prioritätswarteschlange einzufügen.

Ähnlich wie bei der LRU-basierten altruistischen Heuristik haben wir auch hier einen Konflikt zwischen einem lokalen Ersetzungsalgorithmus und einem globalen Optimierungsziel. So bestimmen wir jeweils lokal das Objekt mit der geringsten globalen Temperatur als Opfer; Ziel jedoch wäre es, das Objekt mit der geringsten globalen Temperatur innerhalb des aggregierten, verteilten Caches zu ersetzen. Auch für diese Heuristik stellen wir in Abschnitt 3.5 ein Verfahren vor, das durch asynchrone Migration von Objekten versucht, das globale Optimierungsziel zu erreichen.

Verglichen mit der altruistischen Heuristik aus Abschnitt 3.3.1 ist der insgesamt größere Overhead von Nachteil. So sind – neben der komplexeren Implementierung mit Prioritätswarteschlan-

gen und deren logarithmischer *Worst-Case*-Zeit für Operationen – auch noch die Verfahren für die lokale und globale Hitzebestimmung notwendig. Während jedoch die lokale Hitzeberechnung nur einen zusätzlichen lokalen Berechnungsaufwand verursacht, erhöht die verteilte Berechnung der globalen Hitze auch die Anzahl der Nachrichten zwischen den einzelnen Rechnern.

3.4 Eine kostenbasierte Heuristik

Wie wir bereits bei den statischen Heuristiken gesehen haben, ist eine rein-egoistische oder rein-altruistische Vorgehensweise nur in speziellen Umgebungen und für spezielle Lasten optimal. Daher werden wir in diesem Abschnitt eine Heuristik entwickeln, die versucht – basierend auf einem Kostenmodell – zwischen einer egoistischen und einer altruistischen Methode zu balancieren. Entsprechend unseren Beobachtungen in Abschnitt 2.4 werden wir jedoch nicht auf Systemebene zwischen diesen beiden Extremen wählen. Wir werden vielmehr auf Objektebene jeweils entscheiden, ob ein Objekt so wertvoll ist, daß es repliziert gespeichert werden soll oder ob es ausreichend ist, es zur Erhöhung der globalen Trefferrate nur einfach im aggregierten Cache zu halten. Eine zentrale Bedeutung für die Beschreibung der Heuristik nimmt der Begriff des *Nutzens* ein, den wir daher zuerst definieren wollen.

Definition 3.1: (Nutzen)

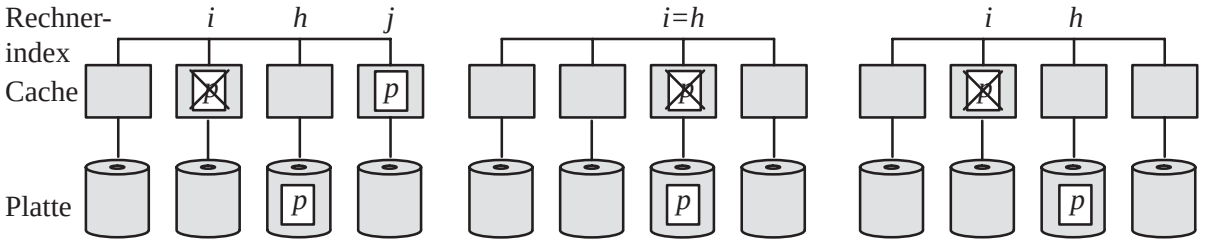
Der **Nutzen eines Objekts *j* im Cache des Rechners *i*** entspricht der Differenz in den mittleren Zugriffskosten zwischen der Situation, in der das Objekt *j* lokal gelöscht wird, und der Situation, in der das Objekt *j* im lokalen Cache gehalten wird.

■

Mit Hilfe dieser Definition können wir nun die Maximierung des Nutzens im Gesamtsystem als das Optimierungsziel unserer kostenbasierten Heuristik formulieren. Analog zu den temperaturbasierten Verfahren kann man dies erreichen, indem man jeweils lokal die Objekte mit dem geringsten Nutzen pro Größeneinheit als Ersetzungsopfer auswählt. Ähnlich wie bei den altruistischen Methoden in Abschnitt 3.3.2 kann auch hier das Problem auftreten, daß das ideale Ersetzungsopfer auf einem anderen Rechner liegt. Aus Skalierbarkeitsgründen werden wir uns auch bei dieser Heuristik darauf beschränken, das lokal am besten geeignete Opfer zu bestimmen. Um einen globalen Ausgleich zwischen den verschiedenen Rechnern zu erzielen, werden wir in Abschnitt 3.5 Methoden zur Lastverteilung beschreiben.

3.4.1 Approximative Berechnung des Nutzens

Zur Berechnung des Nutzens benutzen wir einen inkrementellen Ansatz, bei dem wir die vor dem Löschen des Objekts gültigen Kosten von den nach dem Löschen gültigen Kosten abziehen. Durch diese Vorgehensweise brauchen wir für jede Berechnung nur jeweils die Kostenterme zu betrachten, die durch das Löschen der Cache-Kopie auf dem Rechner *i* beeinflußt werden. Diese Kostenterme können wir in zwei Gruppen aufteilen, nämlich diejenigen, die Zugriffe auf das Objekt *p* selbst betreffen und solche, die andere Objekte referenzieren und daher nur aus den geänderten Zugriffskosten auf die betreffenden Ressourcen resultieren. Bei der Berechnung des Nutzens eines Objekts *p* auf einem Rechner *i* unterscheiden wir zwischen drei Fällen, die wir in Abbildung 3.7 skizziert haben und die wir nun nacheinander betrachten wollen.



(a) Objekt *p* ist ein Replikat

(b) Objekt *p* ist ein Unikat und die Cache- und Plattenkopie befinden sich auf dem gleichen Rechner.

(c) Objekt *p* ist ein Unikat und die Cache- und Plattenkopie befinden sich auf verschiedenen Rechnern.

Abbildung 3.7: Berechnung des Nutzens für das Objekt *p*

Berechnung des Nutzens für Replikate

Im ersten Fall (Abbildung 3.7(a)) berechnen wir den Nutzen für ein Objekt *p* auf dem Rechner *i*, wobei wir annehmen, daß sich das Objekt in den Caches der Rechner *i* und *j* befindet. Betrachten wir zunächst die Änderung in den Kosten für das Lesen des Objekts *p*. Da vor dem Löschen noch beide Kopien von *p* auf den Rechnern *i* und *j* existieren, können wir die mit der lokalen Hitze gewichteten Kosten für die Zugriffe auf *p* folgendermaßen berechnen:

$$- f_{ip} \times c_{lc,i}^{(old)}(p) \tag{3.7}$$

$$- f_{jp} \times c_{lc,j}^{(old)}(p) \tag{3.8}$$

$$- \sum_{k \neq i, k \neq j} f_{kp} \times c_{rc}^{(old)}(p) \quad \text{Rechner } k \text{ liest Objekt } p \text{ aus einem nichtlokalen Cache} \tag{3.9}$$

Für die Rechner *i* und *j* fallen jeweils die Kosten für einen lokalen Cache-Zugriff (Terme 3.7 und 3.8) an, während auf allen anderen Rechnern die Kosten für einen nichtlokalen Cache-Zugriff (Term 3.9) anfallen. Hierbei berücksichtigen wir bereits die Vereinfachungen der Zugriffskosten, die wir in Abschnitt 2.2.4 eingeführt haben und die uns erlauben die nichtlokalen Zugriffe auf die Kopien der Rechner *i* und *j* gleich zu behandeln. Da diese Kosten zu der ursprünglichen Situation vor dem Löschen korrespondieren und da wir an der Differenz zwischen der alten Situation und der Situation nach dem Löschen interessiert sind, werden diese Terme negativ gewichtet. Zur Berechnung des Nutzens brauchen wir dann abschließend nur die verschiedenen Kostenterme aufzuaddieren.

Betrachten wir nun die Situation nach dem Löschen der Kopie auf dem Rechner *i*, so erhalten wir die folgenden Kosten für die verschiedenen Zugriffe auf das Objekt *p*:

$$+ f_{jp} \times c_{lc,j}^{(new)}(p) \tag{3.10}$$

$$+ \sum_{k \neq j} f_{kp} \times c_{rc}^{(new)}(p) \quad \text{Rechner } k \text{ liest Objekt } p \text{ aus dem remote Cache von } j \tag{3.11}$$

In dieser Situation können nur noch die auf dem Rechner *j* initiierten Zugriffe lokal beantwortet werden; alle übrigen Zugriffe müssen als nichtlokale Zugriffe an den Rechner *j* gerichtet werden. Bei diesen Termen sehen wir, daß sich auch die Kosten für die verschiedenen Stufen der Speicherhierarchie selbst ändern können. Der Grund für diese Veränderung liegt darin, daß nun die Zugriffe des Rechners *i* auch nichtlokal beantwortet werden müssen und sich daher die Auslastungen des Netzwerks und der CPUs an den Rechnern *i* und *j* ändern können.

Diese veränderten Auslastungen der CPUs und die höhere Auslastung des Netzwerks bewirken außerdem, daß sich durch Warteschlangeneffekte die Antwortzeit aller Zugriffe, die diese Ressourcen benutzen, verändern können. Diese Änderung der Kosten können wir durch die folgenden Ausdrücke berechnen, wobei die ersten beiden Terme die neue Auslastungen der CPUs der Rechner *i* und *j* berücksichtigen, während die beiden folgenden aus den Änderungen der Netzauslastung resultieren:

Der Rechner *i* bzw. *j* liest ein Objekt *q* aus dem lokalen Cache:

$$\sum_{k\in\{i,j\},q\neq p} f_{kq} \times \left(c_{lc,i}^{(new)}(q) - c_{lc,i}^{(old)}(q)\right) \tag{3.12}$$

Der Rechner *i* bzw. *j* liest ein Objekt *q* aus dem lokalen Cache:

$$\sum_{k\in\{i,j\},q\neq p} f_{kq} \times \left(c_{ld,i}^{(new)}(q) - c_{ld,i}^{(old)}(q)\right) \tag{3.13}$$

Ein Rechner *k* liest ein Objekt *q* aus einem nichtlokalen Cache:

$$\sum_{k,q\neq p} f_{kq} \times \left(c_{rc}^{(new)}(q) - c_{rc}^{(old)}(q)\right) \tag{3.14}$$

Ein Rechner *k* liest ein Objekt *q* von der nichtlokalen Platte des Rechners *l*:

$$\sum_{k,q\neq p} f_{kq} \times \left(c_{rd,l}^{(new)}(q) - c_{rd,l}^{(old)}(q)\right) \tag{3.15}$$

Addieren wir alle diese Terme ((3.7) bis (3.15)) auf, so erhalten wir den exakten Nutzen des Objekts *p* auf dem Rechner *i*. Da die Berechnung dieser Summe jedoch viel zu zeitaufwendig für eine Online-Auswertung ist und darüber hinaus die lokale Kenntnis aller in den Termen vorkommenden Werte einer skalierbaren Berechnung widerspricht, werden wir nun einige Vereinfachungen vorstellen, die sowohl den Berechnungs-Overhead als auch den Informationsbedarf reduzieren.

Betrachten wir die Änderung der Auslastungen, die aus dem Löschen des Objekts *p* auf dem Rechner *i* resultiert, so können wir annehmen, daß diese Änderung – verglichen mit der Gesamtauslastung der betroffenen Ressourcen – vernachlässigbar gering ist. Daher setzen wir die Auslastungen vor dem Löschen gleich den Auslastungen nach dem Löschen ($c = c^{(new)} = c^{(old)}$). Dies hat eine Reihe von Vereinfachungen zur Folge:

- Die Kostenterme (3.12) bis (3.15) ergeben sich zu Null,
- die Terme (3.8) und (3.10) heben sich gegenseitig auf und
- die Summe der Terme (3.9) und (3.11) ist gleich: $-f_{ip} \times c_{rc}(p)$

Fassen wir die verbleibenden Terme zusammen, so brauchen wir für die Berechnung des Nutzens eines Objekts *p*, das sich in den Caches des Rechners *i* und mindestens eines anderen Rechners befindet, nur die folgende Formel auszuwerten:

$$Nutzen_{ip} = f_{ip} \times \left(c_{rc}(p) - c_{lc}(p)\right) \tag{3.16}$$

Berechnung des Nutzens für Unikate

Analog zu der obigen Berechnung des Nutzens für ein Replikat, bestimmen wir nun den Nutzen eines Objekts *p*, das sich nur in dem Cache eines einzigen Rechners – nämlich dem des Rechners *i* – befindet. Bei dieser Berechnung gehen wir davon aus, daß nur eine Plattenkopie existiert, die

jedoch beliebig plaziert werden kann. Auch hier betrachten wir zunächst wieder die Veränderungen der Lesekosten, die aus der neuen Abarbeitung von Zugriffen auf das Objekt p resultieren. Solange das Objekt p auf dem Rechner i noch vorhanden ist, können alle auf dem Rechner i gestarteten Zugriffe auf dieses Objekt lokal beantwortet werden. Von allen anderen Rechner aus kann das Objekt durch einen nichtlokalen Cache-Zugriff aus dem Speicher von i gelesen werden. Somit erhalten wir vor dem Löschen die folgenden beiden Kostenterme:

$$- f_{ip} \times c_{lc}^{(old)}(p) \tag{3.17}$$

$$- \sum_{k \neq i} f_{kp} \times c_{rc}^{(old)}(p) \text{ Rechner } k \text{ liest Objekt } p \text{ aus dem nichtlokalen Cache von } i \tag{3.18}$$

Betrachten wir als nächstes die anfallenden Kosten, wenn wir das Objekt p aus dem Cache des Rechners i und damit komplett aus dem aggregierten Cache des Gesamtsystems entfernen würden. In diesem Fall könnten alle Zugriffe, die auf dem Rechner h gestartet würden, das Objekt p von der lokalen Platte lesen, und alle anderen Rechner müßten das Objekt von der nichtlokalen Platte des Rechners h lesen. Damit ergeben sich die folgenden Kosten

$$+ f_{hp} \times c_{ld,h}^{(new)}(p) \tag{3.19}$$

$$+ \sum_{k \neq h} f_{kp} \times c_{rd,h}^{(new)}(p) \text{ Rechner } k \text{ liest das Objekt } p \text{ von der Platte des Rechners } h \tag{3.20}$$

Ähnlich wie bei der Berechnung des Nutzens bei einem Replikat, können sich auch hier durch das Löschen des Objekts p die Zugriffskosten ändern. Im Gegensatz zu der obigen Berechnung kann sich jedoch zusätzlich zu der Auslastung des Netzwerks und der CPU auch noch die Auslastung der Festplatte, auf der das Objekt p liegt, ändern. Die Kostenerhöhung durch diese Änderungen der Auslastungen berücksichtigen die nächsten Kostenterme:

Der Rechner i bzw. h liest ein Objekt q aus dem lokalen Cache:

$$+ \sum_{k \in \{i,h\}, q \neq p} f_{kq} \times \left(c_{lc}^{(new)}(q) - c_{lc}^{(old)}(q) \right) \tag{3.21}$$

Der Rechner i bzw. h liest ein Objekt q aus einem nichtlokalen Cache:

$$+ \sum_{k \in \{i,h\}, q \neq p} f_{kq} \times \left(c_{rc}^{(new)}(q) - c_{rc}^{(old)}(q) \right) \tag{3.22}$$

Ein Rechner k liest ein Objekt q aus einem nichtlokalen Cache:

$$+ \sum_{k, q \neq p} f_{kq} \times \left(c_{rc}^{(new)}(q) - c_{rc}^{(old)}(q) \right) \tag{3.23}$$

Ein Rechner k liest ein Objekt q von der nichtlokalen Platte des Rechners l:

$$+ \sum_{k, q \neq p} f_{kq} \times \left(c_{rd,l}^{(new)}(q) - c_{rd,l}^{(old)}(q) \right) \tag{3.24}$$

Der Rechner h liest ein Objekt q von der lokalen Platte

$$+ \sum_{q \neq p} f_{hq} \times \left(c_{ld,h}^{(new)}(q) - c_{ld,h}^{(old)}(q) \right) \tag{3.25}$$

Analog zu der vorangegangenen Berechnung nehmen wir auch hier an, daß die Änderungen der Auslastungen – verglichen mit der Gesamtauslastung der entsprechenden Ressourcen – vernachlässigbar sind. Daher berücksichtigen wir im folgenden die Terme (3.21) bis (3.25) nicht weiter und wir benutzen wiederum c an Stelle von $c^{(old)}$ bzw. $c^{(new)}$. Bei der weiteren Berechnung des Nutzens müssen wir unterscheiden, ob der Rechner i auch gleichzeitig die Plattenkopie des Objekts p besitzt, oder nicht. Betrachten wir zuerst den in der Abbildung 3.7(b) skizziert Fall, in dem der Rechner i das Objekt p auf Platte gespeichert hat.

Fall 1: Der Rechner i besitzt das Objekt p auf Platte

In Abschnitt 3.1.4 haben wir zur Gewährleistung der Skalierbarkeit angenommen, daß jeweils nur die Kosten von lokal initiierten Zugriffen bekannt sind. Daher sind auf dem Rechner *h* die Kosten für nichtlokale Zugriffe auf die Festplatte von *h* unbekannt. Dieses Problem können wir jedoch umgehen, wenn wir die Terme (3.18) und (3.20) auf die folgende Art zusammenfassen:

$$\begin{aligned}
&\sum_{k \neq h} f_{kp} \times c_{rd,h}(p) - \sum_{k \neq h} f_{kp} \times c_{rc}(p) \\
\approx\ &\sum_{k \neq h} f_{kp} \times \left(c_{rc}(p) + c_{ld,h}(p)\right) - \sum_{k \neq h} f_{kp} \times c_{rc}(p) \\
=\ &\sum_{k \neq h} f_{kp} \times c_{ld,h}(p) + \sum_{k \neq h} f_{kp} \times c_{rc}(p) - \sum_{k \neq h} f_{kp} \times c_{rc}(p) \\
=\ &\sum_{k \neq h} f_{kp} \times c_{ld,h}(p) \\
=\ &\left(\sum_{k \neq h} f_{kp}\right) \times c_{ld,h}(p)
\end{aligned} \tag{3.26}$$

Bei dieser Berechnung beruht die Approximation auf der Annahme, daß die Kosten eines nichtlokalen Zugriffs auf die Festplatte des Rechners *h* ungefähr gleich der Summe der Kosten für einen nichtlokalen Cache-Zugriff und den Kosten eines lokalen Festplattenzugriffs an dem Rechner *h* sind. Dies ist eine berechtigte Annahme, da der Cache-Zugriff die Transferkosten des Netzwerks und der lokale Plattenzugriff die Lesekosten eines nichtlokalen Plattenzugriffs berücksichtigt.

Fassen wir die nach den Vereinfachungen verbleibende Terme (3.17), (3.19) und (3.26) zusammen, so erhalten wir für die Berechnung des Nutzens die folgende Formel:

$$Nutzen_{ip} = \left(\sum_{k} f_{kp}\right) \times c_{ld,i}(p) - f_{ip} \times c_{lc}(p) \tag{3.27}$$

In Formel (3.27) erkennen wir, daß es nicht notwendig ist, die einzelnen lokalen Hitzen des Objekts *p* auf allen anderen Rechnern *k* zu kennen. Vielmehr ist die lokale Kenntnis der globalen Hitze und der lokalen Hitze auf dem Rechner *i* ausreichend.

Fall 2: Rechner i besitzt keine Plattenkopie des Objekts p

Als letzten Fall betrachten wir nun noch die in Abbildung 3.7(c) skizzierte Situation. Problematisch bei diesem Fall ist, daß wir in den Formeln (3.19) und (3.20) die lokalen Hitzen des nichtlokalen Rechners *h* benötigen. Da wir jedoch eine Verteilung von lokalen Hitzen auf andere Rechner aus Kostengründen vermeiden wollen, fassen wir die Terme (3.19) und (3.20) zu einem einzigen Term zusammen:

$$+ \left(\sum_{k} f_{kp}\right) \times c_{rd,h}(p) \tag{3.28}$$

Durch dieses Zusammenfassen führen wir den folgenden Fehler in unsere Berechnung ein:

$$f_{hp} \times \left(c_{rd,h}(p) - c_{ld,h}(p)\right)$$

Die relative Größe dieses Fehlers wird im allgemeinen sehr klein sein, da die Kosten für einen Plattenzugriff im Mittel sehr viel größer sind als für den Netztransfer. In Umgebungen, in denen dies nicht gilt, ist der Einsatz von verteiltem Caching ohnehin unsinnig, da gerade dieser Unterschied in den Zugriffskosten die Motivation für verteiltes Caching geliefert hat.

Unter dieser Voraussetzung können wir auch hier wiederum die Terme (3.17), (3.19) und (3.28) zusammenfassen, und wir erhalten in diesem Fall als Formel für die Berechnung des Nutzens:

$$Nutzen_{ip} = \left(\sum_k f_{kp}\right) \times c_{rd,h}(p) - f_{ip} \times c_{lc,i}(p) - \left(\sum_{k \neq i} f_{kp}\right) \times c_{rc}(p)$$
$$= \left(\sum_k f_{kp}\right) \times \left(c_{rd,h}(p) - c_{rc}(p)\right) + f_{ip} \times \left(c_{rc}(p) - c_{lc,i}(p)\right) \quad (3.29)$$

Da wir den Nutzen nur für solche Objekte berechnen müssen, die sich aktuell in dem Cache irgendeines Rechners befinden, haben wir hiermit alle Fälle abgedeckt. Wie wir aus den Formeln (3.16), (3.27) und (3.29) erkennen, brauchen wir für die Berechnung des Nutzens im allgemeinen die folgenden Informationen:

- Den **Status des Objekts *p***, der aussagt, ob es sich bei dem betreffenden Objekt um ein Unikat oder ein Replikat handelt. Diese Information wird benötigt, um entscheiden zu können, welche der Formeln anzuwenden ist. Den Status können wir mit Hilfe des Protokolls aus Abschnitt 3.1.3 lokal verfügbar machen. Im Falle eines Unikates müssen wir für die korrekte Auswahl der Formel auch noch wissen, ob der lokale Rechner gleichzeitig eine Plattenkopie dieses Objekts besitzt. Diese Information können wir – wie wir in Kapitel 4 noch sehen werden – dem lokalen Katalog entnehmen.
- Die **lokale und die globale Hitze des Objekts *p***. Eine Approximation über die lokale Hitze liefert uns die LRU-k Methode aus Abschnitt 3.1.1, und die Methode aus Abschnitt 3.1.2 berechnet die globale Hitze und macht sie lokal verfügbar.
- Die **erwarteten Zugriffskosten für das Objekt *p***. Diese Kosten können wir durch die Methoden aus Abschnitt 3.1.4 lokal schätzen, und somit sind auch die Werte dieser Variabeln lokal verfügbar.

Der Algorithmus 3.4 zeigt den Pseudocode für die Berechnung des Nutzens eines Objekts *p*. Hierbei nehmen wir an, daß die Funktionen local_heat und global_heat die entsprechenden Hitzen und daß die Funktionen loc_cache_cost, loc_disk_cost, rem_cache_cost und rem_disk_cost die geschätzten Kosten für das Lesen des entsprechenden Objekts von den verschiedenen Stufe der Speicherhierarchie zurückliefern.

```
void compute_benefit (Rechner i, Objekt p)
begin
    if (Objekt p ist ein Replikat) then
        benefit_ip = loc_heat(p) × (rem_cache_cost(p) - loc_cache_cost(p));
    elsif (Objekt p ist ein Unikat) then
        if (Rechner i ist der Heimatrechner des Objekts p) then
            benefit_ip = glob_heat(p) × loc_disk_cost(p)
                       - loc_heat(p) × loc_cache_cost(p);
        else
            benefit_ip = glob_heat(p) × (rem_disk_cost(p,h) - rem_cache_cost(p))
                       + loc_heat(p) × (rem_cache_cost(p) - loc_cache_cost(p));
        endif;
    endif
end;
```

Algorithmus 3.4: Berechnung des Nutzens für das Objekt *p* auf dem Rechner *i*

3.4.2 Der lokale Cache-Ersetzungsalgorithmus

Nachdem wir in dem vorangegangenen Abschnitt die Berechnung des Nutzens beschrieben haben, wollen wir nun noch auf die Ersetzungsstrategie für die kostenbasierte Heuristik eingehen. Wie bereits bei den temperaturbasierten Methoden ist es auch hier möglich, die Objektgröße zu berücksichtigen. Daher berechnen wir die Wichtigkeit eines Objekts als den Quotienten aus seinem Nutzen und seiner Größe. Um eine schnelle Opferauswahl zu gewährleisten, verwalten wir alle Objekte sortiert nach diesem Wert in einer Prioritätswarteschlange. Die Bestimmung eines Opfers bei einer Cache-Ersetzung ist dann in logarithmischer *Worst-Case*-Zeit möglich.

Da der Nutzen eines Objekts sowohl von den Objekt- als auch von den Systemeigenschaften abhängig ist, müssen bei Bedarf entsprechende Neuberechnungen durchgeführt werden. Während bei den Änderungen der Objekteigenschaften – Status, lokale und globale Hitze – eine gezielte Neuberechnung nur für das betroffene Objekt möglich ist, verursacht eine Änderung der Systemeigenschaften (z.B. Änderung der Zugriffskosten durch eine Schwankung in der Ressourcenauslastung) eine Neuberechnung für eine sehr große Anzahl von Objekten. Um nicht auf jede kleine Änderung der Systemeigenschaften mit einer sehr großen Anzahl von Neuberechnungen reagieren zu müssen, aber trotzdem Schwankungen der Systemeigenschaften berücksichtigen zu können, führen wir in einem festen Zeitraster eine Neuberechnung des Nutzens der im Cache befindlichen Objekte durch.

Da jede Änderung des Nutzens prinzipiell zu einer neuen Reihenfolge innerhalb der lokalen Prioritätswarteschlange führen kann, ist es unter Umständen sinnvoll die Anzahl der Umsortierungen und damit die CPU-Auslastung auf Kosten der Genauigkeit zu reduzieren. Dies können wir erreichen, wenn wir – ähnlich wie bei der Berechnug der globalen Temperatur – eine Umsortierung nur dann durchführen, wenn die Änderung im Nutzen größer ist als ein vorgegebener Schwellwert.

3.5 Eine Erweiterung zur Lastverteilung

Bei der Beschreibung der altruistischen und der kostenbasierten Cache-Verwaltungsverfahren haben wir gesehen, daß wir zur Bestimmung des optimalen Opfers alle im aggregierten, verteilten Cache gespeicherten Objekte betrachten müßten. Da dies aber zu inakzeptabel langen Antwortzeiten für Cache-Einfügeoperationen führen würde, haben wir uns bei diesen Methoden auf eine lokale Opferauswahl beschränkt. Diese Einschränkung kann jedoch dazu führen, daß das Potential des Gesamtsystems nicht optimal ausgenutzt wird. Dieser Fall tritt insbesondere dann auf, wenn die Auslastungen der Rechner innerhalb des Netzwerks sehr unterschiedlich sind.

Betrachten wir dazu die in Abbildung 3.8 skizzierte Situation. Wir nehmen an, daß alle Objekte die Größe eins besitzen und daß jeder Rechner einen Cache der Größe zwei hat, der entsprechend der temperaturbasierten altruistischen Heuristik verwaltet wird. Neben den aktuellen Cache- und Platteninhalten sind auch noch die jeweiligen lokalen und globalen Hitzen (in diesem Fall identisch mit der Temperatur) skizziert. Unter diesen Voraussetzungen müßte eine bezüglich der altruistische Heuristik optimale Allokation die 6 Objekte mit der höchsten Temperatur im Cache halten. In der Abbildung erkennen wir jedoch, daß die aktuelle Allokation nicht optimal ist, da

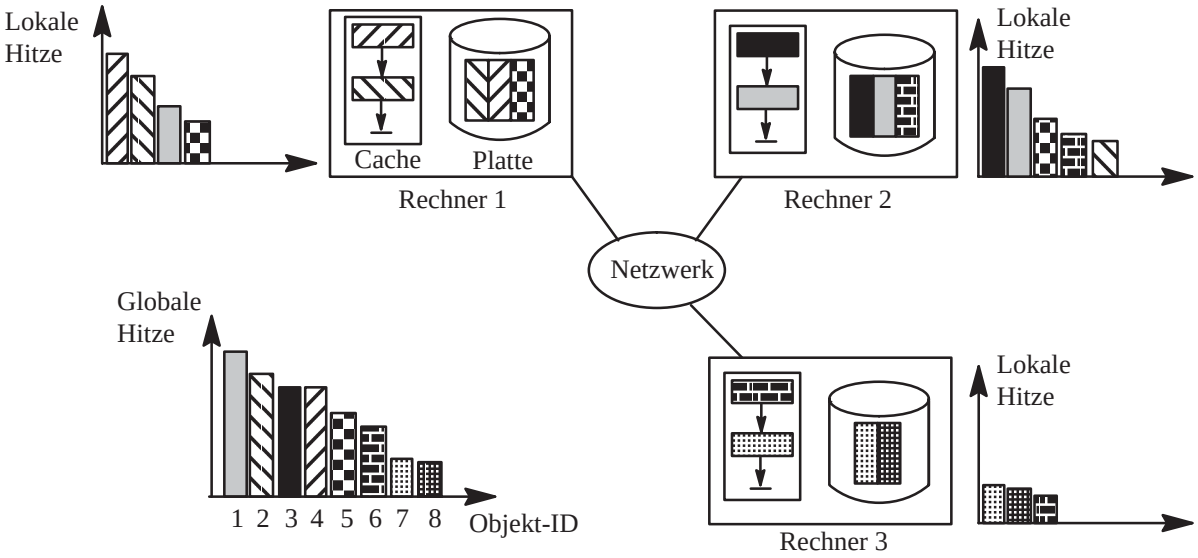


Abbildung 3.8: Suboptimale Allokation durch Unbalanciertheit des Systems

das Objekt 5 nicht im Cache gehalten wird, obwohl es zu den 6 heißesten Objekten gehört. Der Grund dafür ist, daß auf den Rechnern 1 und 2 jeweils global noch heißere Objekte zugegriffen werden, so daß das Objekt 5 dort immer wieder aus dem Cache verdrängt wird. Daher müßte das Objekt 5 im Cache des Rechners 3 gehalten werden. Da dieser Rechner jedoch nie auf das Objekt 5 zugegreift, wird es auch nie in den lokalen Cache geladen. Um nun doch dieses Objekt im aggregierten, verteilten Cache halten zu können, müßte bei einer entsprechenden Cache-Ersetzung auf den Rechnern 1 oder 2 versucht werden, das Objekt 5 auf den Rechner 3 zu migrieren.

In diesem Abschnitt werden wir zwei Methoden vorstellen, die es uns erlauben, eine solche Migration von Objekten zwischen den Rechnern durchzuführen. Beide Verfahren werden jeweils aktiv, wenn ein Unikat aus dem lokalen Cache verdrängt wird, und sie unterscheiden sich lediglich in der Art, wie der Zielrechner für eine solche Migration bestimmt wird.

Bevor wir diese Verfahren genauer beschreiben, wollen wir noch verdeutlichen, warum die Beschränkung auf Unikate gerechtfertigt ist. Während diese Beschränkung bei den altruistischen Verfahren direkt aus dem Optimierungsziel folgt, ist es a priori nicht klar, warum bei der kostenbasierten Heuristik nicht auch Replikate migriert werden sollten. Betrachten wir dazu die Abbildung 3.9, in der wir die zwei Möglichkeiten nach dem Löschen eines Replikates aufgezeigt haben. An Hand dieser Abbildung zeigen wir, daß die Migration eines Replikates *p* zu keiner Ver-

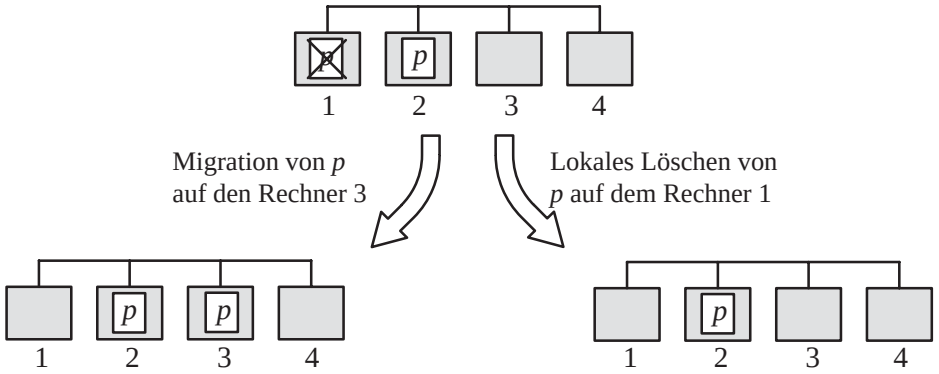


Abbildung 3.9: Mögliche Reaktionen auf das lokale Löschen eines Replikates

größerung des Nutzens im Gesamtsystem führen kann und daher eine solche Migration nicht durchgeführt werden sollte. Betrachten wir die Zugriffskosten auf das Objekt *p* von den Rechnern 1 und 4 aus, nachdem das Objekt *p* auf dem Rechner 1 gelöscht wurde, so sehen wir, daß wir hierfür in beiden skizzierten Fällen jeweils einen nichtlokalen Cachezugriff durchführen müssen und daß daher die Kosten für diese Zugriffe in unserem Modell unabhängig von der Migration von *p* sind. Wir erkennen ebenfalls, daß das Objekt *p* auf dem Rechner 2 in jedem Fall aus dem lokalen Cache gelesen werden kann und sich die Kosten daher auch für den Rechner 2 nicht ändern. Da keiner der bisher betrachteten Rechner von einer Migration des Objekts *p* auf den Rechner 3 profitiert, ist eine Erhöhung des Nutzens nur noch dadurch möglich, wenn sich der Nutzen des Objekts *p* auf dem Rechner 3 durch die Migration erhöht.

Da das Objekt *p* nach einer Migration auf den Rechner 3 weiterhin ein Replikat wäre, können wir seinen Nutzen auf dem Rechner 3 entsprechend der Formel (3.16) berechnen:

$$Nutzen_{3p} = f_{3p} \times \left(c_{rc}(p) - c_{lc}(p)\right)$$

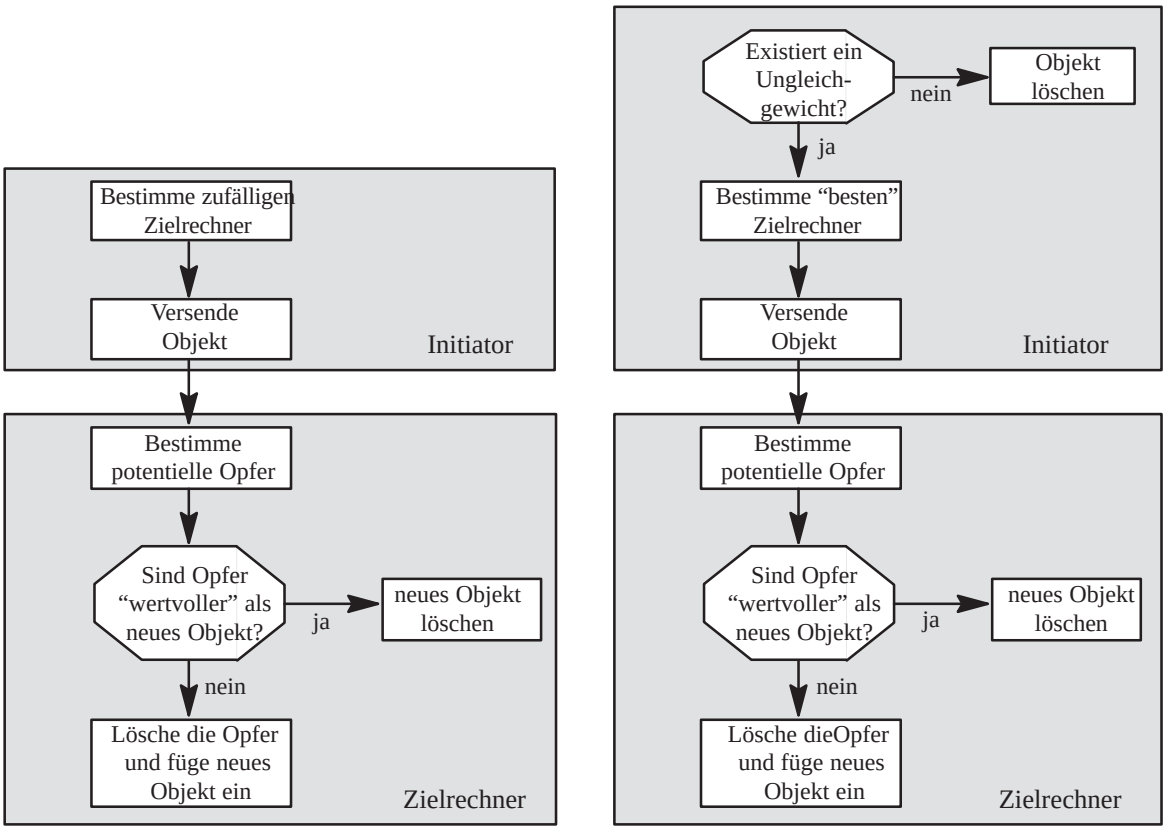
An dieser Formel erkennen wir, daß ein positiver Nutzen nur dann erzielt werden kann, wenn die lokale Hitze des Objekts *p* auf dem Rechner 3 größer als Null ist. Daher brauchen wir nun den Fall zu betrachten, in dem von Rechner 3 aus Zugriffe auf das Objekt *p* erfolgen. Da sich durch die Migration des Objekts *p* weder die lokale Objekthitze noch die Faktoren für die Berechnung der erwarteten Zugriffskosten verändern, bleibt auch der Nutzen des Objekts *p* auf dem Rechner 3 von der Migration unbeeinflußt. Hieraus folgt jedoch direkt, daß eine Migration des Objekts *p* unsinnig ist, da sich das Objekt – wenn es einen genügend großen Nutzen besäße – bereits im Cache des Rechners 3 befinden müßte. Damit haben wir gezeigt, daß bei allen betrachteten Heuristiken nur die Migration von Unikaten zu einer Verbesserung – entsprechend der verschiedenen Optimierungsziele – beitragen kann.

3.5.1 Ein zufälliges Lastverteilungsverfahren

Bei diesem Verfahren bestimmen wir jedesmal, wenn ein Unikat aus dem lokalen Cache entfernt werden soll, einen zufälligen Zielrechner, an den das zu verdrängende Objekt gesendet wird. Abhängig von der verwendeten Cache-Heuristik wird auf dem Zielrechner überprüft, ob das Einfügen des empfangenen Objekts in den Cache vorteilhaft für das Gesamtsystem ist. Die prinzipielle Vorgehensweise dieser Migrationsmethode ist in der Abbildung 3.10(a) gezeigt. Wie entschieden werden kann, ob das Einfügen des neuen Objekts auf dem Zielrechner vorteilhaft für das Gesamtsystem ist, wollen wir nun für die verschiedenen Cache-Heuristiken untersuchen.

Im Falle der LRU-basierten altruistischen Ersetzungsstrategie können wir eine Verbesserung entsprechend des Optimierungszieles dadurch erreichen, daß wir die Anzahl der Unikate im System erhöhen. Dazu brauchen wir bei einer Migration auf dem Zielrechner nur zu überprüfen, ob das Einfügen des neuen Objekts nur Replikate verdrängt. Ist dies der Fall, so übernehmen wir das Objekt in den lokalen Cache und ansonsten löschen wir das Objekt.

Bei den beiden anderen Verfahren – altruistisch mit Berücksichtigung der Temperatur und kostenbasiert – haben wir jeweils ein quantitatives Maß (die globale Hitze bzw. den Nutzen), das uns ermöglicht abzuschätzen, ob ein lokales Einfügen vorteilhaft ist. Dazu bestimmen wir jeweils



(a) zufälliges Lastverteilungsverfahren

(b) lastabhängiges und zielgerichtetes Lastverteilungsverfahren

Abbildung 3.10: Ablauf einer Migration

zuerst die Objekte, die entfernt werden müßten, damit das neue Objekt in den Cache eingefügt werden könnte. Bei der altruistischen Ersetzungsstrategie fügen wir das neue Objekt nur dann in den Cache ein, wenn erstens alle Ersetzungsopfer Replikate sind und wenn zweitens die aggregierte globale Hitze der Opfer kleiner ist als die globale Hitze des einzufügenden Objekts. Ist eine dieser Bedingungen verletzt, so löschen wir auch hier das empfangene Objekt.

Bei der kostenbasierten Cache-Strategie ist der Status des Objekts bereits bei der Berechnung des Nutzens berücksichtigt worden. Daher ist es hier ausreichend zu überprüfen, ob die Summe der Nutzen über alle Ersetzungsopfer kleiner ist als der Nutzen des migrierten Objekts. Im Gegensatz zu der globalen Hitze ist der Nutzen eines Objekts jedoch von dem betrachteten Rechner abhängig. Daher müssen wir vor diesem Test eine lokale Neuberechnung des Nutzens für das einzufügende Objekt durchführen.

Nachteilig an diesem zufälligen Lastverteilungsverfahren ist, daß der Transfer eines lokal gelöschten Unikates zu einem Zielrechner in jedem Fall durchgeführt wird und erst der Zielrechner entscheidet, ob die Migration erfolgreich abgeschlossen oder abgebrochen wird. Dies kann dazu führen, daß in einem balancierten System durch den Transfer der Objekte die Netzlast soweit erhöht wird, daß sich die Gesamtleistung verschlechtert. Aber selbst in einem unbalancierten System kann die zufällige Auswahl eines Zielrechners dazu führen, daß Migrationen zu den bereits überlasteten Rechnern durchgeführt werden und gering belastete Rechner nicht in entsprechendem Maße berücksichtigt werden.

Dieses letzte Problem können wir auf Kosten eines stärkeren Verbrauchs an Netzwerkbandbreite abschwächen, indem wir das Verfahren – ähnlich zu dem in [DWA*94] vorgestellten *N-Chance*-Algorithmus – erweitern. Dabei erhält jedes Objekt einen zusätzlichen Zähler und bei jedem mißglückten Migrationsversuch wird dieses Zählers um eins erhöht. Erfolgt ein Zugriff auf das Objekt so wird der Zähler wieder auf Null gesetzt. Ein endgültiges Entfernen eines Objekts aus dem aggregierten Cache wird bei dieser Heuristik nur dann durchgeführt, wenn dieser Zähler einen vorher festgelegten Wert erreicht hat.

3.5.2 Ein lastabhängiges und zielgerichtetes Lastverteilungsverfahren

Bei dem Verfahren, das wir jetzt vorstellen werden, benötigen wir eine quantitative Metrik, die den Wert eines Objekts bezeichnet. Daher ist dieses Verfahren nur in Verbindung mit der temperaturbasierten altruistischen und der kostenbasierten Cache-Strategie möglich. Die entsprechende Metrik (globale Hitze bzw. Nutze) wird jeweils ausgenutzt, um zu erkennen, ob das System ungleichmäßig ausgelastet und daher der zusätzliche Aufwand für eine Migration gerechtfertigt ist. Darüber hinaus können wir auch einen bezüglich der Metrik idealen Zielrechner für die Migration bestimmen und so die Überlastung von bereits belasteten Rechnern vermeiden.

Aus den oben angeführten Gründen wird auch dieses Verfahren nur bei der Verdrängung von Unikaten aktiv. Bei jeder Ersetzung eines solchen Objekts bestimmt der Rechner zuerst den optimalen Zielrechner für eine Migration. Dies ist entweder der Rechner mit der minimalen aggregierten Temperatur bzw. mit dem minimalen aggregierten Nutzen pro Größeneinheit. Die Normierung der Metrik bezüglich der Cache-Größe erlaubt die Betrachtung von Rechnernetzen mit unterschiedlich konfigurierten Rechnern. Da der Wert dieser Metrik alle lokal im Cache gehaltenen Objekte berücksichtigt, ändert sich diese Information im allgmeinen nicht abrupt, und daher können wir den Aufwand für die Verteilung dieser Information gering halten. Eine Möglichkeit zur Propagierung der Lastvektoren besteht in dem randomisierten Verfahren von [BaSh85].

Hat der lokale Rechner nun den optimalen Empfänger bestimmt, so vergleicht er seine lokale Temperatur bzw. seinen Nutzen pro Größeneinheit mit dem entsprechenden Wert des Empfängers. Dies erlaubt uns, Migrationen in einem balancierten System zu vermeiden und somit eine Verschwendung von Netzbandbreite zu verhindern. Nur wenn der Unterschied zwischen dem lokalen und dem Wert des Zielrechners größer ist als ein vorgegebener Schwellwert, wird eine Migration durchgeführt.

Völlig analog zu dem in Abschnitt 3.5.1 beschriebenen Verfahren, testen wir auch auf der Empfängerseite an Hand der aktuellen lokalen Information, ob ein Einfügen des betreffenden Objekts in den lokalen Cache durchgeführt werden soll. Nur in den Fällen, in denen das Einfügen vorteilhaft ist, wird auch wirklich eine Ersetzung durchgeführt; ansonsten löschen wir das empfangene Objekt.

Eine zu dem *N-Chance*-Algorithmus vergleichbare Erweiterung, bei der das Objekt anstatt auf dem Empfänger gelöscht zu werden an einen anderen Rechner weitergesendet wird, kann hier benutzt werden, um Ungenauigkeiten bei der Verteilung der Lastwerte auszugleichen. So kann

z.B. ein Empfänger feststellen, daß das empfangene Objekt zwar lokal nicht eingefügt werden kann, daß jedoch innerhalb des Systems ein Rechner existiert, der entsprechend seiner lokalen Information einen noch geringeren Wert für seine Temperatur bzw. seinen Nutzen pro Größeneinheit besitzt. In diesem Fall kann der Empfänger das Objekt direkt an diesen Rechner weiterschicken. Da wir jedoch davon ausgehen, daß Fehlentscheidungen auf Grund von falscher Information bei der Bestimmung des optimalen Empfängers nur selten auftreten, betrachten wir diese Erweiterung im folgenden nicht.

Kapitel 4

Die Prototyp-Implementierung

Nachdem wir in Kapitel 3 die Cache-Ersetzungsstrategien zusammen mit den von ihnen benötigten Hilfsprotokollen hergeleitet haben, beschreiben wir nun die Einbettung dieser Methoden und Protokolle in eine prototypische Implementierung.

In Abbildung 4.1 ist die Architektur unserer Implementierung gezeigt. Wir betrachten ein System, das aus mehreren Rechnern besteht, die durch ein schnelles Netzwerk miteinander verbunden sind. Jeder einzelne Rechner besteht aus den folgenden Software- und Hardwarekomponenten:

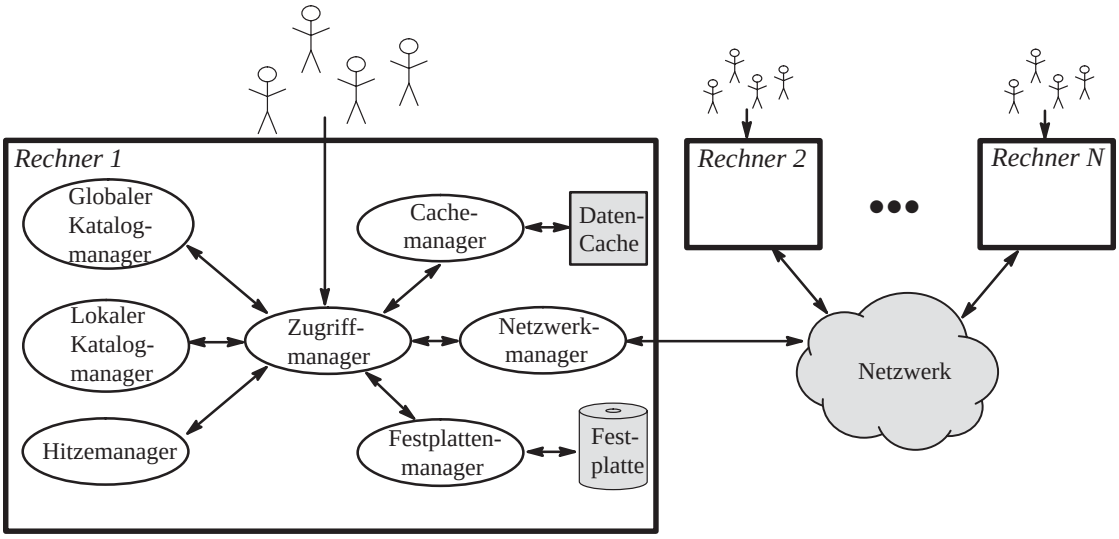


Abbildung 4.1: Allgemeine Architektur des Prototyps

- dem Zugriffmanager, der für die Abarbeitung und Synchronisation von Lese- und Schreibzugriffen zuständig ist,
- den Managern für den globalen und lokalen Katalog, die gemeinsam für die Verwaltung der Buchhaltungsinformation notwendig sind,
- dem Hitzemanager, der die Hitzen und die Zugriffskosten entsprechend der Verfahren aus Abschnitt 3.1 berechnet und verwaltet,
- einem Cachemanager, der die Verwaltung des lokalen Daten-Caches übernimmt,
- einem Festplattenmanager, der ein Interface zu der Festplatte bereitstellt und
- einem Netzwerkmanager, der eine Schnittstelle zu dem lokalen Netzwerk darstellt.

Auf jedem Rechner können Benutzer Zugriffe auf Objekte initiieren, unabhängig davon, wo diese Objekte innerhalb des Systems im Cache gehalten werden oder auf Platte gespeichert sind.

In dem Abschnitt 4.1 beschreiben wir detailliert die verschiedenen Manager. Da uns für eine systematische Evaluation nicht die entsprechende Hardware zur Verfügung steht – hier seien insbesondere neue Netzwerktechnologien und Netzwerkprotokolle genannt–, sind wir gezwungen, auf simulierte Hardware zurückzugreifen.[3] Die Modellierung der verschiedenen Hardwareressourcen beschreiben wir in Abschnitt 4.2. Um auch bezüglich der Last und der Daten eine systematische Untersuchung der Performance durchführen zu können, beschreiben wir in Abschnitt 4.3 einen Lastgenerator und stellen abschließend ein synthetisches Datenmodell vor. Als Hilfmittel zur Implementierung der Simulationsumgebung und der simulierten Komponenten haben wir die C++-Bibliothek *CSIM18* [Schw96] benutzt.

4.1 Die Softwarekomponenten

4.1.1 Der globale Katalogmanager

Zur Verwaltung der Buchhaltungsinformation benutzen wir eine Kombination von lokalem und globalem Katalog. Der globale Katalog verwaltet dabei die Daten, die zur rechnerübergreifenden Synchronisation und Abarbeitung von Zugriffen notwendig sind. Dies sind unter anderem die Verwaltungsinformationen, die wir bereits bei der Einführung der Heimatrechner angesprochen und bei der Beschreibung der Protokolle in Abschnitt 3.1 benutzt haben. In den lokalen Katalogen hingegen verwalten wir die Daten für die lokale Synchronisation und für den direkten Zugriff auf lokale Cache- oder Plattenkopien. In diesem Abschnitt betrachten wir zunächst den globalen Katalogmanager. Eine mögliche Implementierung eines lokalen Kataloges beschreiben wir anschließend in Abschnitt 4.1.2.

Der globale Katalog erlaubt eine einheitliche Sicht auf das Gesamtsystem. So muß es für jeden Rechner möglich sein mit Hilfe des globalen Katalogs, jedes im System vorhandene Objekt ausfindig zu machen. Für diesen Zweck verwaltet der globale Katalog für jedes Objekt eine Liste, in der alle momentanen Besitzer einer Cache-Kopie gespeichert sind. Diese Liste nennen wir im folgenden *Owner-Liste*. Das Verfahren zur Aktualisierung dieser Liste beschreiben wir bei der Abarbeitung der Zugriffe in Abschnitt 4.1.3.

Neben den Cache-Kopien müssen wir auch Buch über die Plazierung der Plattenkopien führen. Bisher haben wir immer angenommen, daß die Objekte jeweils auf einer beliebigen Platte gespeichert werden können. Innerhalb unseres Prototyps gehen wir jedoch von der Vereinfachung aus, daß die Objekte jeweils auf der Platte des Heimatrechners alloziert werden. Dies führt zu einigen Vereinfachungen bei der Abarbeitung von Zugriffen, ist jedoch keine prinzipielle Einschrän-

3. Eine Implementierung unseres Verfahrens auf realer Hardware ist in [Stru98] beschrieben. In dieser Arbeit wird verteiltes Caching benutzt, um einen verteilten HTTP-Server zu beschleunigen. Aus den bereits angesprochenen Gründen muß jedoch dort auf ein relativ langsames Netzwerk (10 MBit/s *Ethernet*) zurückgegriffen werden, so daß die dort erzielten Ergebnisse nur einen grundsätzlichen “Proof of Concept” liefern können.

kung. Will man jedoch auch eine Replikation von Kopien auf Festplatten zulassen, so muß die Herleitung zur Berechnung des Nutzens in Kapitel 3 verallgemeinert werden.

Neben diesen Daten verwalten wir in dem globalen Katalog auch noch die zur Synchronisation der Zugriffe notwendigen Informationen. Da jedoch eine genaue Beschreibung dieser Informationen zum Verständnis der Verwaltungsmethoden für den globalen Katalog nicht notwendig ist, stellen wir diese erst in Abschnitt 4.1.3 vor.

Wichtiger als der Aufbau eines einzelnen Katalogeintrages ist in diesem Abschnitt die Verwaltung der verschiedenen Einträge. Prinzipiell können wir hier zwischen zwei verschiedenen Ansätzen wählen. Bei der ersten Möglichkeit bestimmen wir einen dedizierten Server, der alle Katalogeinträge speichert, während bei der zweiten Möglichkeit eine Verteilung der Einträge über alle – oder aber zumindest über mehrere – Rechner vorgenommen wird. Die Vorteile eines dedizierten Katalog-Servers sind die leichte Implementierung und das einfache Protokoll für Anfragen und Änderungen. So weiß z.B. jeder Rechner im System, an welchen Rechner er Kataloganfragen zu senden hat. Ein effizienter Zugriff auf den gesuchten Eintrag kann durch eine Hashtabelle realisiert werden. Problematisch an diesem Ansatz ist jedoch, daß dieser dedizierte Katalog-Server sehr schnell zum Engpaß werden kann, wenn das System hochskaliert wird [Stan97]. Dieses Problem kann zwar prinzipiell durch das Caching von Katalogeinträgen abgeschwächt werden, da jedoch in unserem System jedes Einfügen bzw. Löschen eines Objekts einer Änderung des Katalogeintrages entspricht, müssen wir mit einer sehr hohen Änderungsrate rechnen. Daher ist der Gewinn durch Caching sehr gering und kann sich durch das aufwändigere Protokoll sogar nachteilig auswirken.

Bei einem verteilten Katalog werden die Einträge auf mehrere Rechner aufgeteilt. Daher besteht hier das Problem, daß jeder Rechner feststellen können muß, an wen er seine Kataloganfragen bzw. Änderungsnachrichten für einen Katalogeintrag senden muß. In der Literatur werden zwei prinzipiell verschiedene Methoden unterschieden. Bei der ersten Methode erhalten alle Objekte einen lokalitätsabhängigen Namen, d.h. jeder Rechner kann lokal aus dem Namen des Objekts die Adresse des Heimatrechners und damit den Besitzer des Katalogeintrages für das betreffende Objekt ableiten. Dies kann man erreichen, indem man entweder die Adresse in den Namen hineincodiert, wie dies z.B. bei den *System-Wide-Names* [Lind81, ChSe91] gemacht wird, oder aber jeder Rechner kennt die Funktion (z.B. eine Hashfunktion), nach der die Katalogeinträge über die verschiedenen Rechner verteilt sind. Eine hiervon grundsätzlich abweichende Methode wird durch die Benutzung von lokalitätsunabhängigen Namen erreicht. Hierbei werden verteilte Suchstrukturen benutzt, um den zu einem Objekt zugehörigen Katalogeintrag zu finden [KrWi94, LNS94, VBW94, EKK97].

Während bei den lokalitätsabhängigen Methoden jeder Katalogeintrag nach höchstens einem Netzzugriff gefunden wird, brauchen die lokalitätsunabhängigen Verfahren unter Umständen mehrere Netzzugriffe. Diese höhere Anzahl kann nur dann vorteilhaft sein, wenn sich die Transferkosten für die Verbindungsstrecken innerhalb des Netzwerks unterscheiden. In diesem Fall können mehrere Netztransfers über schnelle Verbindungsstrecken günstiger sein als ein einziger teuerer Transfer. Da wir in unserem Modell jedoch ein Bus-Netzwerk verwenden, d.h. ein Netz, in dem die Übermittlungskosten zwischen allen Paaren von Rechnern nahezu gleich sind, macht die Verwendung von lokalitätsunabhängigen Namen keinen Sinn. In unserem Prototyp verteilen

wir daher die Katalogeinträge entsprechend einer Hashfunktion über die verschiedenen Rechner und machen die Definition dieser Hashfunktion auf jedem Rechner bekannt.

Wie wir bereits weiter oben beschrieben haben, werden wir eine Evaluation unseres Prototyps durch Simulationen durchführen. Für eine exakte Modellierung ist es daher notwendig, die CPU-Kosten für die Ausführung der Katalogoperationen mit zu berücksichtigen. Um diese Kosten zu bestimmen, haben wir den Zeitbedarf für das Einfügen, Suchen und Löschen eines Eintrages in bzw. aus einer *LEDA*-Hashtabelle [MNU97] gemessen. Dazu haben wir die Anzahl der Einträge in der Hashtabelle variiert und jeweils die Zeit für die entsprechende Operation mit Hilfe der *Solaris*-Betriebssystemroutine *getrusage* auf einer Workstation des Typs *SUN Sparc 4* bestimmt. Die Ergebnisse dieser Messung sind in der Abbildung 4.2 aufgezeigt. Bis auf einige Ausreißer können wir in dieser Abbildung sehr gut erkennen, daß die Kosten für alle Operationen nahezu unabhängig von der Anzahl der Einträge sind, was auch mit dem theoretischen Ergebnis über die amortisierten Kosten für Hashoperationen übereinstimmt. Für unsere Experimente veranschlagen wir für jedes Einfügen eines Eintrages 5 µs, für jedes Suchen 4 µs und für jedes Löschen 7 µs CPU-Overhead.

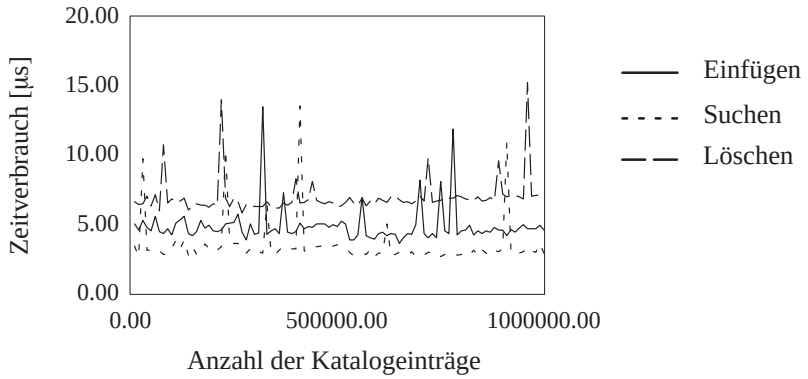


Abbildung 4.2: CPU-Kosten für die unterschiedlichen Katalogoperationen

4.1.2 Der lokale Katalogmanager

Im vorangegangenen Abschnitt haben wir gesehen, daß wir mit Hilfe des globalen Katalogmanagers den Rechner lokalisieren können, auf dem sich entweder eine Cache- oder eine Plattenkopie des gesuchten Objekts befindet. Um jedoch auf die Objekte zugreifen zu können, benötigen wir die exakte Adresse des Objekts im Cache bzw. auf Platte. Diese Information verwalten wir in dem lokalen Katalog. Da dieser somit Informationen über alle lokal im Cache bzw. auf Platte befindlichen Objekte führt, können wir auf diese Objekte lokal zugreifen, ohne daß eine Anfrage an den globalen Katalog notwendig wird.

Neben diesen Informationen zur Lokalisierung der Daten auf dem Rechner verwalten wir auch noch die für die verschiedenen Cache-Heuristiken notwendigen Daten im lokalen Katalog. Dies sind je nach der gewählten Heuristik die lokale Hitzeinformation, der lokal gespeicherte Wert über die globale Hitze, der Status und der berechnete lokale Nutzen eines Objekts. Zusätzlich müssen wir – analog zu dem globalen Katalogmanager – auch hier Informationen zur Synchronisation aktuell laufender Zugriffe auf ein Objekt mitführen. Eine genaue Beschreibung der dazu notwendigen Daten verschieben wir wiederum auf den Abschnitt 4.1.3.

Betrachten wir nur die Cache-Heuristiken, die ohne die lokale oder globale Hitze auskommen, so ist es ausreichend, den lokalen Katalogeintrag für ein Objekt jeweils nur solange zu behalten, wie das Objekt entweder lokal im Cache oder lokal auf Platte gespeichert ist. Im Gegensatz dazu haben wir bei der Hitzeberechnung in Abschnitt 3.1.1 gesehen, daß die Information zur Approximation der lokalen Hitze auch über den Zeitraum, in dem das Objekt nicht lokal vorhanden ist, gemerkt werden muß. Um jedoch den Overhead für die Speicherung zu vieler Katalogeinträge zu reduzieren, wurde in Abschnitt 3.1.1 der Aufräumdämon eingeführt. Dieser Dämon löscht die Katalogeinträge von lokal nicht mehr verfügbaren Objekten, die seit einer bestimmten Zeitspanne nicht mehr zugegriffen wurden. Zur Bestimmung einer sinnvollen Zeitspanne kann die 5-Minuten-Regel von [GrPu87, GrGr97] benutzt werden. Diese Regel leitet in Abhängigkeit von den Hardwarepreisen für Hauptspeicher und Festplatten eine Mindesthitze für solche Objekte her, die im Hauptspeicher gehalten werden sollten. Entsprechend dieser Regel sollten alle Objekte, deren mittlere Zeitspanne zwischen zwei Zugriffen größer als 5 Minuten ist, nicht mehr im Cache gehalten werden. Da wir unter dieser Annahme für diese Objekte auch keine Hitzeinformation mehr benötigen, entfernt unser Aufräumdämon die lokalen Katalogeinträge solcher Objekte.

Analog zur Implementierung des globalen Kataloges benutzen wir eine Hashtabelle auf jedem Rechner, die uns bei Anfragen auf einem lokalen Katalog einen schnellen Zugriff auf den gesuchten Eintrag gewährleistet. Zur Berücksichtigung des Overheads von lokalen Kataloganfragen können wir daher auch hier die für den globalen Katalog gemessenen Werte für die Bearbeitungszeit übernehmen.

4.1.3 Der Zugriffmanager

Der Zugriffmanager ist für die korrekte Abarbeitung der Lese- und Schreibzugriffe zuständig. Für die Lesezugriffe muß er dabei die Zugriffshierarchie entsprechend des verteilten Cachings berücksichtigen, d.h. ein lokaler Cache-Zugriff wird einem nichtlokalen Cache-Zugriff vorgezogen und nur wenn sich das Objekt in keinem Cache befindet, erfolgt ein Zugriff auf die Platte des Rechners, der das Objekt gespeichert hat.

Um die verschiedenen Cache-Heuristiken auch unter realistischen Bedingungen vergleichen zu können, erlauben wir in unserem Prototypen – im Gegensatz zu den theoretischen Betrachtungen in den Kapiteln 2 und 3 – auch Schreibzugriffe. Neben der korrekten Abarbeitung der Lesezugriffe muß der Zugriffmanager daher auch die Kohärenz der Daten sicherstellen, d.h. es darf nicht vorkommen, daß ein Zugriff eine veralteten Kopie eines Objekts liest. Ein Protokoll, das dies garantiert, ist das sogenannte *Callback-Locking*-Protokoll [HKM*88, Rahm93, Fran96].

Neben den Lese- und Schreibzugriffen muß der Zugriffmanager auch die Migrationen eines Objekts aus Gründen der Lastverteilung entsprechend den in dem Abschnitt 3.5 vorgestellten Verfahren durchführen. Insbesondere muß dabei berücksichtigt werden, daß es trotz möglicher Überschneidungen von Migrationen mit regulären Zugriffen zu keinen Inkohärenzen kommen darf.

Wir beschreiben zunächst die zur Synchronisation notwendigen Datenstrukturen und werden anschließend detailliert auf die verschiedenen Zugriffsarten eingehen.

Datenstrukturen zur lokalen und globalen Synchronisation

Auf jedem Rechner läuft ein lokaler Synchronisationsprozeß (LSP), an den alle lokal gestarteten Zugriffe weitergereicht werden. Da Zugriffe auf unterschiedliche Objekte unabhängig voneinander sind, muß der LSP nur jeweils die Zugriffe auf das gleiche Objekt synchronisieren. Dazu führen wir für jedes Objekt eine Variable *Valid* und eine Variable *Lock* ein. Die Variable *Valid* gibt an, ob sich das entsprechende Objekt im lokalen Cache befindet (*Valid=true*) oder nicht, und die Variable *Lock* zeigt an, ob das Objekt bereits durch einen anderen Zugriff gesperrt ist. Bei den Sperren unterscheiden wir zwischen drei verschiedenen Typen: Lese- (*R-Lock*), Schreib- (*W-Lock*) und Migrationssperren (*M-Lock*). Während die Lesesperren *Shared-Sperren* sind, d.h. mehrere Lesezugriffe können gleichzeitig auf ein Objekt zugreifen, sind die Schreib- und Migrationssperren exklusiv. Zur Verwaltung von Zugriffen, die auf Grund der aktuellen Werte von *Valid* und *Lock* am Weiterlaufen gehindert werden müssen, weil z.B. eine exklusive Sperre gesetzt ist oder das Objekt gerade von Platte gelesen wird, besitzt der LSP für jedes Objekt eine eigene Warteschlange, die wir *Local-Queue* nennen. Um den Speicher-Overhead gering zu halten, werden diese Warteschlangen nach Bedarf dynamisch kreiert bzw. gelöscht. Mit Hilfe dieser Daten, die wir jeweils im lokalen Katalogeintrag des betreffenden Objekts ablegen, können wir nun eine Übergangsfunktion definieren. Beim Auftreten bestimmter Ereignisse wird entsprechend der aktuellen Variablenwerte ein neuer Zustand berechnet, und zusätzlich können noch weitere Aktionen angestoßen werden. Betrachten wir dazu das folgende Beispiel:

Beispiel 4.1:

Der Ausgangszustand entsprechend der Abbildung 4.3 besagt, daß das betreffende Objekt lokal vorhanden ist, daß zur Zeit bereits drei Lesezugriffe auf diesem Objekt laufen und daß ansonsten keine weiteren Zugriffe für dieses Objekt auf die Bearbeitung warten. Trifft nun ein neuer Lesezugriff an dem LSP ein, so werden entsprechend der Übergangsfunktion die in der Abbildung gezeigten Aktionen durchgeführt:

Abbildung 4.3: Übergang des lokalen Synchronisationsprozesses

Da Lesezugriffe gleichzeitig auf ein Objekt zugreifen dürfen, müssen wir in diesem Fall nur die Anzahl der Lesesperren um eins erhöhen und können danach den neu eingetroffenen Lesezugriff, der während der Berechnung der Übergangsfunktion angehalten wurde, sofort wieder aktivieren.

■

Wegen des sehr großen Umfanges der kompletten Übergangstabelle, die zur Definition der Übergangsfunktion benutzt wird, verzichten wir an dieser Stelle auf eine ausführliche Beschreibung und verweisen auf den Anhang C.

Analog zur lokalen müssen wir auch die globale Synchronisation sicherstellen. Dies geschieht durch den globalen Synchronisationsprozeß (GSP). Im Gegensatz zu dem oben beschriebenen LSP benötigen wir einen solchen Prozeß nur auf den Rechnern, die Heimatrechner für mindestens ein Objekt sind. Auch dieser Prozeß benutzt wieder eine *Lock*-Variable und eine Warteschlange (*Global-Queue*) um die eintreffenden Zugriffe zu synchronisieren. Zusätzlich muß der GSP auch die Liste alle Rechner besitzen, die dieses Objekt im Cache halten. Dies ist zum einen notwendig um Lesezugriffen an einen Besitzer einer Cache-Kopie weiterzuleiten, und zweitens müssen im Falle eines Schreibzugriffs sämtliche Cache-Kopien von den entsprechenden Rechnern zurückgerufen werden. Da die globale Synchronisation der Zugriffe auf ein Objekt immer auf dem Heimatrechner geschieht, können wir die in Abschnitt 4.1.1 beschriebene *Owner-List* für diese Aufgabe benutzen. Analog zu dem LSP definieren wir für alle verschiedenen Kombinationen der *Global-Queue*, der *Lock*-Variablen und der *Owner-List* eine Übergangstabelle, deren komplette Beschreibung wiederum im Anhang C erfolgt.

Bevor wir die Standardabarbeitung der Lese- und Schreibzugriffe und der Objektmigrationen genauer beschreiben, wollen wir noch auf den durch den Zugriffmanager verursachten CPU-Overhead eingehen. Zur Berechnung der Übergangsfunktion muß der Synchronisationsprozeß zuerst den aktuellen Zustand bestimmen. Dazu muß er einen Zugriff auf den lokalen bzw. globalen Katalogeintrag des entsprechenden Objekts durchführen. Die Kosten für die Durchführung von Kataloganfragen werden jedoch schon bei den entsprechenden Katalogmanagern berücksichtigt. Da auch der Aufwand für das Versenden der Nachrichten – wie wir später noch sehen werden – separat erfolgt, können wir den Overhead durch den Zugriffmanager vernachlässigen.

Die Abarbeitung von Lesezugriffen

Bei der Abarbeitung von Lesezugriffen müssen wir die Zugriffshierarchie des verteilten Cachings berücksichtigen. Entsprechend dieser Hierarchie unterscheiden wir zwischen den drei Fällen: lokaler Cache-, nichtlokaler Cachezugriff und Plattenzugriff, die wir nun nacheinander beschreiben. Bei den Abbildungen, die wir zur Illustration der Vorgehensweise benutzen, entsprechen die Pfeile jeweils einem Nachrichtenaustausch. Hierbei können wir zwischen lokalen Nachrichten, die zwischen verschiedenen Komponenten des gleichen Rechners ausgetauscht werden und daher zu keiner Netzwerkbelastung führen, und globalen Nachrichten, die zwischen verschiedenen Rechnern versendet werden, unterscheiden. Die fettgedruckten Nachrichtentypen korrespondieren zu den entsprechenden Einträgen in den Übergangstabellen der Synchronisationsprozesse im Anhang C.

Fall 1*: *Das Objekt wird aus dem lokalen Cache gelesen
Betrachten wir zunächst den Fall, in dem sich das Objekt bereits in dem lokalen Cache befindet (Abbildung 4.4). Nach der Generierung eines lokalen Zugriffs wird dieser an den LSP übergeben (Nachricht 1). Befindet sich das Objekt im lokalen Cache und ist nicht exklusiv gesperrt, so wird eine Lesesperre erworben. Anschließend wird der Anwendung mitgeteilt, daß das Objekt im Cache vorhanden ist und daß sie daher weiterlaufen kann (2). Ist das Objekt jedoch exklusiv gesperrt, so fügen wir den Zugriff in die lokale Warteschlange ein und testen bei jeder Beendigung eines lokalen Zugriffs auf diesem Objekt, ob ein oder mehrere der wartenden Zugriffe reaktiviert werden können. Nachdem die Anwendung ihre Bearbeitung auf dem Objekt abgeschlossen hat, muß sie ihre Lesesperre wieder zurückgeben. Dies geschieht mit Hilfe der Nachricht (3).

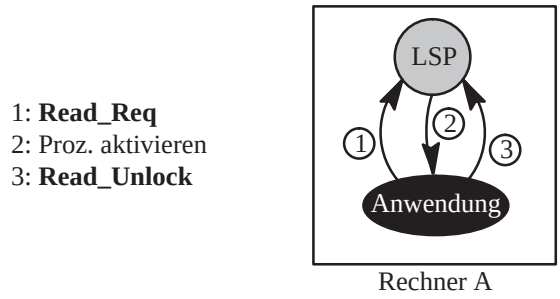
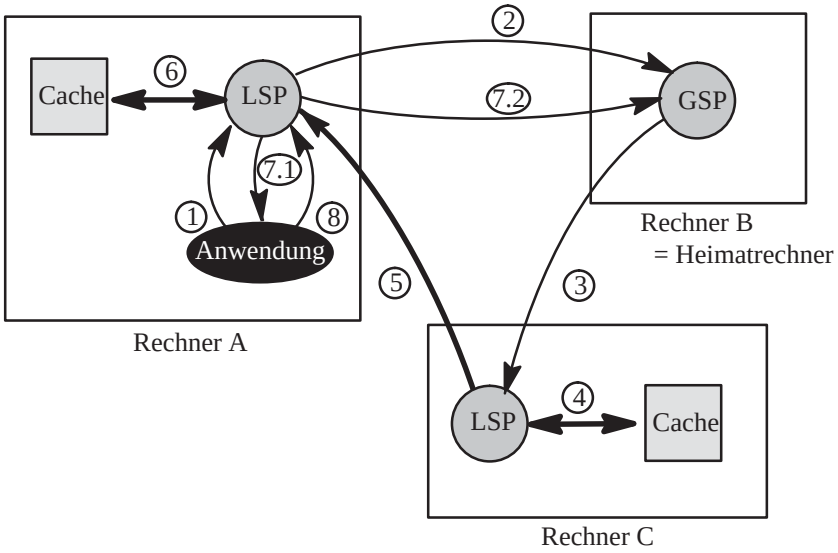


Abbildung 4.4: Lesender lokaler Cache-Zugriff

Fall 2*: *Das Objekt wird aus einem nichtlokalen Cache gelesen

Ist das Objekt nicht in dem lokalen jedoch in einem nichtlokalen Cache vorhanden, so können wir die Abarbeitung an Hand der Abbildung 4.5 verdeutlichen. Auch hier wird wiederum zuerst die Anfrage an den LSP weitergegeben (1). Da dieser aber feststellt, daß sich das Objekt nicht im lokalen Cache befindet, leitet er die Nachricht an den Heimatrechner weiter (2). Auf dem Heimatrechner kann nun der GSP in der Owner-Liste nachschauen. Nachdem in dem GSP eine Lesesperre erworben wurde, kann die Nachricht an einen Besitzer einer Cache-Kopie weitergegeben werden (3).[4] Der Rechner C liest nun das Objekt aus seinem Cache (4) und sendet es direkt an den Initiator des Zugriffs (5). Der fette Pfeil der Nachricht (5) signalisiert, daß diese Nachricht ein Objekt beinhaltet. Sie ist daher im allgemeinen sehr viel größer als die bislang nur vorkommenden Kontrollnachrichten. Auf dem Rechner A wird das Objekt in den lokalen Cache eingefügt (6), eine Lesesperre erworben und der Anwendung mitgeteilt, daß sie fortgeführt werden kann (7.1). Gleichzeitig hierzu können wir dem Heimatrechner mitteilen, daß nun auch Rechner A Besitzer einer Cache-Kopie ist (7.2), damit die Owner-Liste entsprechend aktualisiert werden kann. Diese Nachricht signalisiert auch, daß der Heimatrechner nun seine Lesesperre im GSP freigeben



1: **Read_Req**
2: **Read_Home**
3: **Read_Forward**
4: Objekt lesen
5: **Read_Reply**
6: Objekt einfügen
7.1: Aktivieren
7.2 **Cache_Ins**
8. **Read_Unlock**

Abbildung 4.5: Lesender nichtlokaler Cache-Zugriff

4. Obwohl die Rechner B und C identisch sein können, betrachten wir hier nur den Fall, in dem sich die beiden Rechner unterscheiden. Die Situation, in welcher der Rechner B gleich dem Rechner C ist, betrachten wir in Fall 3.

kann. Diese Lesesperre verhindert, daß Schreibzugriffe während der Abarbeitung des betrachteten Lesezugriffs parallel laufen, und dies durch die veraltete Owner-Liste zu Inkohärenzen führt. Am Ende der Bearbeitung auf dem Rechner A wird die lokale Lesesperre wieder freigegeben (8).

Fall 3*: *Das Objekt wird von Platte oder aus dem Cache des Heimatrechners gelesen
Abschließend betrachten wir die Situation, in der entweder der Heimatrechner selbst eine Cache-Kopie besitzt oder aber überhaupt keine Cache-Kopie im System existiert (Abbildung 4.6). Diese Fälle betrachten wir zusammen, da beide Abarbeitungen fast identisch ablaufen. Nachdem festgestellt wird, daß das Objekt nicht lokal vorhanden ist, wird die Nachricht an den Heimatrechner weitergeleitet (1+2). Befindet sich das Objekt auch nicht im aggregierten, verteilten Cache, was leicht mit Hilfe der Owner-Liste auf dem Heimatrechner festgestellt werden kann, so wird das Objekt von Platte gelesen und in den lokalen Cache eingefügt (3). (Hier fließt jetzt die weiter oben getroffene Einschränkung ein, daß jedes Objekt auf der Platte seines Heimatrechners gespeichert ist.) Ab diesem Zeitpunkt läuft der Zugriff so weiter, als ob bereits von Anfang an eine Cache-Kopie am Heimatrechner existiert hätte. Von dem Heimatrechner werden die Daten an den Initiator zurückgesendet (4) und dort in den Cache eingefügt (5). Anschließend kann die Anwendung, die das Objekt angefordert hat, fortgesetzt werden (6.1). Aus den gleichen Gründen wie bei dem oben skizzierten nichtlokalen Cache-Zugriff, wird auch hier auf dem Heimatrechner eine Lesesperre erworben, die erst dann wieder freigegeben werden kann, wenn das Objekt am Initiatorrechner eingefügt und die Owner-Liste am Heimatrechner aktualisiert wurde (6.2). Wie bei den vorangegangenen Zugriffen muß die Anwendung nach der Bearbeitung des Objekts die lokale Lesesperre wieder freigeben. Sind die Rechner A und B identisch, so besteht der einzige Unterschied zu dem allgemeinen Fall darin, daß wir an Stelle der Netzwerknachrichten lokale Nachrichten benutzen.

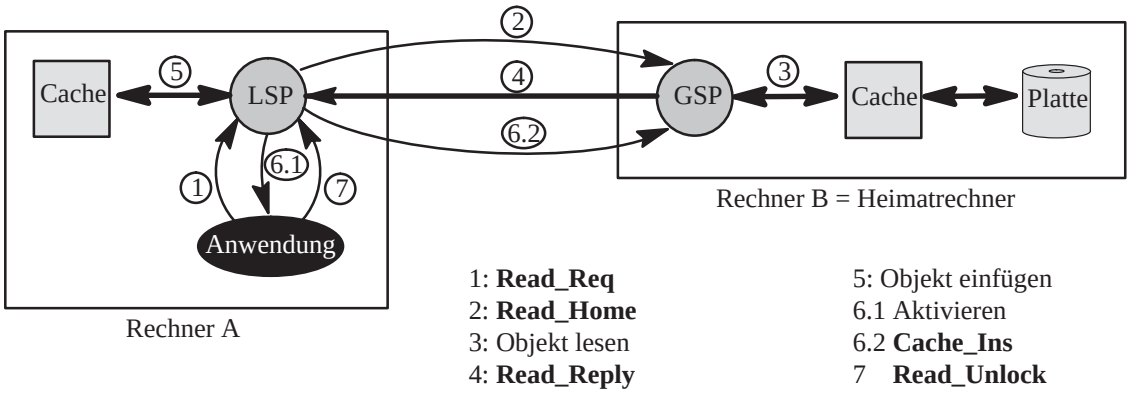


Abbildung 4.6: Lesender Cache- oder Plattenzugriff auf dem Heimatrechner

Die Abarbeitung von Schreibzugriffen

Nach der Beschreibung der verschiedenen Arten von Lesezugriffen wollen wir nun die Abarbeitung von Schreibzugriffen genauer untersuchen. Abbildung 4.7 illustriert die Aktionen im Schreibfall. Um zu verhindern, daß mehrere Rechner gleichzeitig Änderungen auf dem selben Objekt durchführen, müssen Schreibzugriffe innerhalb des Callback-Protokolls immer von dem Heimatrechner bearbeitet werden. Dies ist im Gegensatz zu den Leseanforderungen auch dann notwendig, wenn der lokale Rechner bereits eine Kopie des zu schreibenden Elementes besitzt.

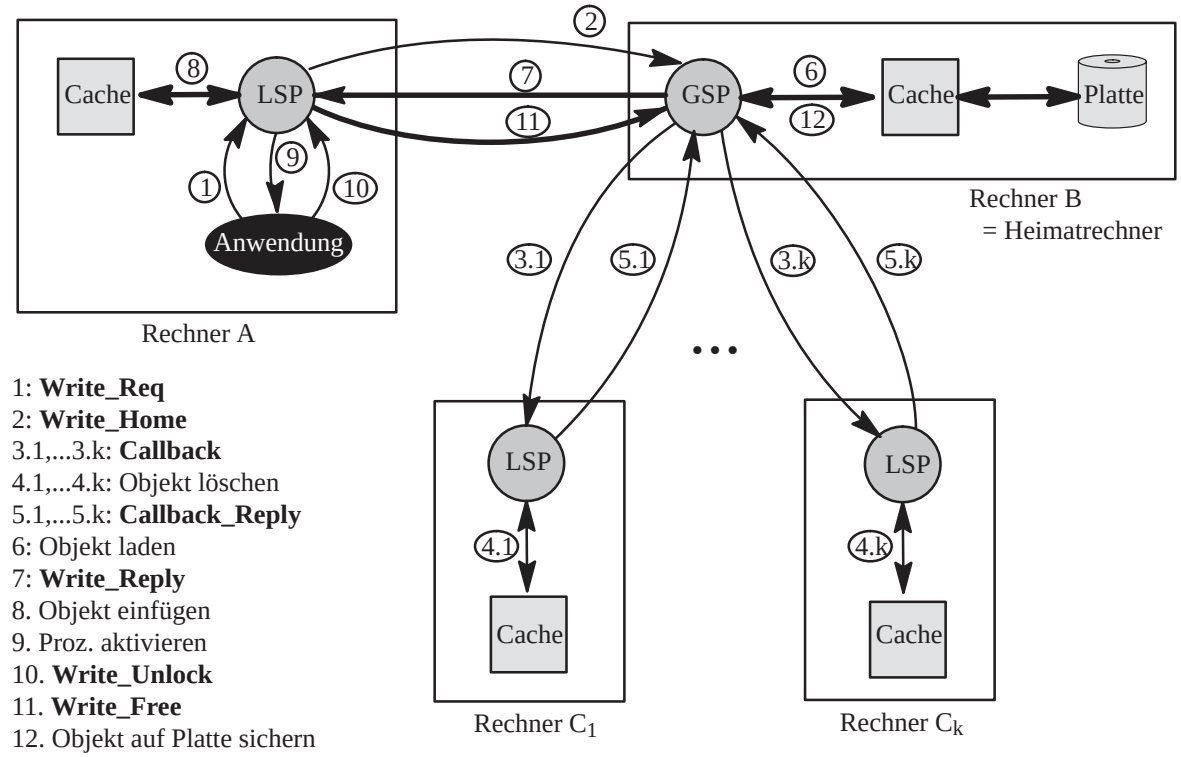


Abbildung 4.7: Schreibzugriff

Der LSP auf dem Initiatorrechner hat lediglich die Aufgabe sicherzustellen, daß die Reihenfolge der lokal erzeugten Lese- und Schreibzugriffe gleich bleibt (1). Empfängt der Heimatrechner eine Anfrage für einen Schreibzugriff von einem anderen Rechner (2), so kann er mit Hilfe der Owner-Liste feststellen, welche Rechner noch Kopien dieses Objekts im Cache halten. An alle diese Rechner sendet nun der Heimatrechner eine Nachricht, die besagt, daß der Rechner das entsprechende Objekt aus dem Cache entfernen soll (3.*). Empfängt ein Rechner, der das Objekt noch im Cache besitzt, eine solche Nachricht, so wartet er, bis eventuell laufende lokale Zugriffe beendet sind und ihre Sperren freigegeben haben. Anschließend löscht er das Objekt aus dem Cache (4.*) und sendet eine Bestätigung zurück an den Heimatrechner (5.*). Dieser Rückruf (*Callback*) aller Kopien im System gibt diesem Kohärenzprotokoll auch seinen Namen. Durch Netzverzögerungen kann es jedoch vorkommen, daß ein Rechner eine Callback-Nachricht erhält, obwohl er das Objekt bereits aus seinem lokalen Cache verdrängt hat. In diesem Fall kann er sofort eine Bestätigung an den Heimatrechner zurücksenden. Hat der Heimatrechner von allen potentiellen Besitzern einer Cache-Kopie jeweils eine Bestätigung erhalten, wird auf dem Heimatrechner eine Schreibsperre erworben, das Objekt von Platte gelesen (6) und dem Initiator wird zusammen mit der Schreiberlaubnis auch das Objekt übermittelt (7). Auf dem Initiatorrechner wird nun ebenfalls eine Schreibsperre erworben, und das Objekt wird in den Cache eingefügt (8). Nachdem die Anwendung aufgeweckt wurde (9) und ihre Änderungen des Objekts durchgeführt hat, wird zuerst wieder die lokale Sperre freigeben (10) und dann das geänderte Objekt an den Heimatrechner übermittelt (11). Gleichzeitig wird das Objekt auf dem Rechner A aus dem lokalen Cache gelöscht. Auf dem Heimatrechner wird das Objekt in den Cache eingetragen und um die Änderung dauerhaft zu machen, wird das Objekt auf Platte durchgeschrieben (12). Anschließend können wir auch auf dem Heimatrechner die Schreibsperre freigeben.

Dieses Protokoll hat den Nachteil, daß es für jeden Schreibzugriff zwei Festplattenzugriffe erfordert. Um dies zu vermeiden, wollen wir nun noch zwei Verbesserungen beschreiben. Betrachten wir zunächst den Plattenzugriff, der notwendig ist, um nach dem Callback das Objekt wieder in den Cache zu laden. Hier können wir eine Sonderbehandlung einführen, wenn sich das Objekt bereits im Cache des Initiator- oder des Heimatrechners befindet. Anstatt es auf diesen Rechnern zu löschen, können wir bei der Durchführung des Callbacks eine Sperre auf diesem Objekt allozieren und dadurch verhindern, daß andere Zugriffe auf dieses Objekt erfolgen können. Befindet sich das Objekt auf dem Initiatorrechner, so können wir neben dem Plattenzugriff auch den Datentransfer über das Netzwerk vermeiden. Dazu muß der Initiatorrechner bei der Bestätigung des Callbacks dem Heimatrechner nur eine entsprechende Mitteilung zukommen lassen. Besitzen weder der Heimatrechner noch der Initiatorrechner eine Cache-Kopie, so könnte man zusammen mit der Callback-Nachricht eine Aufforderung für das Senden des Objekts an einen oder mehrere Kopienbesitzer mitschicken. Wird eine solche Nachricht nur an einen Rechner gesendet, so kann dies zu einem Fehlzugriff führen, da dieser Rechner seine lokale Kopie schon gelöscht haben kann. Auf der anderen Seite kann eine Aufforderung an alle Besitzer einer Cache-Kopie zu einem sehr großen Netzoverhead führen, da mehrere Besitzer das Objekt zurücksenden können. Wegen dieser Problematik haben wir diese Erweiterung nicht in unser Protokoll aufgenommen. Eine einfache Möglichkeit, Plattenzugriffe in den meisten Fällen auch dann zu vermeiden, wenn sich das Objekt in dem Cache eines Rechners befindet, der weder der Initiator- noch der Heimatrechner ist, besteht darin, vor einem Schreibzugriff einen Lesezugriff auf das entsprechende Objekt durchzuführen und es dadurch in den lokalen Cache des Initiatorrechners zu transferieren.

Das “Durchschreiben” von Änderungen auf Platte, das ebenfalls einen Plattenzugriff für jede Schreiboperation verursacht, können wir umgehen, wenn wir mehrere aufeinanderfolgende Änderungen eines Objekts im Hauptspeicher erlauben. Den Verlust von bereits durchgeführten Änderungen können wir dabei durch das folgende Protokoll vermeiden. Wird ein Objekt durch eine Schreiboperation geändert, so wird es wie bisher auf den Heimatrechner transferiert und in den Cache eingefügt. Anstatt das Objekt jedoch auf Platte zu sichern, wird es nur als geändert markiert. Ein anderer Schreibzugriff kann nun seinerseits dieses Objekt ändern, ohne daß ein Plattenzugriff notwendig wird. Erst wenn das Objekt aus dem Cache des Heimatrechners verdrängt werden soll, müssen wir die aktuellen Daten für das Objekt auf Platte sichern.

Treten keine Rechner- oder Plattenausfälle auf, so garantiert sowohl das Durchschreiben als auch das Markieren von Objekten im Cache die Dauerhaftigkeit von Änderungen. Da in realen Umgebungen – im Gegensatz zu unserer Simulation – Ausfälle jedoch nicht ausgeschlossen werden können, muß die Dauerhaftigkeit von Schreiboperationen auf einer höheren Ebene durch eine eigene Recovery-Komponente sichergestellt werden. Der Aufwand für diese Recovery-Komponente ist jedoch nur von dem Protokoll zur Abarbeitung der Schreibzugriffe abhängig. Da dieses jedoch für alle Caching-Heuristiken, die wir in dieser Arbeit betrachten, gleich ist, berücksichtigen wir diesen zusätzlichen Aufwand in unserer Simulation nicht.

Die Abarbeitung von Objektmigrationen

Als letztes wollen wir die Aktionen beschreiben, die notwendig sind, um aus Gründen der Lastbalancierung die Migration eines Objekts durchzuführen. An Hand der Abbildung 4.8 beschreiben

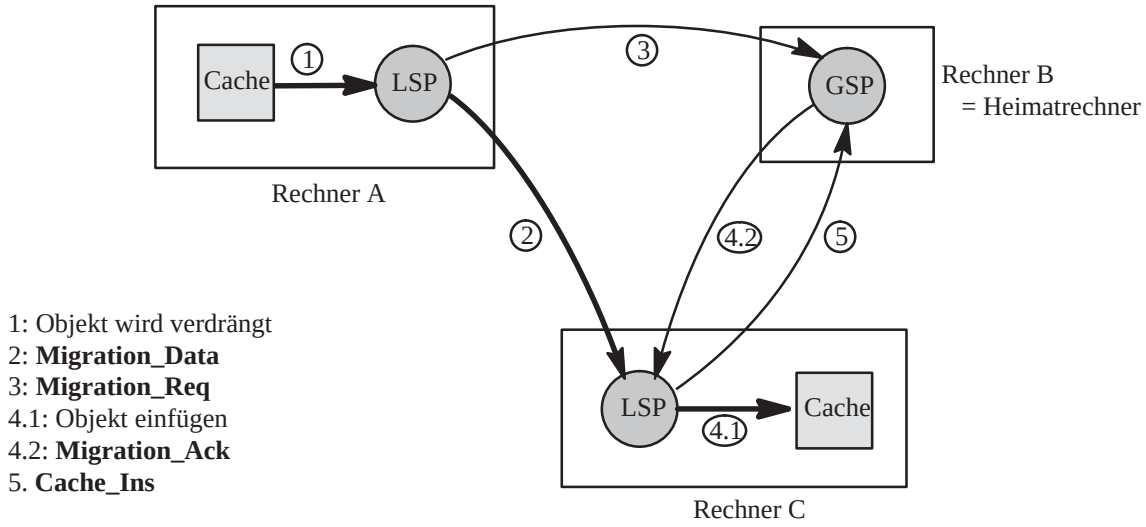


Abbildung 4.8: Eine Objektmigration

wir eine erfolgreiche Objektmigration. Die Aktionen, die bei einer fehlgeschlagenen Migration nötig werden, können den Übergangstabellen im Anhang C entnommen werden. Im Gegensatz zu den Lese- und Schreibzugriffen wird eine Migration nicht durch einen Anwendungsprozeß gestartet, sondern durch das Löschen eines Unikates aus dem Cache (1). Nachdem der LSP die Daten an den Zielrechner gesendet hat (2), wird eine Nachricht (3) an den Heimatrechner geschickt. Während nun überprüft wird, ob das empfangene Objekt für den Zielrechner C rentabel ist und daher in den Cache mit einer Migrationssperre eingefügt werden soll (4.1), wird auf dem Heimatrechner nachgeschaut, ob noch andere Zugriffe auf diesem Objekt arbeiten. Eine entsprechende Mitteilung wird an den Rechner C gesendet (4.2). Laufen aktuell keine weiteren Zugriffe auf dem Objekt, so wird bei dem Empfang dieser Nachricht die Migrationssperre aufgehoben, ansonsten wird das Objekt wieder aus dem Cache gelöscht (wenn es aus Rentabilitätsgründen zuvor eingefügt wurde). Die abschließende Nachricht (5) teilt dem Heimatrechner die entsprechende Aktion mit, damit dort die Sperre ebenfalls aufgehoben und gegebenenfalls die Owner-Liste aktualisiert werden kann.

4.1.4 Der Hitzemanager

Die Aufgabe des Hitzemanagers ist die Berechnung der für die verschiedenen Heuristiken notwendigen Informationen. Darunter fällt sowohl die Berechnung der lokalen (Abschnitt 3.1.1) und globalen Hitze (Abschnitt 3.1.2), als auch im Falle der kostenbasierten Heuristik die Protokollierung der Zugriffskosten (Abschnitt 3.1.4) und die Berechnung des Nutzens (Abschnitt 3.4.1). Die Berechnung und Verbreitung der Lastwerte der Rechner, die wir für die Lastverteilungsmethoden aus Abschnitt 3.5 benötigen, werden ebenfalls von dem Hitzemanager durchgeführt. Da wir die einzelnen Protokolle für diese Verfahren bereits im Kapitel 3 ausführlich beschrieben haben, wollen wir in diesem Abschnitt nur den Einfluß der verschiedenen Parameter auf die Genauigkeit der Approximation und die damit verbundene Leistungsfähigkeit der Methoden untersuchen. Zusätzlich müssen wir die Kosten zur Berechnung dieser Informationen bestimmen, um innerhalb unserer Simulation den durch die entsprechenden Methoden verursachten Aufwand berücksichtigen zu können.

Betrachten wir zunächst die zur Berechnung der lokalen Hitze notwendigen Parameter. Wie wir bereits in Abschnitt 3.1.1 gesehen haben, erhöht ein hoher Wert für den Parameter *k* die Genauigkeit der Hitzeapproximation während durch einen kleinen Wert eine bessere Adaptivität erreicht wird. Als guter Kompromiß zwischen diesen beiden gegenläufigen Zielen hat sich LRU-3 in unseren Messungen bewährt. Ein weiterer Parameter ist das Zeitraster, in dem Pseudozugriffe generiert werden. Auch hier müssen wir zwischen dem zusätzlichen Berechnungsaufwand und einer schnellen Adaption bei Laständerungen abwägen. Bei unseren Experimenten hat sich gezeigt, daß die Wahl dieses Parameters sehr unkritisch ist und daß ein Zeitraster von einer Sekunde – selbst bei sehr drastischen Laständerungen – eine genügend gute Adaptivität sicherstellt.

In Abschnitt 3.1.2 haben wir für das Protokoll zur Berechnung der globalen Hitze die beiden Schwellwerte Θ_{local} und Θ_{global} eingeführt. Obwohl wir diese Werte zur Herleitung einer Mindestgenauigkeit brauchen, hat der konkrete Wert wegen des zusätzlich verwendeten *Piggyback*-Mechanismus keinen großen Einfluß auf die erzielte Genauigkeit der Approximation. Der Einfluß auf die Anzahl der zu sendenden Nachrichten ist jedoch sehr viel größer, da durch einen höheren Schwellwert die lokalen Schwankungen der Hitze ohne zusätzlichen Nachrichtenaufwand ausgeglichen werden können. In unseren Simulationen setzen wir die (relativen) Schwellwerte jeweils auf 10 % des aktuellen Wertes.

Für die Protokollierung der Zugriffskosten wurden in Abschnitt 3.1.4 zwei unterschiedliche Verfahren vorgestellt. Um die Güte dieser Approximationsverfahren zu untersuchen, haben wir in Abbildung 4.9 die Antwortzeiten 200 aufeinander folgender Zugriffe auf die lokale Festplatte für Objekte mit variabler Größe gemessen. Neben den beobachteten Antwortzeiten sind auch die berechneten approximierten Geraden entsprechend der beiden Verfahren eingezeichnet. In der Abbildung erkennen wir sehr gut die bereits in Abschnitt 3.1.4 angesprochene Unzulänglichkeit der Methode der exponentiellen Glättung für Umgebungen mit variabel großen Objekten. Während die Methode der minimalen Fehlerquadrate sehr genau den größenunabhängigen Anteil der Rotationsverzögerung und der Suchzeit (12 ms) erkennt, führt die Approximation durch eine Ursprungsgerade bei der Methode der exponentiellen Glättung zwangsläufig zu einer Unterschätzung der Kosten für kleine Objekte. Daher benutzen wir das Verfahren mit der exponentiellen Gewichtung nur in Umgebungen mit Objekten konstanter Größe, während wir für variabel großen Objekte die Methode der minimalen Fehlerquadrate verwenden.

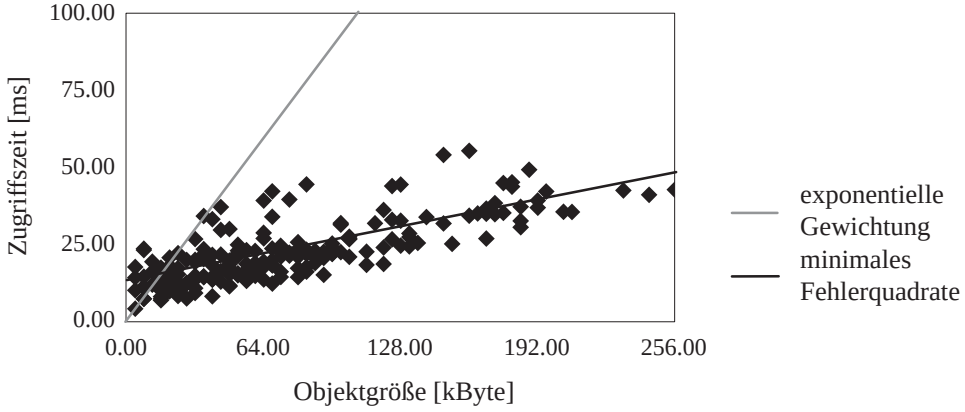


Abbildung 4.9: Vergleich der Approximationsverfahren für die Berechnung der Zugriffskosten

Um die Schwankungen in den Zugriffskosten abzuschwächen, die durch Warteschlangeneffekte auf den verschiedenen Ressourcen entstehen können, haben wir den Gewichtungsfaktor γ bei der exponentiellen Gewichtung auf 0.01 und die Fenstergröße w bei der Methode der minimalen Fehlerquadrate auf 50 gesetzt. Diese Werte glätten die Zugriffskosten genügend stark, ohne jedoch die Adaptivität bei Lastwechsel zu stark zu behindern.

Um den Overhead der Berechnungen in unserer Simulation berücksichtigen zu können, bestimmen wir abschließend die CPU-Kosten für die verschiedenen Aktionen. Auch hier haben wir wiederum Messungen auf einer *SUN*-Workstation durchgeführt und die Ergebnisse dieser Messungen sind in Tabelle 4.1 zusammengefaßt.

Lokale Hitzeberechnung (k=3)	1.0 µs
Durchführung eines Pseudozugriffs	0.5 µs
Kostenberechnung mit exponentieller Glättung	0.4 µs
Kostenberechnung mit minimalem Fehlerquadrat	2.5 µs
Berechnung des Nutzens	0.3 µs

Tabelle 4.1: CPU-Kosten für die Durchführung der Aktionen des Hitzemanagers

4.1.5 Der Cache-Manager

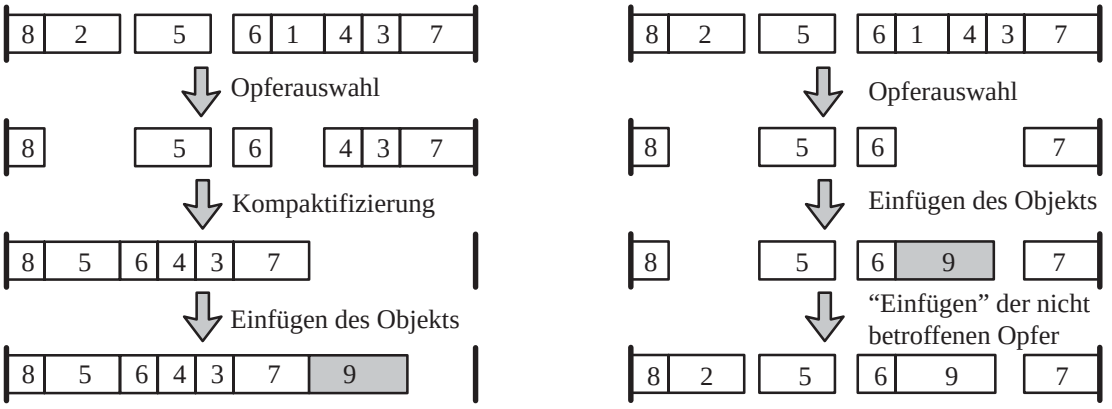
Der Cache-Manager implementiert eine Schnittstelle für das Einfügen, Lesen und Löschen eines Objekts in den bzw. aus dem lokalen Daten-Cache. Betrachten wir nur konstant große Objekte, so können diese ohne externe Fragmentierung [Tane92] im Cache plaziert werden. Im Falle einer Einfügung ersetzt ein Objekt genau ein anderes Objekt. Komplizierter wird das Cache-Management, wenn wir variable Objektgrößen zulassen. In diesem Fall verwalten wir eine Liste mit den freien Bereichen des Caches. Will ein Prozeß nun ein neues Objekt einfügen, so durchläuft er diese Liste und sucht sich entsprechend einer einfachen Heuristik einen passenden Bereich aus. Mögliche Heuristiken sind z.B. *first-fit*, bei der wir den ersten Bereich, oder *best-fit*, bei der wir den kleinsten Bereich auswählen, der groß genug für das Objekt ist. Wird kein passender Bereich gefunden, so müssen Ersetzungsopfer bestimmt werden, die dann aus dem Cache entfernt werden und dadurch Platz für das neue Objekt machen. Wird ein Cache-Bereich durch eine Löschoperation wieder freigegeben, so wird dieser zu der Liste der freien Bereiche hinzugefügt. Gleichzeitig wird versucht eventuell benachbarte freie Bereiche miteinander zu verschmelzen, um jeweils möglichst große zusammenhängende Speicherbereiche zu erhalten.

Die Reihenfolge, in der die im Cache vorhandenen Objekte für eine Ersetzung in Betracht gezogen werden, erfolgt entsprechend der in den Abschnitten 3.2, 3.3 und 3.4 vorgestellten Cache-Heuristiken. Während wir bei Objekten mit konstanten Größen für jedes neu einzufügende Objekt auch nur genau ein Objekt entfernen müssen, ist es bei variabel großen Objekten im allgemeinen notwendig, mehrere Ersetzungsopfer zu bestimmen. Da wir die kompletten Objekte nach Möglichkeit zusammenhängend auf Platte speichern, sollte für das Lesen eines Objekts auch jeweils nur ein einziger Plattenzugriff durchgeführt werden. Um dies zu ermöglichen, muß jedoch ein genügend großer, zusammenhängender Bereich im Hauptspeicher zur Verfügung gestellt

werden.[5] Zwei unterschiedliche Methoden, die jeweils einen solchen zusammenhängenden Bereich erzeugen, wollen wir nun näher beschreiben. Die Vorgehensweisen dieser Methoden sind an Hand eines Beispielszenarios in Abbildung 4.10 gezeigt. Die Zahlen in den Objekten geben jeweils deren Wert bezüglich der gewählten Cache-Ersetzungsstrategie an. Besitzt ein Objekt *p* eine höhere Zahl als ein Objekt *q*, so ist auch der Wert von *p* größer als der Wert von *q* und daher sollte das Objekt *q* vor dem Objekt *p* aus dem Cache entfernt werden.

Bei der ersten Methode (Abbildung 4.10(a)) werden jeweils so viele Opfer bestimmt, bis die Summe über alle Einträge in der Liste der freien Bereiche größer als die Größe des neuen Objekts ist. In unserem Beispiel sind die Objekte 1 und 2 zusammen mit den bereits existierenden Lücken größer als das neue Objekt 9. Da diese Bereiche jedoch im allgemeinen nicht zusammenhängend sind, müssen wir den Speicher kompaktifizieren [Tane92], d.h. die Objekte im Speicher werden so umkopiert, bis nur noch ein zusammenhängender, freier Bereich existiert. Vorteilhaft an diesem Verfahren ist, daß die Cache-Ersetzungsstrategie nicht beeinflußt wird. So sind die ausgewählten Opfer jeweils die Objekte im Cache, die den geringsten Wert entsprechend der gewählten Ersetzungsstrategie besitzen. Nachteilig sind jedoch die hohen Kosten für Einfügungen, die aus dem Umkopieren der Objekte im Speicher resultieren.

Bei der zweiten Methode (Abbildung 4.10(b)) vermeiden wir daher ein Umkopieren der Objekte. Dazu bestimmen wir so lange Ersetzungsopfer, bis durch das Löschen dieser Opfer ein genügend großer, zusammenhängender Bereich erzeugt wird. In diesen Bereich fügen wir dann das neue Objekt ein. Da jedoch im allgemeinen mehr Opfer bestimmt werden, als zu der benötigten Lücke beitragen, können wir diejenigen Opfer, die sich nicht mit dem neu eingefügten Objekt im Speicher überschneiden, weiterhin im Cache halten. So kann in unserem Beispiel das Objekt 2 weiterhin im Cache verbleiben, da es sich nicht mit dem neu eingefügten Objekt 9 überschneidet. Im



(a)Einfügeoperation mit Kompaktifizierung

(b)Einfügeoperation ohne Kompaktifizierung

Abbildung 4.10: Mögliche Cache-Verwaltungsstrategien für variabel große Objekte

5. Erlaubt das Betriebssystem die Benutzung von *Virtual DMA* [HePa96], so besteht die Möglichkeit, diese Restriktion zu umgehen. Bei Virtual DMA braucht die Festplatte nur einen zusammenhängenden virtuellen Speicherbereich als Ziel, während die zu diesem Speicherbereich korrespondierenden physikalischen Seiten beliebig über den Cache verteilt sein können.

Gegensatz zu dem Verfahren mit Kompaktifizierung verfälscht somit diese Methode die gewählte Ersetzungsstrategie, da Objekte als Opfer bestimmt werden können, die wertvoller sind als die im Cache verbleibenden Objekte. In unserem Beispiel werden die Objekte 3 und 4 aus dem Cache verdrängt, obwohl sie wertvoller als das Objekt 2 sind.

Neben der unbedingten Einfügeoperation, bei der ein Objekt in jedem Fall in den Cache eingefügt werden muß, haben wir in Abschnitt 3.5 auch die Notwendigkeit einer bedingten Einfügeoperation gesehen. Diese Operation führen wir aus, in dem wir zunächst nach einer der beiden soeben vorgestellten Verfahren die potentiellen Ersetzungsopfer bestimmen und – in Abhängigkeit von dem Vergleich des Wertes der Opfer und des neuen Objekts – fügen wir entweder das neue Objekt ein, oder wir löschen es.

Wie wir bei der Beschreibung des Zugriffsmanagers in Abschnitt 4.1.3 gesehen haben, schreiben wir geänderte Objekte nicht direkt auf Festplatte zurück, sondern markieren diese im Cache lediglich als geändert. Wird nun ein solches Objekt als Ersetzungsopfer ausgewählt, so müssen wir vor dem Löschen die geänderten Daten auf Festplatte sichern. Dies führt jedoch zu einer sehr starken Verzögerung des Zugriffs, der ein neues Objekt in den Cache einfügen will. Um dies zu vermeiden, werden nicht geänderte Objekte bei der Ersetzung bevorzugt, da wir diese nicht auf Platte zurückschreiben müssen. Diese Bevorzugung führt jedoch zu einer sehr starken Veränderung der Caching-Strategie, da nicht mehr die wertvollsten Objekte, sondern nur noch bereits geänderte Objekte im Cache gehalten werden. Um dies zu vermeiden, besitzt jeder Cache-Manager einen Hintergrundprozeß, der jeweils den Anteil geänderter Objekte im Cache überprüft. Falls dieser Anteil eine bestimmte Schranke übersteigt, werden so lange geänderte Objekte auf Platte geschrieben, bis eine vorgegebene untere Schranke erreicht wird. Dies ist ein übliches Verfahren zur Vermeidung von synchronen Festplattenzugriffen bei Cache-Ersetzungen und wird auch in kommerziellen Systemen (z.B. in DB2 [Mull93b] oder Oracle [CoGu93]) verwendet.

Um den CPU-Overhead für Änderungsoperationen auf dem Cache abschätzen zu können, haben wir eine Reihe von Messungen für jede Ersetzungsstrategie durchgeführt. Hierbei haben wir für die verschiedenen Ersetzungsstrukturen entsprechende *LEDA*-Klassen [MNU97] benutzt. Für die LRU-basierten Heuristiken benutzen wir als Datenstruktur eine *Queue*, während wir für die übrigen Heuristiken jeweils *Priority-Queues* benötigen. In der Abbildung 4.11 sind die CPU-Kosten für das Einfügen in bzw. das Löschen eines Objekts aus der entsprechenden Ersetzungsstruktur zu sehen. Bis auf einige “Ausreißer” sind die Kosten für die verschiedenen Operationen unab-

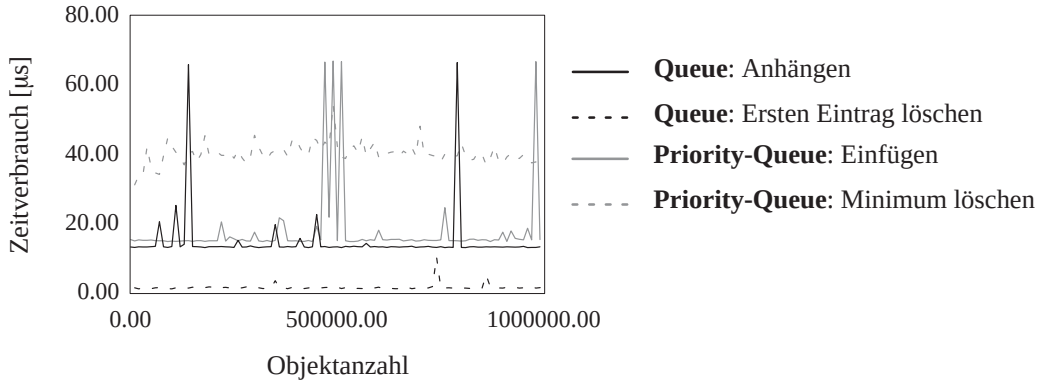


Abbildung 4.11: CPU-Kosten für die unterschiedlichen Cache-Operationen

hängig von der Anzahl der Objekte im Cache. Entsprechend dieser Messungen veranschlagen wir in unseren Simulationen 13 µs für das Anhängen an eine Queue und 1.5 µs für das Löschen des ersten Eintrages aus einer Queue. Bei den Priority-Queues belegen wir die CPU für 15 µs wenn wir ein Objekt einfügen und 40 µs wenn wir den minimalen Eintrag löschen.

4.1.6 Der Festplattenmanager

Da wir davon ausgehen, daß das Betriebssystem eine Schnittstellen zum Lesen und Schreiben der Festplatte bereitstellt, untersuchen wir in diesem Abschnitt nur den durch die Software verursachten CPU-Aufwand, damit dieser in unserer Simulation berücksichtigt werden kann. Zur Bestimmung dieses Overheads haben wir eigene Messungen auf einer *SUN*-Workstation durchführen. Um dabei den Einfluß des *UNIX*-File-Caches zu umgehen, messen wir – in Abhängigkeit von der Objektgröße – den CPU-Zeitbedarf für das Schreiben und Lesen auf eine *Raw*-Partition. Die Ergebnisse dieser Messungen sind in der Abbildung 4.12 zusammengefaßt. Entsprechend diesen Messungen approximieren wir den CPU-Overhead für das Lesen und Schreiben eines Objekts der Größe s_p durch die folgende Gleichung, die wir aus der Abbildung 4.12 abgeleitet haben:

$$T_{disk\_read} = T_{disk\_write} = \begin{cases} 12 \times s_p + 275\ \mu s & \text{falls } s_p < 128 \text{ kByte} \\ 1700\ \mu s & \text{sonst} \end{cases}$$

Überraschend ist, daß ab einer gewissen Objektgröße (128 kByte) die CPU-Kosten nicht mehr von der Größe abhängig sind.

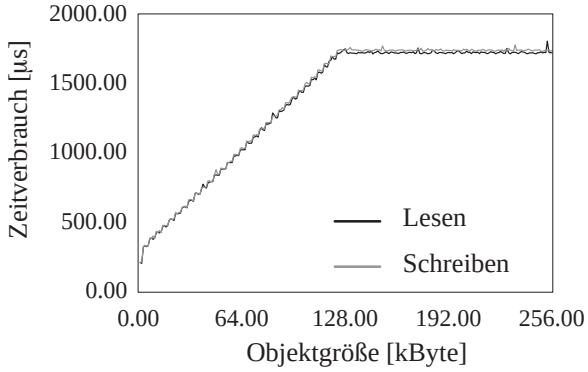


Abbildung 4.12: CPU-Kosten für die verschiedenen Festplattenoperationen

4.1.7 Der Netzwerkmanager

Auch bei dem Netzwerkmanager beschränken wir uns – ähnlich wie bei dem Festplattenmanager – auf die Untersuchung des CPU-Overheads für die Abarbeitung des Netzwerkprotokolls. Da dieser Mehraufwand von der benutzten Rechnerkonfiguration abhängt, haben wir Messungen an unserem lokalen Netzwerk durchgeführt. Dazu haben wir den Zeitbedarf für das Senden und das Empfangen verschieden großer Nachrichten gemessen. Der Rechner, auf dem wir die Messungen durchgeführt haben war eine Workstation vom Typ *SUN Sparc 4*, die an ein lokales *Ethernet*-LAN angeschlossen ist. Als Netzwerkprotokoll benutzen wir *TCP/IP* unter dem Betriebssystem

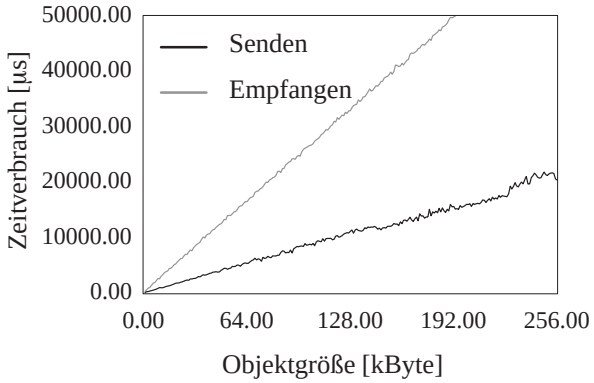


Abbildung 4.13: CPU-Kosten für die verschiedenen Netzwerkoperationen unter *TCP/IP*

Solaris 2.5. Die Ergebnisse der Messungen sind in der Abbildung 4.13 zusammengefaßt. Auch hier approximieren wir wiederum die CPU-Kosten für das Empfangen und für das Senden eines Objekts der Größe s_p durch eine Gerade. Die Koeffizienten für diese Geraden haben wir aus den in Abbildung 4.13 gzeigten Messungen abgeleitet:

$$T_{net\_recv} = 170 \times s_p + 85\ \mu s$$
$$T_{net\_send} = 85 \times s_p + 80\ \mu s$$

Wie wir in Abbildung 4.13 erkennen können, ist die Abarbeitung des *TCP/IP*-Netzwerkprotokolls eine sehr zeitaufwendige Aktion. In diesem Fall macht bei kleinen Nachrichten die Abarbeitung des Protokolls den größten Anteil an den Kommunikationskosten aus und verhindert dadurch, daß eine Erhöhung der Netzwerkgeschwindigkeit in einem gleichen Maße zu einer Reduktion der Übermittlungszeit führt. Aus diesem Grund wurden für schnelle Netzwerke (wie z.B. *Fast Ethernet*, *ATM* [Tane97] oder *Myrinet* [BCF*95]) neue Netzwerkschnittstellen und Protokolle entwickelt, die die Latenzzeit von Nachrichten reduzieren [EBBV95, PLC95, WBE96, WBE97, RAC97]. Hierbei können Gewinne vor allen Dingen dadurch erreicht werden, daß das Kopieren der Daten auf den unterschiedlichen Stufen des Protokolls vermieden und daß die Verwaltung der Netzwerkschnittstelle aus dem Betriebssystemkern herausgezogen wird. Mit dem in [WBE97] vorgestellten *U-Net*-Verfahren für *Fast-Ethernet* kann der CPU-Bedarf für das Senden bzw. das Empfangen von Nachrichten der Größe s_p entsprechend der folgenden zwei Formeln bestimmt werden:

$$T_{net\_recv} = \left\lfloor s_p / MaxPacketSize \right\rfloor \times 26 + \left(s_p \text{ div } MaxPacketSize\right) \times 1.5 + 5\ \mu s$$
$$T_{net\_send} = 5\ \mu s$$

Bei den Empfangskosten entspricht der erste Summand der Zeit für das Empfangen von Paketen der maximalen Größe *MaxPacketSize*, während die beiden folgenden Summanden die Kosten für das letzte – möglicherweise kleinere – Teilpaket berücksichtigen. Besonders erwähnenswert ist, daß die Kosten für das Senden bei diesem Verfahren unabhängig von der Größe des Objekts sind.

Ein asynchroner Sendepuffer

Für Umgebungen, in denen es nicht möglich ist auf ein modernes Netzwerkprotokoll umzusteigen, kann ein asynchroner Sendepuffer genutzt werden, um den Overhead für die Abarbeitung des *TCP/IP*-Protokolls zu reduzieren. Dazu wird die Methode des *Piggybackings* genutzt. Bei

dieser Methode werden Nachrichten, deren Übermittlung nicht extrem zeitkritisch ist, zunächst verzögert. Erst wenn eine zeitkritische Nachricht an den gleichen Empfänger gesendet werden muß, werden alle “aufgestauten”, an diesen Rechner adressierten Nachrichten mitübermittelt. Dadurch fällt der Overhead für die Abarbeitung des Netzwerkprotokolls nur einmal an. Um jedoch eine zu große Verzögerung einer Nachricht zu verhindern, wird der Sendepuffer durch einen Hintergrunddämon geleert, wenn entweder die Nachrichten ein bestimmtes Alter überschritten haben oder wenn der Puffer voll ist.

Um diese Methode effizient durchführen zu können, verwaltet jeder Netzwerkmanager eine Hashtabelle, in der die Zielrechner der Nachrichten als Schlüssel dient. Sollen Daten per *Piggybacking* verschickt werden, so wird ein entsprechender Eintrag in der Hashtabelle kreiert, und die Daten in den Puffer transferiert.[6] Vor dem Versenden einer Nachricht wird nun jeweils in der Hashtabelle nachgeschaut, ob Daten für den entsprechenden Zielrechner im Sendepuffer gespeichert sind. Falls solche Daten existieren, werden diese zusammen mit der ursprünglichen Nachricht weggeschickt. Die in unseren Experimenten benutzten Parameter für den asynchronen Sendepuffer sind in der Tabelle 4.2 zusammengefaßt.

Puffergröße	4096 Byte
Maximales Alter einer Nachricht	50 ms

Tabelle 4.2: Parameter des asynchronen Sendepuffers

4.2 Die simulierten Hardwarekomponenten

Da wir nicht die entsprechende Hardware zur Verfügung haben, um eine systematische Untersuchung der verschiedenen Caching-Heuristiken durchzuführen, sind wir gezwungen, die notwendige Hardware zu simulierten. In diesem Abschnitt werden wir die Modellierungen dieser Komponenten genauer untersuchen.

4.2.1 Modellierung des Daten-Caches

Der Daten-Cache besteht aus einem zusammenhängenden Speicherbereich, der durch seine Größe in Byte beschrieben ist. Da wir in unseren Experimenten keine realen Daten verwalten, ist es nicht notwendig diesen Speicherbereich wirklich zu allozieren, sondern es ist ausreichend die Adressen der freien und der belegten Bereiche zu protokollieren und dem Cache-Manager zugänglich zu machen.

Je nach verwendeter Cache-Verwaltungsmethode können jedoch Kopierkosten im Hauptspeicher auftreten, die wir in unserer Simulation berücksichtigen müssen. Während bei der Methode ohne Kompaktifizierung (siehe Abschnitt 4.1.5) jedes Objekt nur einmal in den Speicher geschrieben wird, und wir diese Kosten bei dem entsprechenden Platten- bzw. Netzzugriff berück-

6. Zur Bestimmung der CPU-Kosten für die Operationen auf der Hashtabelle können wir wiederum auf unsere Messungen aus der Abbildung 4.2 zurückgreifen. Da wir in unserer Anwendung nur Kontrollnachrichten asynchron versenden und diese jeweils nur aus 2 bzw. 3 skalaren Werte bestehen, vernachlässigen wir den Overhead für das Kopieren der Daten in den Sendepuffer.

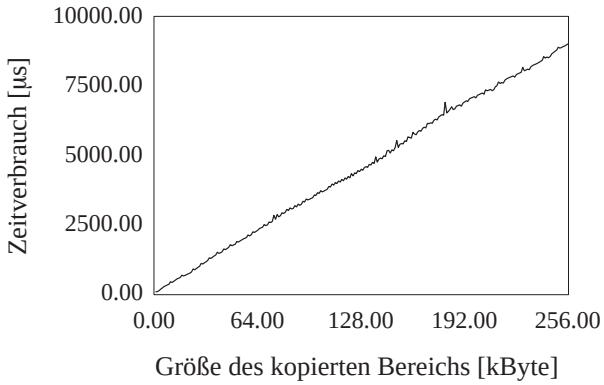


Abbildung 4.14: CPU-Kosten für das Umkopieren von Objekten im Hauptspeicher

sichtigen, können durch die Kompaktifizierung Objekte mehrfach im Cache umkopiert werden. Um auch diesen Aufwand in unserer Simulation berücksichtigen zu können, haben wir in Abbildung 4.14 die Zeiten für das Kopieren unterschiedlich großer Speicherbereiche mittels einer *for*-Schleife aufgetragen.[7] Aus dieser Abbildung können wir die folgende Formel zur Berechnung des CPU-Overheads für das Kopieren eines Speicherbereichs der Größe *size* kByte ableiten:

$$T_{copy} = 35 \times size + 60\ \mu s$$

4.2.2 Modellierung der Festplatte

In diesem Abschnitt wollen wir die Modellierung der Komponente untersuchen, die das Verhalten der Festplatten simulieren soll. In unserem Prototyp haben wir die Wahl zwischen zwei unterschiedlich detaillierten Modellen. Diese wollen wir zunächst vorstellen und anschließend quantitativ vergleichen.

Eine Festplattensimulation unter Benutzung der *DevSim* Bibliothek

Die Bibliothek *DevSim* [GiGr96] stellt Routinen bereit, die eine detaillierte Simulation der Hardware von Festplatten und Tertiärspeichermedien ermöglicht. Da wir in unseren Experimenten keine Tertiärspeicher benutzen, beschränken wir uns in diesem Abschnitt auf die Beschreibung der Festplattenmodellierung [RuWi94].

In Abbildung 4.15 ist der prinzipielle Aufbau einer Festplatte gezeigt. Physikalisch besteht eine Platte aus mehreren übereinander rotierenden Magnetscheiben. Für jede Oberfläche dieser Magnetscheiben existieren eigene Schreib-Leseköpfe, die jedoch gemeinsam auf einem Zugriffsarm befestigt sind. Jede einzelne Magnetscheibe ist logisch in konzentrische Kreise aufgeteilt, die *Spuren* genannt werden. Jede Spur wiederum besteht aus mehreren *Blöcken*, die die kleinsten Zugriffseinheiten der Festplatte darstellen. Da alle Schreib-Leseköpfe auf einem gemeinsamen Arm montiert sind, können alle Spuren, die exakt übereinander liegen, gelesen werden, ohne daß eine Bewegung des Armes notwendig wird. Die Menge aller dieser Spuren wird *Zylinder* genannt.

7. Unsere Messungen haben gezeigt, daß die *UNIX*-Systemroutine *memcpy* einen größeren Zeitverbrauch als eine *for*-Schleife besitzt. Daher betrachten wir hier nur die Ausführungszeit der *for*-Schleife.

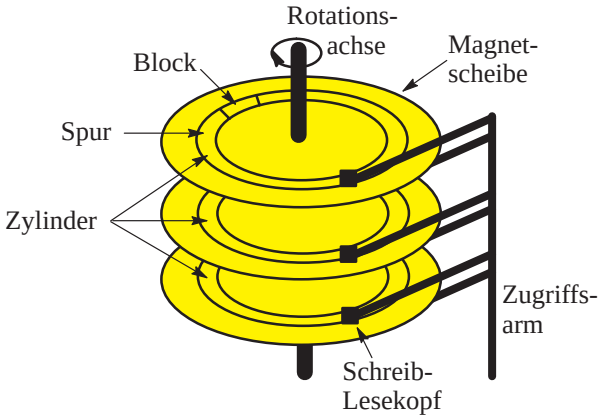


Abbildung 4.15: Prinzipieller Aufbau einer Festplatte

Um einen Zugriff auf einen Datenblock durchführen zu können, müssen wir zuerst den Schreib-Lesekopf über die entsprechende Spur bringen. Die Zeit, die hierfür notwendig ist, wird als *Suchzeit* bezeichnet. Die Suchzeit hängt ab von der Entfernung, die der Schreib-Lesekopf von der aktuellen Spur bis zur Zielspur zurücklegen muß und von den physikalischen Eigenschaften des Festplattenlaufwerkes. Allgemein können wir die Suchzeit in die folgenden Teile zerlegen:

- Beschleunigung des Arms auf seine maximale Geschwindigkeit,
- Bewegung des Arms mit seiner maximalen Geschwindigkeit,
- Abbremsen des Arms und
- genaues Positionieren des Arms über der Zielspur.

Nachdem diese Aktionen durchgeführt wurden, befindet sich der Schreib-Lesekopf über der korrekten Spur. Bevor jedoch der gesuchte Datenblock gelesen werden kann, muß noch gewartet werden, bis sich die rotierenden Magnetscheiben so weit gedreht haben, daß sich der gesuchte Block unter dem Schreib-Lesekopf befindet. Diese Zeit nennen wir *Rotationsverzögerung*. Die Größe der Rotationsverzögerung ist abhänigg von der Rotationsgeschwindigkeit der Platte und von dem momentanen Abstand des gesuchten Blocks vom Schreib-Lesekopf.

Die letzte Komponente, die zur Bestimmung der Zeit für einen Festplattenzugriff berücksichtigt werden muß, ist die *Transferzeit*. Diese Zeit bestimmt wie schnell die Datenblöcke von der Festplatte in den Rechner übertragen werden können. Auch diese Größe ist wiederum von der Umdrehungsgeschwindigkeit, aber auch von der Packungsdichte auf der Festplatte und der Bus-Geschwindigkeit abhängig. Darüberhinaus ist dies auch die einzige Komponente, die proportional zu der Größe des Objekts ist. Bei der Suchzeit und der Rotationsverzögerung geht die Größe nur in soweit ein, als diese Zeiten eventuell mehrfach anfallen können, falls das Objekt so groß ist, daß es über mehrere Spuren verteilt gespeichert ist.

Da in der heutigen Festplattentechnologie die Suchzeit zusammen mit der Rotationsverzögerung einen großen Anteil an der Zugriffszeit ausmachen, besitzen fast alle moderne Festplatten Caches. Eine häufig benutzte Methode zur Verwaltung solcher Platten-Caches ist die *A-Track-At-A-Time*-Strategie [Tane92]. Bei dieser Strategie wird bei einem Zugriff jeweils die gesamte Spur eingelesen, auf der sich der gesucht Block befindet. Dies verhindert, daß die Suchzeit und die

Rotationsverzögerung erneut anfallen, wenn eine kurze Zeit später ein neuer Zugriff einen anderen Block dieser Spur lesen muß, der Schreib-Lesekopf jedoch schon auf eine andere Spur bewegt wurde. Diese Strategie ist oftmals sinnvoll, da viele Anwendungen eine solche räumliche Lokalität in den Festplattenzugriffen aufweisen.

Zur Charakterisierung einer Festplatte innerhalb des soeben beschriebenen Modells benötigen wir die in der Tabelle 4.3 zusammengefaßten Parameter. Die zu den Parametern korrespondierenden Werte spezifizieren eine SCSI-Festplatte des Typs Seagate ST-15150.

Kapazität	4013 MByte
Blockgröße	512 Byte
Anzahl Schreib-Leseköpfe	21
Anzahl der Zylinder	3711
Anzahl der Blöcke pro Zylinder	2562
Größe des Caches	0 bzw. 17 Spuren
Rotationsgeschwindigkeit	7200 1/s
Suchzeit über einen Zylinder	0.6 ms
Suchzeit über alle Zylinder	17 ms
Strecke für Beschleunigungs- und Abbremsphase	370 Spuren

Tabelle 4.3: Parameter für die komplexe Modellierung einer Festplatte

Eine einfache Festplattensimulation

Neben der sehr detaillierten Modellierung der Festplatte durch die Bibliothek *DevSim* haben wir zusätzlich eine einfache Simulation der Festplatte implementiert. Der Grund hierfür sind die relativ aufwendigen Berechnungen für das detaillierte Modell. Bei dieser einfachen Modellierung vernachlässigen wir den nichtlinearen Anteil der Suchzeit, der aus der Beschleunigungs- und Abbremsphase resultiert. Daher berechnet sich die Zeit für das Bewegen des Schreib-Lesekopfes von der Spur *t1* auf die Spur *t2* durch die folgende Formel:

$$T_{seek} = \begin{cases} (|t2 - t1| - 1) \times \dfrac{\left(T_{full\_seek} - T_{cyl\_seek}\right)}{Z - 2} + T_{cyl\_seek} & \text{falls } t_1 \neq t_2 \\ 0 & \text{sonst} \end{cases} \qquad (4.1)$$

In dieser Formel ist Z die Anzahl der Zylinder, T_{full_seek} die Zeit für die Kopfbewegung vom ersten bis zum letzten Zylinder und T_{cyl_seek} die Zeit für die Kopfbewegung über einen einzelnen Zylinder.

Zur Modellierung der Rotationsverzögerung nehmen wir an, daß jeweils zwei aufeinander folgende Zugriffe unabhängig voneinander erfolgen. Unter dieser Annahme können wir die Rotationsverzögerung durch eine Gleichverteilung über dem Intervall von Null bis zur Zeit für eine komplette Rotation modellieren. Nehmen wir weiterhin an, daß die Transfergeschwindigkeit der Platte gegeben ist, so können wir schließlich die Zeit für den Transfer des Objekts *p* in den Hauptspeicher durch den Quotient aus der Objektgröße und der Transfergeschwindigkeit berechnen.

Einen Festplatten-Cache berücksichtigen wir in unserer einfachen Simulation nicht, da wir bei der Beschreibung des Lastgenerators in Abschnitt 4.3.1 sehen werden, daß wir die Zugriffe unab-

hängig voneinander generieren. Dies verhindert eine Lokalität der Zugriffe auf der Platte. Daher würde der Festplatten-Cache nur eine sehr geringe Trefferrate besitzen. Insgesamt brauchen wir daher für die einfache Modellierung der Festplatte nur die in Tabelle 4.4 aufgeführten Parameter.

Kapazität	4013 MByte
Blockgröße	512 Byte
Anzahl der Zylinder	3711
Rotationsgeschwindigkeit	7200 1/s
Suchzeit über einen Zylinder	0.6 ms
Suchzeit über alle Zylinder	17 ms

Tabelle 4.4: Parameter für die einfache Modellierung einer Festplatte

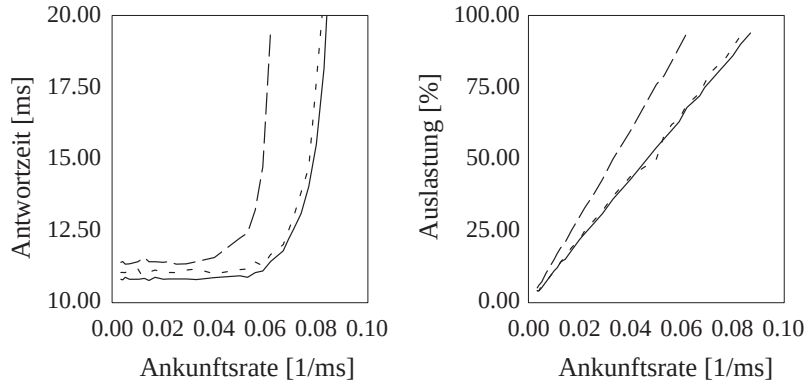
Quantitativer Vergleich der Festplattensimulationen

Um zu überprüfen, ob die einfache Festplattensimulation für unsere Experimente genügend genaue Ergebnisse liefert, werden wir in diesem Abschnitt die beiden Modellierungen an Hand von Simulationen miteinander vergleichen. Bevor wir jedoch die Ergebnisse vorstellen, müssen wir zuerst noch die Datenplazierung auf der Festplatte beschreiben, da diese einen entscheidenden Einfluß auf die Größe der Suchzeit hat.

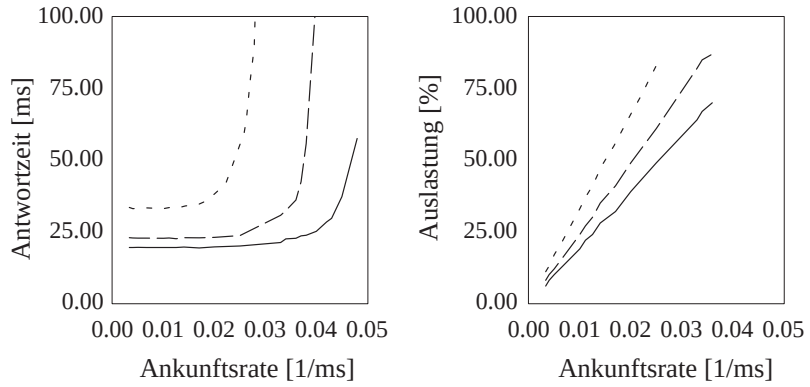
Wir nehmen an, daß die Objekte auf jeder Platte zufällig verteilt sind, daß jedoch alle Blöcke eines Objekts zusammenhängend gespeichert sind. Dies minimiert die Anzahl der Suchvorgänge während des Lesens eines Objekts. Durch Fragmentierung kann dies in einem realen Filesystem nicht immer gewährleistet werden, und die daraus resultierenden zusätzlichen Kopfbewegungen beim Lesen der einzelnen Blöcke eines Objekts können zu einer erheblichen Verschlechterung der Zugriffszeiten führen. Da unsere kostenbasierte Caching-Heuristik jedoch die Differenz zwischen den Zugriffszeiten auf den nichtlokalen Cache und auf die Festplatte ausnutzt, um die Gesamtleistung des Systems zu erhöhen, stellt die vorgestellte zusammenhängende Speicherung der Objekte den ungünstigsten Fall für unsere Heuristik dar.

In Abbildung 4.16 sind die Ergebnisse von zwei Experimentreihen aufgeführt. In der ersten Versuchsreihe (Abbildung 4.16(a)) betrachten wir nur Objekte der konstanten Größe 4 kByte. Eine unterschiedlich starke Auslastung der Festplatte erreichen wir dadurch, daß wir Zugriffe mit unterschiedlicher exponential verteilter Zwischenankunftszeit generieren. Aus den Diagrammen der Abbildung 4.16(a) können wir erkennen, daß die einfache Modellierung relativ gut mit der exakten Modellierung ohne Cache übereinstimmt. Die schlechten Ergebnisse bei der Benutzung des Platten-Caches können wir auf die oben angesprochene Cache-Strategie zurückführen. Da jeweils die komplette Spur – auch bei Anforderung nur eines Blocks dieser Spur – eingelesen wird, werden neu ankommende oder in der Warteschlange auf eine Bearbeitung wartende Zugriffe verzögert. Diese Zusatzkosten des Platten-Caches werden – wegen der fehlenden Lokalität im Zugriffsverhalten – nicht durch eine entsprechende Trefferrate ausgeglichen, so daß sich das Caching der Festplatte insgesamt negativ auswirkt.

In dem zweiten Experiment (Abbildung 4.16(b)) nehmen wir eine exponential verteilte Objektgröße mit Mittelwert 64 kByte an. Auch hier variieren wir wiederum den Mittelwert der Zwischenankunftszeit, um so die Festplatte unterschiedlich stark auszulasten. Im Gegensatz zu dem vorherigen Experiment erkennen wir nun deutliche Unterschiede zwischen den Ergebnissen von



(a) Objekte mit der konstanten Größe 4 kByte



(b) Objekte mit exponentialverteilter Größe mit Mittelwert 64 kByte

——— Lineare Approximation - - - - - *DevSim* — — — *DevSim* mit Cache

Abbildung 4.16: Vergleich der unterschiedlichen Festplattensimulationen

DevSim und unserer einfachen Simulation. Eine Ursache hierfür ist, daß in dem einfachen Modell für jeden Objektzugriff nur genau eine Suchzeit und eine Rotationsverzögerung auftritt, während *DevSim* auch die Aktionen berücksichtigt, die notwendig sind, wenn ein Objekt über eine Spur- oder Zylindergrenze hinausgeht. Auffallend an diesem Experiment ist, daß bei diesen großen Objekten nun auch der Einsatz eines Caches zu einer Verbesserung der Antwortzeit führt. Dies hängt damit zusammen, daß nach unserer Annahme alle Blöcke, die zu einem Objekt gehören, physikalisch hintereinander auf Platte liegen. Besteht nun ein Objekt aus mehreren Blöcken, so befinden sich diese Blöcke sehr wahrscheinlich auf der selben Spur. Wird nun – entsprechend der Caching-Strategie der Festplatte – die komplette Spur gelesen, so ist der Anteil der Blöcke, die von dem aktuellen Zugriff wirklich benötigt werden, größer. Obwohl auch hier die Trefferrate auf den Platten-Cache gering ist, ist sie doch groß genug, um den Zusatzaufwand für das Caching mehr als auszugleichen.

In den beiden beschriebenen Experimenten berechnet die einfache Simulation stets eine geringere Zugriffszeit als die detaillierte Simulation durch *DevSim*. Aus den bereits oben angeführten Gründen profitiert unsere kostenbasierte Heuristik jedoch von langsamen Festplattenzugriffen. Daher können wir trotz der relativ großen Abweichung bei großen Objekten immer die einfache

Simulation benutzen; eine detailliertere Simulation müßte – wegen der größeren Zugriffszeiten auf Platte – zu einer weiteren Verbesserung unseres Verfahrens führen.

4.2.3 Modellierung des Netzwerks

Die Komponente, welche die Kommunikation zwischen den verschiedenen Rechnern ermöglicht, ist der Netzwerkmanager. Ähnlich wie bei dem Festplattenmanager haben wir auch hier die Wahl zwischen zwei Modellen, die sich in ihrem Detaillierungsgrad unterscheiden. Zunächst beschreiben wir die beiden Methoden detailliert und vergleichen sie anschließend an Hand von Messungen miteinander.

Eine Netzwerksimulation unter Benutzung der *NetClient* Bibliothek

Die Bibliothek *NetClient* [Zenn97] stellt eine detaillierte Modellierung des *Internet*-Protokolls zur Verfügung. Die unterste Ebene dieser Protokolle bildet die Mediumzugriffsschicht, die sicherstellt, daß jeweils nur eine Station innerhalb des lokalen Netzwerks sendet. Oberhalb dieser Schicht befindet sich die Übermittlungsschicht, welche die Übertragung und das Routing zwischen den verschiedenen Rechnern bzw. Netzwerken durchführt. Als oberste Schicht stellt die Bibliothek *NetClient* eine an den *UNIX-Sockets* orientierte Transportschicht zur Verfügung. Im folgenden wollen wir diese Schichten und ihre möglichen Implementierungen detaillierter vorstellen.

Für die Mediumzugriffsschicht existieren mehrere Methoden, von denen in der Bibliothek die Verfahren des *Ethernets* mit *CSMA/CD* (IEEE 802.3) und des *Token-Bus* (IEEE 802.4) implementiert sind. Bevor ein Rechner A bei der *CSMA/CD*-Methode (***C****arrier* ***S****ense* ***M****ultiple* ***A****ccess with* ***C****ollision* ***D****etection*) beginnt ein Paket zu senden, hört er den Netzwerk-Bus ab. Stellt er fest, daß bereits ein anderer Rechner sendet, so wartet er bis das Netzwerk wieder frei ist; ansonsten beginnt er zu senden. Durch die endliche Ausbreitungsgeschwindigkeit kann es jedoch dazu kommen, daß bereits ein anderer Rechner B zu senden begonnen hat, ohne daß der Rechner A dies bemerkt hat. Beginnt Rechner A nun seinerseits zu senden, so führt das gleichzeitige Senden der Rechner A und B zu einer Kollision. Durch das Mithören am Bus bemerken die sendenden Rechner, wenn eine Kollision auftritt. Jeder Rechner, der eine Kollision feststellt, bricht seine Übertragung sofort ab und sendet ein spezielles Kollisionssignal. Um sicherzustellen, daß eine Kollision erkannt wird bevor die Übertragung beendet ist, muß jedes Teilpaket bei *CSMA/CD* eine Mindestlänge haben, die von der maximalen räumlichenAusdehnung des Netzwerks abhängig ist. Nach einer Kollision bestimmt jeder beteiligte Rechner einen Startzeitpunkt für einen neuen Sendeversuch, entsprechend des *Exponential Backoff Algorithmus*. Dieser Algorithmus versucht durch eine probabilistische Berechnung des Startzeitpunkts für einen erneuten Sendeversuch, zukünftige Kollisionen zu vermeiden. Eine Anpassung an die Netzlast erfolgt dabei dadurch, daß nach einer Kollision die mittlere Wartezeit exponentiell von der Anzahl lokal erfolgter Kollisionen seit der letzten erfolgreichen Übermittlung eines Paketes abhängt. Die für die Simulation des *Ethernets* notwendigen Parameter – entsprechend des Standards IEEE 802.3 – sind in der Tabelle 4.5 zusammengefaßt.

Transferrate	10 Mbit / s
Minimale Paketgröße	64 Byte
Maximale Paketgröße	1500 Byte
Daten-Overhead	26 Byte

Tabelle 4.5: *Ethernet*-Parameter entsprechend IEEE 802.3

Im Gegensatz zu *Ethernet* vermeidet die *Token-Bus*-Methode Kollisionen vollständig. Ebenso wie bei *Ethernet* sind auch hier alle Rechner an ein Bus-Netzwerk angeschlossen, jedoch nehmen wir nun an, daß alle beteiligten Rechner logisch einen Ring bilden. Demnach besitzt jeder Rechner einen eindeutigen logischen Vorgänger und Nachfolger. Um Kollisionen zu vermeiden, darf jeweils nur der Rechner senden, der im Besitz eines speziellen, innerhalb des Systems nur einmal vorkommenden, *Tokens* ist. Beendet ein Rechner seinen Sendevorgang, so übergibt er das Token an seinen logischen Nachfolger. Besitzt dieser Rechner ebenfalls Daten, die er versenden will, so darf er dies nun tun; ansonsten übergibt er das Token direkt an seinen Nachfolger. Um eine Blockierung des Netzwerks durch einen einzigen Rechner zu vermeiden, darf jeder Rechner maximal eine vorgegebene Zeitspanne im Besitz des Tokens sein, bevor er es an seinen Nachfolger weiterreichen muß. Die für die Simulation des *Token-Bus* benutzten Parameter sind in der Tabelle 4.6 aufgeführt.

Transferrate	10 Mbit / s
Maximale Token-Besitzzeit	10 ms
Maximale Paketgröße	8182 Byte
Daten-Overhead	20 Byte

Tabelle 4.6: *Token-Bus*-Parameter entsprechend IEEE 802.4

Da wir in unseren Simulationen jeweils nur von einem lokalen Netzwerk ausgehen, beschreiben wir die Vermittlungsschicht nicht genauer. Auf unsere Messungen nimmt sie nur in sofern Einfluß, als sie pro Paket einen Overhead von 20 Byte hinzufügt.

Wichtiger ist die Transportschicht, die in der *NetClient*-Bibliothek durch das *TCP*-Protokoll (***T****ransmission* ***C****ontrol* ***P****rotocol*) realisiert ist. Dieses verbindungsorientierte Protokoll ermöglicht eine zuverlässige Kommunikation über unsichere Netzwerke. Zusätzlich erfolgt eine Flußkontrolle, so daß Pufferüberläufe am Empfänger vermieden werden können. Um die zuverlässige Übertragung sicherzustellen, tauschen die an der Verbindung beteiligten Rechner Quittungen für empfangene Datenpakete aus. Durch einen *Time-Out*-Mechanismus wird auf ausbleibende Quittungen mit einer Wiederholung der Übertragung reagiert. Um den dadurch verursachten Overhead gering zu halten, wird versucht, jeweils mehrere Pakete gemeinsam zu bestätigen. Diese Quittungen werden darüberhinaus auch benutzt, um dem Sender jeweils die aktuelle Größe des Empfangspuffers mitzuteilen und dadurch kann verhindert werden, daß der Sender schneller Daten liefert als der Empfänger abnehmen kann. Schließlich fügt das *TCP*-Protokoll die empfangenen Teilpakete wieder entsprechend ihrer korrekten Reihenfolge zusammen und liefert das komplette Paket an die Anwendung weiter. Um alle diese Aufgaben erfüllen zu können, fügt das *TCP*-Protokoll zu jedem Teilpaket einen eigenen Header der Größe 20 Byte hinzu.

Eine einfache Netzwerksimulation

Nachdem wir in dem vorangegangenen Abschnitt eine sehr detaillierte Simulation des *Internet*-Protokolls vorgestellt haben, wollen wir in diesem Abschnitt eine einfache Approximation des

Netzwerkprotokolls beschreiben. Analog zu den Plattensimulationen ist auch diese Vereinfachung durch den hohen Berechnungsaufwand motiviert. In dieser einfachen Simulation betrachten wir nur die Mediumzugriffsschicht des Netzwerks. Das physikalische Netzwerk modellieren wir hierbei durch eine Bedienstation, d.h. zu einem Zeitpunkt kann jeweils nur ein Sendeauftrag bearbeitet werden und alle weiteren Aufträge werden in einer Warteschlange aufgereiht. Nach der Beendigung eines Auftrages wird in dieser nachgeschaut, ob dort noch ein Auftrag wartet, und gegebenenfalls wird der nächste Auftrag bedient. Um eine zu große Verzögerung von kurzen Nachrichten durch sehr lange Übertragungen zu vermeiden, wählen wir als Bedienstrategie *Round-Robin* mit Zeitscheiben. Die Größe der Zeitscheiben ist dabei so gewählt, daß sie der Übertragungszeit eines maximal großen Teilpaketes innerhalb des Mediumzugriffsprotokolls entspricht. Verschickt ein Prozeß Daten, die nicht innerhalb eines einzigen Paketes verschickt werden können, und warten andere Prozeße innerhalb der Warteschlange, so gibt der momentan aktive Prozeß nach einer Zeitscheibe die Bedienstation frei und stellt sich erneut am Ende der Warteschlange an. Um den Overhead durch das Verpacken des Objekts in Teilpakete zu berücksichtigen, erweitern wir alle zu versendenden Teilpakete um die notwendigen Header. Die Größen der entsprechenden Parameter entnehmen wir wiederum der Tabelle 4.5.

Quantitativer Vergleich der Netzwerksimulationen

Um die Genauigkeit unserer einfachen Simulation abschätzen zu können, werden wir in diesem Abschnitt einen quantitativen Vergleich mit der detaillierten Simulation *NetClient* durchführen. Dazu haben wir unseren Prototyp mit den unterschiedlichen Netzwerkmodellen konfiguriert und wir verändern die Belastung des Netzwerks durch eine Variation der Ankunftsrate der Zugriffe. Die Größe der Nachrichten schwankt in dieser Anwendung zwischen 26 Byte für eine einfache Kontrollnachricht und 4 kByte für den Transfer eines Objekts. Die mittlere Größe einer Nachricht beträgt 1 kByte. Die Ergebnisse der Messungen sind in der Abbildung 4.17 zusammengefaßt.

Während die Auslastung durch die reinen Daten für alle drei Verfahren gleich ist, unterscheiden sich die Gesamtauslastungen der verschiedenen Verfahren sehr stark (Abbildung 4.17(b)). So verändert sich die Auslastung in unserer einfachen Approximation proportional zu der Zwischenankunftsrate, während wir bei der *Ethernet*-Simulation eine lineare Abhängigkeit nur bei

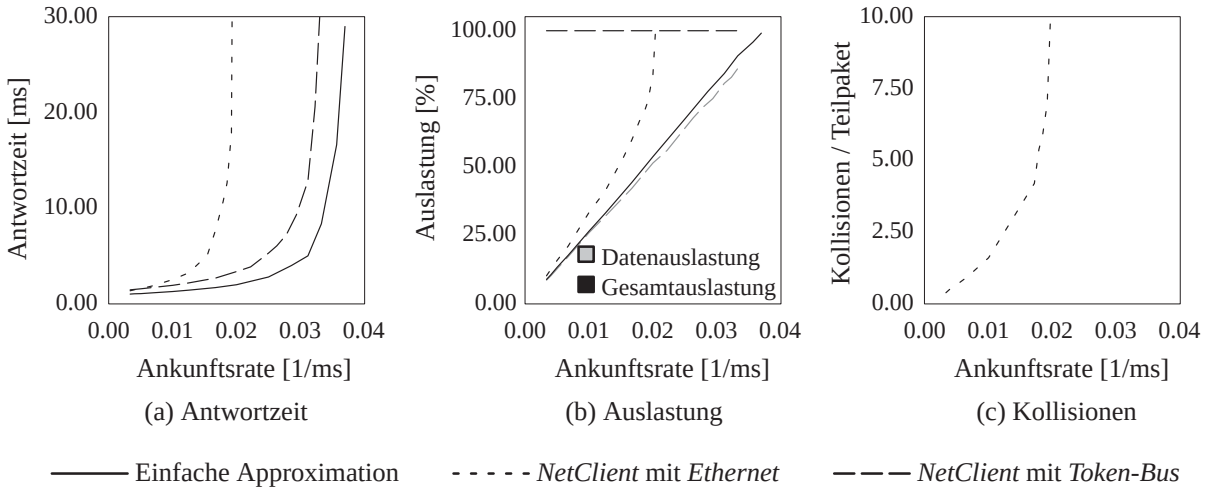


Abbildung 4.17: Vergleich der unterschiedlichen Netzwerksimulationen

leicht belastetem Netzwerk feststellen können. Erhöhen wir die Zwischenankunftsrate bei *Ethernet* weiter, so wächst die Auslastung plötzlich an. Der Grund hierfür liegt in der hohen Anzahl an Kollisionen (Abbildung (c)). Diese Kollisionen bewirken auch, daß die Antwortzeit sehr früh und sehr stark anwächst (Abbildung (a)). Obwohl die Auslastung bei dem *Token-Bus*-Verfahren ständig 100% beträgt (entweder sendet ein Rechner oder das Token rotiert), verhält sich die Antwortzeit bei einer hohen Zwischenankunftsrate besser als bei *Ethernet*, da Kollisionen explizit vermieden werden. Die schlechtere Performance des *Token-Bus* in der detaillierten Simulation gegenüber unserer einfachen Simulation liegt daher hauptsächlich in der Berücksichtigung des zusätzlichen Nachrichtenverkehrs, der durch das *TCP*-Protokoll hervorgerufen wird.

Bei unseren Messungen hat sich gezeigt, daß bei hoher Netzauslastung das *Ethernet*-Protokoll wegen der hohen Anzahl an Kollisionen nicht geeignet ist, während der *Token-Bus* durch die Vermeidung von Kollisionen gute Ergebnisse liefert. Der Nachteil des *Token-Bus* besteht jedoch darin, daß selbst bei einem völlig unbelasteten Netzwerk ein Rechner erst warten muß, bis er im Besitz des Tokens ist. Bei *Ethernet* hingegen kann ein Rechner direkt anfangen zu senden. In einem realen Netzwerk sollte daher die Wahl des Protokolls in Abhängigkeit von der der mittleren Auslastung des Netzwerks durchgeführt werden. In unseren Simulationen werden wir wegen des sehr großen Rechenaufwands der detaillierten Simulationen immer die einfache Approximation wählen. Dies ist gerechtfertigt, da bei geringer Last die Antwortzeit aller Methoden nahezu gleich sind; erst bei höherer Last unterscheiden sich die gemessenen Werte stärker voneinander, wobei jedoch die Antwortzeiten von *Token-Bus* noch hinreichend nahe bei denen unserer einfachen Simulation liegen.

4.3 Das Last- und Datenmodell

Wie wir bereits mehrfach erwähnt haben, läuft unser Prototypen innerhalb einer Simulationsumgebung. Hieraus folgt, daß wir keine realen Benutzer haben, die in unserem System Zugriffe auf reale Daten durchführen. Daher stellen wir in diesem Abschnitt zuerst einen Lastgenerator vor und beschreiben anschließend unser Datenmodell. Ein Vorteil synthetischer Last- und Datenmodelle besteht auch darin, daß diese uns eine systematische Variation über einen weiten Bereich von Last- und Datenkonfigurationen erlauben.

4.3.1 Der Lastgenerator

Die Aufgabe des Lastgenerators ist die Erzeugung einer synthetischen Last auf den einzelnen Rechnern. Dazu läuft auf jedem Rechner ein eigener Prozeß, der rechnerspezifisch konfiguriert werden kann. In diesem Abschnitt erläutern wir die Lastparameter, die wir zur Beeinflussung dieses Prozesses benutzen können. Zuerst stellen wir die Methode vor, nach der die lokalen Hitzen der verschiedenen Objekte berechnet werden. Danach leiten wir verschiedene Möglichkeiten zur Variation der Ähnlichkeit im Zugriffsverhalten zwischen den verschiedenen Rechnern her. Anschließend beschreiben wir, wie der Ankunftsprozeß von Zugriffen modelliert wird und wie wir damit die Rechneraktivität im System steuern können. Schließlich präsentieren wir noch einen Mechanismus, der es uns erlaubt Lasten zu generieren, deren Charakteristika sich dynamisch ändern.

Die Bestimmung der lokalen Hitze

Die Verteilung der lokalen Hitze beschreibt das lokale Zugriffsverhalten eines Rechners auf die im Gesamtsystem vorhandenen Objekte. Zur einfacheren Beschreibung betrachten wir zunächst nur einen einzelnen Rechner und nehmen an, daß die Objektnumerierung so gewählt wird, daß der Index eines Objekts dem Rang in einer absteigenden Sortierung nach der lokalen Hitze entspricht. Unter diesen Voraussetzungen ist die Hitze *heat*(*p*) eines Objekt *p* – entsprechende der Definition der Zipf-Verteilung aus Abschnitt 2.4 – gleich:

$$heat(p) = \frac{1}{C \times p^{\theta}} \text{ mit der Normierungskonstanten } C = \sum_{q=1}^{M} 1/q^{\theta}$$

In dieser Formel ist *M* wiederum die Gesamtanzahl der Objekte und θ bestimmt die Schräge der Verteilung.

Eine Rechtfertigung für die Wahl dieser Funktion können wir z.B. aus der Analyse von Traces gewinnen. In Abbildung 4.18 ist das Zugriffsverhalten des Traces eines HTTP-Servers (*www.leo.org*) skizziert. Da CGI-Skripte nicht gecacht werden können, haben wir diese aus dem Trace entfernt. Zusätzlich sind – um statistisch nicht-signifikante Zugriffe zu vermeiden – nur Zugriffe auf solche Objekte berücksichtigt, die mindestens dreimal angefordert werden. Insgesamt lieferte uns dies ca. 1.7 Millionen Zugriffe. Wie wir in Abbildung 4.18 erkennen können, erreichen wir durch eine geeignete Wahl der Schräge der Zipf-Funktion eine gute Übereinstimmung zwischen dem Trace und der Zipf-Verteilung.

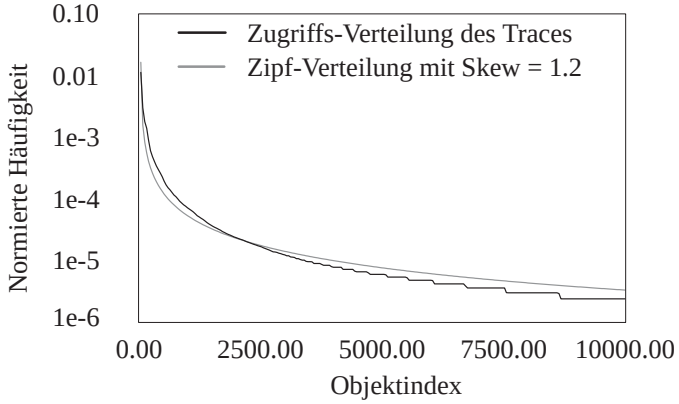


Abbildung 4.18: Zugriffsverteilung eines HTTP-Traces

Bestimmung der Ähnlichkeit im Zugriffsverhalten zwischen den Rechnern

In der vorangegangenen Beschreibung haben wir angenommen, daß die Numerierung der Objekte entsprechend ihres Rangs in der Sortierung nach lokaler Hitze erfolgt. Da jedoch die Zuordnung der Indizes zu den Objekten im gesamten System gleich sein muß, folgt daraus, daß auch die Rangfolge der Objekte auf den verschiedenen Rechnern gleich sein müßte. Dies stellt aber eine zu starke Einschränkung der möglichen Lasten dar. Durch die Einführung einer zusätzlichen Abbildung zwischen den Indizes und den Objekten können wir diese Beschränkung aufheben.

Eine erste Möglichkeit zur Konstruktion einer solchen Abbildung bietet eine *zyklische Verschiebung* der Indizes. Zur Konstruktion einer zyklischen Verschiebung um den Wert σ, benutzen wir

für jeden Rechner i einen Vektor $shift_i$, der folgendermaßen definiert ist:

$$shift_i[p] = (p + (i - 1) \times \sigma) \bmod M \quad \text{für alle } p \in [1, M]$$

Hierbei ist M wiederum die Anzahl der Objekte im System.

Mit Hilfe dieser Verschiebungsvektoren können wir nun aus der allgemeinen Hitzeverteilung – die wir z.B. entsprechend der oben beschriebenen Zipf-Verteilung berechnet haben – die lokalen Hitzen auf den einzelnen Rechnern bestimmen. Die lokale Hitze des Objekts p auf dem Rechner i ist dann definiert durch:

$$heat_i(p) = heat(shift_i[p]).$$

Diese Vorgehensweise verdeutlichen wir an Hand des Beispiels 4.2.

Beispiel 4.2:

In diesem Beispiel nehmen wir an, daß das System aus 3 Rechnern besteht, und daß der komplette Datenbestand 9 Objekte umfaßt. In Abbildung 4.19 ist skizziert, wie für diese Umgebung eine Verschiebung um σ=3 berechnet wird.

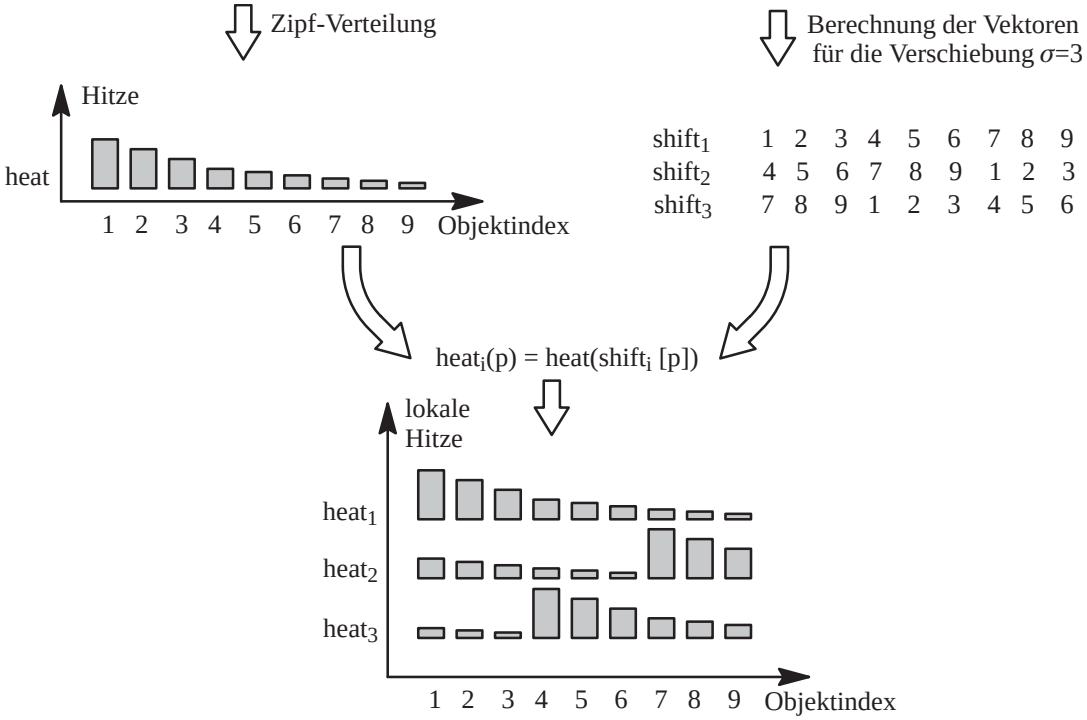


Abbildung 4.19: Variation des Zugriffsverhaltens durch eine zyklische Verschiebung

■

Eine weitere Möglichkeit ein unterschiedliches Zugriffsverhalten auf den verschiedenen Rechnern zu erreichen, besteht darin, anstatt einer Verschiebung eine parametrisierbare Permutation durchzuführen. In [LYW91] wird dazu die *Correlation* eingeführt, die folgendermaßen definiert ist:

Definition 4.1: (Correlation)

Sei M die Gesamtanzahl der Objekte im System und seien die Objekte entsprechend ihrer lokalen Hitze auf dem Referenzrechner i sortiert. Ein System besitzt die **Correlation** ξ, falls für jedes Objekt p, das auf dem Rechner i den Rang r besitzt, der Rang auf allen anderen Rechnern innerhalb des Intervalls $[1, \min(\xi+r-1, M)]$ liegt.

■

Es ist einfach einzusehen, daß entsprechend dieser Definition die Werte für die Correlation zwischen 1 und M liegen müssen. Ein Wert von 1 entspricht dabei einer vollkommenen Übereinstimmung des Zugriffsverhaltens auf allen Rechnern, während eine Correlation mit dem Wert M einer völlig zufälligen Zuordnung entspricht.

Für die Bestimmung eines Zugriffsmusters, das der Correlation ξ entspricht, können wir wiederum ein konstruktives Verfahren angeben, das ähnlich wie bei der zyklischen Verschiebung vorgeht. Auch hier konstruieren wir für jeden Rechner einen eigenen Vektor, der jedoch entsprechend der Definition der Correlation aufgebaut wird, und den wir $corr_i$ nennen:

$$corr_i[p] = \text{random}(1, \min(\xi + p - 1, M)) \quad \text{für alle } p \in [1, M]$$

Die Funktion *random(x,y)* implementiert eine Gleichverteilung auf den ganzen Zahlen zwischen den Grenzen x und y; berücksichtigt jedoch zusätzlich, daß ein Wert nicht mehrfach zurückgeliefert wird. Mit dem Correlation-Vektor können wir nun die lokalen Hitzen auf dem Rechner i entsprechend der folgenden Formel bestimmen:

$$heat_i(p) = heat(corr_i[p]).$$

Auch diese Möglichkeit verdeutlichen wir wiederum an Hand eines Beispiels.

Beispiel 4.3:

Abbildung 4.20 illustriert die Vorgehensweise um für 3 Rechner und 9 Objekte eine Correlation der Größe ξ=3 zu berechnen.

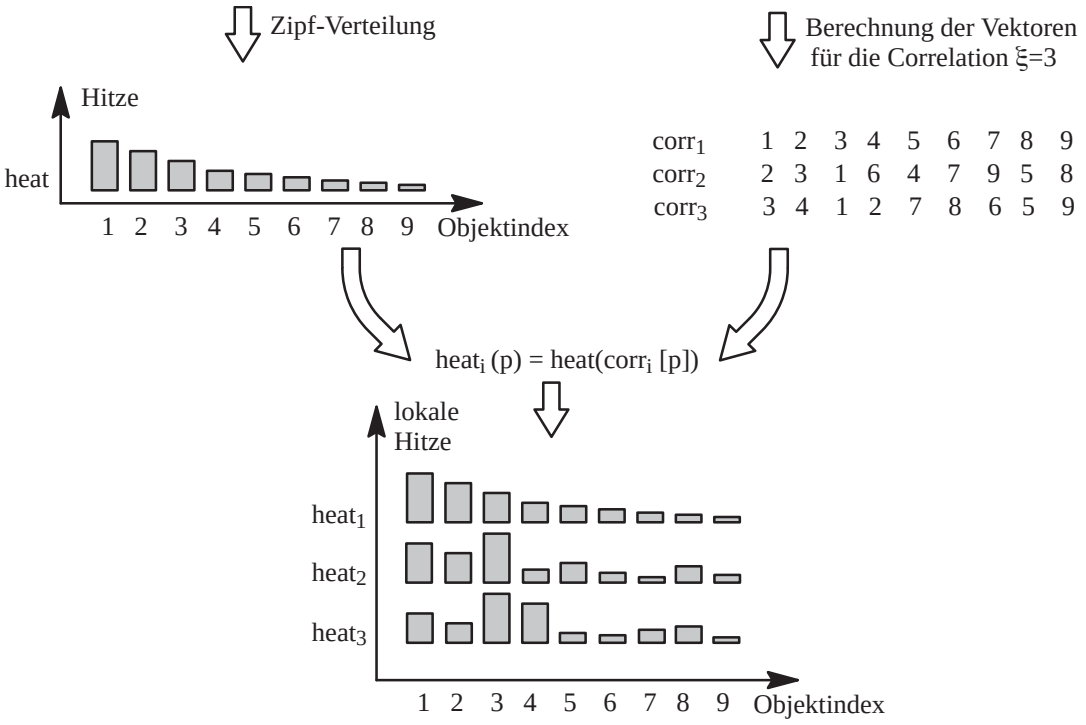


Abbildung 4.20: Variation des Zugriffsverhaltens durch eine Permutation

■

Die Metrik der Correlation besitzt jedoch mehrere Probleme, auf die wir im folgenden näher eingehen wollen. Als erstes hängt sie von der Wahl des Referenzrechners ab, wie an dem folgenden Beispiel deutlichen wird:

Beispiel 4.4:

Betrachten wir ein System, das aus zwei Rechnern besteht, auf denen die Rangfolge der Objekte entsprechend ihrer lokalen Hitze folgendermaßen gegeben ist:

Rechner 1: 1, 2, 3, ..., *M*

Rechner 2: *M*, 1, 2, ..., *M*-1.

Wählen wir Rechner 1 als Referenzrechner, so ist die minimale Correlation, die dieses Zugriffsverhalten generieren kann, gleich 2. Wählen wir hingegen Rechner 2 als Referenz, so können wir diese Situation nur dann erhalten, wenn wir die Correlation *M* annehmen.

■

Ein weiterer Nachteil dieser Metrik entsteht dadurch, daß sie nur die Rangfolge der Objekte betrachtet. Eine Vertauschung zwischen zwei Objekten mit sehr großen Unterschieden in der Hitze kann jedoch einen sehr viel stärkeren Einfluß auf die mittlere Antwortzeit haben, als das Vertauschen zweier Objekte mit fast identischer Hitze. Wegen dieser Unzulänglichkeiten der Correlation führen wir die neue Metrik Deviation als Maß für die Ähnlichkeit der Zugriffsmuster ein.

Definition 4.2: (Deviation)

Sei *M* die Gesamtanzahl der Objekte im System und sei s_p die Größe des Objekts *p*, dann ist die **Deviation δ** durch die folgende Gleichung definiert:

$$\delta \stackrel{def}{=} 1/Size \times \sum_{p=1}^{M} \left(s_p \times std\_dev_p \right)$$

wobei $Size = \sum_{j=1}^{M} s_p$ die Gesamtgröße der Daten und std_dev_p die empirische Standardabweichung der Hitze des Objekts *p* über alle Rechner ist.

■

Diese Metrik vermeidet zwar die Nachteile der Correlation, jedoch besitzen wir kein konstruktives Verfahren, um ein Zugriffsverhalten der Rechner für eine vorgegebenen Deviation zu erzeugen. Daher generieren wir in unseren Experimenten das Zugriffsverhalten jeweils für unterschiedliche Correlations entsprechend des oben beschriebenen Verfahrens und berechnen erst im nachhinein die Deviation.

Bestimmung der Rechnerlast

Wie wir bereits erwähnt haben, besitzt jeder Rechner einen lokalen Lastgenerator. Dieser erzeugt Zugriffe auf Objekte mit exponential verteilter Zwischenankunftszeit. Dadurch, daß bei der Kreation eines neuen Zugriffs der aktuelle Zustand des Systems nicht berücksichtigt wird, kann die Anzahl der im System vorhandenen Aufträge sehr stark schwanken. Diese Schwankungen bewirken, daß die Last für das System schwieriger zu handhaben ist als eine Last, bei der eine Beschränkung der Anzahl gleichzeitig aktiver Zugriffe (***M**ulti-**P**rogramming-**L**evel*) existiert. Der Nachteil einer Zulassungskontrolle durch die Vorgabe eines MPL besteht darin, daß durch eine zu konservative Wahl die Leistungsfähigkeit des Gesamtsystems nicht komplett ausgenutzt wird, da Zugriffe außerhalb des Systems warten müssen, obwohl noch genügend Ressourcen für ihre Bearbeitung bereit stehen. Zwar existieren bereits Ansätze zur automatischen Anpassung des MPL entsprechend der aktuellen Ressourcenauslastungen auf einem einzelnen Server [WHMZ94, Rahm97], jedoch ist eine Anpassung in einem verteilten System wesentlich komplexer. So können insbesondere ungleichmäßige Rechnerauslastungen dazu führen, daß sich an einem Rechner

Zugriffe stauen, während andere Rechner unterlastet sind. Aus diesen Gründen verzichten wir auf eine Zulassungskontrolle und überlassen unserer Caching-Heuristik die komplette Kontrolle über die Abarbeitung der Zugriffe. Da dies – wie bereits oben erwähnt – der schwierigere Fall ist, sollte unser System auch mit der Situation zurecht kommen, in der die Varianz der Last durch eine Zulassungskontrolle reduziert wird.

Da wir sowohl Lese- als auch Schreibzugriffe zulassen, reicht es nicht aus, eine einzelne Zwischenankunftsrate λ_{loc} für das Generieren von Zugriffen anzugeben. Daher spezifizieren wir beim Start des Lastgenerators zusätzlich die Schreibwahrscheinlichkeit π. Bei jedem Start eines Zugriffs würfeln wir entsprechend dieser Wahrscheinlichkeit die Art des neuen Zugriffs. Die daraus resultierenden Zwischenankunftsraten für Lese- bzw. Schreibzugriffe sind entsprechend:

$$\lambda_{loc,read} = \lambda_{loc} \times (1 - \pi)$$
$$\lambda_{loc,write} = \lambda_{loc} \times \pi$$

Zwar haben wir in dem vorangegangenen Abschnitt schon verschiedene Möglichkeiten beschrieben, wie wir das Zugriffsmuster zwischen den Rechnern variieren können, jedoch sind wir bis jetzt davon ausgegangen, daß auf jedem Rechner die gleiche Aktivität herrscht, d.h. daß die mittlere Ankunftsrate von Zugriffen auf allen Rechnern gleich ist. Um auch heterogene Lasten im System modellieren zu können wird in [LYW91] die relative Rechneraktivität vorgeschlagen. Im Gegensatz zu diesem Ansatz betrachten wir hier jedoch die absolute Rechneraktivität, da wir dadurch sicherstellen können, daß die Gesamtlast im System gleich bleibt. Analog zu der Definition der relativen Rechneraktivität definieren wir die (absolute) Rechneraktivität daher folgendermaßen.

Definition 4.3: (Rechneraktivität)

Sei N die Anzahl der Rechner im System, dann ist die **Rechneraktivität** des Rechners i durch die folgende Gleichung definiert:

$$node\_activity_i \stackrel{def}{=} \frac{N}{C \times i^{\eta}} \text{ mit der Normierungskonstanten } C = \sum_{j=1}^{N} 1/j^{\eta}$$

Den Parameter $\boldsymbol{\eta}$ ($\eta > 0$) bezeichnen wir als die **Schräge der Rechneraktivität**.

■

Den Einfluß der Rechneraktivität verdeutlichen wir an Hand des folgenden Beispiels.

Beispiel 4.5:

Für ein Netzwerk aus 10 Rechnern sind in der folgenden Tabelle die Rechneraktivitäten für die Schrägen 0, 1 und 1.5 aufgeführt.

Rechnerindex	1	2	3	4	5	6	7	8	9	10	Σ
Rechneraktivität η=0.0	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	10
Rechneraktivität η=1.0	3.41	1.71	1.14	0.85	0.68	0.57	0.49	0.42	0.38	0.34	10
Rechneraktivität η=1.5	5.00	1.77	0.96	0.63	0.45	0.34	0.27	0.22	0.19	0.16	10

Diese berechneten Rechneraktivitäten können wir nun benutzen, um für jedem Rechner eine individuelle Zwischenankunftsrate zu bestimmen. Dazu multiplizieren wir den entsprechenden Wert der Aktivität mit der global spezifizierten Zwischenankunftsrate. Dadurch

daß die Summe der Aktivitäten über alle Rechner unabhängig von der Schräge ist, stellen wir sicher, daß die Gesamtlast des Systems, d.h. die globale Ankunftsrate von Zugriffen, immer gleich ist.

■

Eine letzte Variationsmöglichkeit, die wir bis jetzt noch nicht besprochen haben, ist die Variation der Last über die Zeit. Dies ist sehr wichtig, um die Adaptivität unseres Verfahrens bei sich dynamisch ändernden Lasten untersuchen zu können. Um diese Variation zu ermöglichen, können für jeden Lastgenerator Phasen bestimmter Last definiert werden. Innerhalb dieser Phasen bleiben alle bisher beschriebenen Parameter konstant und nur beim Übergang zwischen zwei Phasen können sich die Parameter beliebig ändern. In unseren Simulationen werden wir insbesondere die Auswirkungen untersuchen, die sich ergeben, wenn das Zugriffsmuster eines Rechners beim Übergang zwischen zwei Phasen zyklisch verschoben wird.

4.3.2 Das Datenmodell

Abschließend gehen wir nun noch näher auf unser Datenmodell ein. Zuerst stellen wir die möglichen Allokationsarten der Objekte auf die verschiedenen Festplatten der Rechner vor. Die Allokation erfolgt jeweils beim Start der Simulation und bleibt während der gesamten Beobachtungsspanne konstant, d.h. wir betrachten keine Migrationen der Objekte auf Plattenebene. In unseren Experimenten werden wir die folgenden Allokationsmethoden benutzen. Bei der Beschreibung dieser Methoden gehen wir davon aus, daß das betrachtete System aus N Rechnern mit jeweils einer Festplatte besteht und der Datenbestand insgesamt M Objekte umfaßt:

- *Round-Robin-Partitionierung*: Das Objekt p wird auf der Platte des Rechners (p mod N)+1 gespeichert.
- *Bereichspartitionierung*: Auf dem Rechner i werden die Objekte $\{(i \times M/N), \ldots, (i+1) \times M/N - 1\}$ gespeichert. Ist M kein Vielfaches von N, so werden die Objekte so plaziert, daß der Unterschied in der Anzahl der Objekte pro Rechner maximal eins ist.
- *Zufällige Partitionierung mit der Schräge ψ*: Die Objekte werden zufällig über die Platten verteilt. Die Wahrscheinlichkeit, daß das Objekt p auf der Platte der Rechners i alloziert wird, ist gleich $1/\left(C \times i^{\psi}\right)$. In diesem Term dient C wiederum der Normierung und es gilt: $C = \sum_{j=1}^{N} 1/j^{\psi}$.

Da keine unserer Caching-Heuristiken den Inhalt von Objekten berücksichtigt, ist es ausreichend ein Objekt durch seine Größe zu beschreiben. Wie wir bereits in den vorangegangenen Abschnitten erwähnt haben, werden wir neben Experimenten mit Objekten konstanter Größe auch Messungen mit Umgebungen durchführen, in denen die Objektgröße variieren kann. Während wir bei den konstant großen Objekten annehmen können, daß jedes Objekt einer Seite (z.B. 4096 Byte) entspricht, müssen wir bei variabel großen Objekten die Größen entsprechend einer Verteilung bestimmen. Um eine realistische Verteilung zu wählen, haben wir die Objektgrößen des Traces eines HTTP-Servers untersucht. Innerhalb des von uns untersuchten Traces des Servers *www.leo.org* werden 30000 Objekte mit der Gesamtdatengröße 180 MByte zugegriffen. In Ab-

bildung 4.21 ist die mittlere Objektgröße $\bar{s}$ (6 kByte) zusammen mit der Größenverteilung gezeigt. Ähnlich wie bei den Zugriffsmustern in Abschnitt 4.3.1 können wir auch hier eine gute Übereinstimmung zwischen dem Trace und einer entsprechend parametrisierten Zipf-Verteilung erkennen. Trotz der guten Übereinstimmung werden wir in den Experimenten auch Umgebungen untersuchen, in denen die Objektgrößen entsprechend einer Normalverteilung oder einer Exponentialverteilung gewählt wurden, um dadurch die Eignung unserer Heuristik über einen möglichst breiten Bereich zu überprüfen.

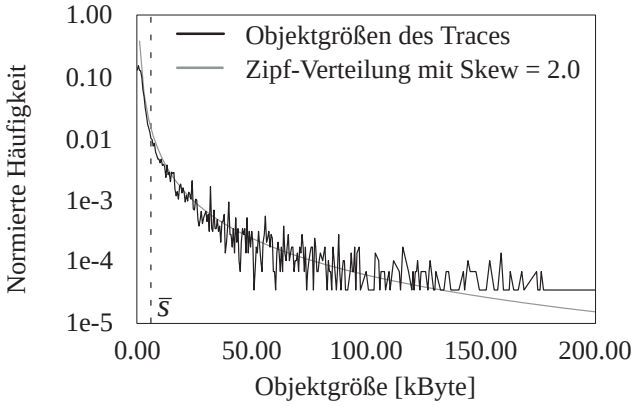


Abbildung 4.21: Objektgrößenverteilung in einem HTTP-Trace

Obwohl die Objektgrößen beliebige integrale Werte annehmen können, haben wir in Abschnitt 4.2.2 gesehen, daß die kleinste Zugriffseinheit auf einer Festplatte ein Block ist. Daher runden wir die Objektgrößen bei der Speicherung auf Festplatte jeweils auf den nächsten Block auf. Im Gegensatz dazu können bei der Speicherung im Cache und bei dem Transfer über das Netzwerk beliebige Größen angenommen werden. An diesen Stellen rechnen wir daher mit den wirklich ermittelten Objektgrößen.

Neben der Verteilung spielt auch die mittlere Objektgröße eine wichtige Rolle. Der Grund hierfür liegt darin, daß verschiedene Kosten unabhängig von der Objektgröße sind, andere aber von davon abhängen. Dies führt dazu, daß das Verändern der mittleren Größe die Leistungsfähigkeit der verschiedenen Heuristiken unterschiedlich stark beeinflußt. Betrachten wir zum Beispiel sehr große Objekte, die jeweils zusammenhängend auf Festplatte gespeichert sind. In diesem Fall ist die Suchzeit und die Rotationsverzögerung gegenüber der Übertragungszeit fast vollständig vernachlässigbar. Betrachten wir hingegen sehr kleine Objekte, so können wir die Übertragungszeit vernachlässigen und die Gesamtzugriffszeit wird durch die Suchzeit und die Rotationsverzögerung dominiert. Dies zeigt, daß die mittlere Objektgröße einen sehr großen Einfluß auf die Performance der Heuristiken nehmen kann, da erst die große Latenz bei Festplattenzugriffen (Rotationsverzögerung und Suchzeit) die Motivation für verteiltes Caching geliefert hat.

Zusätzlich zu der Variation der mittleren Objektgröße spielt natürlich auch das Verhältnis von der Gesamtgröße der Objekte zu dem insgesamt zur Verfügung stehenden Hauptspeicher eine entscheidende Rolle für die Leistungsfähigkeit der Heuristiken. Je größer die Datenmenge im Vergleich zum insgesamt vorhandenen, verteilten Cache wird, umso geringer wird die maximal mögliche Trefferrate und daher die Auswirkung durch die Wahl unterschiedlicher Caching-Heuristiken. Dies können wir an dem folgenden Beispiel verdeutlichen.

Beispiel 4.6:

Nehmen wir an, daß unser System aus 10 Rechnern besteht, von denen jeder 100 Objekte im Cache halten kann. Auf jedem Rechner herrscht das gleiche Zugriffsverhalten mit einer Schräge von θ=0.5. In der folgenden Tabelle sind die Cache-Trefferraten für eine rein-egoistische und eine rein-altruistische Heuristik gezeigt, wenn wir die Anzahl der Objekte von 1000 bis 1 Million variiert.

Objektanzahl	1000	10000	100000	1000000
Altruistische Heuristik	1.00	0.31	0.10	0.03
Egoistische Heuristik	0.30	0.09	0.03	0.01
Differenz	0.70	0.22	0.07	0.02

Tabelle 4.7: Einfluß der Datengröße auf die Cache-Trefferraten

Neben dem generellen Absinken der Trefferraten für die beiden Heuristiken bei einer Vergrößerung des Datenbestands, fällt in der Tabelle 4.7 auch die immer geringer werdende Differenz in den Trefferraten zwischen den beiden Heuristiken auf. Beide Effekte zusammen bewirken, daß der Spielraum für unsere kostenbasierte Heuristik, der aus dem Balancieren zwischen einer egoistischen und einer altruistischen Vorgehensweise resultiert, bei sehr großen Datenmengen immer geringer wird. Die Wahl einer konkreten Caching-Heuristik hat daher in solchen Situationen nur noch einen untergeordneten Einfluß auf die Performance des Gesamtsystems.

■

Kapitel 5

Experimentelle Evaluation

Nachdem wir im vorangegangenen Kapitel die prototypische Implementierung eines verteilten Datenverwaltungssystems mit den für unsere Cache-Heuristiken notwendigen Erweiterungen vorgestellt haben, wollen wir in diesem Kapitel die Leistungsfähigkeit der verschiedenen Heuristiken miteinander vergleichen. Zunächst besprechen wir in Abschnitt 5.1 einige Probleme, die mit dem prinzipiellen Ablauf der Experimente zusammenhängen. In Abschnitt 5.2 untersuchen wir dann den Einfluß der Lastparameter auf die verschiedenen Heuristiken. Um hier eine systematische Untersuchung zu ermöglichen, benutzen wir synthetische Lasten, deren Parameter wir über einen weiten Bereich variieren. Anschließend beschreiben wir in den Abschnitten 5.3 und 5.4 Experimente, in denen wir die Daten- bzw. die Systemparameter variieren. In Abschnitt 5.5 untersuchen wir die Adaptivität der verschiedenen Verfahren bei dynamisch schwankenden Lasten und abschließend fassen wir in Abschnitt 5.6 die Ergebnisse unserer experimentellen Studie zusammen.

5.1 Vorbemerkungen zu den Experimenten

5.1.1 Namenskonventionen

Um die verschiedenen Heuristiken in den Experimenten einfacher bezeichnen zu können, führen wir in diesem Abschnitt passende Abkürzungen für die verschiedenen Verfahren ein. Im folgenden werden wir die LRU-basierte egoistische Heuristik, die wir in Abschnitt 3.2.1 beschrieben haben, mit EGO(lru) bezeichnen. Entsprechend benutzen wir die Abkürzung EGO(temp) für die temperaturbasierte egoistische Heuristik aus Abschnitt 3.2.2. Analog benennen wir die altruistischen Verfahren aus Abschnitt 3.3 mit ALT(lru) und ALT(temp). Die von uns entwickelte kostenbasierte Heuristik aus Abschnitt 3.4 bezeichnen wir mit COST. Auch für die beiden Lastverteilungsverfahren aus Abschnitt 3.5 führen wir Abkürzungen ein. So bezeichnen wir im folgenden die zufällige Lastverteilungsmethode mit RANDOM, während wir die zielgerichtete Methode, welche die Objekte jeweils zu dem an geringsten belasteten Rechner migriert, mit MIN bezeichnen.

5.1.2 Aufwärmphase und statistische Signifikanz

Um aus unseren Experimenten aussagefähige Werte ableiten zu können, führen wir vor jedem Meßdurchgang eine Aufwärmphase durch. Diese Aufwärmphase umfaßt für jede Heuristik zunächst den Zeitraum, der erforderlich ist, um die Caches sämtlicher Rechner im System zu füllen. Darüber hinaus warten wir noch eine zusätzliche Zeitspanne, um sicherzustellen, daß sich das System eingependelt hat. Diese Zeitspanne ist abhängig von der benutzen Caching-Heuristik und wird folgendermaßen gewählt:

- **EGO(lru):** Hier haben wir kein Maß, das uns erlaubt, eine minimale Aufwärmphase zu berechnen. Um jedoch trotzdem die Messung nur an einem eingeschwungenen System durchzuführen, starten wir die Messung erst, nachdem jeder Rechner mindestens 10000 Zugriffe durchgeführt hat.
- **EGO(temp):** Um Aussagen über den stationären Fall treffen zu können, muß die Simulation mindestens solange aufgewärmt werden, bis für alle Objekte, die im Idealfall im Cache gehalten werden sollten, die Hitze mit ausreichender statistischer Signifikanz berechnet werden kann. Dies ist dann erfüllt, wenn auf das Objekt p im Mittel k Zugriffe erfolgt sind, wobei k der Parameter der LRU-k Methode und p das lokal kälteste Objekt ist, das bei einem exakt-egoistischen Caching gerade noch im Cache gehalten würde. Betrachten wir zur Verdeutlichung ein System, bei dem alle Rechner eine lokale Ankunftsrate λ_{loc}=100/Sekunde besitzen, die Zugriffsverteilungen jeweils einer Zipf-Verteilung der Schräge θ=1.0 entsprechen und jeder Rechner B=500 der insgesamt M=10000 vorhandenen Objekte im Cache halten kann. Unter diesen Voraussetzungen berechnet wir die minimale Aufwärmzeit durch:

  $$T_{warmup} = \frac{1}{\lambda_{loc} \times prob(X = B)} \approx 50000\ ms$$

- **ALT(lru):** Bei dieser Methode warten wir, bis die Caches ein altruistisches Verhalten zeigen. Dies können wir im Idealfall daran erkennen, daß in den Caches der einzelnen Rechner nur noch Unikate gespeichert sind. In unserem System kann es jedoch vorkommen, daß durch laufende Zugriffe dieser Idealzustand nie erreicht wird. Daher beenden wir die Aufwärmphase, wenn die Anzahl der Replikate auf jedem Rechner weniger als 1% der insgesamt im Cache gespeicherten Objekte beträgt.
- **ALT(temp):** Bei dieser Methode müssen wir zusätzlich zu dem altruistischen Verfahren auch noch die Berechnung der Hitze bei der Berechnung der Aufwärmphase mit berücksichtigen. Daher kombinieren wir die Methoden, die wir bei EGO(temp) und ALT(lru) vorgestellt haben, indem wir die Messung erst dann starten, wenn beide Bedingungen erfüllt sind. Hierbei müssen wir jedoch beachten, daß bei den altruistischen Verfahren mehr Objekte im aggregierten Cache gehalten werden als bei den egoistischen. Das global kälteste Objekt, das gerade noch im aggregierten Cache gehalten wird, muß daher bei der Abschätzung der Aufwärmphase benutzt werden.
- **COST:** Da unsere kostenbasierte Methode die Hitze zur Berechnung des Nutzens benötigt, müssen wir wiederum sicherstellen, daß die Simulation lange genug läuft, um eine korrekte Bestimmung der Hitze zu erlauben. Daher verwenden wir auch hier das bei EGO(temp) vorgestellte Verfahren zur Berechnung der Länge der Aufwärmphase.

Die Messung führen wir so lange durch, bis der Mittelwert der Antwortzeit eine vorgegebene Konfidenz [Jain91] erreicht hat. Bei unseren Messungen generieren wir jeweils so lange neue Zugriffe, bis die Breite des Konfidenzintervalls für das 98% Niveau kleiner als 2% des gemessenen Mittelwertes ist. Während den Simulationen überprüfen spezielle Funktionen von *CSIM18* [Schw96], ob die Meßwerte schon eine genügend große Zuverlässigkeit erreicht haben. Wird dies festgestellt, so werden die Ergebnisse ausgegeben und die Messung wird abgebrochen. Dies verhindert, daß die Abklingphase, in der die im System vorhandenen Warteschlangen abgearbeitet werden, die gemessenen Ergebnisse verfälscht.

5.1.3 Standardwerte der Parameter

In unseren Experimenten werden wir zum besseren Verständnis der beobachteten Ergebnisse jeweils nur einen Parameter variieren. Alle anderen Parameter halten wir konstant. Die Standardwerte für die Last-, Daten- und Systemparameter haben wir in den Tabellen 5.1, 5.2 und 5.3 zusammengestellt. Wenn wir in einem Experiment zur Verdeutlichung bestimmter Effekte von diesem Standard abweichende Werte wählen, so werden wir dies explizit herausstellen.

In der Tabelle 5.1 sind die Parameter aufgeführten, die wir zur Charakterisierung unserer synthetischen Last – entsprechend des Lastmodells aus Abschnitt 4.3.1 – benötigen. Tabelle 5.2 bestimmt die Parameter, die das Datenmodell aus Abschnitt 4.3.2 beschreiben. Schließlich sind in der Tabelle 5.3 die Parameter des Systems aufgeführt. Hierbei sollte berücksichtigt werden, daß die lokale Cache-Größe und das Verhältnis der Datengröße zur Cache-Größe voneinander abhängen. In unseren Experimenten werden wir daher nur das Verhältnis der Größen variieren und die lokale Cache-Größe in allen Experimenten konstant halten.

Parameter	**Standardwert**
Schräge der Zugriffsverteilung θ	1.0
Schreibwahrscheinlichkeit π	0.0
Zugriffsähnlichkeit (Deviation) δ	0.0
Zyklische Verschiebung σ	0
Relative Rechneraktivität η	0.0

Tabelle 5.1: Standardwerte für die Lastparameter

Parameter	**Standardwert**
Verhältnis der Datengröße zur Cache-Größe ν	2
Mittlere Objektgröße $\bar{s}$	4096 Byte
Objektgrößenverteilung	konstant
Objektallokation auf Platte	Round-Robin
Schräge der Allokation auf Platte ψ	0.0

Tabelle 5.2: Standardwerte für die Objektparameter

Parameter	**Standardwert**
Rechneranzahl N	10
Lokale Cache-Größe B	2 MByte
Netzgeschwindigkeit v_{net}	100 Mbit / s

Tabelle 5.3: Standardwerte für die Systemparameter

5.1.4 Bestimmung der Ankunftsrate

Bei der Beschreibung des Lastgenerators in Abschnitt 4.3.1 haben wir bereits herausgestellt, daß auf jedem Rechner ein eigener Prozeß Aufträge mit negativ exponentialverteilter Ankunftsrate generiert. Ein Problem beim Entwurf der Experimente besteht nun darin, für jede Wahl der Parameter eine solche Ankunftsrate zu bestimmen, so daß die Last bezüglich der vorhandenen Ressourcen sinnvoll dimensioniert ist, d.h. das System soll sich weder in einem total überlasteten, noch in einem sehr stark unterlasteten Zustand befinden. Hierzu benutzen wir die folgende einfache Rechnung, die unter Berücksichtigung der aktuellen Parameter die maximale Ankunftsrate für eine gegebene Auslastung des Netzwerks und der Festplatten berechnet.

Maximale Ankunftsrate für eine gegebene Plattenauslastung

Betrachten wir zuerst die Berechnung der Festplattenauslastung. Ist λ die Gesamtankunftsrate von Aufträgen in unserem verteilten System, so können wir die Gesamtankunftsrate auf alle Festplatten entsprechend der folgenden Gleichung berechnen:

$$\lambda_{disk} = (1 - \mathit{CacheHitRate}) \times \lambda \tag{5.1}$$

Hierbei bezeichnet *CacheHitRate* die Trefferrate auf den aggregierten Cache.

Für die weitere Herleitung nehmen wir an, daß jeder Rechner die gleiche Last besitzt. Darüber hinaus sollen die Objekte bezüglich ihrer Hitze gleichmäßig über die Festplatten verteilt sein. Diese Anforderung erfüllt die von uns standardmäßig benutzte Round-Robin-Verteilung annähernd. Unter diesen beiden Bedingungen können wir die Gleichung (5.1), die für das Gesamtsystem gilt, auf einen einzelnen Rechner übertragen:

$$\lambda_{loc,disk} = (1 - \mathit{CacheHitRate}) \times \lambda_{loc} \tag{5.2}$$

Berücksichtigen wir das Auslastungsgesetz [Jain91], so können wir die Ankunftsrate auf die lokale Festplatte in Abhängigkeit von der Plattenauslastung ρ_{disk} und der mittleren Plattenbedienzeit S_{disk} ausdrücken. Eingesetzt in Gleichung (5.2) erhalten wir:

$$\lambda_{loc} = \frac{\rho_{disk}}{(1 - \mathit{CacheHitRate}) \times S_{disk}} \tag{5.3}$$

In dieser Gleichung sind noch die Cache-Trefferrate und die mittlere Bedienzeit der Platte unbekannt; die maximale Auslastung hingegen wird von uns vorgegeben. Im folgenden werden wir zuerst die Cache-Trefferrate berechnen und anschließend die mittlere Bedienzeit herleiten.

Da wir die maximale Ankunftsrate bestimmen wollen, bei der keine der Heuristiken zu einer Überlastung der Festplatten führt, müssen wir die Gleichung (5.3) für die Heuristik mit der geringsten Cache-Trefferrate auswerten. Da die Cache-Trefferraten der von uns implementierten Heuristiken jedoch nicht so einfach im vorhinein bestimmt werden können, betrachten wir hier nur eine exakt-egoistische bzw. eine exakt-altruistische Heuristik, entsprechend den Beschreibungen in Abschnitt 2.3. Wie wir bereits dort diskutiert haben, besitzt die egoistische Heuristik immer die geringere Cache-Trefferrate, und daher ist es ausreichend die Trefferrate dieser Heuristik zu betrachten. Nehmen wir an, daß das Zugriffsmuster auf allen Rechnern gleich ist, so hält jeder Rechner die gleichen Objekte in seinem lokalen Cache. Daher kommen keine nichtlokalen Cache-Treffer vor. Wählen wir die Zugriffe entsprechend einer Zipf-Verteilung, so können wir

die lokale Cache-Trefferrate und damit auch die gesamte Cache-Trefferrate entsprechend der folgenden Formel berechnen:

$$CacheHitRate = \sum_{j=1}^{B} \frac{1}{C \times j^{\theta}}$$

In dieser Gleichung beschreibt B die Anzahl der Objekte, die jeweils in einen lokalen Cache passen, C ist die Normierungskonstante der Zipf-Verteilung und θ die Schräge der Verteilung.

Nachdem wir die Cache-Trefferrate bestimmt haben, wollen wir nun abschließend noch die mittlere Bedienzeit der Platte S_{disk} herleiten. Dazu werden wir die einzelnen Komponenten (Suchzeit, Rotationsverzögerung und Transferzeit) unserer Plattenmodellierung aus Abschnitt 4.2.2 nacheinander betrachten.

Zur Berechnung der mittleren Suchzeit müssen wir zunächst bestimmen, wie viele Zylinder bei einem Suchvorgang überquert werden müssen. In [Zabb94] wurde gezeigt, daß die Wahrscheinlichkeit, daß bei einer Platte mit Z Zylindern bei einem Suchvorgang z Zylinder überquert werden, durch folgende Formel berechnet werden kann:

$$Prob(i = z) = 2\frac{Z - i}{Z^2}$$

Mit Hilfe dieser Formel können wir nun den Mittelwert $\bar{z}$ berechnen:

$$\bar{z} = \sum_{i=0}^{Z-1} i \times 2\frac{Z - i}{Z^2} = \frac{Z^2 - 1}{3Z}$$

Kennen wir die mittlere Anzahl von Zylindern, die beim Suchen überquert werden müssen, so können wir die mittlere Suchzeit innerhalb unseres Modells berechnen, indem wir diesen Wert in die Gleichung (4.1) für die lineare Approximation der Suchzeit einsetzen.

Die zweite Komponente ist die Rotationsverzögerung. Wie wir in Abschnitt 4.2.2 beschrieben haben, wählen wir diese gleichverteilt aus dem Intervall von 0 bis zur Zeit für eine komplette Rotation. Da wir unabhängige Zugriffe annehmen, ist die mittlere Rotationsverzögerung die Zeit für eine halbe Rotation.

Zur Bestimmung der mittleren Transferzeit müssen wir den Quotienten aus der mittleren Objektgröße $\bar{s}$ und der Transfergeschwindigkeit v_{disk} der Platte berechnen. Kennen wir die mittleren Bedienzeiten für die einzelnen Komponenten, so ergibt sich die mittlere Gesamtbedienzeit der Platte aus der Summe der Einzelkomponenten.

Damit haben wir alle unbekannten Größen der Gleichung (5.3) bestimmt und sind nun in der Lage, die lokale Ankunftsrate zu berechnen, bei der die Festplattenauslastung auf keinem der Rechner eine vorgegebene mittlere Auslastung überschreitet.

Maximale Ankunftsrate für eine gegebene Netzauslastung

Nachdem wir die maximale Ankunftsrate für das Erreichen einer vorgegebenen Festplattenauslastung berechnet haben, wollen wir nun eine analoge Formel für das Netzwerk herleiten. Im Gegensatz zur obigen Berechnung benutzen wir nun die exakt-altruistische Heuristik zur Herleitung einer maximalen Ankunftsrate, da diese Heuristik die größere Netzlast verursacht. Auch hier

nehmen wir wiederum an, daß alle Rechner die gleichen Lastcharakteristika haben und daß die Objekte gleichmäßig über die Festplatten der Rechner verteilt sind. Unter diesen Annahmen können wir davon ausgehen, daß die Objekte in den lokalen Caches bezüglich ihrer Hitze gleichmäßig über alle Rechner verteilt sind. Daher ist die Wahrscheinlichkeit, daß ein Zugriff lokal, d.h. entweder aus dem lokalen Cache oder von der lokalen Platte beantwortet wird, nur von der Rechneranzahl N abhängig:

$$LocalHitRate = 1/N \tag{5.4}$$

Analog zu Gleichung (5.1) können wir nun auch hier die Ankunftsrate von Aufträgen an das Netzwerk in Abhängigkeit von der Gesamtankunftsrate λ ausdrücken:

$$\lambda_{net} = (1 - LocalHitRate) \times \lambda = (1 - LocalHitRate) \times N \times \lambda_{loc} \tag{5.5}$$

Unter Ausnutzung des Auslastungsgesetzes und nach dem Einsetzen der Gleichungen (5.4) und (5.5) erhalten wir somit:

$$\lambda_{loc} = \frac{\rho_{net}}{(N - 1) \times S_{net}} \tag{5.6}$$

Auch hier ist wiederum die mittlere Bedienzeit des Netzwerks unbekannt. Ist die mittlere Objektgröße $\bar{s}$ und die Transfergeschwindigkeit des Netzwerks v_{net} bekannt, so können wir diese jedoch als den Quotienten dieser Werte berechnen

Mit Hilfe der Formeln (5.3) und (5.6) können wir die lokalen Ankunftsraten für eine vorgegebene maximale Auslastung berechnen. Um keine der betrachteten Ressourcen zu stark zu belasten, wählen wir den kleineren der beiden berechneten Werte als die maximale Ankunftsrate jedes einzelnen Rechners im System. Exemplarisch können wir diese Vorgehensweise an Hand des folgenden Beispiels verdeutlichen.

Beispiel 5.1:

Wir variieren die Schräge der Zugriffsverteilung und wählen für alle anderen Parameter die in Abschnitt 5.1.3 beschriebenen Standardwerte. Für jede betrachtete Schräge bestimmen wir die maximale Ankunftsrate, bei der weder die Festplatten noch das Netzwerk eine Auslastung von mehr als 90% besitzen.[8] Dazu müssen wir zuerst die mittleren Bedienzeiten S_{disk} und S_{net} bestimmen. Entsprechend der oben angegebenen Formeln erhalten wir:

$$S_{disk} = S_{seek} + S_{rot\_latency} + S_{transfer} = 6.1 + 4.2 + 0.5 = 10.8\ ms \text{ und}$$

$$S_{net} = \frac{\bar{s}}{v_{net}} = 0.3\ ms.$$

Durch Einsetzen dieser mittleren Bedienzeiten in die Formeln (5.3) und (5.6) können wir nun die maximalen Ankunftsraten für die verschiedenen Schrägen berechnen. Die ermittelten Werte sind in der Tabelle 5.4 zusammengefaßt.

8. Eine Auslastung von 90% erscheint auf den ersten Blick übertrieben hoch. Da wir jedoch bei der Herleitung der Formeln von einer exakt-egoistischen bzw. einer exakt-altruistischen Cache-Heuristik ausgegangen sind, werden bei unseren Online-Implementationen der Heuristiken diese hohen Auslastungen nicht erreichen.

Schräge	λ_{disk} [1/ms]	λ_{net} [1/ms]	λ_{max} [1/ms]
0.00	0.09	0.3	0.09
0.25	0.10		0.10
0.50	0.11		0.11
0.75	0.15		0.15
1.00	0.29		0.29
1.25	0.83		0.30
1.50	3.33		0.30

Tabelle 5.4: Bestimmung der Ankunftsrate für eine gegebene maximale Ressourcenauslastung

In dieser Tabelle können wir gut erkennen, daß sich der Engpaß – je nach gewählter Schräge – verändert. Während bei geringer Schräge die Auslastung der Platte die maximale Ankunftsrate beschränkt, ist bei großer Schräge die Auslastung des Netzwerks der begrenzende Wert. Bemerkenswert ist auch, daß die Auslastung des Netzwerks bei der altruistischen Heuristik und damit der Wert von λ_{net} unabhängig von der Schräge der Zugriffsverteilung ist.

■

Wählen wir die Ankunftsraten innerhalb eines Experiments entsprechend dieser Vorgehensweise, so kann es vorkommen, daß sich die beobachteten Antwortzeiten für die verschiedenen Werte des variierten Parameters sehr stark unterscheiden. Um trotzdem eine deutliche Darstellung für jeden Parameterwert innerhalb eines einzigen Diagramms zu erhalten, werden wir in den entsprechenden Experimenten eine normierte Darstellung der Antwortzeit benutzen. Die Normierung erfolgt dabei bezüglich der Antwortzeit von EGO(lru). Dies bedeutet, daß EGO(lru) immer den Wert 1 erhält. Werte größer 1 entsprechen einer schlechteren und Werte kleiner 1 einer besseren Antwortzeit.

5.2 Variation der Lastparameter

5.2.1 Variation der Schräge im Zugriffsverhalten

In diesem ersten Experiment variieren wir die Schräge des Zugriffsverhaltens von einer Gleichverteilung (θ=0) bis hin zu einer sehr schrägen Verteilung mit θ=1.5. Die lokalen Ankunftsraten haben wir entsprechend der Vorgehensweise aus Abschnitt 5.1.4 berechnet. Da sich – abhängig von der Schräge – die mittleren Zugriffszeiten in den verschiedenen Messungen sehr stark unterscheiden können, sind in Abbildung 5.1 die relativen, auf die EGO(lru)-Heuristik normierten, Antwortzeiten skizziert.

In dieser Abbildung können wir erkennen, daß – ähnlich wie bei dem entsprechenden Experiment mit den statischen Kosten in Abschnitt 2.4 – die Reihenfolge der egoistischen und altruistischen Verfahren von der Schräge der Verteilung abhängt. Auch hier sind die altruistischen Verfahren besser, wenn die Schräge gering ist. Erst bei wachsender Schräge verbessern sich die Antwortzeiten der egoistischen Verfahren gegenüber den altruistischen Methoden. Das Verhalten der kosten-

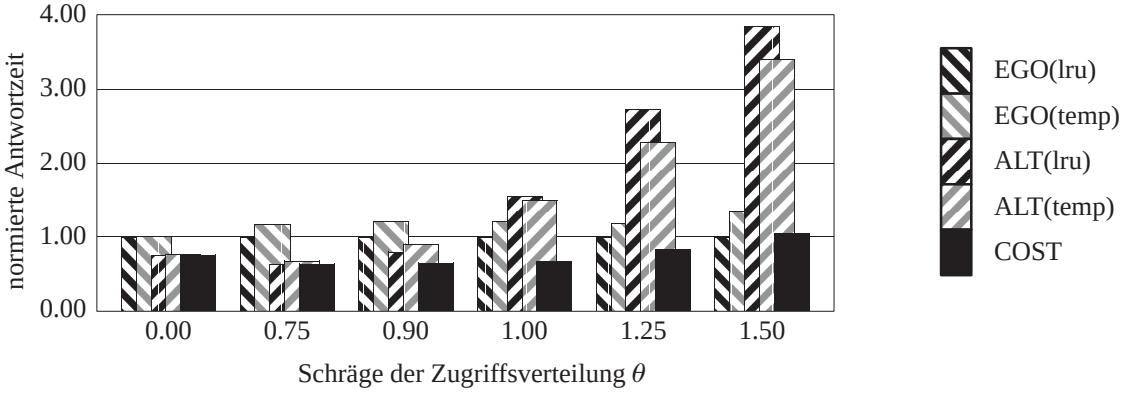


Abbildung 5.1: Einfluß der Schräge auf die Antwortzeit

basierten Heuristik verhält sich ähnlich wie die optimale Cache-Strategie aus Abschnitt 2.2. Für die beiden Extremwerte der Schräge "konvergiert" COST entweder zu den egoistischen oder altruistischen Methoden, je nachdem welches Paradigma für diese Situation besser geeignet ist. Zwischen diesen Extremen erkennen wir jedoch, daß die kostenbasierte Heuristik eine entscheidende Verbesserung gegenüber allen einfachen Caching-Verfahren erzielt.

Zur Erkärung dieses Verhaltens haben wir in Abbildung 5.2 die Trefferrate der verschiedenen Heuristiken für unterschiedliche Schrägen skizziert. Zusätzlich zu diesen beobachteten Trefferraten haben wir auch die für die jeweilige Schräge θ maximal erreichbaren lokalen bzw. globalen Cache-Trefferraten nach den folgenden Formeln bestimmt:

$$\text{Maximale lokale Cache-Trefferrate} = \sum_{j=1}^{B} \frac{1}{C \times j^{\theta}} \quad \text{und}$$

$$\text{Maximale globale Cache-Trefferrate} = \sum_{j=1}^{N \times B} \frac{1}{C \times j^{\theta}}$$

In diesen Formeln ist B wiederum die Anzahl der Objekte, die im lokalen Cache gehalten werden können, C ist die Normierungskonstante der Zipf-Verteilung und N die Gesamtanzahl der Rechner im System.

Betrachten wir zunächst die Abbildung (a), in der die Trefferraten bei einer Gleichverteilung dargestellt sind. In dieser Abbildung erkennen wir, daß zwar alle Verfahren die maximal mögliche

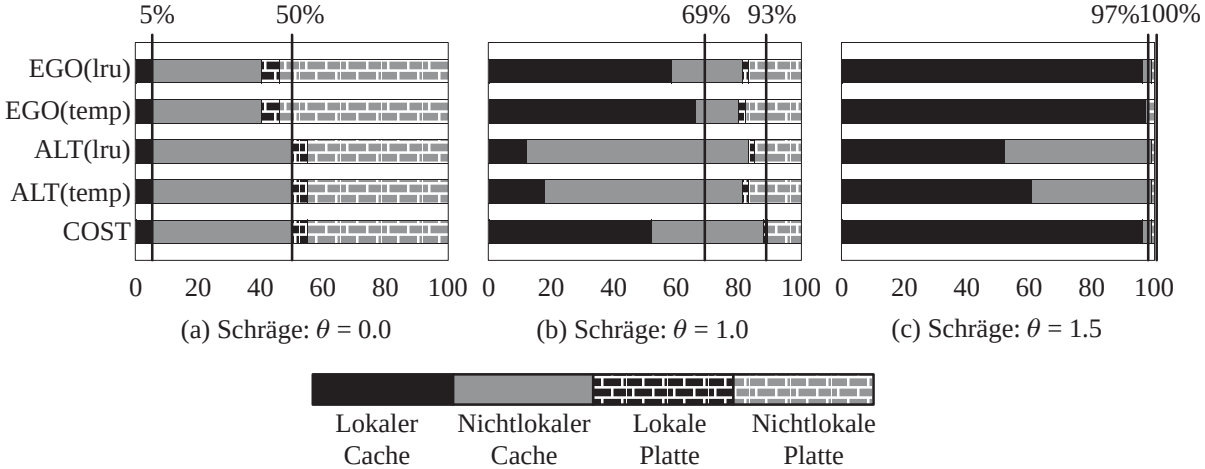


Abbildung 5.2: Trefferraten auf den verschiedenen Hierarchiestufen

lokale Cache-Trefferrate von 5% erreichen, daß jedoch nur die altruistischen und die kostenbasierte Methode die maximale globale Trefferrate von 50% erreichen. Die schlechte globale Trefferrate der egoistischen Heuristiken kommt daher, daß die Caches sehr viele (im Mittel 37%) Replikate enthalten. Daher ist der Gesamtanteil der im Cache gehaltenen Objekte mit ca. 40% sehr viel kleiner, als der entsprechende Anteil bei den altruistischen bzw. der kostenbasierten Heuristik mit jeweils ca. 50%. Diese geringere Cache-Trefferrate der egoistischen Methoden resultiert in einer höheren Plattenauslastung. Da die Platten die Engpaßressourcen sind, führt dies zu der schlechteren mittleren Antwortzeit, die wir in Abbildung 5.1 beobachten können.

Betrachten wir nun die Trefferraten, die zu der Situation mit einer sehr hohen Schräge korrespondieren (Abbildung 5.2(c)). Das Vermeiden von Replikationen bei sehr hoher Schräge führt dazu, daß bei den altruistischen Verfahren sehr viele Zugriffe auf lokal heiße Objekte nur durch den nichtlokalen Cache befriedigt werden können. Dies wiederum führt dazu, daß die Netzauslastung bei diesen Methoden sehr stark anwächst (größer als 40%) und zur Engpaßressource wird, während die egoistischen Heuristiken und unsere kostenbasierte Heuristik durch die sehr große lokale Cache-Trefferrate Netzauslastungen kleiner als 5% erzielen. Der Vorteil der etwas höheren globalen Trefferrate der altruistischen Heuristiken kann diesen großen Nachteil nicht ausgleichen. Dies ist die Ursache für die sehr schlechte Antwortzeiten der altruistischen Heuristiken bei großen Schrägen in der Zugriffsverteilung.

Nachdem wir die beiden Extremsituationen besprochen, und gesehen haben, wie sich unsere kostenbasierte Heuristik jeweils einer der beiden Heuristiken angepaßt hat, und dadurch die Auslastung der Engpaßressource minimierte, wollen wir nun untersuchen, was zwischen diesen beiden Extremen passiert. Dazu haben wir in Abbildung 5.2(b) die Trefferraten für die Schräge θ=1.0 skizziert. Die kostenbasierte Heuristik kann jeweils auf Objektebene bestimmen, ob das betreffende Objekt repliziert oder nur einfach im Cache gehalten werden soll. Daher erzielt sie eine größere lokale Cache-Trefferrate als die altruistischen Heuristiken (sehr heiße Objekte werden repliziert in den Caches gehalten), und gleichzeitig eine größere globale Cache-Trefferrate als die egoistischen Heuristiken (kalte Objekte werden nur einmal im System gehalten). Diese Flexibilität in der separaten Bestimmung des Replikationsgrads für jedes Objekt führt dazu, daß unsere kostenbasierte Heuristik eine sehr viel bessere Performance liefert als die einfachen Heuristiken.

Betrachten wir weiterhin den Fall θ=1.0, so erscheint es jedoch merkwürdig, daß die kostenbasierte Heuristik auch eine größere globale Trefferrate als die altruistischen Heuristiken erzielt, obwohl durch die bevorzugte Verwendung von Replikaten bei der Cache-Ersetzung die altruistischen Methoden einen größeren Anteil der Objekte im aggregierten Cache halten (49.8% bei den beiden altruistischen Methoden verglichen mit 43.8% bei der kostenbasierten Heuristik). Der Grund hierfür liegt in der geringeren aggregierten Temperatur der Objekte im Cache, die durch die Vorgehensweise bei der Opferbestimmung der altruistischen Verfahren bedingt wird. Betrachten wir dazu das in Abbildung 5.3 skizzierte Beispiel. Zuerst speichert Rechner 1 das Objekt *p* als Unikat in seinem Cache. Durch den nichtlokalen Zugriff ändert sich der Status des Objekts auf Replikat und das Objekt wird an den Rechner 2 gesendet. Die Wahrscheinlichkeit, daß das Objekt auf Rechner 1 als Ersetzungsopfer ausgewählt wird, ist nun sehr groß, da die Anzahl der Objekte in der Replikat-Struktur immer sehr gering ist. Nach dem Löschen des Objekts auf Rechner 1 wird Rechner 2 darüber informiert, daß er nun die einzige Kopie des Objekts besitzt. Ist die Bearbeitung des Objekts auf Rechner 2 jedoch schon beendet bevor diese Unikat-Nachricht ein-

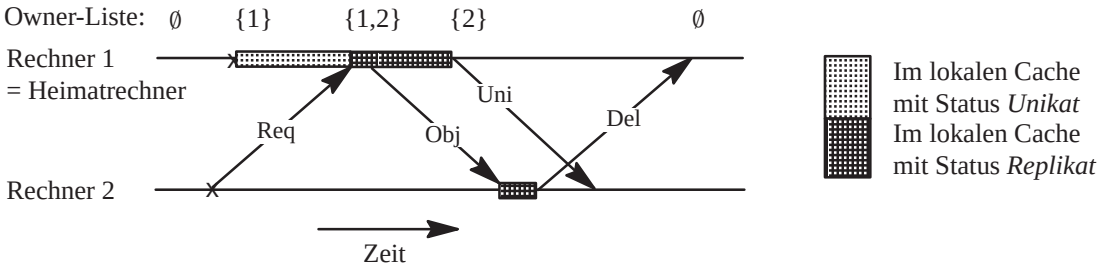


Abbildung 5.3: Ungewolltes Löschen eines Objekts aus dem aggregierten Hauptspeicher durch zeitweise inkonsistente Buchhaltungsinformation

trifft, so wird hier – aus den gleichen Gründen wie auf Rechner 1 – das Objekt sehr wahrscheinlich als Ersetzungsopfer ausgewählt. Dies führt jedoch dazu, daß keine Kopie dieses Objekts mehr im aggregierten Cache existiert. Da dieses Verhalten unabhängig von der Hitze der Objekte auftreten kann, werden auch heiße Objekte aus dem verteilten Cache verdrängt. Somit sinkt die aggregierte Temperatur der im Cache vorhandenen Objekte. Dieses Verhalten wird verstärkt durch eine hohe Netzauslastung und durch eine hohe Ankunftsrate. Beide Effekte erhöhen die Anzahl der Opferbestimmungen, die während einer Phase inkonsistenter Buchhaltungsinformation durchgeführt werden.

Um diesen Effekt genauer zu untersuchen, haben wir bei sonst gleichen Parametern die Ankunftsrate der Zugriffe variiert. Die gemessenen mittleren Antwortzeiten sind in der Abbildung 5.4 skizziert. Während bei sehr kleiner Ankunftsrate die altruistischen Methoden noch besser als die egoistischen Methoden abschneiden, werden durch Erhöhung der Ankunftsrate die altruistischen Methoden sehr schnell schlechter. Obwohl das Problem der zeitweise inkonsistenten Buchführungsinformation auch die Berechnung des Nutzens bei unserer kostenbasierten Heuristik beeinflußt, erfolgt dort – durch die zusätzliche Berücksichtigung der Objekthitzen – keine unbedingte Bevorzugung von Replikaten. Dadurch ist die Wahrscheinlichkeit, daß ein sehr heißes Objekt auf Grund der oben beschriebenen Situation ungewollt aus dem Cache verdrängt wird, sehr viel geringer. Daher reagiert unsere kostenbasierte Heuristik nicht so empfindlich auf eine Erhöhung der Ankunftsrate.

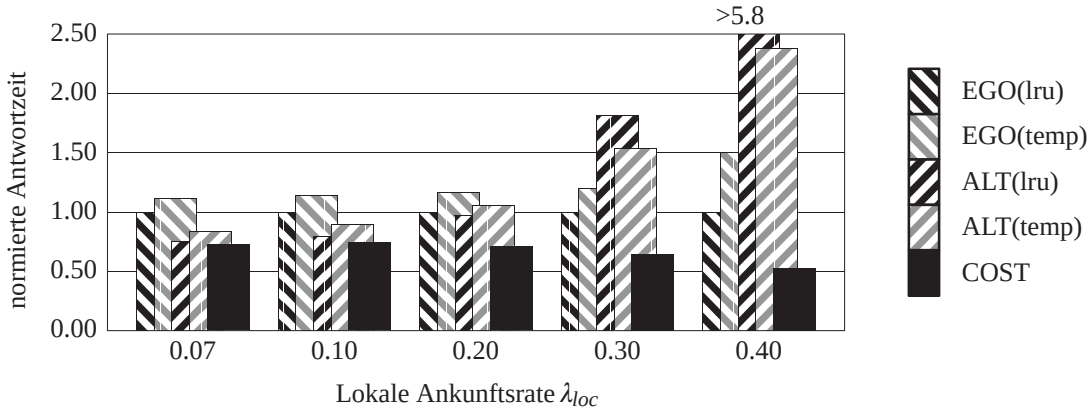


Abbildung 5.4: Einfluß der Ankunftsrate auf die Antwortzeit

Bislang sind wir noch nicht auf einen Vergleich der verschiedenen Implementierungen der egoistischen bzw. altruistischen Heuristiken untereinander eingegangen. Betrachten wir zunächst die beiden egoistischen Verfahren. In Abbildung 5.1 können wir erkennen, daß EGO(temp) über den

gesamten Bereich eine schlechtere Performance als EGO(lru) liefert, obwohl EGO(temp) eine höhere lokale Trefferrate erzielt und damit das exakt-egoistische Verhalten besser approximiert als EGO(lru). In dieser Situation führt dies jedoch dazu, daß der Replikationsgrad der Objekte weiter erhöht wird und dadurch die globale Trefferrate sinkt (Abbildung 5.2(b)). Dieses Verhalten wir um so ausgeprägter, je besser wir die Hitze approximieren. Je nachdem, ob eine exakt-egoistische Heuristik vorteilhaft für die aktuellen Parameter ist oder nicht, kann also eine bessere Approximation des egoistischen Verhaltens sogar zu einer Verschlechterung der Antwortzeit führen. Zusätzlich zu diesem Phänomen, das von den aktuellen Parametern abhängt, trägt jedoch auch der Berechnungs-Overhead für die teuereren Ersetzungsoperationen im Cache (siehe Abschnitt 4.1.5) zu einer Verschlechterung der Ausführungszeit jedes einzelnen Zugriffs bei.

Bei den altruistischen Methoden kann eine solche über den ganzen Bereich gültige Aussage nicht gemacht werden. Zwar verursacht der zusätzliche Berechnungsaufwand bei ALT(temp) im Vergleich zu ALT(lru) ebenfalls eine Verlangsamung eines einzelnen Zugriffs, jedoch führt bei hoher Schräge die Berücksichtigung der Hitze dazu, daß die lokale Trefferrate steigt (Abbildung 5.2(b) und (c)). Dadurch wird die Netzlast reduziert, was letztendlich zu einer geringeren mittleren Antwortzeit führt.

Ein möglicher Einwand gegen unsere bisherigen Ergebnisse könnte darin bestehen, daß die Ursache für das schlechte Abschneiden der einfachen Heuristiken – und hier insbesondere der altruistischen Verfahren – in der speziellen Implementierung der Zusatzprotokolle (siehe Abschnitt 3.1) liegt. Um dieses Argument zu entkräften, haben wir das Experiment mit globalem Wissen bezüglich lokaler und globaler Hitze und Objektstatus wiederholt. Die Antwortzeiten, die wir dieses Mal bezüglich der Antwortzeit von EGO(exakt) normiert haben, und die erzielten Trefferraten dieser Messung sind in den Abbildungen 5.5 bzw. 5.6 gezeigt. Neben den Verfahren mit dem exakten Wissen haben wir auch die Ergebnisse unserer kostenbasierten Heuristik eingezeichnet, bei der wir die benötigten Informationen mittels der in Abschnitt 3.1 vorgestellten Protokolle bestimmen.

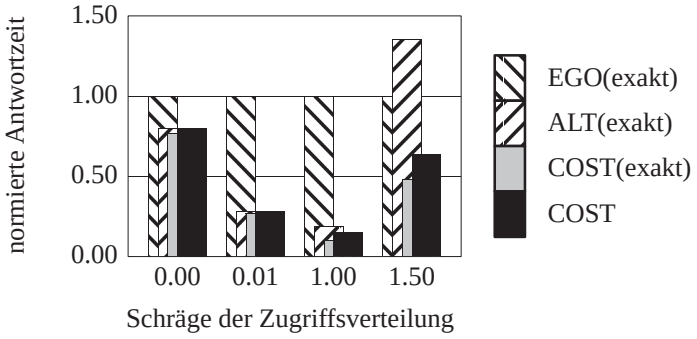


Abbildung 5.5: Vergleich der Verfahren bei exakter Information

In Abbildung 5.5 sehen wir, daß sich – verglichen mit dem ursprünglichen Experiment – zwar die Abstände zwischen den Heuristiken verändert haben, daß jedoch die weiter oben getroffenen qualitativen Aussagen, bestehen bleiben. Insbesondere verhält sich die altruistische Heuristik bei geringer Schräge besser, während die egoistische Heuristik Vorteile bei großer Schräge bietet. Eine Besonderheit ist der große Unterschied in dem relativen Verhalten zwischen der Schräge 0.0 und 0.01. Die Ursache hierfür liegt in der sehr unterschiedlichen Cache-Trefferrate des egois-

tischen Verfahrens in diesen beiden Fällen. Da bei der Schräge 0.01 bereits eine Rangfolge bezüglich der Hitze zwischen den Objekten besteht und jeder Rechner das gleiche Zugriffsverhalten besitzt, werden jeweils die heißesten Objekte repliziert in den Caches gehalten. Existieren M Objekte im System, und kann jeder Rechner B Objekte im Cache halten, dann berechnet sich die globale Cache-Trefferrate, die in diesem Fall gleich der lokalen Cache-Trefferrate ist, durch:

$$GlobCacheHitrate = \sum_{j=1}^{B} \frac{1}{C \times j^{\theta}} \approx \frac{B}{M} = 5\%$$

Diese Approximation gilt für sehr kleine θ, da in diesem Fall j^{θ} gegen 1 und die Normierungskonstante C gegen M konvergiert.

Anders verhält sich die Situation bei einer Gleichverteilung. Hier existiert keine Rangfolge bezüglich der Hitzen zwischen den Objekten. Daher können wir nicht mehr davon ausgehen, daß jeder Rechner die gleichen Objekte in seinem Cache hält. Zur Berechnung der globalen Cache-Trefferrate müssen wir zunächst bestimmen wieviele verschiedene Objekte sich im Mittel in den Caches der Rechner befinden. Betrachten wir dazu die Caches der unterschiedlichen Rechner nacheinander. Auf dem ersten Rechner befinden sich B verschiedene Objekte. Auf dem zweiten Rechner können jedoch Objekte vorkommen, die sich bereits im Cache des ersten Rechners befinden. Die Wahrscheinlichkeit dafür, daß sich ein Objekt bereits auf dem ersten Rechner befindet, ist $(1\text{-}B/M)$. Entsprechend befinden sich im Mittel $B(1\text{-}B/M)$ neue Objekte auf dem zweiten Rechner. Verallgemeinern wir diese Vorgehensweise, so erhalten wir als erwartete Anzahl unterschiedlicher Objekte in einem System mit N Rechnern:

$$NoDifferentObjects = \sum_{i=0}^{N-1} B(1 - B/M)^{i} = B\left(1 - (1 - B/M)^{N}\right)$$

Die globale Trefferrate ist dann entsprechend:

$$GlobCacheHitrate = \frac{NoDifferentObjects}{NoObjects} \approx 40.1\%$$

Diese berechneten Resultate bestätigen die Abbildungen 5.6(a) und (b). Außerdem erkennen wir in Abbildung 5.6, daß durch die Verwendung der exakten Hitze- und Statusinformation generell die gemessenen Trefferraten (lokale bei der egoistischen und globale bei der altruistischen Methode) mit den theoretisch berechneten Werten sehr gut übereinstimmen. Besonders hervorzuheben ist, daß sich unsere kostenbasierte Heuristik relativ unkritisch bezüglich der Ungenauigkeit der Approximation der Lastinformation verhält. Weder die Antwortzeiten in Abbildung 5.5 noch

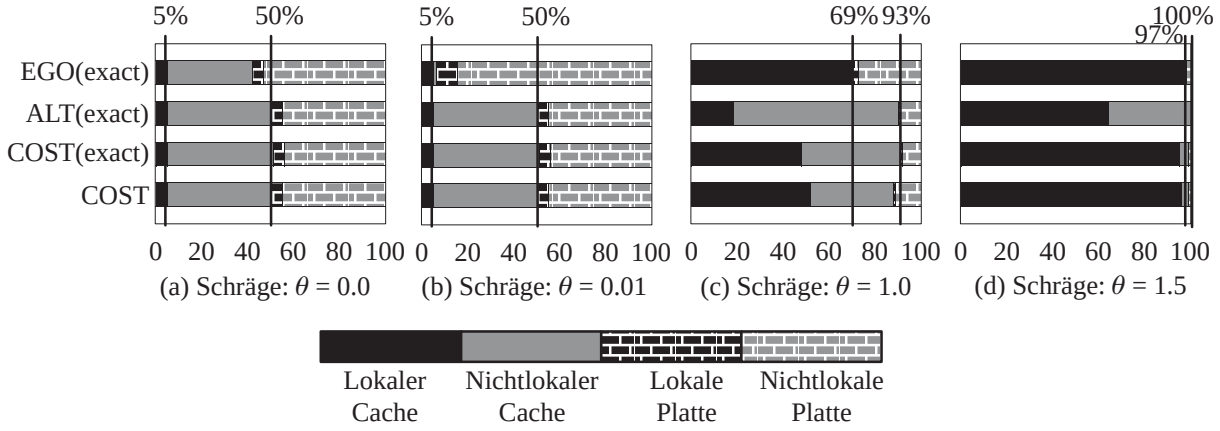


Abbildung 5.6: Trefferraten bei exakter Information

die Trefferraten in Abbildung 5.6 zeigen starke Unterschiede zwischen unserer kostenbasierten Heuristik mit exaktem und mit approximiertem Wissen.

Overhead der verschiedenen Heuristiken

Betrachten wir abschließend für dieses Experiment noch den durch die verschiedenen Heuristiken verursachten Overhead. Für die Schräge θ=1 zeigen wir dazu in Tabelle 5.5 neben den mittleren Gesamtauslastungen der verschiedenen Ressourcen im System auch eine nach den Verursachern differenzierte Liste der Teilauslastungen.

Bei den CPUs führen wir zunächst die Auslastung für die Durchführung der *Anwendung* auf. Als minimale Anwendung nehmen wir an, daß das zugegriffene Objekt zumindest einmal komplett gelesen wird. Als nächstes zeigen wir in der Tabelle den CPU-Aufwand für die Abarbeitung des *Netzwerkprotokolls* und für das Management von *Cache* und *Platte*. Der CPU-Bedarf für die Bearbeitung von Anfragen an den *lokalen* und *globalen Katalog* ist ebenfalls separat in der Tabelle aufgeführt. Bei den auf der Temperatur und den Kosten basierenden Heuristiken verursacht die Berechnung der *Hitze* bzw. des *Nutzens* ebenfalls eine Zusatzlast auf der CPU. *S*chließlich können bei einigen Verfahren Änderungen der Objekt- (z.B. Status, Hitze) oder Systemeigenschaften (z.B. Auslastungen) *Umsortierungen im Cache* notwendig machen, ohne daß das betreffende Objekt lokal selbst zugegriffen wurde.

Neben der CPU-Auslastung können wir auch die Netzwerkauslastung in unterschiedliche Klassen aufteilen. Neben den *Transfers der Objektdaten* tragen auch die Protokolle zur verteilten globalen *Hitzeberechnung* und zur Propagierung des *Objektstatus* zu einer Erhöhung der Netzlast bei.

	EGO(lru)	**EGO(temp)**	**ALT(lru)**	**ALT(temp)**	**COST**
Anwendung	4.7 %	4.7 %	4.7 %	4.7 %	4.7 %
Netzwerkprotokoll	1.6 %	1.3 %	3.8 %	3.4 %	2.0 %
Cache-Management	0.5 %	1.9 %	0.7 %	2.6 %	2.2 %
Plattenmanagement	1.3 %	1.5 %	1.2 %	1.3 %	0.9 %
Anfragen an den lokaler Katalog	0.8 %	0.6 %	1.1 %	0.8 %	0.7 %
Anfragen an den globaler Katalog	0.2 %	0.2 %	0.3 %	0.3 %	0.2 %
Hitze- und Nutzenberechnung	–	0.3 %	0.3 %	0.3 %	0.3 %
Cache-Umsortierungen	–	0.5 %	0.2 %	1.7 %	3.0 %
Gesamtauslastung der CPU	**9.0 %**	**10.9 %**	**12.1 %**	**17.6 %**	**14.2 %**
Datentransfers	37.7	30.7 %	83.0 %	76.6 %	45.7 %
Verteilte Hitzeberechnung	–	–	–	1.3 %	1.7 %
Verteilte Statusberechnung	–	–	0.4 %	0.3 %	0.2 %
Gesamtauslastung des Netzwerks	**37.7 %**	**30.7 %**	**83.4 %**	**78.2 %**	**47.6 %**
Gesamtauslastung der Festplatten	**53.9 %**	**59.9 %**	**49.4 %**	**54.5 %**	**37.2 %**

Tabelle 5.5: Auslastungen der verschiedenen Ressourcen

Vergleichen wir zunächst einmal die Gesamtauslastungen der verschiedenen Ressourcen. Hier können wir in Tabelle 5.5 erkennen, daß die Gesamtauslastung der CPU verglichen mit den ande-

ren Ressourcen relativ gering ist, d.h. die CPU ist nicht die Engpaßressource. Desweiteren sehen wir, daß der Aufwand für die Anwendung und für die Anfragen an die verschiedenen Kataloge für alle Verfahren etwa gleich ist. Im Gegensatz dazu schwanken die CPU-Auslastungen für das Management der Platten und des Netzwerks sehr stark in Abhängigkeit von der benutzten Heuristik. Dies können wir dadurch erklären, daß Anzahl der Zugriffe (erkennbar in den unterschiedlichen Auslastungen des Netzwerks und der Festplatten in Tabelle 5.5) auf die entsprechende Ressource auch von der Heuristik abhängt. Die höheren Auslastungen durch das Cache-Management für die auf der Temperatur und auf den Kosten basierenden Heuristiken resultiert aus den aufwendigeren Operationen auf der Cache-Ersetzungsstruktur.

Neben diesen Aktionen, die zur Abarbeitung der Zugriffe notwendig sind und die daher unabhängig von der benutzten Heuristik immer durchgeführt werden müssen, benötigen manche Heuristiken zusätzliche Aktionen, die ebenfalls zu einer Erhöhung der CPU-Auslastung beitragen. In der Tabelle erkennen wir jedoch, daß der Aufwand für die Neuberechnung der Hitzen und des Nutzens verschwindend gering ist. Die unterschiedlich starke Auslastung durch das Umsortieren bei den verschiedenen Heuristiken können wir durch die unterschiedlichen Ursachen für eine solche Aktion erklären. So ist bei ALT(lru) nur dann ein Umsortieren notwendig, wenn sich der Status eines Objekts ändert. Im Gegensatz dazu muß bei EGO(temp), ALT(temp) und COST auch dann ein Umsortieren durchgeführt werden, wenn sich die Hitze eines Objekts ändert. Während jedoch bei EGO(temp) eine Hitzeänderung immer nur einen Einfluß auf den lokalen Cache hat, kann bei ALT(temp) und COST ein Schwanken der Hitze zu Rangfolgeänderungen auf mehreren Rechnern führen. Bei COST kommt zusätzlich noch die Berücksichtigung der Ressourcenauslastungen hinzu, so daß hier die Anzahl der Umsortierungen am größen ist. Aber selbst in diesem Fall ist der verursachte Overhead verglichen mit der Gesamtauslastung gering.

Auch bei der Auslastung des Netzwerks erkennen wir, daß der Overhead durch die Protokolle zur verteilten Berechnung der Hitze und des Status eines Objekts nicht ins Gewicht fällt. Wichtig für das Erzielen des geringen Aufwandes bei der Hitzeberechnung ist das lokale Abfangen von geringfügigen Schwankungen in der Hitze durch das schwellwertbasierte Protokoll aus Abschnitt 3.1.2.

5.2.2 Variation der Ähnlichkeit der Zugriffsmuster

Nachdem wir den Einfluß der Schräge im Zugriffsverhalten dargestellt haben, wollen wir in diesem Experiment untersuchen, wie die verschiedenen Heuristiken auf eine Variation in der Ähnlichkeit des Zugriffsmusters reagieren. Dazu permutieren wir jeweils die lokale Rangfolge der Objekte auf den verschiedenen Rechnern zufällig und bestimmen für die generierte Verteilung den Wert der Deviation entsprechend der Definition 4.2. Die mittlere Ankunftsrate – und damit die Gesamtlast auf den einzelnen Rechnern – bleibt dadurch konstant. Da wir bei diesem Verfahren die Trefferrate im vorhinein nicht berechnen können, ist es nicht möglich, die Methode zur Bestimmung der Ankunftsrate aus Abschnitt 5.1.4 zu benutzen. Daher wählen wir unabhängig von der gewählten Schräge eine konstante Ankunftsrate. Die Antwortzeiten der verschiedenen Heuristiken in Abhängigkeit von der Deviation sind in der Abbildung 5.7 gezeigt.

Auffällig bei diesem Experiment ist der nicht-monotone Verlauf einiger Antwortzeitkurven. Besonders bei den altruistischen Verfahren erkennen wir, daß eine Erhöhung der Deviation zuerst zu

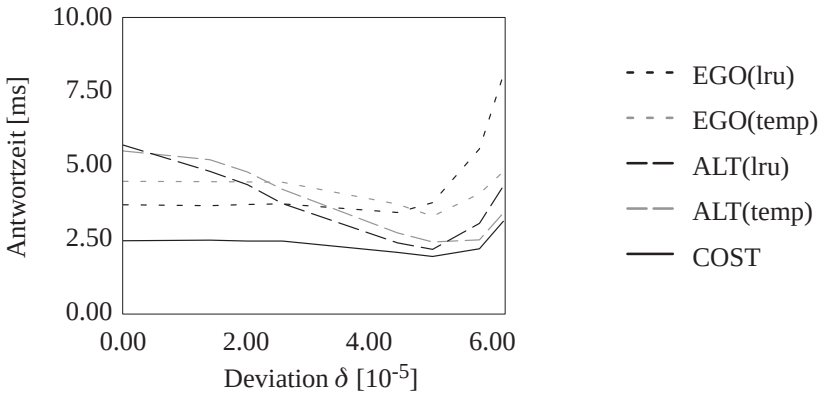


Abbildung 5.7: Einfluß der Ähnlichkeit im Zugriffsverhalten auf die Antwortzeit

einer deutlichen Reduktion der Antwortzeit führt. Erst bei einem sehr hohen Wert für die Deviation, der einer komplett zufälligen Permutation der Zugriffsmuster entspricht, tritt wieder eine Verschlechterung ein. Bei den egoistischen Verfahren und bei unserem kostenbasierten Verfahren ist eine solch starke Verbesserung bei geringer Deviation nicht zu beobachten, jedoch können wir auch hier bei sehr großer Deviation eine spürbare Verschlechterung der Antwortzeit erkennen.

Dieses Verhalten können wir wiederum an Hand der Trefferraten erklären, die wir in Abbildung 5.8 skizziert haben. Betrachten wir zunächst die altruistischen Verfahren. Wie wir bereits in dem vorangegangenen Experiment gesehen haben, wirkt sich die explizite Vermeidung von Replikaten bei der von uns standardmäßig gewählten Schräge von θ=1.0 negativ auf die Antwortzeit aus. Erhöhen wir die Deviation, so vermindert sich die “Überlappung” der heißen Bereiche auf den verschiedenen Rechnern. Es ist daher eher möglich, die jeweils heißesten Objekte auf den verschiedenen Rechnern im Cache zu halten, ohne daß dadurch das altruistische Paradigma verletzt wird. Dies erkennen wir daran, daß die Anzahl der lokalen Cache-Treffer von Abbildung (a) nach Abbildung (b) ansteigt, während gleichzeitig die globale Cache-Trefferrate nahezu konstant bleibt. Erhöhen wir die Deviation jedoch noch weiter (Abbildung (c)), so werden irgendwann die Zugriffsmuster der Rechner so unterschiedlich, daß andere Rechner kaum von den im nichtlokalen Cache gehaltenen Objekten profitieren können. Dadurch wird die lokale Cache-Trefferrate zwar weiter gesteigert, jedoch ist diese Verbesserung nicht mehr in der Lage, die Verschlechte-

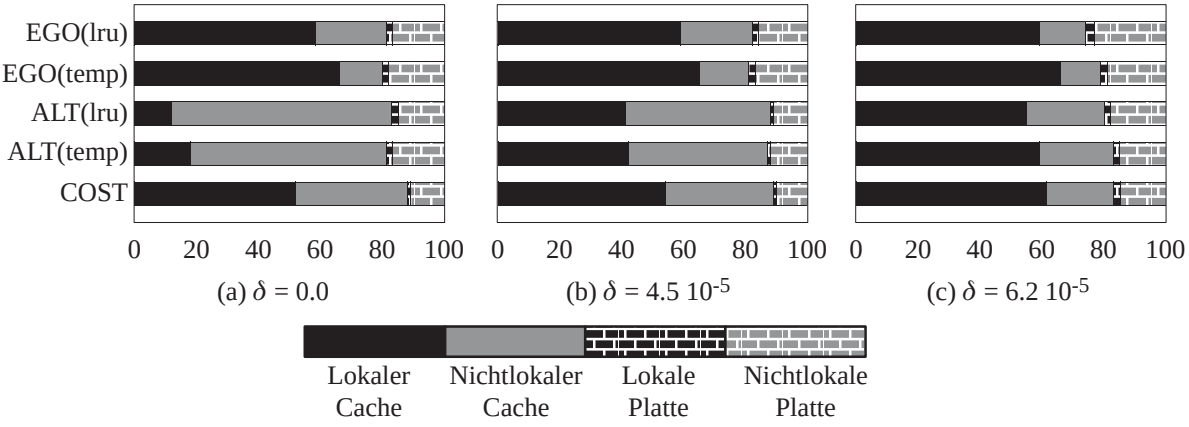


Abbildung 5.8: Trefferraten auf den verschiedenen Hierarchiestufen

rung durch die geringere globale Cache-Trefferrate auszugleichen. Dies ist der Grund für die Verschlechterung der Antwortzeit, die wir in Abbildung 5.7 beobachten können.

Bei den egoistischen Methoden ändert sich die lokale Cache-Trefferrate nicht, da unabhängig von der Deviation jeder Rechner jeweils seinen heißen Bereich an Objekten im Cache hält. Eine Verschlechterung der Antwortzeit tritt bei diesen Verfahren erst dann ein, wenn wiederum das Zugriffsverhalten so unterschiedlich ist, daß die Rechner nicht mehr von den im nichtlokalen Cache gehaltenen Objekten profitieren können. Ähnlich wie die egoistischen Verfahren liefert auch die kostenbasierte Heuristik nahezu eine konstante Antwortzeit über den beobachteten Bereich, jedoch ist die Antwortzeit insgesamt wesentlich besser. Bei sehr großen Werten für die Deviation steigt auch bei COST – aus den bereits oben angesprochenen Gründen – die Antwortzeit an. In Abbildung (c) können wir auch erkennen, daß sich die Heuristiken bezüglich ihrer Trefferraten immer stärker einander annähern. Dies kommt daher, daß sich bei einer komplett zufälligen Permutation der Zugriffsmuster egoistische und altruistische Ziele sich nicht mehr widersprechen.

5.2.3 Variation der Rechneraktivität

In diesem Experiment untersuchen wir die Reaktion der unterschiedlichen Cache-Heuristiken auf heterogene Lasten. Dazu variieren wir die in Abschnitt 4.3.1 eingeführte Rechneraktivität. Um einen Lastausgleich zwischen den Rechnern zu ermöglichen, werden wir in diesem Experiment für die altruistischen Heuristiken und für unsere kostenbasierte Heuristik jeweils eines der Lastverteilungsverfahren aus Abschnitt 3.5 benutzen. Dabei verwenden wir die zufällige Lastverteilungsmethode (RANDOM) zusammen mit ALT(lru) und die lastabhängige und zielgerichtete Verteilungsmethode (MIN) zusammen mit ALT(temp) und COST. Für die egoistischen Heuristiken benutzen wir keine Lastverteilungsmethode, da wir in diesen Heuristiken keine Informationen besitzen, nach denen eine globale Lastverteilung durchgeführt werden kann. Darüber hinaus widerspricht eine Lastverteilung auch dem egoistischen Paradigma.

Um die Arbeitsweise der verschiedenen Methoden besser illustrieren zu können, haben wir in diesem Experiment eine von unserem Standard abweichende Last gewählt. Das Zugriffsmuster der verschiedenen Rechner haben wir zyklisch verschoben, so daß sich die heißen Bereiche der Rechner maximal unterscheiden. Um zu vermeiden, daß durch das Lesen eines Objekts von der nichtlokalen Platte bereits eine implizite Lastbalancierung erreicht wird, haben wir die Objekte entsprechend einer Bereichspartitionierung so auf die Platten verteilt, daß sich die jeweils lokal heißesten Objekte auch auf der lokalen Platte befinden.

Während der Aufwärmphase benutzen wir – unabhängig von der spezifizierten Rechneraktivität – jeweils eine homogene Last. Diese Vorgehensweise soll verhindern, daß schon vor dem eigentlichen Meßintervall ein Lastausgleich durchgeführt wird. Erst zu Beginn der Messung ändern wir die Last entsprechend der vorgegebenen Rechneraktivität. Bei der Variation der Rechneraktivität wird – entsprechend der Definition 4.3 – darauf geachtet, daß die Gesamtlast im System konstant bleibt und sich lediglich die Verteilung der Last auf die Rechner verändert. Um einen fairen Vergleich der Verfahren zu ermöglichen, wird für jede Heuristik eine fest vorgegebene Anzahl von Zugriffen durchgeführt. In Abbildung 5.9 sehen wir die Antwortzeiten der verschiedenen Heuristiken, wenn wir die Schräge der Rechneraktivität von η=0 (Lastgleichgewicht) bis η=1.5 variieren.

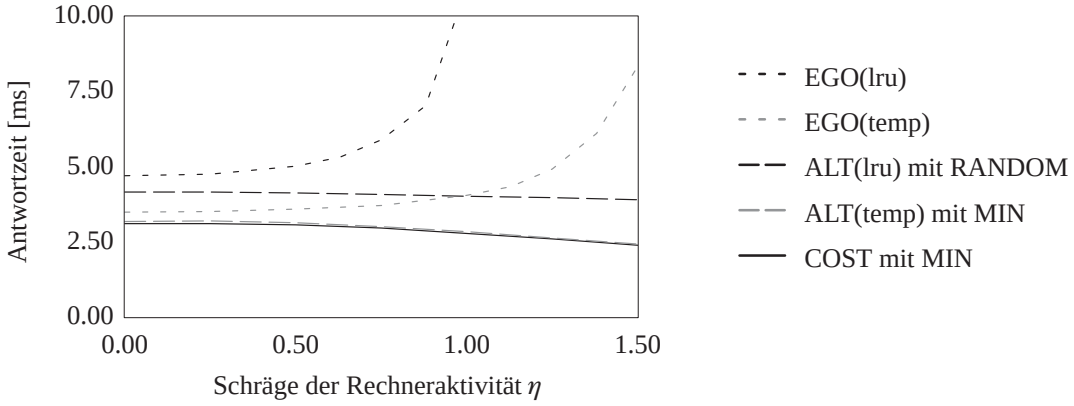


Abbildung 5.9: Variation der Rechneraktivität

Betrachten wir zunächst einmal die egoistischen Verfahren, die – wie wir oben erwähnt haben – ohne Lastbalancierung auskommen müssen. Auffällig ist hier, daß EGO(temp) sehr viel besser abschneidet als EGO(lru). Wie wir bereits im ersten Experiment gezeigt haben, ist EGO(temp) durch die umfangreichere statistische Information in der Lage, die lokale Wichtigkeit eines Objekts genauer abzuschätzen. In den vorangegangenen Experimenten führte diese bessere Approximation zu einem höheren Replikationsgrad und damit zu einer schlechteren Antwortzeit. Durch die zyklische Verschiebung der Last ist dies jedoch hier nicht der Fall, und somit führt EGO(temp) zu einer besseren lokalen Cache-Trefferrate, ohne daß die globale Cache-Trefferrate dadurch verringert wird. Obwohl sich die beiden egoistischen Heuristiken bezüglich ihrer absoluten Antwortzeit deutlich unterscheiden, ist doch der generelle Verlauf der beiden Kurven ähnlich. So können wir in Abbildung 5.9 beobachten, daß sich die Antwortzeiten bei geringer Rechneraktivität zwar noch relativ konstant verhalten, daß sie jedoch bei größerer Schräge der Aktivität sehr schnell ansteigen. Zur Erklärung dieses Verhaltens ist in Abbildung 5.10 die Differenz der Antwortzeiten zwischen dem am stärksten und dem am geringsten belasteten Rechner aufgetragen.

In dieser Abbildung ist sehr deutlich zu erkennen, daß durch die fehlende Möglichkeit der Lastverteilung die Differenz zwischen den Antwortzeiten auf den verschiedenen Rechnern stetig anwächst. Da zusätzlich die Gesamtlast im System gleich bleibt, führt dies dazu, daß der am stärksten belastete Rechner einen immer größeren Anteil der Last bearbeiten muß. Dieser gelangt

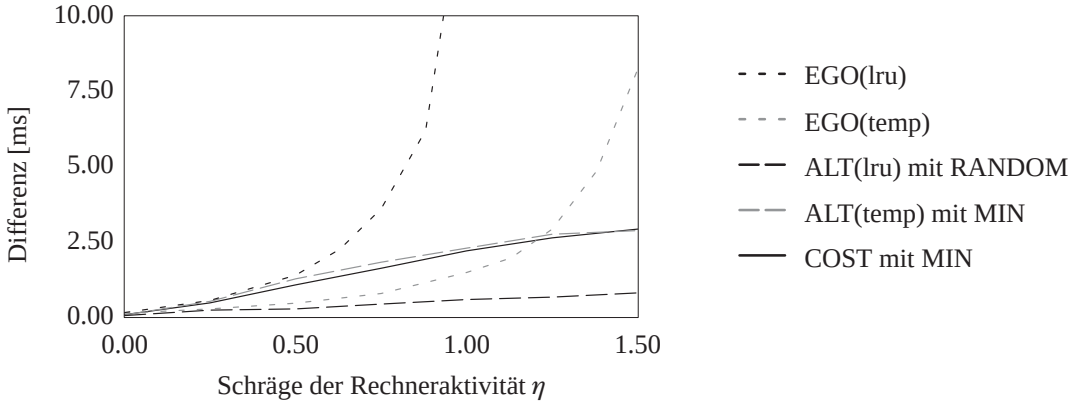
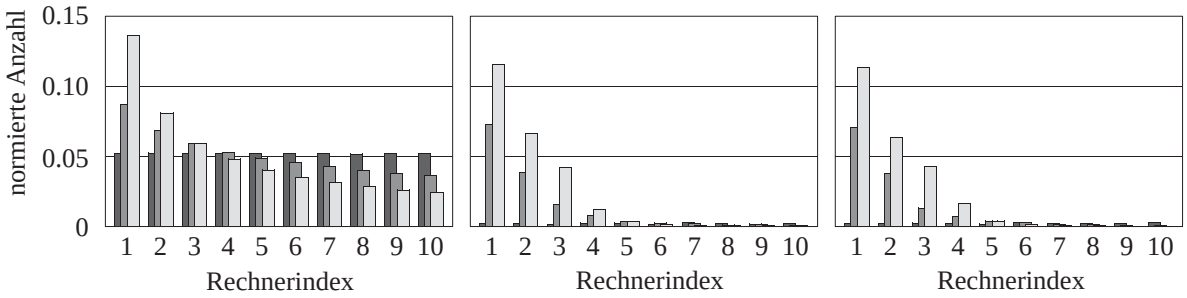


Abbildung 5.10: Differenz in der Antwortzeit zwischen höchst- und niedrigstbelastetem Rechner

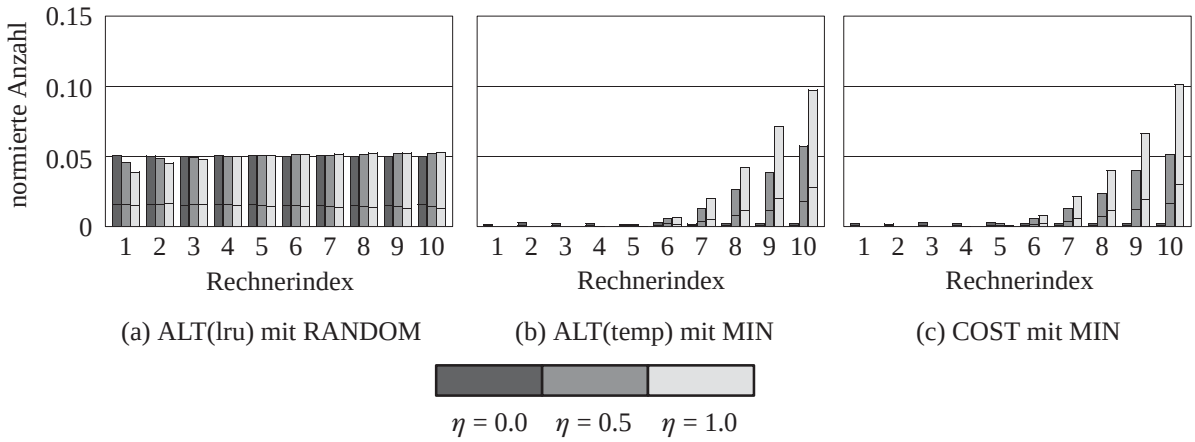
schließlich an seine Leistungsgrenze. So ist bei EGO(lru) bereits bei einer Schräge von θ=0.75 die maximale Plattenauslastung größer als 80%, während die durchschnittliche Auslastung im Gesamtsystem lediglich 47% beträgt. Das gleiche Phänomen ist auch die Ursache für den Anstieg der Antwortzeit bei EGO(temp), jedoch tritt ein vergleichbarer Plattenengpaß durch die bessere Cache-Trefferrate erst später auf.

Zum besseren Verständnis der Verhaltensweisen der anderen Heuristiken sind in den Abbildungen 5.11 und 5.12 die Aktivitäten der Lastverteilung skizziert. In der Abbildung 5.11 haben wir die Anzahl der auf den einzelnen Rechnern initiierten Migrationen aufgetragen, während wir in der Abbildung 5.12 die Anzahl der zu den Rechnern migrierten Objekte zeigen. Da nicht jedes Objekt, das zu einem Rechner migriert wird, auch in den Cache dieses Rechners eingefügt wird, ist in der Abbildung 5.12 die Gesamtanzahl in erfolgreiche (unten) und fehlgeschlagene (oben) Migrationen aufgeteilt. Um eine von der Laufzeit unabhängige Darstellung zu erhalten, sind in beiden Abbildungen die Größen bezüglich der Gesamtanzahl der Zugriffe im System normiert.



(a) ALT(lru) mit RANDOM (b) ALT(temp) mit MIN (c) COST mit MIN

Abbildung 5.11: Migrationsaktivitäten auf dem Initiatorrechner



(a) ALT(lru) mit RANDOM (b) ALT(temp) mit MIN (c) COST mit MIN

Abbildung 5.12: Migrationsaktivitäten auf dem Zielrechner

Betrachten wir zunächst den Fall η=0. In dieser Situation haben alle Rechner die gleiche Last. Da eine Migration zu einer Zusatzlast führt, sollten keine Migrationen von Objekten durchgeführt werden. In Abbildung 5.11(a) können wir jedoch erkennen, daß bei ALT(lru) in Verbindung mit der RANDOM-Lastverteilungsmethode sehr viele Migrationen initiiert werden. Dies liegt daran, daß bei RANDOM unabhängig vom Systemzustand Migrationen gestartet werden. Diese vielen unsinnigen Objektverschiebungen führen dazu, daß die Migration von Objekten das Netzwerk mit 29% sehr viel stärker auslastet, als der reguläre Datentransfer mit 17%. Im Gegensatz dazu

sehen wir in den Abbildungen 5.11 (b) und (c), daß die MIN-Lastverteilungsmethode das Lastgleichgewicht im System erkennt und daher nur sehr wenige Migrationen initiiert. Dieser Unterschied in der Netzwerkauslastung ist zusammen mit der besseren Cache-Trefferrate der temperaturbasierten Verfahren der Grund für die bessere Performance von ALT(temp) und COST verglichen mit ALT(lru).

Betrachten wir als nächstes die Situationen, in denen ein Ungleichgewicht vorliegt. Bei allen Verfahren können wir nun erkennen, daß die Anzahl der Initiierungen von dem Rechner abhängt. Bei ALT(lru)/RANDOM ist diese Abhängigkeit jedoch alleine durch die ungleich hohe Anzahl von Zugriffen auf den Rechnern bewirkt. So initiiert jeder Rechner – unabhängig von der Last der anderen Rechner – eine Migration, wenn ein Unikat aus dem Cache verdrängt wird. Im Gegensatz dazu erkennen wir bei ALT(temp)/MIN und COST/MIN, daß gering belastete Rechner praktisch keine Migrationen starten. Auf diesen Rechnern erkennt MIN, daß alle anderen Rechner ungefähr genauso stark oder sogar stärker belastet sind. Daher ist die Wahrscheinlichkeit sehr gering, daß eine Migration zu einer besseren Lastverteilung führt.

In Abbildung 5.12 erkennen wir auch sehr deutlich die Berücksichtigung der Rechnerlast auf der Empfängerseite der Migrationen. Während die Anzahl der Objekte, die zu einem bestimmten Rechner gesendet werden, bei ALT(lru)/RANDOM unabhängig von dem Ungleichgewicht der Last im System ist, erhalten bei ALT(temp)/MIN und COST/MIN die unterlasteten Rechner sehr viel mehr Objekte. Weiterhin beobachten wir, daß bei diesen beiden Verfahren nicht nur die Gesamtanzahl der zu den unterlasteten Rechnern migrierten Objekte größer ist, sondern daß auch der Anteil der auf den unterlasteten Rechnern wirklich in den Cache eingefügten Objekte größer ist als bei ALT(lru)/RANDOM. Durch die unzureichende Information bei ALT(lru)/RANDOM ist es auf dem Empfängerrechner unmöglich, den genauen Wert eines migrierten Objekts abzuschätzen. Dies führt dazu, daß selbst kalte Objekte, die zu einem sehr stark belasteten Rechner migriert werden, in den dortigen Cache eingefügt werden, wenn in diesem wenigstens ein Replikat existiert. Dies wird selbst dann durchgeführt, wenn das verdrängte Objekt sehr viel heißer ist als das migrierte Objekt. Dadurch führt ALT(lru)/RANDOM zwar zu einer Situation, in der die Antwortzeiten auf allen Rechnern sehr gut balanciert sind (siehe Abbildung 5.10), jedoch sind die Antwortzeiten im Mittel sehr viel schlechter als die von ALT(temp)/MIN und COST/MIN.

Die Wirksamkeit der durchgeführten Migrationen können wir auch an Hand der Abbildung 5.13 zeigen. Hier ist für jeden Rechner der Anteil der Objekte am verteilten aggregierten Cache gezeigt, der sich auf seiner lokalen Platte befinden. Da die Objekte, die sich auf der lokalen Platte befinden, auch gleichzeitig die Objekte sind, auf die am häufigsten lokal zugegriffen wird, zeigt uns diese Abbildung wie groß der jeweils im aggregierten, verteilten Cache gehaltene *heiße Bereich* eines Rechners ist. Während wir bei den egoistischen Verfahren sehen, daß jeder Rechner den gleichen Anteil des aggregierten Caches benutzt – nämlich gerade seinen lokalen Cache –, erkennen wir bei den anderen Verfahren, daß die Rechner, die eine höhere Last haben, mehr Objekte ihres heißen Bereichs im aggregierten Cache halten. Dies kommt daher, daß durch die Migration Objekte von den höher belasteten Rechnern auf die weniger belastete Rechnern ausgelagert werden. Dies führt bei den stärker belasteten Rechnern zu einer Vergrößerung der Cache-Trefferrate, und dadurch wird ein Engpaß vermieden. Gleichzeitig erfolgt eine Verkleinerung der Cache-Bereiche für die weniger belasteten Rechner, die zu einer Verschlechterung der Antwortzeit auf diesen Rechnern führt. Da wir jedoch die mit der Ankunftsrate gewichtete mittlere Ant-

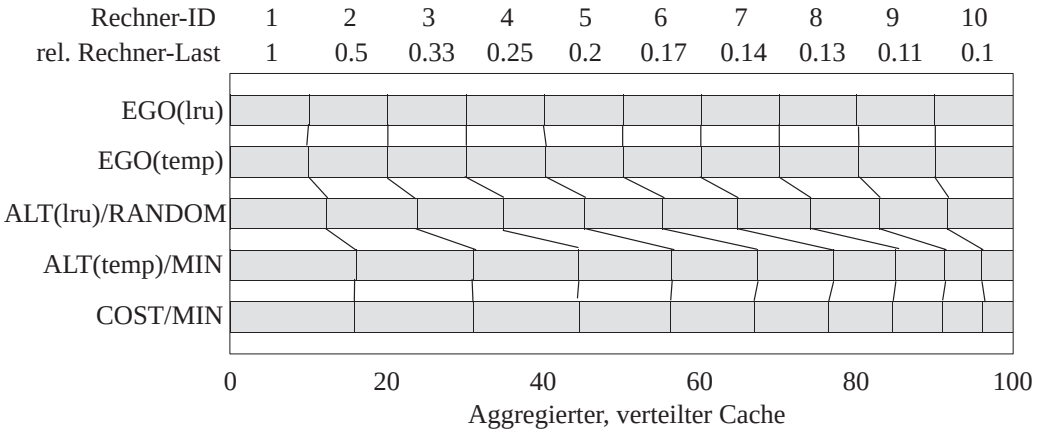


Abbildung 5.13: Aufteilung des globalen Caches bei der Rechneraktivität η=1.0

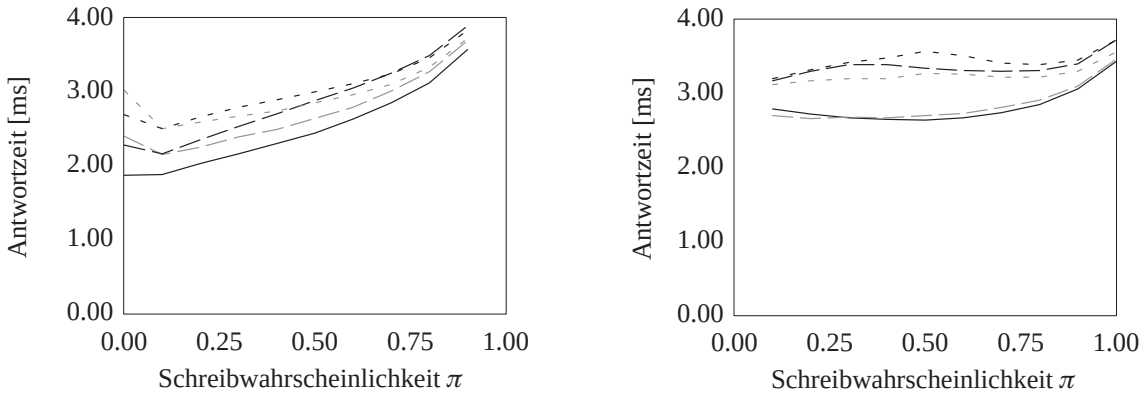
wortzeit als Optimierungsmetrik betrachten, fällt diese Verschlechterung nicht so stark ins Gewicht wie der Gewinn auf den stark belasteten Rechnern. Dies ist der Grund für das Fallen der mittleren Antwortzeit bei hohen Schrägen, das wir in Abbildung 5.9 beobachten können. Obwohl wir eine Ausnutzung der nichtlokalen Caches sowohl durch die RANDOM als auch durch die MIN-Lastverteilungsmethode erreichen, erkennen wir in Abbildung 5.13 doch, daß MIN mehr Objekte von stark belasteten Rechnern auslagert und damit zu einer besseren Performance führt.

5.2.4 Variation der Schreibwahrscheinlichkeit

Bislang haben wir in unseren Experimenten nur Lesezugriffe betrachtet. Wichtig ist jedoch auch das Zusammenspiel von Lese- und Schreibzugriffen. Um eine zu starke Verzögerung einer Cache-Ersetzung zu vermeiden, bevorzugen wir bei einer Ersetzung nicht-geänderte Objekte. Geänderte Objekte werden nach Möglichkeit asynchron von einem Hintergrundprozeß auf Platte gesichert. In unserem Experiment wird dieser Prozeß jeweils aktiv, wenn mehr als 60% des lokalen Caches mit schmutzigen Objekten gefüllt sind. Der Prozeß sichert dann so lange schmutzige Objekte auf Platte, bis dieser Anteil unter 40% sinkt. Dabei wird ein Zurückschreiben von geänderten Objekten jedoch nur dann durchgeführt, wenn nicht schon andere, reguläre Aufträge an der Platte auf eine Bearbeitung warten. Durch diese Bevorzugung von nicht-schmutzigen Objekten bei der Cache-Ersetzung verändern die Schreibzugriffe die Caching-Strategien, da nun ein Objekt verdrängt wird, obwohl ein entsprechend der Caching-Heuristik weniger wichtiges, jedoch schmutziges Objekt im Cache gehalten wird. In diesem Experiment wollen wir diesen Einfluß untersuchen. Dazu variieren wir die Schreibwahrscheinlichkeit π von 0 bis 1.[9] In der Abbildung 5.14 sind die mittleren Antwortzeiten für Lese- und Schreibzugriffe bei unterschiedlicher Schreibwahrscheinlichkeit aufgetragen.

Betrachten wir zunächst die Kurve mit den Antwortzeiten für Lesezugriffe. Außer bei sehr geringer Schreibrate steigen die Antwortzeiten mit wachsender Schreibwahrscheinlichkeit monoton

9. Obwohl in der Realität eine Schreibrate größer als π=0.3 kaum vorkommen wird, variieren wir die Schreibrate bis zu zu dem Extrem π=1.0, in dem nur noch Schreibzugriffe und keine Lesezugriffe mehr vorkommen. Dies tun wir, um die Verhaltensweisen der verschiedenen Heuristiken auch unter extremen Streßbedingungen zu testen.



(a) Antwortzeit für Lesezugriffe

(b) Antwortzeit für Schreibzugriffe

EGO (lru) EGO (temp) ALT (lru) ALT (temp) COST

Abbildung 5.14: Einfluß der Schreibwahrscheinlichkeit auf die Antwortzeit

an. Der Grund für diesen Anstieg liegt in einer wachsenden Netzauslastung. Obwohl wir die Ankunftsrate in allen Experimenten gleich lassen, wird das Netzwerk bei einer höheren Schreibwahrscheinlichkeit stärker belastet. Eine der Ursachen liegt in der Abarbeitung unseres Kohärenzprotokolls. So muß nach einem erfolgten Schreibzugriff das geänderte Objekt wieder an den Heimatrechner gesendet werden. Dieses Zurücksenden der schmutzigen Objekte ist zwar für die Korrektheit unseres Protokolls nicht unbedingt notwendig, jedoch erleichtert es eine mögliche Erweiterung unseres Prototyps um eine Logging-Komponente am Heimatrechner, die man für eine Daten-Recovery bräuchte.

Eine weitere Ursache, die eine wachsende Netzlast vor allem bei den egoistischen Heuristiken und bei unserer kostenbasierten Heuristik bewirkt, ist die Elimination von Replikaten durch das Callback-Protokoll. Wie wir in Abschnitt 4.1.3 gesehen haben, müssen zur Sicherstellung der Kohärenz vor einem Schreibzugriff zunächst alle Replikate des betreffenden Objekts gelöscht werden. Dadurch verringert sich die lokale Cache-Trefferrate, und es müssen mehr Objekte über das Netzwerk zugegriffen werden. Dieses Verhalten erkennen wir sehr gut in der Abbildung 5.15. Während sich die Trefferraten bei einer Last ohne Schreibzugriffe (Abbildung (a)) noch sehr stark unterscheiden, erkennen wir in den Abbildungen (b) und (c), daß sich die Verfahren mit wachsender Schreibwahrscheinlichkeit immer stärker annähern. Auch an den Antwortzeiten in Abbildung 5.14 erkennen wir eine Annäherung der Verfahren für hohe Schreibraten. Jedoch erzielen die Verfahren, welche die Hitze der Objekte mit berücksichtigen, eine etwas bessere Antwortzeit, da sie durch die umfangreichere statistische Information die heißeren Objekte im Cache halten.

Nachdem wir den generellen Anstieg der Antwortzeit für Lesezugriffe beschrieben haben, wollen wir nun die Ursache für die Verbesserung der Leseantwortzeit in dem Intervall von π=0 bis π=0.15 untersuchen. Wie wir bereits weiter oben beschrieben haben, bewirken die Schreibzugriffe bei den egoistischen Verfahren eine Reduzierung des Replikationsgrads von Objekten. Während dies im allgemeinen zu einer Verschlechterung der Antwortzeit führt, da das Netz stärker ausgelastet wird, ist bei sehr geringer Schreibrate die Gesamtauslastung des Netzes so gering, daß der Vorteil durch die größere globale Cache-Trefferrate den Nachteil der geringeren lokalen

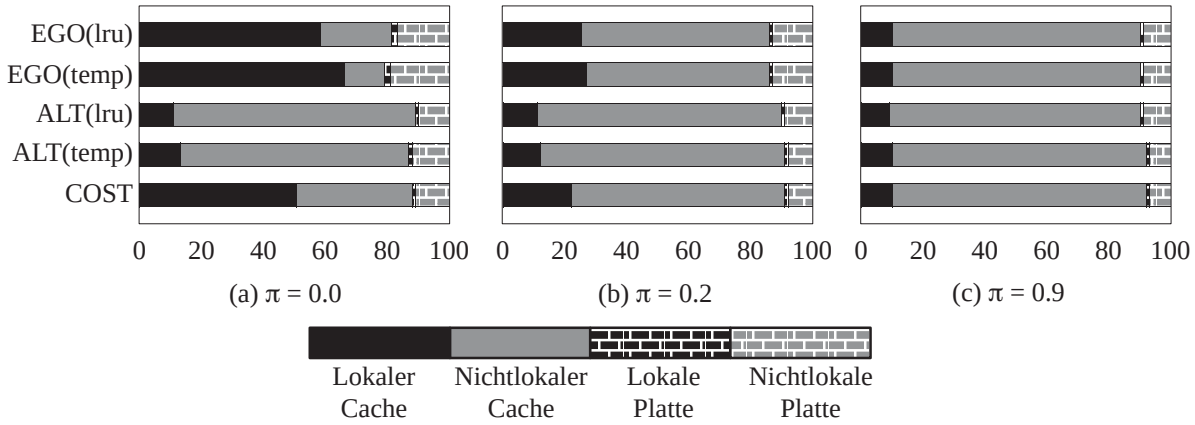


Abbildung 5.15: Lese-Trefferraten auf den verschiedenen Hierarchiestufen

Cache-Trefferrate dominiert. Dies führt zu der Verbesserung der Leseantwortzeit bei geringer Schreibrate, die wir in Abbildung 5.14 beobachten können.

Betrachten wir nun die Antwortzeiten für Schreibzugriffe. In Abbildung 5.14(b) erkennen wir, daß wir die Heuristiken grob in zwei Klassen aufteilen können. Die erste Klasse wird von ALT(temp) und COST gebildet. Diese liefert über den ganzen Bereich eine bessere Performance als die zweite Klasse, in der wir EGO(lru), EGO(temp) und ALT(lru) zusammenfassen. Analog zu den Lesezugriffen haben wir auch bei den Schreibzugriffen mitprotokolliert, von welcher Stufe der Speicherhierarchie die Objekte vor einem Zugriff jeweils gelesen werden. Die daraus abgeleiteten Trefferraten haben wir in der Abbildung 5.16 skizziert. In dieser Abbildung sehen wir, daß bei geringer Schreibwahrscheinlichkeit die lokalen Trefferraten sich noch stärker unterscheiden, während bei reinen Schreibzugriffen die lokalen Cache-Trefferraten für alle Verfahren fast gleich sind. Der Grund hierfür ist die unterschiedliche Anzahl von Replikaten, die – bei geringer Schreibwahrscheinlichkeit – von den Heuristiken kreiert werden. Treten jedoch sehr viele Schreibzugriffe auf, so wirken die Callbacks einer Replikation entgegen. Daher existieren unabhängig von der gewählten Heuristik fast nur noch Unikate im System. Bei sehr hohen Schreibraten erkennen wir – ähnlich wie bei den Lesezugriffen – die Überlegenheit der Verfahren, die die Objekthitzen berücksichtigen. Durch die bessere statistische Information können sie die wertvolleren Objekte im Cache halten, und dadurch ist die globale Cache-Trefferrate größer (Abbildung

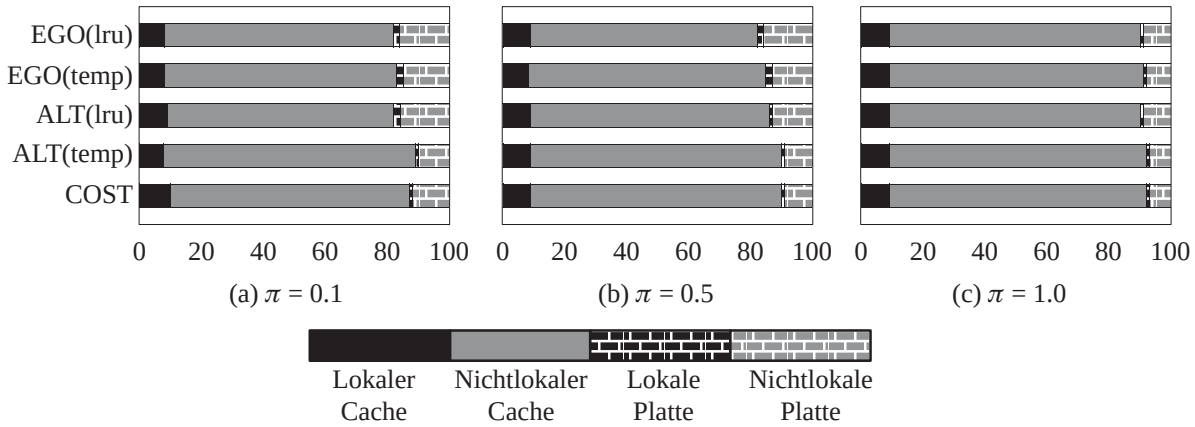


Abbildung 5.16: Schreibzugriff-Trefferraten auf den verschiedenen Hierarchiestufen

5.16). Dies bewirkt eine Entlastung der Engpaßressource “Platte” und damit eine Verbesserung der mittleren Antwortzeit.

5.3 Variation der Datenparameter

Neben den Lastparametern beeinflussen auch die Datenparameter die Leistungsfähigkeit der verschiedenen Heuristiken. Dazu werden wir in diesem Abschnitt den Einfluß der wichtigsten Datenparameter systematisch untersuchen. In einer ersten Meßreihe variieren wir die Größe des Verhältnisses zwischen der Gesamtdatenmenge und dem insgesamt vorhandenen aggregierten Cache. Danach untersuchen wir den Einfluß der Objektgröße, indem wir zunächst nur die mittlere Objektgröße und danach die Wahrscheinlichkeitsverteilung, nach der die Objektgrößen generiert werden, verändern. Abschließend betrachten wir Umgebungen, in denen die Objekte ungleichmäßig auf den Platten alloziert werden.

5.3.1 Variation des Verhältnisses von Datengröße zu Cache-Größe

In diesem Experiment untersuchen wir den Einfluß des Verhältnises der Datengröße zu der Cache-Größe. Um dieses Verhältnis zu verändern, halten wir die Cache-Größe der Rechner konstant und variieren die Objektanzahl. Da sich auch hier wieder die Antwortzeiten je nach gewähltem Größenverhältnis sehr stark unterscheiden, normieren wir die Antwortzeiten wieder bezüglich EGO(lru). Die Ergebnisse der Experimente, wenn wir das Größenverhältnis der Daten zu dem aggregierten Cache von ν=1 bis 20 variieren, sind in Abbildung 5.17 gezeigt.

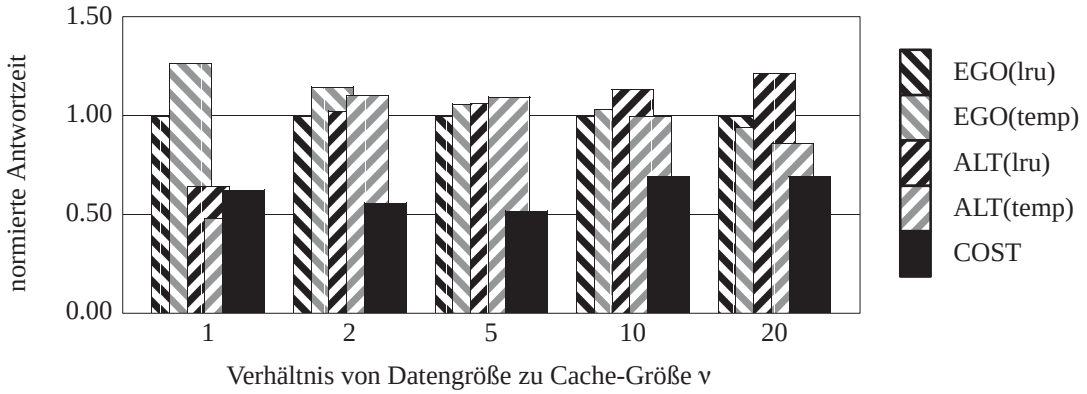


Abbildung 5.17: Einfluß des Verhältnisses von Datengröße zu Cache-Größe auf die Antwortzeit

Betrachten wir in dieser Abbildung zunächst die beiden egoistischen Verfahren. Bei kleinen Werten für ν liefert EGO(temp) die schlechtere mittlere Antwortzeit. Der Grund hierfür ist der bereits mehrfach angesprochene höhere Replikationsgrad, der durch die bessere Approximation des rein-egoistischen Verfahrens verursacht wird. Durch diese höhere Replikation ist der Anteil der im Cache gehaltenen Objekte geringer und damit auch die globale Cache-Trefferrate, was zu der schlechteren Performance führt. Bei großen Datenbeständen verhält sich EGO(temp) jedoch besser als EGO(lru). Zwar führt EGO(temp) immer noch zu einer größeren Replikation, jedoch ist die LRU-Ersetzungsstrategie bei wachsender Datengröße immer schlechter in der Lage, die heißen Objekte der Rechner im Cache zu halten. Der Grund hierfür liegt darin, daß durch die Erhö-

hung der Objektanzahl die Anzahl der Zugriffe, die zwischen zwei Zugriffen auf das gleiche Objekt erfolgen, im Mittel immer größer wird. Dadurch sinkt die Wahrscheinlichkeit, daß ein zweiter Zugriff ein Objekt liest, bevor dieses aus dem Cache verdrängt wird. Dies führt dazu, daß sich das Verhalten der LRU-Ersetzung immer mehr einer FIFO-Strategie annähert. Anders ist dies bei der LRU-k Methode, die von EGO(temp) benutzt wird. Da hier auch Informationen über den Zeitraum des eigentlichen Cachings hinaus gehalten werden, ist diese Methode sehr viel besser in der Lage, zwischen wichtigen und unwichtigen Objekten zu unterscheiden.

Bei den altruistischen Verfahren fällt zuerst die sehr gute Performance auf, wenn das Verhältnis der Größen gleich 1 ist. In diesem Fall paßt der komplette Datenbestand in den Cache, und beide altruistischen Verfahren können durch das Vermeiden von Replikationen fast alle Objekte im Cache halten. So hält ALT(lru) 98% und ALT(temp) 95% aller Objekte im Cache. Dadurch können Plattenzugriffe fast vollkommen vermieden werden. Da das Netzwerk bei der betrachteten Ankunftsrate nicht überlastet wird, erreichen diese beiden Verfahren eine sehr geringe mittlere Antwortzeit. Vergrößern wir jedoch die Objektanzahl, so erkennen wir in Abbildung 5.17, daß die relative Leistungsfähigkeit von ALT(lru) immer schlechter wird. Der Grund hierfür ist wiederum das bereits oben angesprochene Problem der LRU-Strategie beim Erkennen der wichtigen Objekte bei sehr großer Objektanzahl. Auch hier profitiert ALT(temp) von der größeren statistischen Information und liefert daher eine bessere mittlere Antwortzeit.

Unsere kostenbasierte Heuristik ist in dem Fall, in dem alle Objekte in den Cache passen, etwas schlechter als ALT(temp), da COST die Erzeugung von Replikaten nicht explizit verhindert und daher nur 75% der Objekte – verglichen mit 95% bei ALT(temp) – im Cache hält. Diese Differenz in der Anzahl der im Cache gehalten Objekte scheint sehr groß. Betrachten wir jedoch die globale Cache-Trefferrate so erkennen wir, daß COST mit 95% nur um 3% schlechter abschneidet als ALT(temp). Dieser Unterschied in der globalen Cache-Trefferrate und die damit verbundene geringere Plattenauslastung ist der Grund für die bessere Performance von ALT(temp). Bei einer Vergrößerung des Datenbestandes erkennen wir jedoch, daß die Berücksichtigung der Beschaffungskosten unserer kostenbasierten Heuristik sehr viel bessere Ergebnisse liefert als die einfacheren Heuristiken.

Bemerkenswert ist auch, daß bei einem immer größer werdenden Datenbestand immer mehr der Unterschied zwischen temperaturbasiert bzw. kostenbasiert und LRU-basiert zu Tage tritt. Gleichzeitig wird die Unterscheidung zwischen altruistisch und egoistisch immer unwichtiger.

5.3.2 Variation der Objektgröße

Bereits in Abschnitt 4.3.2 haben wir erläutert, daß nicht nur das relative Größenverhältnis zwischen dem aggregierten Cache und der Gesamtdatenmenge einen Einfluß auf die Verfahren hat, sondern auch die absolute Größe der Objekte. Um dies zu belegen, haben wir in einer Meßreihe die Objektgröße von 1/2 kByte bis 32 kByte variiert; die Größen innerhalb einer Messung sind jedoch für alle Objekte gleich. Die Größe der Caches und das relative Verhältnis zwischen der gesamten Cache-Größe und der Datengröße haben wir ebenfalls konstant gehalten. Die relativen Antwortzeiten der verschiedenen Verfahren sind in der Abbildung 5.18 gezeigt.

Deutlich ist zu erkennen, daß mit zunehmender Objektgröße die altruistischen Verfahren immer schlechter werden. Auch die relative Performance unserer kostenbasierten Heuristik verschlech-

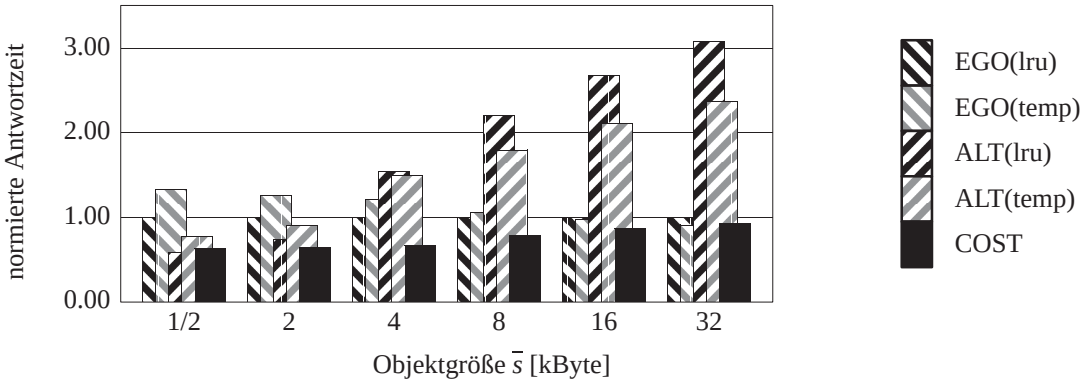


Abbildung 5.18: Einfluß der Objektgröße auf die Antwortzeit

tert sich mit größer werdenden Objekten und nähert sich dabei immer stärker den egoistischen Verfahren an. Durch größere Objekte wachsen die Transferzeiten für das Netzwerk und für die Festplatten an; die Latenzzeit für das Anfahren der Zielspur und für die Rotationsverzögerung auf den Platten wachsen jedoch nicht. Dadurch wird der Einfluß der größenunabhängigen Kosten immer geringer. Dies führt dazu, daß der Unterschied in den Zugriffskosten zwischen dem nichtlokalen Cache und der Platte immer kleiner wird. Dies erkennen wir auch in Tabelle 5.6. Während bei relativ kleinen Objekten (2 kByte) ein nichtlokaler Cache-Zugriff bei allen Verfahren sehr viel schneller ist als ein Plattenzugriff, dreht sich dies bei großen Objekte (32 kByte) und den altruistischen Verfahren um.

Verfahren	**Lokaler Cache**		**Nichtlokaler Cache**		**Lokale Platte**		**Nichtlokale Platte**	
	2 kB	**32 kB**	**2 kB**	**32 kB**	**2 kB**	**32 kB**	**2 kB**	**32 kB**
EGO (lru)	0.1 ms	0.1 ms	0.5 ms	6.8 ms	18.6 ms	13.0 ms	19.1 ms	18.8 ms
EGO (temp)	0.1 ms	0.1 ms	0.5 ms	6.0 ms	20.3 ms	12.1 ms	20.4 ms	17.3 ms
ALT (lru)	0.3 ms	2.3 ms	0.7 ms	18.8 ms	13.8 ms	14.6 ms	14.4 ms	32.0 ms
ALT (temp)	0.1 ms	0.7 ms	0.8 ms	15.2 ms	15.4 ms	12.3 ms	15.9 ms	26.8 ms
COST	0.1 ms	0.1 ms	0.7 ms	7.4 ms	14.2 ms	12.2 ms	14.8 ms	19.1 ms

Tabelle 5.6: Gemessene Zugriffszeiten bei unterschiedlicher Objektgröße

Diese Umkehrung der Zugriffskosten tritt nur bei den altruistischen Verfahren auf, da das Optimierungsziel dieser Verfahren davon ausgeht, daß Zugriffe auf den nichtlokalen Cache sehr viel günstiger sind als Plattenzugriffe. Daher versuchen diese Heuristiken die globale Trefferrate zu maximieren, ohne jedoch zu bemerken, daß die Annahmen, die zu diesem Optimierungsziel geführt haben, nicht mehr gültig sind. Durch die höhere Anzahl von Netzzugriffen wird die Auslastung des Netzwerks weiter erhöht, was zu einer zusätzlichen Verschlechterung von nichtlokalen Cache-Zugriffen führt.

Da unsere kostenbasierte Heuristik jeweils die Kosten für die Zugriffe auf die unterschiedlichen Stufen der Speicherhierarchie mit in die Berechnung des Objektwertes einbezieht, bemerkt sie eine Veränderung in den Abständen zwischen den Speicherhierarchiestufen und kann geeignet darauf reagieren. Dies führt dazu, daß in dem Fall sehr großer Objekte unsere kostenbasierte Heuristik eine fast komplett egoistische Verhaltensweise annimmt.

5.3.3 Variabel große Objekte

Nachdem wir in dem vorangegangenen Experiment den Einfluß einer konstanten Objektgröße untersucht haben, wollen wir nun variabel große Objekte zulassen. Unabhängig von der Objektgrößenverteilung wird die Gesamtdatengröße jeweils doppelt so groß wie der aggregierte Hauptspeicher gewählt. Die Objektgrößen generieren wir entsprechend den folgenden Verteilungsfunktionen:

- eine konstante Verteilung mit der Objektgröße 16 kByte,
- eine Zipf-Verteilung mit der mittleren Objektgröße 16 kByte und der Schräge 1.0,
- eine Exponentialverteilung mit der mittleren Objektgröße 16 kByte und
- eine Normalverteilung mit der mittleren Objektgröße von 16 kByte und einer Standardabweichung von ebenfalls 16 kByte.

Da eine Normalverteilung auch negative Werte liefert, dies aber als Objektgröße keinen Sinn ergibt, werden solche Werte bei der Bestimmung der Objektgröße einfach verworfen, und es wird neu gewürfelt. Ein ähnliches Problem tritt auch mit zu großen Objekten auf. So können z.B. in unserer Architektur nie Objekte in den Cache geladen werden, die größer als die lokalen Caches sind. Da alle drei Verteilungen mit variabler Größe prinzipiell beliebig große Werte generieren können, verwerfen wir auch hier sämtliche Werte, die größer als eine vorgegebene Maximalgröße (ein Viertel der lokalen Cache-Größe) sind, und würfeln neue Werte. Diese Vorgehensweise verändert zwar die Mittelwerte und die Varianzen der generierten Größenverteilungen leicht, jedoch bleibt die prinzipielle “Form” bestehen.

Die Zuordnung der Größen zu den Objekten erfolgt jeweils zufällig und damit unabhängig von der Hitze der betreffenden Objekte. Dadurch ist es nicht mehr ausreichend, nur eine einzige Messung mit einer genügend großen Konfidenz durchzuführen, sondern wir müssen mehrere Messungen mit unterschiedlicher Zuordnung der Größen durchführen und über diese mitteln. In diesem Experiment haben wir für jeden Parameterwert jeweils so viele Messungen durchgeführt, bis die Breite des 95% Konfidenz-Intervalls geringer als 5% des ermittelten Mittelwertes war.

In Abschnitt 4.1.5 haben wir zwei unterschiedliche Verfahren zur Verwaltung eines Caches mit variabel großen Objekten vorgestellt. Während sich das Verfahren mit Kompaktifizierung streng an die durch die Heuristik vorgegebene Ersetzungsreihenfolge hält, verändert das Verfahren ohne Kompaktifizierung im allgemeinen diese Reihenfolge. Obwohl somit das Verfahren mit Kompaktifizierung vorzuziehen ist, hat sich in einem ersten Experiment gezeigt, daß die CPU durch die sehr hohen Kosten für das Umkopieren der Objekte zum Engpaß wird. Daher haben wir für die folgende Meßreihe die Variante ausgewählt, die ohne Umkopieren auskommt. Die Ergebnisse dieser Messung sind in Abbildung 5.19 aufgeführt.

In dieser Abbildung erkennen wir, daß bei variabel großen Objekten die temperaturbasierten Verfahren gegenüber den LRU-basierten im Vorteil sind. Den Grund hierfür können wir aus den Trefferraten ableiten. Im Gegensatz zu unseren bisherigen Experimenten müssen wir jetzt zwischen der Trefferraten bezüglich der Anzahl und der Größe von Zugriffen unterscheiden. Während die Trefferrate bezüglich der Anzahl unserer bisherigen Definition entspricht, können wir

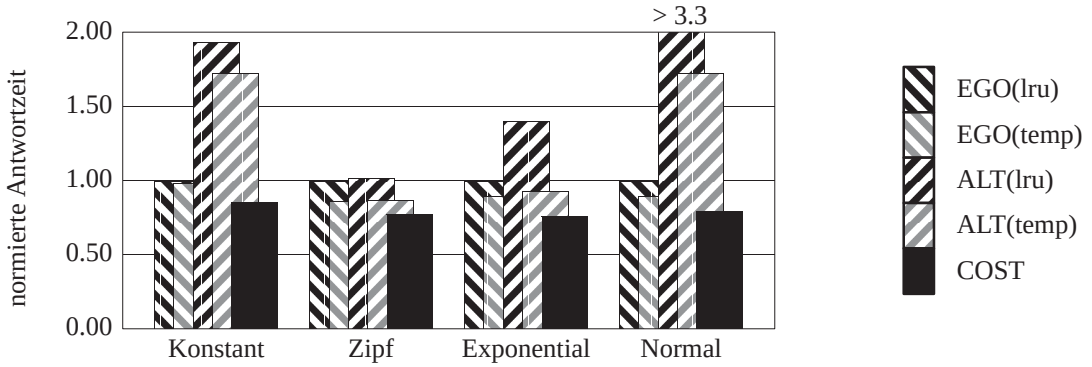


Abbildung 5.19: Einfluß der Objektgrößenverteilung auf die Antwortzeit

die lokale Cache-Trefferrate bezüglich der Größe durch den folgenden Quotienten definieren:

$$\frac{\textit{Summe der im lokalen Cache zugegriffen Bytes}}{\textit{Summe der insgesamt zugegriffenen Bytes}}$$

Analog können wir die entsprechenden Trefferraten für den nichtlokalen Cache und für die Festplatten definieren.

In Abbildung 5.20 sind für die Verteilungen, bei denen sich diese Werte unterscheiden, jeweils oben die Trefferrate bezüglich der Größe und darunter die bezüglich der Anzahl gezeigt. Deutlich erkennbar ist, daß die Anzahl der Cache-Treffer bei allen Verfahren größer ist als die Größe der Cache-Treffer. Daraus können wir ableiten, daß alle Heuristiken jeweils kleine Objekte bevorzugt im Cache halten. Bei den temperaturbasierten Verfahren ist dabei die Differenz in den Trefferraten zwischen der Größe und der Anzahl jeweils noch etwas größer, und dies ist der Grund für die bessere Antwortzeit.

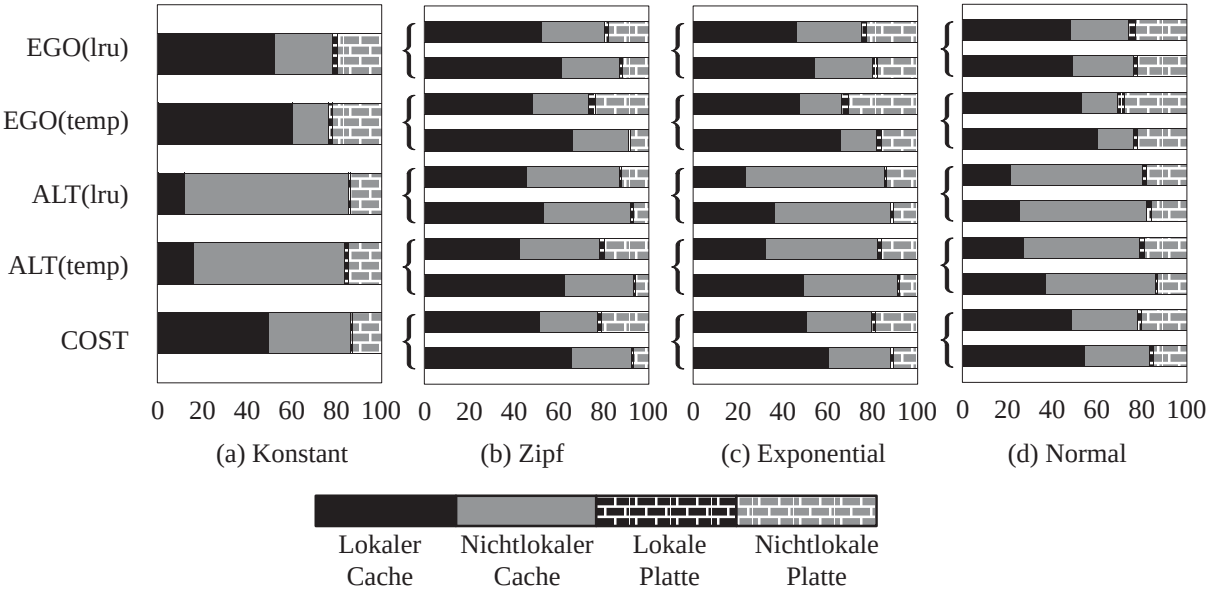


Abbildung 5.20: Trefferraten bezüglich der Größe und der Anzahl auf den verschiedenen Hierarchiestufen

Die Ursache für diese Bevorzugung von kleinen Objekten liegt darin, daß bei den temperaturbasierten Verfahren jeweils die Temperatur (=Hitze/Größe) eines Objekts als Maß für die Wichtig-

keit genommen wird. Dadurch werden kleinere Objekte bei gleicher Hitze wertvoller und somit ist die Wahrscheinlichkeit größer, daß sie im Cache gehalten werden.

Überraschend ist jedoch, daß sich auch bei den LRU-basierten Verfahren die Trefferraten bezüglich Größe und Anzahl unterscheiden. Allein auf Grund der LRU-Strategie müßte man erwarten, daß die mittlere Größe der Objekte im Cache gleich der mittleren Objektgröße über alle Objekte ist. Daher sollten auch die Trefferraten bezüglich Größe und Anzahl gleich sein. Dieses Phänomen kann jedoch durch unseren Cache-Verwaltungsalgorithmus erklärt werden. Wie wir in Abschnitt 4.1.5 gesehen haben, werden solange Opferkandidaten bestimmt, bis ein genügend großer zusammenhängender Bereich im Cache gefunden wird. In diesen Bereich wird das neue Objekt kopiert. Alle Opferkandidaten, die sich im Cache nicht mit dem neuen Objekt überschneiden, können weiterhin im Cache verbleiben. Durch diese Vorgehensweise ist jedoch die Wahrscheinlichkeit, daß ein Objekt als ein Ersetzungsopfer aus dem Cache entfernt wird, von seiner Objektgröße abhängig. Dies erklärt die Unterschiede in den Trefferraten, die wir in der Abbildung 5.20 beobachten können.

Wegen dieser Beeinflussung der Opferauswahl durch die Cache-Verwaltung haben wir die Messungen mit einem *idealen* Cache-Verwaltungsalgorithmus wiederholt. Bei einem solchen Algorithmus kann jeweils – unabhängig von der Fragmentierung des Speichers – ein Objekt in den Cache eingefügt werden, wenn die Summe der freien Bereiche größer oder gleich der Größe des einzufügenden Objekts ist.[10] Die normierten mittleren Antwortzeiten für diese Meßreihe sind in der Abbildung 5.21 skizziert.

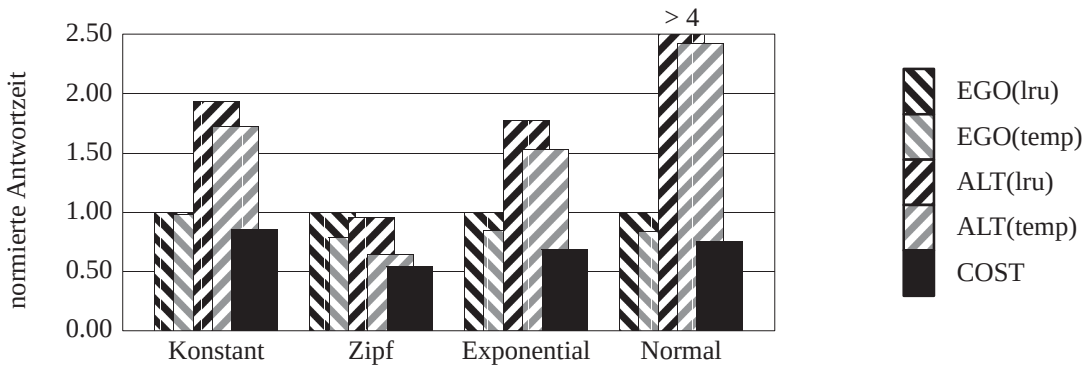


Abbildung 5.21: Antwortzeiten für unterschiedliche Objektgrößenverteilungen bei einem idealen Cache-Verwaltungsalgorithmus

Auch hier erkennen wir, daß die temperaturbasierten Verfahren jeweils ihren LRU-basierten Gegenstücken deutlich überlegen sind. Die Ursache für diese Überlegenheit können wir wiederum an Hand der Trefferraten (Abbildung 5.22) erklären. Analog zu der Abbildung 5.20 haben wir auch hier die Trefferraten bezüglich der Größe und der Anzahl eingezeichnet. In dieser Abbildung erkennen wir nun, daß bei den LRU-basierten Verfahren die Trefferraten bezüglich der Größe und der Anzahl fast identisch sind und somit den oben angesprochenen Erwartungen entsprechen. Ebenfalls entsprechend unseren Erwartungen ist bei den temperaturbasierten Verfahren die Cache-Trefferrate bezüglich der Anzahl sehr viel höher als die bezüglich der Größe. Im Gegen-

10. Wie wir in Abschnitt 4.1.5 gesehen haben, ist der Einsatz dieses idealisierten Verfahrens z.B. dann möglich, wenn das Betriebssystem *virtual DMA* unterstützt.

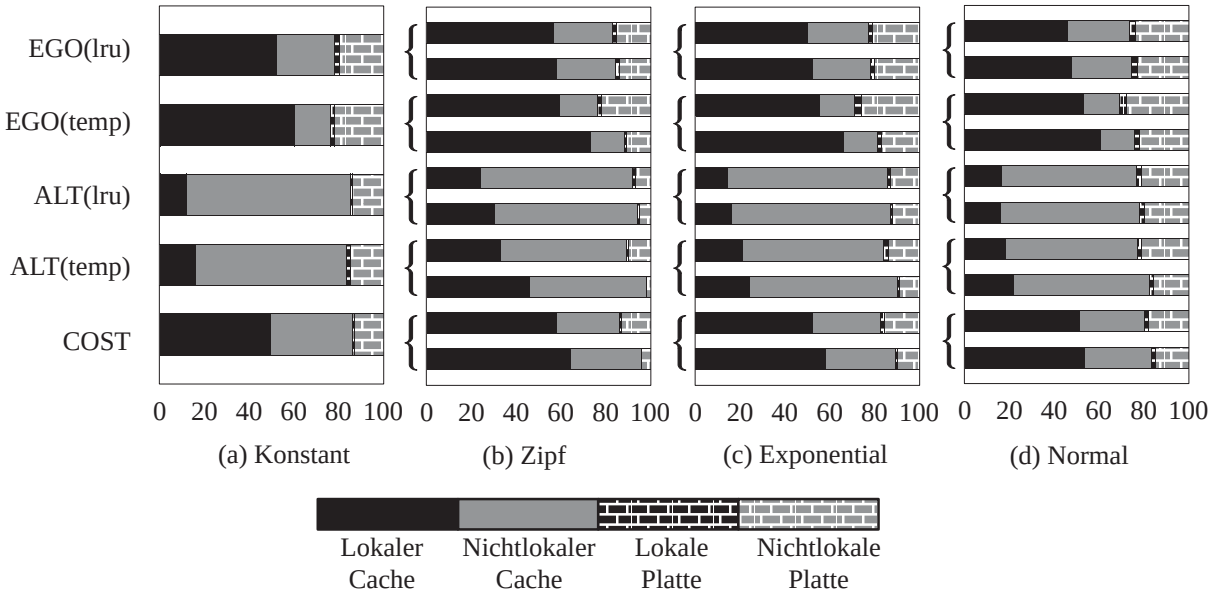


Abbildung 5.22: Trefferraten bezüglich der Größe und der Anzahl auf den verschiedenen Hierarchiestufen

satz zu dem obigen Experiment ist dieser Unterschied jedoch alleinig auf die Berücksichtigung der Größe der Objekte – und die dadurch bedingte Bevorzugung von kleinen Objekten – zurückzuführen. Die höhere Anzahl der Treffer auf die Caches führt zu den geringeren relativen Antwortzeiten, die wir für die temperaturbasierten Verfahren in Abbildung 5.21 beobachten können.

5.3.4 Variation der Datenallokation auf Platte

Bislang haben wir – bis auf die Ausnahme bei dem Experiment mit unterschiedlicher Rechnerlast – immer angenommen, daß die Objekte gemäß einer Round-Robin-Partitionierung auf die Festplatten verteilt sind. Dies gewährleistet, daß sich die Zugriffe sehr gleichmäßig auf die Platten verteilen und somit die Auslastungen aller Platten im Gesamtsystem annähernd gleich sind. In diesem Experiment wollen wir nun untersuchen, wie die Cache-Heuristiken auf eine ungleichmäßige Plattenallokation reagieren. Dazu verteilen wir die Objekte entsprechend einer Zipf-Verteilung zufällig über die Platten und variieren die Schräge der Plattenverteilung von ψ=0 bis ψ=1.5. Die Bestimmung der Hitze der Objekte entsprechend des Zugriffsmusters erfolgt unabhängig von der Allokation der Objekte auf der Festplatte. Die ermittelten Antwortzeiten für unterschiedlich schräge Plattenallokationen sind in Abbildung 5.23 gezeigt.

In dieser Abbildung fällt zuerst auf, daß die Antwortzeiten der egoistischen Heuristiken ab einer gewissen Schräge sehr schnell ansteigen. Der Grund hierfür liegt in dem Anwachsen der Auslastung der Platte, die die meisten Objekte speichert. Das etwas frühere Ansteigen der Antwortzeit für EGO(temp) verglichen mit EGO(lru) resultiert daraus, daß EGO(temp) das egoistische Verhalten besser approximiert und dadurch die Anzahl der Replikate im aggregierten Hauptspeicher erhöht. Durch die größere Anzahl an Replikaten reduziert sich die Cache-Trefferrate, und dies bewirkt eine weitere Erhöhung der Plattenauslastung, so daß die Vollauslastung der Engpaßplatte früher erreicht wird.

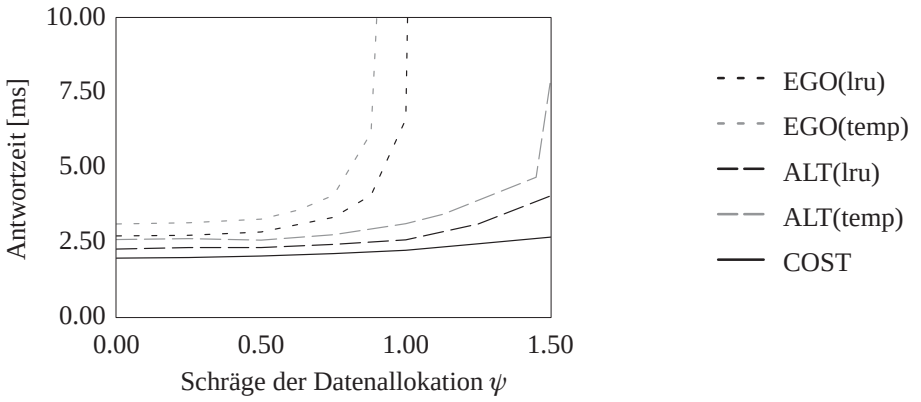


Abbildung 5.23: Variation der Datenallokation auf Platte

Die bessere Performance der altruistischen Methoden resultiert daraus, daß diese Verfahren versuchen, die globale Cache-Trefferrate zu maximieren. Die hohe Cache-Trefferrate führt dazu, daß die Platten insgesamt – und damit insbesondere auch die Engpaßplatte – auf Kosten des Netzwerks entlastet werden. Da jedoch die altruistischen Methoden, genauso wie die egoistischen Verfahren, die Kosten für die Beschaffung eines Objekts nicht in dem Objektwert berücksichtigen, führen auch diese Verfahren ab einer gewissen Schräge zu einer Überlastung der Engpaßplatte. Dies geschieht selbst dann, wenn die mittlere Auslastung über alle Platten noch sehr gering ist. So ist z.B. bei einer Schräge der Allokation von ψ=1.5 die Auslastung der Engpaßplatte für ALT(lru) gleich 83% (bzw. 88% bei ALT(temp)), während die mittlere Auslastung über alle Platten bei beiden Verfahren nur 16% beträgt.

Im Gegensatz zu den einfachen Verfahren berücksichtigt unsere kostenbasierte Heuristik die Beschaffungskosten. Dies können wir sehr gut an der unterschiedlichen Anzahl der verschiedenen Objekte im aggregierten Cache erkennen, die in der Abbildung 5.24 gezeigt sind. Jeder Rechner besitzt einen Cache für maximal 512 Objekte. Da das System aus 10 Rechnern besteht, können somit maximal 5120 verschiedene Objekte im Cache gehalten werden. In der Abbildung sehen wir, daß die altruistischen Methoden dieses Maximum auch annähernd erreichen. Die kostenbasierte Heuristik hält dageben insgesamt weniger verschiedene Objekte im aggregierten Cache, da die heißesten Objekte repliziert gespeichert werden. Zusätzlich zu der Gesamtzahl der im Cache gehaltenen Objekte ist in Abbildung 5.24 auch der Anteil der Objekte von den verschiedenen Platten gezeigt. Die Anteile sind dabei so eingezeichnet, daß von links nach rechts die Auslastung der Platten abnimmt. Es ist deutlich zu erkennen, daß die Anzahl der im Cache gehaltenen Objek-

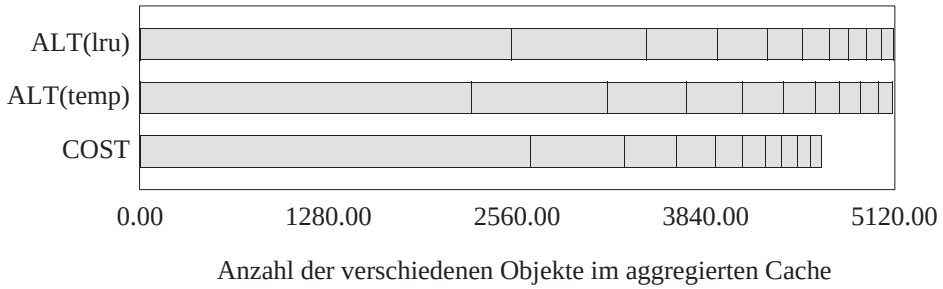


Abbildung 5.24: Anzahl der im Cache gehaltenen Objekte von den verschiedenen Platten für eine Schräge der Plattenallokation von ψ=1.5

te für die verschiedenen Platten unterschiedlich ist. Dies hängt damit zusammen, daß sich auf den stark ausgelasteten Platten mehr Daten befinden. Damit steigt die Wahrscheinlichkeit, daß auf ein Objekt dieser Platten zugegriffen wird. Bemerkenswert ist jedoch, daß unsere kostenbasierte Heuristik die mit Abstand meisten Objekte der Engpaßplatte im aggregierten Cache hält, obwohl sie insgesamt weniger verschiedene Objekte im Cache hält als die anderen Heuristiken. Dies ist dadurch zu erklären, daß COST die Beschaffungskosten der einzelnen Objekte berücksichtigt und dadurch erhält ein Objekt, das von einer stark belasteten Platte stammt, einen höheren Wert. Dies führt dazu, daß die Differenz zwischen der am stärksten und der am geringsten belasteten Platte reduziert wird, und somit eine Balancierung zwischen den verschiedenen im System vorhanden Platten erreicht wird. Durch diese Balancierung erreicht unsere kostenbasierte Heuristik eine nahezu gleichbleibende Antwortzeit, unabhängig von der Schräge der Plattenallokation.

5.4 Variation der Systemparameter

Nachdem wir in den vorangegangenen Abschnitten die Last- und Objektparameter variiert haben, wollen wir nun den Einfluß der Rechneranzahl und der Netzgeschwindigkeit auf die verschiedenen Caching-Heuristiken untersuchen.

5.4.1 Variation der Netzgeschwindigkeit

Im ersten Experiment variieren wir – analog zu der entsprechenden Messung mit den statischen Zugriffskosten in Abschnitt 2.4.2 – die Netzgeschwindigkeit von der traditionellen, wenngleich heute veralteten *Ethernet*-Geschwindigkeit (10 MBit/s), bis zur Geschwindigkeit von *ATM*-Netzwerken (160 MBit/s). Auch hier haben wir wiederum die Ankunftszeit entsprechend der Methode aus Abschnitt 5.1.4 bestimmt. Die von den Heuristiken erzielten und auf die egoistische Heuristik normierten Antwortzeiten sind in der Abbildung 5.25 aufgezeigt.

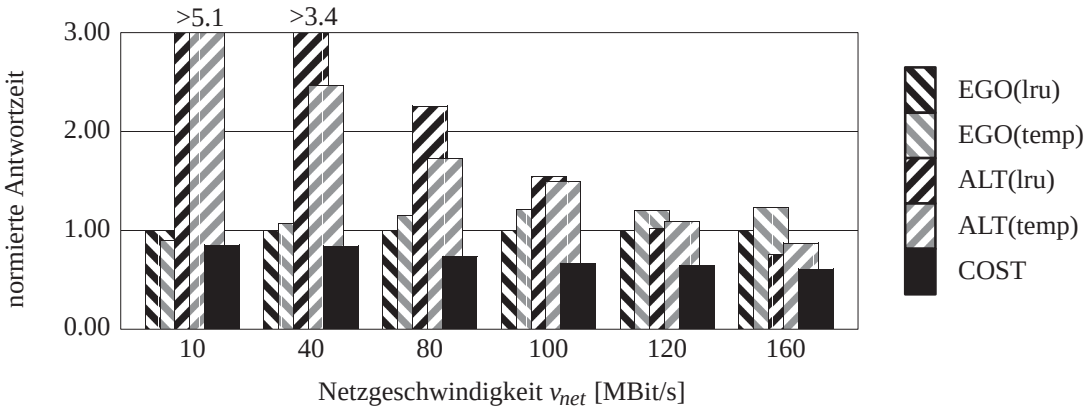


Abbildung 5.25: Einfluß der Netzgeschwindigkeit auf die Antwortzeit

In dieser Abbildung ist deutlich zu erkennen, daß bei einem langsamen Netzwerk die altruistischen Verfahren sehr viel schlechter abschneiden als die egoistischen Methoden. Dies liegt an der Maximierung der globalen Cache-Trefferrate, die bei altruistischen Verfahren ohne Berücksichtigung der lokalen Trefferrate erfolgt. Dies führt dazu, daß selbst sehr heiße Objekte nicht repliziert im Cache gehalten werden. Somit müssen sehr viele Zugriffe über das Netzwerk erfolgen

und dadurch steigt die Netzwerkauslastung (92% bei ALT(lru) und 87% bei ALT (temp)). Der bei dieser Netzgeschwindigkeit ohnehin schon geringe Unterschied zwischen nichtlokalen Cache-Zugriffen und lokalen Plattenzugriffen nimmt dadurch weiter ab. Ähnlich wie im Experiment mit sehr großen Objekten kann dies im Extremfall dazu führen, daß bei den altruistischen Verfahren die Kosten für einen Plattenzugriff geringer als für einen nichtlokalen Cache-Zugriff werden.

Diese Umkehrung der Hierarchiestufen wird auch hier von unserer kostenbasierten Heuristik erkannt und durch eine Replikation der heißesten Objekte verhindert. Durch diese Replikation verringert sich bei COST die Auslastung des Netzwerks entscheidend (38%) und dadurch bleiben die mittleren Kosten für einen nichtlokalen Cache-Zugriff unter den Kosten für das Lesen von Platte. Dies erklärt die wesentliche Verbesserung gegenüber der altruistischen Verfahren. Wir können jedoch auch erkennen, daß unsere Heuristik – verglichen mit den egoistischen Verfahren – eine bessere Performance liefert. Dies kommt daher, daß eine Reduktion der Netzbelastung nur soweit durchgeführt wird, wie dies für die Gesamt-Performance sinnvoll ist. Während die egoistischen Methoden die lokale Trefferrate unabhängig von der Plattenauslastung maximieren, wägt unser Verfahren zwischen der Platten- und der Netzlast ab.

Mit größer werdender Netzgeschwindigkeit wird die Verzögerungszeit einer Nachricht immer kleiner. Gleichzeitig vergrößert sich die Bandbreite für Übertragungen, und dadurch verringert sich die Auslastung.[11] Beide Phänomene führen dazu, daß nun die Kosten für einen nichtlokalen Cache-Zugriff sehr viel geringer als für einen Plattenzugriff sind. Somit ist es in diesem Fall günstiger, die globale Cache-Trefferrate zu maximieren. Dies führt – im Vergleich zu den egoistischen Heuristiken – zu der Verbesserung der relativen Antwortzeit der altruistischen Verfahren, die wir in Abbildung 5.25 beobachten können. Aber selbst bei dem sehr schnellen Netzwerk mit 160 MBit/s zeigt es sich, daß ein völliges Vermeiden von Replikationen, und die daraus resultierende Vernachlässigung der lokalen Cache-Trefferrate, nur zu einem suboptimalen Ergebnis führt. So kann unsere kostenbasierte Heuristik durch eine gezielte Replikation der wichtigsten Objekte deutliche Vorteile gegenüber den altruistischen Heuristiken erzielen.

Die Netzgeschwindigkeit wird von unserer Heuristik in der Berechnung des Nutzens berücksichtigt und wirkt sich dadurch in dem Caching-Verhalten aus. Dies erkennen wir sehr gut in der Abbildung 5.26. Während die Cache-Trefferraten der egoistischen und altruistischen Heuristiken nahezu unabhängig von der Netzgeschwindigkeit sind und daher nur einmal in der Abbildung aufgeführt sind, müssen wir bei unserer kostenbasierten Heuristik zwischen der Trefferrate bei *Ethernet*- und *ATM*-Geschwindigkeit unterscheiden. Bei einer geringeren Netzwerkgeschwindigkeit agiert COST stärker egoistisch, indem es die lokale Cache-Trefferrate auf Kosten der globalen Trefferrate erhöht. Dies ist vorteilhaft, da hierdurch das Netzwerk, das in dieser Situation den Engpaß darstellt, entlastet wird. Bei einem schnellen Netzwerk jedoch ist es günstiger, die Platten zu entlasten. Um dies zu erreichen, reduziert COST den Replikationsgrad und erhöht dadurch die globale Cache-Trefferrate.

---

11. Hier sei nochmals darauf hingewiesen, daß sich der Gewinn durch ein schnelleres Netzwerk nur dann auszahlt, wenn wir ein optimiertes Netzwerkprotokoll benutzen. Ansonsten kann es vorkommen, daß die Protokollabarbeitung den Engpaß bei der Übertragung darstellt und sich daher eine reine Beschleunigung der Transferrate auf dem Netzwerk nicht in einer merklichen Verringerung der Übermittlungszeit auswirkt.

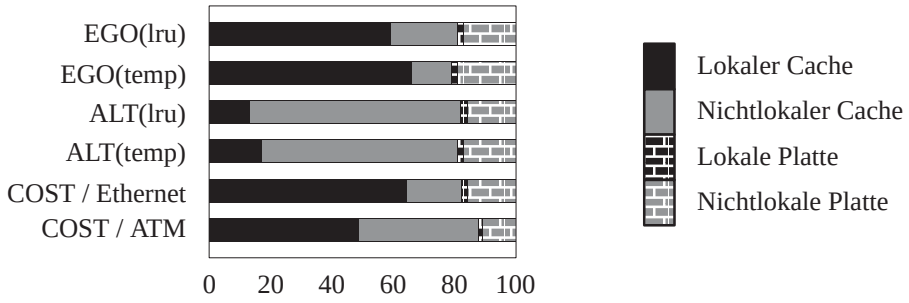


Abbildung 5.26: Trefferraten auf den verschiedenen Hierarchiestufen

5.4.2 Skalierbarkeit

In diesem Experiment untersuchen wir die Skalierbarkeit der verschiedenen Heuristiken. Dazu erhöhen wir die Rechneranzahl und die Größe des Datenbestandes jeweils proportional zueinander. Zusammen mit der Erhöhung der Rechneranzahl führen wir auch eine Erhöhung der Last durch, da wir auf jedem Rechner Zugriffe mit der gleichen mittleren Ankunftsrate generieren. In Abbildung 5.27 haben wir die Antwortzeitkurven für die verschiedenen Heuristiken für Systemgrößen von 5 bis 30 Rechner skizziert.

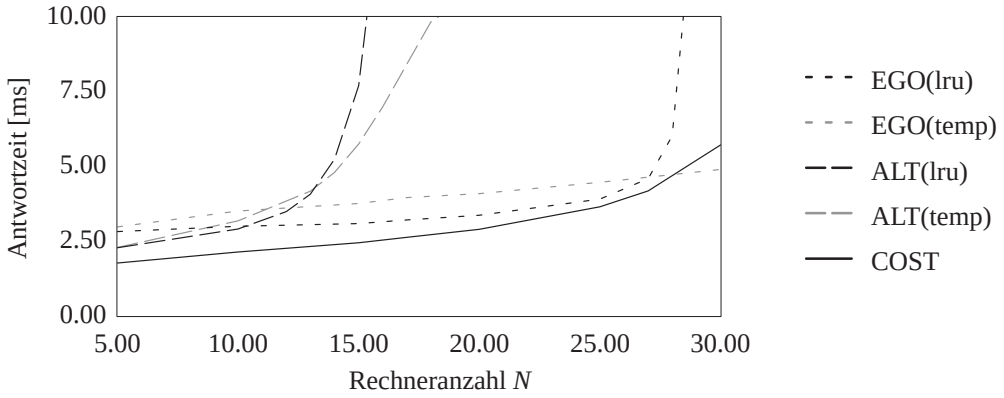
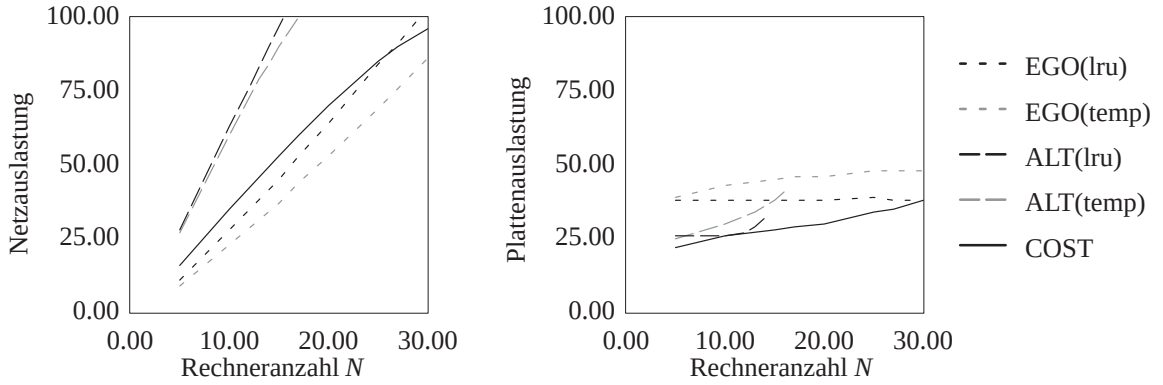


Abbildung 5.27: Skalierbarkeit der Heuristiken

Bei geringer Rechneranzahl liefert unsere kostenbasierte Heuristik die beste Performance, gefolgt von den altruistischen und den egoistischen Heuristiken. Erhöhen wir jedoch die Rechneranzahl, so erkennen wir, daß die Antwortzeiten bei den altruistischen Verfahren sehr stark ansteigen, während bei den anderen Verfahren der Anstieg der Antwortzeiten sehr viel geringer ist. Erklären können wir dieses Verhalten an Hand der Ressourcenauslastungen, die wir in Abbildung 5.28 dargestellt haben. Hier erkennen wir, daß die starken Netzwerkauslastungen der Grund für die schlechte Antwortzeiten der altruistischen Methoden sind. So ist das Netzwerk bei diesen Methoden bereits ab 14 bzw. 15 Rechnern voll ausgelastet. Dadurch wächst die Warteschlange am Netzwerk ständig an, und das System erreicht keinen stabilen Zustand. Die Ursache für diese hohe Auslastung ist das Bus-Netzwerk, das nicht mit der Rechneranzahl skaliert. Durch eine Erhöhung der Rechneranzahl steigt jedoch die Anzahl der Zugriffe, die über das Netzwerk befriedigt werden müssen. In der Abbildung erkennen wir zwar auch, daß die Auslastung des Netzwerks bei den egoistischen Heuristiken ebenfalls linear ansteigt, jedoch ist der Gradient dieses Anstiegs we-

sentlich geringer. Bei einer weiteren Erhöhung der Rechneranzahl ist daher zu erwarten, daß auch bei diesen Verfahren das Netzwerk zu einem Engpaß wird. Bemerkenswert ist auch die nicht linear verlaufende Netzauslastung bei der kostenbasierten Heuristik. So versucht unsere Heuristik – bei sehr hoher Netzlast – die Netzauslastung auf Kosten der Plattenlast zu reduzieren. Diese Reduktion der Netzlast ist daher auch die Ursache für die bessere Antwortzeit von COST verglichen mit EGO(lru), die wir in Abbildung 5.27 beobachten konnten.



(a) Netzwerkauslastung (b) Festplattenauslastung

Abbildung 5.28: Ressourcenauslastungen

In Abbildung 5.28(b) erkennen wir ein Anwachsen der Festplattenauslastung mit wachsender Rechneranzahl. Dies scheint verwunderlich, da gleichzeitig mit der Erhöhung der Rechneranzahl auch die Anzahl der Platten wächst. Betrachten wir zunächst die egoistischen Verfahren. Bei einem exakt-egoistischen Verfahren ist die Cache-Trefferrate nur abhängig von der maximalen lokalen Cache-Größe aller Rechner und der Gesamtgröße der Daten, jedoch unabhängig von der Rechneranzahl. Erhöhen wir nun die Rechneranzahl, so wächst die Objektanzahl und damit auch die Datengröße; die Cache-Größe jedes einzelnen Rechners bleibt jedoch gleich. Dadurch verringert sich die Cache-Trefferrate, und die Anzahl der Zugriffe auf Platte erhöht sich. Dieses Phänomen ist um so ausgeprägter, je besser ein exakt-egoistisches Verhalten approximiert wird. Daher erkennen wir in Abbildung 5.28(b) diesen Anstieg bei EGO(temp) auch sehr viel deutlicher als bei EGO(lru). Derselbe Grund bewirkt auch den Anstieg der Plattenauslastung bei unserer kostenbasierten Heuristik. Da jedoch nur ein Teil der Objekte repliziert in den Caches gehalten wird, ist der Anstieg bei geringer Rechneranzahl nicht so steil wie bei EGO(temp). Bei einer höheren Rechneranzahl wächst jedoch die Plattenauslastung von COST sehr viel schneller an, um – wie bereits oben erwähnt – eine Überlastung des Netzwerks zu verhindern.

Im Gegensatz zu den egoistischen Heuristiken ist die Cache-Trefferrate der alturistischen Verfahren von der Gesamtgröße des verteilten Caches und von der Datenmenge abhängig. Da die Größe des verteilten Caches – und damit die Anzahl der im Cache gehaltenen Objekte – jedoch linear mit der Rechneranzahl anwächst, wäre zu erwarten, daß die Cache-Trefferrate gleich bleiben würde. In unseren Experimenten konnten wir auch beobachten, daß der Anteil der Daten, die im aggregierten Cache gehalten werden, konstant 50% ist, jedoch nimmt die globale Cache-Trefferrate mit steigender Rechneranzahl ab. Eine Erklärung hierfür ist die bereits in Abschnitt 5.2 angesprochene zeitweilige Ungenauigkeit der Buchhaltungsinformation und das damit verbundene ungewollte Löschen von “wertvollen” Unikaten bei den altruistischen Methoden. Mit wachsender

Rechneranzahl nimmt die Netzauslastung zu, und daher brauchen die Nachrichten zur Aktualisierung der Buchhaltungsinformation immer länger. Dies führt dazu, daß die Zeiträume, in denen das System falsche Informationen besitzt, immer größer werden. Das Problem des ungewollten Löschens von Unikaten tritt dadurch immer stärker auf, und dies reduziert die aggregierte Hitze und damit die Cache-Trefferrate.

Neben den reinen Bus-Netzwerken, wie wir sie in unserer Simulation betrachten, existieren auch Weiterentwicklungen, mit denen sich eine bessere Skalierbarkeit erreichen läßt. Bei dieser Technologie werden die Rechner durch einen *Switch* [Tran97] miteinander verbunden. Ein Switch gewährleistet durch eine entsprechende Hardware, daß mehrere Übertragungen gleichzeitig durchgeführt werden können, solange die an den Übertragungen beteiligten Rechner unterschiedlich sind. Zwar verschiebt diese Technologie den Netzengpaß, an der prinzipiellen Möglichkeit eines überlasteten Netzwerks ändert sie jedoch nichts. Eine umfangreichere Diskusion einer Erweiterung auf solche Netzwerktechnologien erfolgt in Kapitel 7.

5.5 Dynamische Variation der Last

Abschließend untersuchen die Adaptivität der vorgestellten Heuristiken auf dynamische Laständerungen. Dazu haben wir die Simulation in drei Phasen aufgeteilt und die Lastparameter für die einzelnen Phasen folgendermaßen gewählt:

- In der ersten Phase haben alle Rechner das gleiche Zugriffsmuster. Die Parameter dieser Phase werden entsprechend der in Abschnitt 5.1.3 beschriebenen Standardwerte gewählt.
- Zu Beginn der zweiten Phase führen wir eine zyklische Verschiebung der lokalen Hitzen entsprechend der Methode aus Abschnitt 4.3.1 durch. Da bei dieser Methode die Größe der Verschiebung von dem Index des betreffenden Rechners abhängt, sind nach der Verschiebung die heißen Bereiche der Rechner unterschiedlich. Um eine maximale Differenz zwischen den heißen Bereichen zu erhalten, haben wir als Betrag der Verschiebung $\sigma=M/N$ gewählt, wobei M die Anzahl der Objekte und N die Rechneranzahl ist.
- In der dritten Phase heben wir die in der vorangegangenen Phase durchgeführte zyklische Verschiebung wieder auf. Dies bedeutet, daß nun alle Rechner wieder das gleiche Zugriffsmuster besitzen.

In Abbildung 5.29 ist der Verlauf der mittleren Antwortzeiten für die verschiedenen Caching-Heuristiken über die Zeit aufgetragen.

Beim Übergang von Phase 1 nach Phase 2 erkennen wir bei allen Heuristiken eine sprunghafte Verschlechterung der mittleren Antwortzeit. Durch die Verschiebung des Zugriffsmusters befinden sich nur noch “fast wertlose” Objekte im Cache und daher sinkt die Trefferrate plötzlich sehr stark ab. Die dadurch verursachten Plattenzugriffe führen zu langen Warteschlangen und damit zu den schlechten Antwortzeiten, die wir in der Abbildung erkennen können. Die Heuristiken, die am schnellsten auf diese Laständerung reagieren, sind die beiden LRU-basierten Heuristiken ALT(lru) und EGO(lru). Beide Verfahren profitieren davon, daß LRU durch seine auf minimaler statistischer Information beruhende Rangfolge der Objekte sehr schnell auf Laständerungen reagieren kann. Ein weiterer Vorteil von ALT(lru) besteht darin, daß die Leistungsfähigkeit dieser

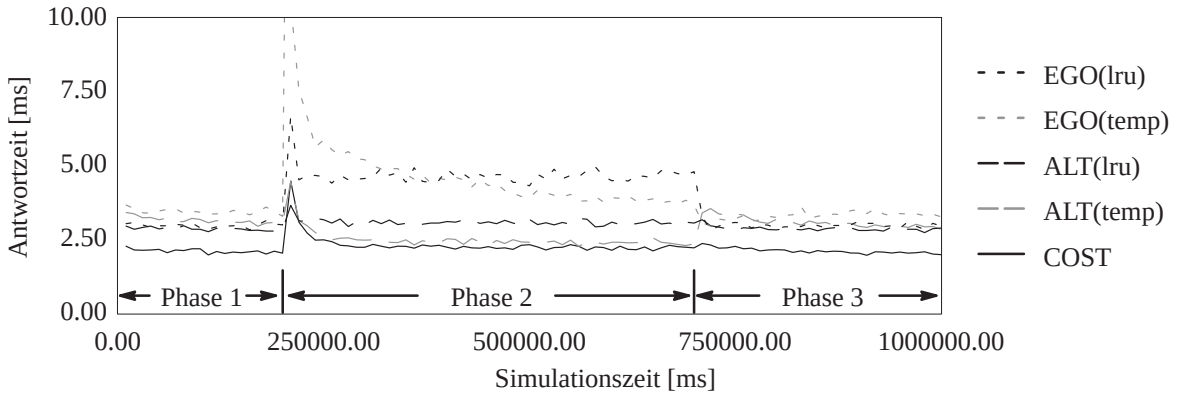


Abbildung 5.29: Einfluß von dynamischen Änderungen der Last auf die Antwortzeit

Heuristik hauptsächlich durch die Statusinformation der Objekte bestimmt wird und diese Information durch die Lastverschiebung nicht verändert wird. Anders verhält es sich bei den Heuristiken, die die lokale oder globale Temperatur benötigen. Durch die Verschiebung der Last entspricht die gesammelte Hitzeinformation nicht mehr dem aktuellen Systemzustand. Daher basieren die Entscheidungen dieser Heuristiken über einen gewissen Zeitraum auf falschen Informationen. Dies ist deutlich an dem sehr hohen Anstieg und dem langsamen Abfallen von EGO(temp) nach dem Lastwechsel zu erkennen. Die beiden anderen Verfahren (ALT(temp) und COST), die ebenfalls die Temperatur benötigen, adaptieren sehr viel schneller auf die Laständerung. Der Grund hierfür liegt wiederum in der zusätzlichen Berücksichtigung des Objektstatus. Wie wir bereits gesagt haben, wird der Status durch die Laständerung nicht beeinflußt. Daher ist dieser Teil der Information korrekt und führt zu einer genaueren Abschätzung des Objektwertes.

Am Ende der zweiten Phase haben sich alle Heuristiken bis auf EGO(temp) wieder eingependelt. Die Rangfolge der Verfahren hat sich jedoch – verglichen mit der Situation in der Phase 1 – geändert. Dies kommt daher, daß in der zweiten Phase das Zugriffsmuster auf den verschiedenen Rechnern unterschiedlich ist. Wie wir bereits in Abschnitt 5.2 bei der Variation der Zugriffsähnlichkeit gesehen haben, ist es bei sich nicht überschneidenden heißen Bereichen am besten, wenn jeder Rechner seine lokale Cache-Trefferrate maximiert. Da die temperaturbasierten Verfahren umfangreichere Informationen zur Bestimmung der lokalen Wichtigkeit besitzen, können diese Verfahren die lokal heißen Objekte besser bestimmen und führen daher zu einer besseren Performance. In Abbildung 5.29 können wir dies für ALT(temp) und COST erkennen. Zwar liefert EGO(temp) selbst am Ende von Phase 2 immer noch eine schlechtere Antwortzeit als ALT(lru), jedoch erkennen wir in Abbildung 5.29, daß die Antwortzeit immer noch sinkt und EGO(temp) daher noch keinen stationären Zustand erreicht hat. Die bessere Performance von ALT(lru) – verglichen mit EGO(lru) – kommt daher, daß in dieser speziellen Situation zwar beide in gleichem Maße die lokale Trefferrate maximieren, daß jedoch ALT(lru) durch die explizite Vermeidung der Replikation zusätzlich eine etwas bessere globale Trefferrate erzielt.

Beim Übergang von der zweiten zur dritten Phase ist kein so starker Ausschlag in den Antwortzeiten – wie bei dem ersten Lastwechsel – zu beobachten. Zwar ändern sich auch bei diesem Übergang die heißen Bereiche und damit sinken kurzzeitig die lokalen Cache-Trefferraten der verschiedenen Rechner, jedoch ist es nun nicht nötig, die nach der Änderung heißen Objekte zuerst von Platte zu laden. Dies kommt daher, daß bei dem Übergang zur Phase drei alle Rechner

wieder das Zugriffsmuster von Rechner 1 aus Phase zwei übernehmen. Da somit Rechner 1 schon die global heißesten Objekte im lokalen Cache besitzt, können die anderen Rechner diese Objekte aus diesem für sie nichtlokalen Cache lesen. Durch die Überschneidung der heißen Bereiche, die in Phase 3 wieder auftritt, verschlechtert sich auch wieder die Antwortzeit von ALT(temp). Durch die explizite Vermeidung von Replikaten müssen nun nämlich selbst die lokal sehr heißen Objekte wieder über das Netzwerk gelesen werden. Die Verbesserungen der egoistischen Verfahren kommen daher, daß nun wieder die verschiedenen Rechner von den im nichtlokalen Cache gehaltenen Objekten profitieren können. Dies korrespondiert mit den Beobachtungen, die wir bei der Variation des Zugriffsmusters machen konnten. Die Antwortzeiten unserer kostenbasierten Heuristik und von ALT(lru) bleiben fast unverändert durch den Lastwechsel, wobei jedoch COST insgesamt eine sehr viel bessere Performance liefert.

5.6 Zusammenfassung und Fazit

In diesem Abschnitt wollen wir die Beobachtungen aus den durchgeführten Experimenten noch einmal zusammenfassen. Eine wichtiger Punkt betrifft dabei die Metrik des Nutzens. In den verschiedenen Experimenten konnten wir erkennen, daß unsere kostenbasierte Heuristik durch die Bestimmung des Nutzens als Maß für den Objektwert, eine einheitliche Betrachtung der folgenden Probleme erlaubt:

- **Replikationskontrolle:** Im Gegensatz zu den egoistischen und altruistischen Methoden, die ihre Caching-Strategie unabhängig von den aktuellen Parametern verfolgen, ist unsere kostenbasierte Heuristik in der Lage, den Replikationsgrad separat für jedes einzelne Objekt entsprechend der aktuellen Last zu bestimmen. Dies ermöglicht eine Balancierung zwischen der Platten- und Netzauslastung. Sehr heiße Objekte werden repliziert gespeichert. Da hierdurch die lokale Cache-Trefferrate erhöht wird, reduziert sich die Netzlast. Auf der anderen Seite werden nicht ganz so heiße Objekte nur einfach im aggregierten Cache gehalten. Durch die höhere Anzahl von Objekten, die dadurch im Cache gehalten werden können, steigt die globale Cache-Trefferrate an und reduziert die Plattenlast im System. In den Experimenten haben wir gesehen, daß die Grenze zwischen den replizierten und den einfach im Cache gehaltenen Objekten, von den aktuellen Last-, Daten- und Systemparametern abhängig ist. Da unsere Heuristik online die Kosten für Zugriffe auf die verschiedenen Stufen der Speicherhierarchie mitprotokolliert, ist sie in der Lage, eine – für die aktuelle Last – geeignete Grenze für die Replikation dynamisch abzuleiten.
- **Ausnutzung von freiem Speicher:** In dem Experiment mit heterogener Last hat sich gezeigt, daß sich der im lokalen Cache eines Rechners aggregierte Nutzen auch als Maß für die Charakterisierung der Last dieses Rechners eignet und somit als Grundlage für die Steuerung der Migration von Objekten benutzt werden kann. Dies erlaubt uns vor einer Migration zu überprüfen, ob ein Ungleichgewicht im System existiert. Dadurch vermeiden wir eine Vielzahl von unsinnigen Migrationen. Zusätzlich können Migrationen zielgerichtet durchgeführt werden, da wir auf Grund des jeweils lokal aggregierten Nutzens entscheiden können, ob ein Rechner eine Über- bzw. eine Unterlast besitzt. Weiterhin berücksichtigen wir den Nutzen eines einzelnen Objekts, um auf dem Zielrechner zu entscheiden, ob dieses spezielle Objekt so wertvoll ist, daß es in den Cache eingefügt werden soll. So kann verhindert

werden, daß relativ wertlose Objekte nur dadurch, daß sie von einem überlasteten Rechner migriert wurden, auf einem anderen Rechner eingefügt werden.

- **Berücksichtigung variabel großer Objekte:** Da wir jeweils den Nutzen pro Objektgröße als Maß für den Wert betrachten, berücksichtigen wir auch die Speicherkosten, die durch das Caching des betreffenden Objekts verursacht werden. Dadurch werden kleinere Objekte bei gleicher Hitze bevorzugt im Cache gehalten und erhöhen dadurch die Anzahl der Cache-Treffer.
- **Unabhängigkeit von der gewählten Plattenallokation:** Da bei der Berechnung des Nutzens auch die Auslastung der Festplatte, auf der das entsprechende Objekt liegt, mit berücksichtigt wird, ist unsere kostenbasierte Heuristik unabhängig von der gewählten Plattenallokation. In dem Experiment mit variabler Plattenallokation haben wir gesehen, daß durch die Berücksichtigung der Plattenauslastung unsere kostenbasierte Heuristik sogar eine Balancierung zwischen den Platten erreicht. So erhalten die Objekte, die auf einer sehr stark ausgelasteten Platte liegen, einen höheren Nutzen als gleichheiße Objekte auf einer nicht so stark ausgelasteten Platte. Diese Objekte werden daher mit einer größeren Wahrscheinlichkeit im Cache gehalten.

Neben diesen Eigenschaften, die alle direkt aus der Betrachtung des Nutzens resultieren, konnten wir in den Experimenten einige weitere Beobachtungen durchführen, die wir im folgenden noch einmal herausstellen wollen:

- **Geringer Overhead für CPU und Netzwerk:** Wichtig für den sinnvollen Einsatz einer Heuristik ist, daß der verursachte Zusatzaufwand nicht größer ist, als der Gewinn, den man erzielt. Daher haben wir in den Experimenten sowohl den CPU- als auch den Netzwerk-Overhead unseres Verfahrens untersucht. Diese Beobachtungen haben gezeigt, daß der CPU-Overhead, der durch die Berechnung der Hitze und des Nutzens verursacht wird, sehr gering ist. Da darüberhinaus zu erwarten ist, daß die Geschwindigkeiten der CPUs weiterhin schnell wachsen ([HePa96]), fällt dieser Overhead in Zukunft noch weniger ins Gewicht. Auch der Netzwerk-Overhead, der aus der Berechnung der globalen Hitze und der Bestimmung des Objektstatus resultiert, ist – verglichen mit der Auslastung des Netzwerks durch den Datentransfer – vernachlässigbar. In unseren Experimenten konnten wir außerdem zeigen, daß unsere Heuristik sehr unempfindlich gegenüber approximierter Information ist.
- **Dynamische Adaptivität auf Laständerungen:** Die gute Adaptivität unseres Verfahrens konnten wir innerhalb eines Experiments mit einer dynamischen Last beobachten. Durch die Berücksichtigung der Hitze adaptiert unser Verfahren zwar langsamer als die Verfahren, die nur auf LRU basieren, jedoch immer noch hinreichend schnell. Hier besteht auch ein Zusammenhang zwischen der Adaptivität und dem Overhead der Heuristik. Wenn wir das Zeitraster des Hitzedämons (siehe Abbschnitt 3.1.1) kleiner wählen, werden Hitzeänderungen schneller bemerkt, und es wird eine schnellere Adaption erreicht. Gleichzeitig wächst jedoch der CPU-Overhead. In unseren Experimenten konnten wir aber beobachten, daß selbst bei einer sehr starken und plötzlichen Änderung im Zugriffsverhalten, bereits mit einem sehr geringen Overhead, eine gute Adaptivität erreicht werden kann. Dies ist möglich, da neben der Hitze auch der Status bei der Berechnung des Nutzens mitberücksichtigt wird.

Dieser ist von einer Änderung des Zugriffsverhaltens unabhängig. Die optimale Performance erzielt unsere Heuristik jedoch erst, wenn sich auch die Hitzen der Objekte an die neue Last angepaßt haben.

- **Zusammenwirken mit Kohärenzprotokoll:** Wichtig für den Einsatz einer Caching-Heuristik ist auch, daß das Vorhandensein von Schreibzugriffen die Cache-Strategie nicht zu stark beeinflußt. In unseren Experimenten haben wir gesehen, daß das Kohärenzprotokoll durch das Löschen von mehrfach im Cache vorhandenen Objekten der Replikationskontrolle unserer kostenbasierten Heuristik entgegenwirkt. Dieser Einfluß der Schreibzugriffe wirkt sich um so stärker aus, je größer die Schreibwahrscheinlichkeit wird. Wichtig ist jedoch, daß sich die Performance unserer Caching-Heuristik dadurch nicht abrupt verschlechtert. So erzielt unsere Heuristik für Lesezugriffe über den ganzen Bereich die beste Antwortzeit; auch bei den Schreibzugriffen liefert COST zusammen mit ALT(temp) die beste Performance.

Fassen wir diese einzelnen Punkte zusammen, so stellt sich unsere kostenbasierte Heuristik als der klare Gewinner dieser experimentellen Studie dar. Dies bestätigt unsere Überlegungen und Ziele bei dem Entwurf einer adaptiven Online-Heuristik für verteiltes Caching. Über alle Experimente betrachtet liefert die kostenbasierte Heuristik nie eine spürbar schlechtere Performance als die in der betreffenden Situation beste einfache Heurstik; jedoch ist sie häufig entscheidend besser als alle anderen Heuristiken. Eine leichte Verschlechterung tritt nur in sehr extremen und daher vielfach auch unrealistischen Situationen auf, z.B. wenn der komplette Datenbestand in den aggregierten Cache paßt. Zwar verhält sich unsere Heuristik in solchen Situationen fast wie die in dieser Situation beste einfache Heuristik, durch ein nicht ganz so aggresives Verfolgen des entsprechenden Optimierungszieles und den etwas höheren Overhead erzielt die kostenbasierte Heuristik in diesen Extremfällen jedoch eine marginal schlechtere Performance. In Umgebungen jedoch, in denen weder ein rein-egoistisches noch ein rein-altruistisches Verhalten optimal ist, erzielt unsere kostenbasierte Heuristik beträchtliche Gewinne. Durch die gute Adaptivität unseres Verfahrens gilt dies auch insbesondere für Systeme, in denen sich die Parameter dynamisch ändern können.

Somit stellt unsere kostenbasierte Heuristik einen weiteren Schritt in die Richtung eines sich selbst optimierenden Systems dar. Das Ziel eines solches System besteht darin, die komplizierte und daher fehleranfällige und kostenintensive Kontrolle und Optimierung durch einen Administrator überflüßig machen. So erkennt das Systems automatisch mögliche Engpäße auf Grund von Netzwerk- oder Plattenüberlast und reagiert selbständig – in unserem Fall durch eine Veränderung des Replikationsgrads oder durch eine Migration von Objekten – auf solche Problemsituationen.

Kapitel 6

Eine Erweiterung zur Sicherstellung von Antwortzeitzielen

Nachdem wir in den vorangegangenen Kapiteln eine kostenbasierte Heuristik für verteiltes Caching vorgestellt haben, welche die mittlere Antwortzeit über alle Zugriffe minimiert, leiten wir in diesem Kapitel eine Erweiterung her, die durch eine geeignete Partitionierung des aggregierten Caches in der Lage ist, vorgegebene *Antwortzeitziele* für verschiedene *Lastklassen* zu erfüllen. In Abschnitt 6.1 motivieren wir, warum ein solcher Ansatz in modernen Anwendungen benötigt wird. Anschließend geben wir in Abschnitt 6.2 einen Überblick über bereits existierende Ansätze für ein solches zielorientiertes Lastmanagement. Nachdem wir in Abschnitt 6.3 die System- und Lastcharakteristika beschrieben haben, werden wir in Abschnitt 6.4 den Algorithmus zur Berechnung einer optimalen Cache-Partitionierung vorstellen. Die Einbettung dieses Algorithmus in unseren Prototyp aus Kapitel 4 beschreiben wir in Abschnitt 6.5. Um die Vorteile unserer kostenbasierten verteilten Cache-Heuristik auch bei der Verwaltung der dedizierten Caches ausnutzen zu können, verallgemeinern wir in Abschnitt 6.6 die Heuristik durch das Berücksichtigen klassenspezifischer Hitzen. Eine experimentelle Evaluation dieser zielorientierten Methode für verteiltes Caching erfolgt in [Köni98, SiKö98].

6.1 Motivation

Unterschiedlicher Ressourcenverbrauch und verschiedene Benutzerprioritäten können bei Datenbankanwendungen zu extrem stark variierenden Antwortzeiten führen. So werden in wachsendem Maße neben einfachen *OLTP*-Anfragen auch komplexe *Decision-Support*-Anfragen auf ein und derselben Datenbank gestellt. Wird keine entsprechende Laststeuerung durchgeführt, so kann der sehr hohe Ressourcenverbrauch solcher komplexen Anfragen dazu führen, daß andere Anfragen stark ausgebremst werden. Neben der Komplexität kann jedoch auch die Priorität einen Einfluß auf die benötigten Ressourcen nehmen. Eine Anfrage, deren Ergebnis für eine aktuelle Entscheidung dringend gebraucht wird, sollte nach Möglichkeit die benötigten Mittel erhalten, während auf der anderen Seite eine Anfrage mit einer unkritischen Bearbeitungszeit mit den noch nicht genutzten Ressourcen auskommen muß. Aus diesen Gründen ist es sinnvoll, eine Aufteilung in unterschiedliche Klassen vorzunehmen, wobei für jede Klasse Benutzeranforderungen durch sogenannte *Service-Level-Agreements* [Noon89] ausgedrückt werden können. Eine Mög-

lichkeit, eine solche Anforderungen zu spezifizieren, besteht in der Vorgabe von Antwortzeitzielen für die einzelnen Lastklassen.

Unter diesen Voraussetzungen ist eine einfache Minimierung der mittleren Antwortzeit für alle Anfragen nicht mehr ausreichend. Da sehr oft die Bearbeitungszeit von den notwendigen Plattenzugriffen dominiert wird, stellt die explizite Kontrolle der Cache-Trefferrate eine geeignete Methode dar, um Antwortzeitziele sicherzustellen. Eine Möglichkeit, die Cache-Trefferrate einer Klasse von Anfrage zu kontrollieren, besteht darin, einen dedizierten Hauptspeicherbereich für diese Klasse zu reservieren. Innerhalb dieses Bereichs dürfen dann nur solche Objekte gehalten werden, die von Anfragen dieser Klasse zugegriffen werden. Während jedoch ein solcher dedizierter Bereich die Anfragen der betreffenden Klasse beschleunigt, führt er gleichzeitig zu einer potentiellen Verlangsamung anderer Klassen, da die Gesamtgröße des Hauptspeichers unverändert bleibt. Zwar werden Mechanismen zur Partitionierung des vorhandenen Caches bereits in kommerziellen Datenbanksystemen (z.B. DB2 [Mull93b]) bereitgestellt, jedoch ist die Bestimmung der Partitionsgrößen – selbst wenn alle Lastparameter konstant und dem Datenbankadministrator im vorhinein bekannt sind – ein sehr schwieriges Optimierungsproblem. Suboptimale Partitionierungen können jedoch zu einer schlechten Systemleistung führen, da manche Cache-Bereiche unterlastet sind, während andere Klassen über einen zu kleinen Bereich verfügen, um ihre Antwortzeitziele zu erreichen. Dieses Problem wird sogar noch schwieriger, wenn sich die Lastparameter über die Zeit ändern können.

Weitere Komplikationen für diese – von einen Administrator vorgenommene – statische Partitionierung treten auf, wenn wir an Stelle des Hauptspeichers eines einzelnen Servers den aggregierten Cache eines Netzwerks von Rechnern benutzen wollen. Zusätzlich zu der zweistufigen Hierarchie im Falle eines einzelnen Servers, muß bei einem Rechnernetz eine weitere Hierarchiestufe – nämlich der nichtlokale Cache – bei der Bestimmung der Cache-Partitionierung mit berücksichtigt werden.

Die Summe dieser Probleme macht eine automatische und dynamische Anpassung der Cache-Partitionierung in Abhängigkeit von den aktuellen Lastparameter zu dem einzig möglichen Ansatz für ein zielorientiertes Performance-Tuning in einem Rechnernetzwerk. In diesem Kapitel werden wir daher ein Caching-Verfahren herleiten, das die folgenden Eigenschaften erfüllt:

- Das Verfahren bestimmt *selbständig* die Größen und die Plazierung der dedizierten Bereiche der verschiedenen Klassen, indem es eine (approximative) Lösung des zugrundeliegenden kombinatorischen Problems berechnet. Dadurch muß ein Administrator nicht mehr selbst die Partitionierung festzulegen, sondern es ist ausreichend wenn er die gewünschten Performance-Ziele spezifizieren; die daraus resultierende Partitionierung berechnet das System selbst. Dies stellt eine erhebliche Erleichterung der Administration dar.
- Last- oder Zieländerungen werden von dem Verfahren erkannt, und die Partitionierung wird *dynamisch* an die neue Situation angepaßt. Dies wird durch einen Regelkreis erreicht, der kontinuierlich die Erfüllung der Antwortzeitziele überwacht und bei Verletzungen der Ziele den Algorithmus zur Neuberechnung der Cache-Partitionierung aufruft.
- Das Verfahren verursacht nur einen *geringen Overhead*. Da die Eingabedaten für den Optimierungsprozeß über alle Rechner verteilt sind, verwenden wir nur approximative Informa-

tionen, die mit Hilfe spezieller Protokolle im System verteilt werden und dadurch die Anzahl der notwendigen Nachrichten reduzieren. Die Fehler, die aus diesen Approximationen entstehen können, werden durch den bereits angesprochenen Regelkreis ausgeglichen.

- Schließlich ist das Verfahren für *beliebige Lasten* geeignet. So fordern wir insbesondere nicht, daß die Objektmengen, auf die von den verschiedenen Klassen zugegriffen wird, disjunkt sein müssen. Dies erlaubt uns, auch Anwendungen zu betrachten, in denen Anfragen aus verschiedenen Klassen auf gemeinsame Daten zugreifen.

6.2 Verwandte Arbeiten

Allgemeine Methoden zum automatischen Tuning von Datenbanksystemen werden in [RFG*89, MöWe92, WHMZ94, Rahm97] beschrieben. In [RFG*89] wird ein *Framework* für zielorientiertes Lastmanagement vorgestellt. Innerhalb eines Regelkreises werden – basierend auf aktuellen Beobachtungen – Engpäße erkannt und Aktionen zur Behebung dieser gestartet. Der entscheidende Vorteil dieses Verfahrens ist, daß Zielvorgaben nicht mehr durch das Fein-Tuning vieler Parameter erreicht werden, sondern daß nur noch die Benutzerziele (wie Antwortzeit oder Durchsatz) vorgegeben werden müssen. Wie die Parameter zur Erfüllung dieser Ziele in Abhängigkeit von der aktuellen Situation gewählt werden müssen, erkennt das System selbständig. In [MöWe92] und [WHMZ94] wird ebenfalls ein Regelkreis genutzt um Systemeigenschaften sicherzustellen. In diesen Ansätzen werden Sperrengpäße lokalisiert und einem möglichen *Trashing* durch Sperrkonflikte wird durch eine automatische Anpassung des *Multi-Programming-Levels* entgegengewirkt. Eine Berücksichtigung von vorgegebenen Antwortzeitzielen erfolgt in diesem Ansatz jedoch nicht. In [Rahm97] wird ein ähnlicher Ansatz verfolgt, jedoch wird nun explizit die Erfüllung von Antwortzeitzielen überwacht. Als mögliche Reaktionen auf eine Verletzung der Zielvorgaben stehen neben einer Variation des *Multi-Programming-Levels* auch eine Änderung der Prioritäten für Transaktionen zur Verfügung. Keines dieser Verfahren berücksichtigt jedoch ein automatisches Tuning der Cache-Allokation, um dadurch Antwortzeitziele sicherzustellen.

Antwortzeitziele für unterschiedliche Transaktionsklassen in einem *shared-nothing OLTP*-System werden auch in [FGND93] betrachtet. In dieser Arbeit haben die Autoren eine Metrik eingeführt, die sie *Performance-Index* nennen und die für jede Klasse *k* durch den Quotienten aus der mittleren beobachteten Antwortzeit und der vorgegebenen Zielantwortzeit der Klasse *k* definiert ist. Eine optimale Erfüllung der Ziele des Systems kann durch die Minimierung des maximalen Performance-Index erreicht werden. Drei unterschiedliche zielorientierte Algorithmen werden in der Arbeit vorgeschlagen, die sich in ihrer Komplexität und in ihrem Informationsbedarf unterscheiden. Durch Simulationen wird die Überlegenheit dieser Algorithmen gegenüber anderen Verfahren gezeigt, die nur versuchen, die mittlere Antwortzeit insgesamt zu minimieren. Im Gegensatz zu unserem Ansatz wird die Antwortzeit jedoch nur durch die Beeinflussung von Routing-Entscheidungen (also die Zuteilung einer Transaktion zu einem Rechner) gesteuert und nicht durch eine dynamische Partitionierung der Caches.

Eine Partitionierung eines zentralen Caches zur Gewährleistung von Antwortzeitzielen auf einem einzelnen Rechner ist in einer Reihe von Papieren betrachtet worden [BCDM93, Brow95,

BCL96, CFW*95]. In [BCDM93] wird das sogenannte *Fragment-Fencing* eingeführt, das in [BCL96] durch *Class-Fencing* ersetzt wird. Das Ziel beider Verfahren ist die Minimierung der Antwortzeit der "freien" Klasse, die alle Anfragen ohne Zielvorgabe enthält, unter der Bedingung, daß alle anderen Klassen ihre vorgegebenen Ziele erfüllen. Um dies zu erreichen, kann der Cache dynamisch in Bereiche partitioniert und für bestimmte Klassen dediziert werden. Erfüllt z.B. eine Klasse *k* nicht ihre Zielantwortzeit, so wird ein bestimmter Bereich des Caches exklusiv für Objekte der Klasse *k* reserviert. Nehmen wir an, daß die Ausführungszeiten der Transaktionen durch die Plattenzugriffe dominiert sind, so erhöhen wir durch diesen dedizierten Cache-Bereich die Trefferrate und reduzieren dadurch die Antwortzeit. Entsprechend reduzieren wir die Größe des dedizierten Cache-Bereichs, falls die Antwortzeit für eine Klasse besser als die Zielantwortzeit ist. Von dieser Reduktion der Größe profitiert die freie Klasse, da diese keinen dedizierten Cache-Bereich allozieren darf und somit nur den Bereich benutzen kann, der nicht von anderen dedizierten Cache-Bereichen eingenommen wird. *Fragment-Fencing* und *Class-Fencing* unterscheiden sich in der Art, in der sie den notwendigen Speicherbedarf für die dedizierten Cache-Bereiche abschätzen. Während *Fragment-Fencing* eine direkte Proportionalität zwischen der Größe der Cache-Bereiche und der Antwortzeit annimmt, geht *Class-Fencing* nur von einer Proportionalität zwischen der Cache-Fehlerrate und der Antwortzeit aus. Die bei *Class-Fencing* weiterhin zu bestimmende Abhängigkeit zwischen der Fehlerrate und der Cache-Größe wird durch eine lineare Extrapolation von früher gemessenen Fehlerraten in Abhängigkeit von der Cache-Größe abgeleitet. Diese Methode garantiert eine schnelle Konvergenz unter der Voraussetzung, daß die Fehlerrate als Funktion der Cache-Größe konkav ist. In [Brow95] wurde durch eine empirische Untersuchung die Gültigkeit dieser Annahme für mehrere häufig benutzte Cache-Ersetzungsstrategien gezeigt. Ist die Antwortzeit der Transaktionen nicht nur durch die Größe des Cache-Bereichs für Plattenzugriffe bestimmt, sondern auch durch den Cache-Bereich, der für Berechnungen zur Verfügung steht (z.B. Hash-Joins), so ist die bisher betrachtete Reduktion der Plattenzugriffe durch eine Vergrößerung der Caches nicht mehr ausreichend. Daher wird in [BMCL94] *Fragment-Fencing* mit einer automatischen Steuerung des *Multi-Programming-Levels* kombiniert.

In [CFW*95] ist der *Dynamic-Tuning*-Algorithmus beschrieben. Auch bei diesem Ansatz wird der lokal zur Verfügung stehende Cache in dedizierte Bereiche aufgeteilt, um so die Antwortzeit einzelner Klassen zu reduzieren. Analog zu [FGND93] basiert diese Methode auf den Performance-Indizes der Klassen und versucht einen Zustand zu finden, in dem das Maximum über alle Performance-Indizes minimal ist. Der Algorithmus berechnet die Auswirkungen von geringfügigen Änderungen innerhalb der Cache-Partitionierung auf den Performance-Index und führt die Änderungen aus, die den Systemzustand – entsprechend des oben skizzierten Optimierungsziels – verbessern.

6.3 System- und Lastcharakteristika

Analog zu den Systemannahmen in den vorangegangenen Kapiteln nehmen wir an, daß das Netzwerk aus *N* Rechnern besteht, die durch ein schnelles Netzwerk miteinander verbunden sind, daß an jedem Rechner eine Festplatte angeschlossen ist und daß die insgesamt vohandenen *M* Objekte über diese Festplatten verteilt sind. Die einzelnen Rechner können jeweils unterschiedlich große Hauptspeicher besitzen. Die Cache-Größe des Rechners *i* bezeichnen wir mit B_i.

Der insgesamt lokal verfügbare Cache wird – abhängig von der Anzahl der vorhandenen dedizierten Cache-Partitionen – entweder von einem oder von mehreren Cache-Managern verwaltet. Als einzige Bedingung an die Cache-Manager fordert unser Verfahren, daß eine Vergrößerung eines dedizierten Cache-Bereichs einer Klasse *k* nicht zu einer Verschlechterung der Antwortzeit für diese Klasse führen darf. Obwohl in [BNS69] mit einer FIFO-Ersetzungsstrategie auf einem einzelnen Rechner ein Gegenbeispiel für diese Bedingung konstruiert wird, sollte diese Annahme von nahezu allen praktisch benutzten Ersetzungsstrategien erfüllt werden.

Wir nehmen an, daß sich die Gesamtlast aus mehreren Anfragetypen zusammensetzt, die an jedem der *N* Rechner im System initiiert werden können. Jede Anfrage kann wiederum in mehrere Objektzugriffe aufgeteilt werden. Die einzelnen Objektzugriffe erfolgen durch *Data-Shipping*, d.h. die Objekte werden jeweils zu dem Rechner gesendet, der den Zugriff initiiert hat. Eine wichtige Annahme für das Funktionieren unserer Methode ist, daß die Anfragen den größten Teil ihrer Ausführungszeit für das Lesen der Objekte von Platte benötigen. Nur in diesem Fall können wir durch die Variation der Cache-Trefferrate die Antwortzeit merklich beeinflussen.

Zur Vereinfachung werden wir in diesem Kapitel nur Lesezugriffe auf die Objekte betrachten. Schreibzugriffe auf einzelne Objekte können direkt mit Hilfe des Kohärenzprotokolls aus Abschnitt 4.1.3 durchgeführt werden. Da wir in diesem Abschnitt jedoch Anfragen, d.h. eine Menge von mehreren, logisch zusammenhängenden Zugriffen, betrachten, tritt bei allgemeinen Schreibzugriffen das Problem der Concurrency-Kontrolle und der Atomarität von Anfragen auf. Obwohl unser Verfahren durch die Benutzung eines (verteilten) 2-Phasen-Lockings [EGLT76] für die Concurrency-Kontrolle und eines 2-Phasen-Commit-Protokolls [Gray79] für die Atomarität erweitert werden könnte, betrachten wir dies hier nicht, da der Schwerpunkt unserer Arbeit auf dem Erzielen eines guten Caching-Verhaltens liegt.

Entsprechend der vorgegebenen Antwortzeitziele können wir die Anfragen in Klassen einteilen. Die Klassen, die Anfragen mit Zielvorgaben enthalten, bezeichnen wir als *Zielklassen*, während wir alle übrigen Anfragen in der sogenannten *freien Klasse* zusammenfassen.

Obwohl es prinzipiell möglich ist beliebige Anfragen mit den gleichen Antwortzeitzielen in einer Klasse zusammenzufassen, ist es doch günstig eine gewisse Homogenität der Anfragen innerhalb einer Klasse zu berücksichtigen. Eine Sicherstellung der Zielantwortzeit durch eine Partitionierung des Caches ist am einfachsten möglich, wenn bei der Aufteilung der Anfragen in die verschiedenen Klassen die folgenden Anforderungen erfüllt werden:

(1) Anfragen aus unterschiedlichen Klassen arbeiten nicht auf den selben Objekten,

(2) Anfragen, die zur selben Klasse gehören, arbeiten auf den selben Objekten und

(3) Anfragen, die zur selben Klasse gehören, haben den gleichen Ressourcenverbrauch, d.h. die Anzahl der Objektzugriffe ist für alle Anfragen einer Klasse ungefähr gleich.

Die Auswirkungen, die durch das Verletzen dieser verschiedenen Anforderungen entstehen können, wollen wir an Hand der 3 folgenden Beispiele verdeutlichen. In Beispiel 6.1 untersuchen wir eine Situation, in der Anfragen aus verschiedenen Klassen auf die gleichen Objekte zugreifen, in Beispiel 6.2 fassen wir in einer Klasse Anfragen zusammengefaßt, die auf diskjunkte Objektmengen zugreifen und in Beispiel 6.3 untersuchen wir schließlich die Auswirkungen, falls Anfragen mit unterschiedlichem Ressourcenverbrauch in einer Klasse zusammengefaßt werden.

Beispiel 6.1:

In diesem Beispiel nehmen wir an, daß die beiden Anfragetypen t_1 aus der Klasse k_1 und t_2 aus der Klasse k_2 auf die gleichen Objekte zugreifen. Desweiteren sei die Zielantwortzeit der Klasse k_1 wesentlich geringer als die der Klasse k_2. Um nun die Zielantwortzeit für die Klasse k_1 zu erzielen, muß der dedizierte Cache-Bereich für diese Klasse vergrößert werden. Da wir aus Effizienzgründen ein repliziertes Caching eines Objekts in mehreren dedizierten Cache-Bereichen des gleichen Rechners verbieten, greifen auch die Anfragen des Typs t_2 auf die in dem Cache der Klasse k_1 gehaltenen Objekte zu. Ist der Unterschied zwischen den Zielantwortzeiten der Klasse k_1 und k_2 sehr groß, so kann der große, dedizierte Cache-Bereich der Klasse k_1 dazu führen, daß die Anfragen des Typs t_2 ebenfalls sehr stark von diesem Cache profitieren und dadurch ihre Zielvorgabe deutlich unterbieten.

■

Beispiel 6.2:

Nun nehmen wir an, daß die Klasse k die beiden Anfragetypen t_1 und t_2 enthält. Beide Anfragetypen greifen jeweils auf die gleiche Anzahl von Objekten zu. Die Mengen der von den Anfragen zugegriffenen Objekte sind jedoch disjunkt. Besitzen nun Anfragen des Typs t_1 eine sehr viel größere Zwischenankunftsrate als die von t_2, so enthalten die dedizierten Cache-Bereiche für die Klasse k fast ausschließlich Objekte, die von t_1-Anfragen zugegriffen werden. Solange die Größe des dedizierten Bereichs für die Klasse k kleiner ist als die Menge der von t_1-Anfragen insgesamt referenzierten Objekte, beeinflussen Veränderungen in der Größe des dedizierten Bereichs somit nur die Antwortzeit für t_1-Anfragen. Dies kommt daher, daß sich nur die Anzahl der Objekte des Anfragetyps t_1 erhöht, während sämtliche Objekte, die von t_2-Anfragen zugegriffen werden, immer noch von Platte gelesen werden müssen. Durch eine entsprechende Wahl der Größe können wir zwar erreichen, daß die mittlere Antwortzeit über beide Anfragetypen t_1 und t_2 mit dem vorgegebenen Ziel übereinstimmt, jedoch trifft dies nicht die Intention des Anwenders, da in diesem Fall die Antwortzeiten der Anfragen des Typs t_1 sehr viel niedriger und die des Typs t_2 sehr viel höher als die Zielantwortzeit sind.

■

Beispiel 6.3:

In diesem Beispiel nehmen wir wiederum an, daß die Klasse k aus zwei Anfragetypen t_1 und t_2 besteht, die jedoch nun auf die gleichen Objekte zugreifen. Die beiden Typen unterscheiden sich lediglich in ihrer Komplexität, d.h. Anfragen des Typs t_1 greifen im Mittel auf sehr viel mehr Objekte zu als Anfragen des Typs t_2. Zur weiteren Vereinfachung nehmen wir an, daß alle Objekte gleichhäufig von den beiden Anfragentypen referenziert werden. Unter diesen Voraussetzungen bewirkt eine Variation der Größe der dedizierten Cache-Bereiche der Klasse k keine Veränderung des relativen Verhältnises in den Antwortzeiten zwischen t_1 und t_2. Da nur die mittlere Antwortzeit einer ganzen Klasse bei unserem Verfahren berücksichtigt wird, wird auch in dieser Situation – ähnlich wie in Beispiel 6.2 – das Ziel dadurch erreicht, daß die Anfragen des Typs t_1 weit oberhalb der Zielvorgabe und die des Typs t_2 unterhalb der Zielvorgabe liegen.

■

Fassen wir die Ergebnisse der verschiedenen Beispiele zusammen, so haben wir in Beispiel 6.1 gesehen, daß das Zugreifen auf die gleichen Objekte von unterschiedlichen Klassen lediglich zu einer übermäßigen Bevorteilung einer Klasse führen kann. Dieser Effekt verursacht jedoch keine Zusatzkosten für das System, da der dedizierte Cache-Bereich der Klasse mit der geringeren Zielantwortzeit in jedem Fall so groß gewählt werden muß. Daher kann unsere Methode zur zielorientierten Cache-Paritionierung auch genutzt werden, wenn Anfragen mit unterschiedlichen Zielvorgaben (z.B. eine *OLTP*- und eine *Decision-Support*-Anwendung) auf den gleichen Datenobjekten operieren. Die Beispiele 6.2 und 6.3 hingegen zeigen Situationen, die im allgemeinen unerwünscht ist, da nur die Mittelwertbildung über sehr unterschiedliche Antwortzeiten der verschiedenen Typen zu der geforderten Zielantwortzeit führt.

Eine einfache Richtlinie für die Einteilung der Klassen kann somit darin bestehen, daß man alle Anfragen, die die gleiche Zielantwortzeit haben, die auf die gleichen Objekte zugreifen und die in ihrer Komplexität vergleichbar sind, in einer Klasse zusammenfaßt. Eine solche Einteilung ist immer möglich, jedoch kann es zu Überschneidungen in den zugegriffenen Objekten kommen. An Hand des Beispiels 6.1 haben wir jedoch gezeigt, daß dies lediglich die Antwortzeit einer anderen Klasse über die vorgebene Zielantwortzeit hinaus beschleunigt, und daher keine negativen Folgen für das Gesamtsystem auftreten. Im folgenden gehen wir davon aus, daß eine Einteilung (z.B. entsprechend dieser Richtlinie) der Anfragen mit vorgegebenen Antwortzeitzielen in K verschiedene Klassen bereits erfolgt ist. Die Zielklassen numerieren wir von 1 bis K und der freien Klasse geben wir den Index Null.

6.4 Approximative Berechnung der optimalen Cache-Partitionierung

In diesem Abschnitt leiten wir eine Methode her, die es uns erlaubt, die Größen der Cache-Bereiche auf den verschiedenen Rechnern für eine Klasse k zu berechnen, so daß die mittlere Antwortzeit über alle Anfragen dieser Klasse ein vorgegebenes Antwortzeitziel erfüllt. Im Gegensatz zu dem zentralisierten Problem, das in [BCL96] betrachtet wird, haben wir bei verteiltem Caching einen zusätzlichen Freiheitsgrad. Neben der Möglichkeit, die Antwortzeit dadurch zu beeinflußen, daß wir die Gesamtgröße der dedizierten Cache-Bereiche ändern, können wir jetzt auch wählen, auf welchen Rechnern eine Änderung des Bereichs durchgeführt bzw. auf welchen Rechnern ein neuer dedizierter Bereich alloziert werden soll.

Um die aggregierte Leistungsfähigkeit der verschiedenen Rechner möglichst gut ausnützen zu können, zielt auch unser Ansatz auf eine Minimierung der mittleren Antwortzeit für die freie Klasse unter der Bedingung, daß die Antwortzeitziele für alle Zielklassen erfüllt werden. Zur Reduktion der Komplexität des Optimierungsproblems nehmen wir an, daß – mit der Ausnahme der gerade betrachteten Klasse k – alle anderen Klassen ihre Antwortzeitziele erfüllen.[12] Für diese Klasse k berechnet unser Algorithmus eine neue Allokation, die idealerweise zu einer Situation

12. Diese Bedingung ist für die theoretische Herleitung notwendig. Bei der in Abschnitt 6.5 beschriebenen Implementation dieses Verfahrens erlauben wir jedoch die gleichzeitige Änderung mehrerer Klassen. Die hierdurch auftretenden Ungenauigkeiten werden durch einen Regelkreis eliminiert.

führen sollte, in der die Klasse *k* ebenfalls ihr Antwortziel erreicht; zumindest aber sollte die Differenz zwischen der mittleren Antwortzeit und der Zielantwortzeit verringert werden. Diesen Algorithmus erhalten wir durch eine Verallgemeinerung des *Class-Fencing*-Algorithmus auf *N* Dimensionen, wobei *N* die Anzahl der Rechner im System ist. Diese Methode, bei der wir die neue Antwortzeit durch Extrapolation von den gemessenen Antwortzeiten früherer Allokationen berechnen, benutzen wir sowohl zur Herleitung der Zielfunktion für die freie Klasse als auch für die Bestimmung der Nebenbedingungen zur Sicherstellung der Antwortzeit für die Zielklasse *k*.

Da wir verteiltes Caching betrachten, ist die lokale Antwortzeit $RT_{k,i}$ einer Anfrage der Klasse *k* auf dem Rechner *i* nicht nur von der lokalen Cache-Trefferrate sondern auch von der globalen Cache-Trefferrate abhängig. Diese Trefferraten hängen ihrerseits von der Größe des dedizierten lokalen Cache-Bereichs ($LM_{k,i}$) bzw. der nichtlokalen Cache-Bereiche ($RM_{k,i}$) der Klasse *k* ab.[13] Ähnlich wie bei *Class-Fencing* mit nur einem Server ist jedoch die Relation, die die Anwortzeit in Abhängigkeit von der Cache-Größe beschreibt, a priori unbekannt. Lediglich einige Punkte dieser Relation, die von früheren Messungen stammen, sind bekannt. Diese Meßpunkte können wir nun benutzen, um die Koeffizienten $\alpha_{k,i}, \beta_{k,i}$ und $\gamma_{k,i}$ einer linearen Approximation dieser unbekannten Antwortzeitfunktion zu berechnen:

$$RT_{k,i}(LM_{k,i}, RM_{k,i}) = \alpha_{k,i} \times LM_{k,i} + \beta_{k,i} \times RM_{k,i} + \gamma_{k,i} \tag{6.1}$$

Da die Größe der nichtlokalen Cache-Bereiche für eine Klasse *k* auf dem Rechner *i* durch die Größe der lokalen Caches auf allen anderen Rechnern bestimmt ist, können wir mit Hilfe von:

$$RM_{k,i} = \sum_{j=1, j\neq i}^{N} LM_{k,j} \tag{6.2}$$

die Gleichung (6.1) so umformen, daß die lokale Antwortzeit nur noch von den Größen der lokalen, dedizierten Cache-Bereiche abhängt:

$$RT_{k,i}(LM_{k,1}, ..., LM_{k,N}) = \alpha_{k,i} \times LM_{k,i} + \beta_{k,i} \times \sum_{j=1, j\neq i}^{N} LM_{k,j} + \gamma_{k,i} \tag{6.3}$$

Da wir für die Klasse *k* nicht auf jedem Rechner ein separates Ziel spezifizieren, sondern nur eine mittlere Zielantwortzeit über alle Rechner, betrachten wir nun die gewichtete Summe der lokalen Antwortzeiten. Als Gewichtungsfaktoren benutzen wir die lokalen Zwischenankunftsraten $\lambda_{k,i}$ von Anfragen der Klasse *k* auf dem Rechner *i*. Unter diesen Voraussetzungen können wir die mittlere Antwortzeit RT_k folgendermaßen berechnen:

$$\begin{aligned} RT_k(LM_{k,1}, ..., LM_{k,N}) &= \sum_{i=1}^{N} \left(\lambda_{k,i} \times RT_{k,i}(LM_{k,1}, ..., LM_{k,N}) \right) \\ &= \sum_{i=1}^{N} \left[\lambda_{k,i} \times \left[\alpha_{k,i} \times LM_{k,i} + \beta_{k,i} \times \sum_{j=1, j\neq i}^{N} LM_{k,j} + \gamma_{k,i} \right] \right] \end{aligned}$$

13. Auch hier gehen wir wieder – wie bei der Herleitung unserer kostenbasierten Heuristik für verteilten Caching – davon aus, daß wir die nichtlokalen Caches zusammenfassen können. Dies ist gerechtfertigt, da wir ein homogenes Netzwerk annehmen, d.h. die Netztransferkosten zwischen allen Paaren von Rechnern sind gleich.

$$= \sum_{i=1}^{N} \underbrace{\left(\lambda_{k,i}\alpha_{k,i} + \sum_{j=1, j\neq i}^{N} \lambda_{k,j}\beta_{k,j} \right)}_{\stackrel{\text{def}}{=} \mu_{k,i}} \times LM_{k,i} + \underbrace{\sum_{i=1}^{N} \lambda_{k,i}\gamma_{k,i}}_{\stackrel{\text{def}}{=} \nu_k} \tag{6.4}$$

Die Geichung (6.4) können wir als *N*-dimensionale Hyperebene interpretieren, welche die Abhängigkeit zwischen der mittleren Antwortzeit und der Größen der lokalen Cache-Bereiche beschreibt. Um dies zu verdeutlichen, haben wir in Abbildung 6.1 sowohl die mittlere Antwortzeitkurve als auch die berechnete lineare Approximation für ein 2-Rechner-Beispiel skizziert. In dieser Abbildung sehen wir, daß die berechnete Hyperebene in allen Dimensionen monoton fallend ist. Dies bedeutet, daß sich die mittlere Antwortzeit verringert, wenn die Größe eines dedizierten Cache-Bereichs auf einem beliebigen Rechner vergrößert wird.

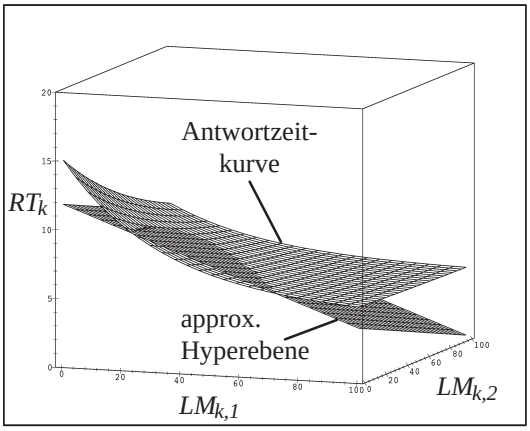


Abbildung 6.1: Approximation der Antwortzeitkurve durch eine Hyperebene

Um sicherzustellen, daß die neue Partitionierung das vorgegebene Ziel erfüllt, müssen wir die Gleichung (6.4) nur gleich der Zielantwortzeit RT_k^{goal} für die Klasse *k* setzen:

$$RT_k^{goal} = \sum_{i=1}^{N} \mu_{k,i} \times LM_{k,i} + \nu_k \tag{6.5}$$

Diese Vorgehensweise können wir wiederum geometrisch an Hand unseres 2-Rechner-Beispiels illustrieren. In Abbildung 6.2 haben wir die approximierte Antwortzeit-Hyperebene, die durch die Gleichung (6.4) beschrieben ist, zusammen mit der Ebene der Zielantwortzeit, die parallel zu den Achsen der “Cache-Größen” verläuft, skizziert. Der Schnitt zwischen diesen beiden Ebenen ist durch die Gleichung (6.5) beschrieben. Er enthält alle möglichen Cache-Partitionierungen für die Klasse *k* auf den Rechnern 1 und 2, die zu einer mittleren Antwortzeit der Klasse *k* führen, die gleich der gewünschten Zielantwortzeit ist. Dies ist jedoch nur korrekt, falls die wirkliche Antwortzeitkurve exakt mit der approximierten Hyperebene übereinstimmen würde. Da diese Bedingung im allgemeinen nicht erfüllt sein wird, kann im allgemeinen nicht erwarten werden, daß bereits nach einer einmaligen Berechnung die gesuchte Partitionierung erreicht wird. In jedem Fall sollte jedoch die neu berechnete Allokationen zu einer Verringerung der Differenz zwischen der aktuellen, mittleren Antwortzeit und der Zielantwortzeit führen.

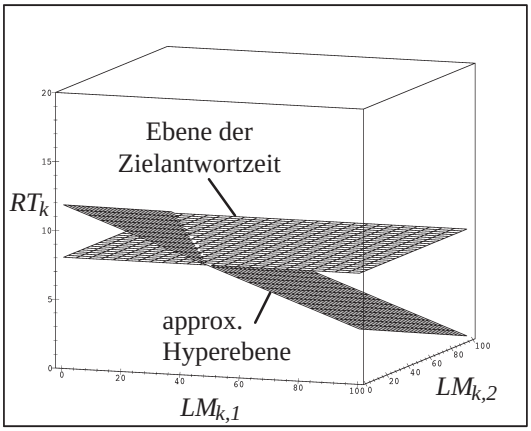


Abbildung 6.2: Schnitt der approximierte Hyperebene mit der Zielantwortzeit

Bevor wir die Zielfunktion für unser Optimierungsproblem herleiten, führen wir zunächst zwei Ungleichungen ein, die aus der beschränkten Größe des Hauptspeichers auf jedem Rechner resultieren. Offensichtlich kann der lokale Cache für eine Klasse *k* auf einem beliebigen Rechner nie kleiner als Null sein. Ebenso kann die Summe aller lokalen Caches nie größer als der lokal reservierte Hauptspeicher (B_i) sein. In unserer Terminologie können wir diese Beschränkungen auf die folgende Art ausdrücken:

$$0 \leq LM_{k,i} \leq B_i - \sum_{l=1,l\neq k}^{K} LM_{l,i} \quad \text{für alle Rechner } i \tag{6.6}$$

Um die gewünschte Antwortzeit für die Klasse *k* zu erzielen, können wir eine beliebige Allokation wählen, die die Gleichungen (6.5) und (6.6) erfüllt. Da wir jedoch gleichzeitig versuchen, die mittlere Antwortzeit der freien Klasse zu minimieren, leiten wir nun eine Zielfunktion für dieses Optimierungsproblem her. Analog zu der Approximation der Antwortzeitfunktion in Abhängigkeit von den Größen der Cache-Bereiche für eine Klasse *k*, können wir die Antwortzeit $RT_{0,i}$ der freien Klasse auf dem Rechner *i* folgendermaßen approximieren:

$$RT_{0,i}(LM_{k,1}, ...LM_{k,N}) = \alpha_{0,i} \times \left(B_i - \sum_{l=1}^{K} LM_{l,i} \right) + \beta_{0,i} \times \sum_{j=1,j\neq i}^{N} \left(B_j - \sum_{l=1}^{K} LM_{l,j} \right) + \gamma_{0,i} \tag{6.7}$$

In dieser Formel nehmen wir an, daß der Cache auf dem Rechner *i*, der den Anfragen der freien Klasse zur Verfügung steht, gleich dem lokalen Hauptspeicher abzüglich der lokal-dedizierten Cache-Bereiche ist. Zusätzlich nehmen wir an, daß wir nur die Allokation einer einzigen Klasse – nämlich der Klasse *k* – ändern. Daher hängt die Antwortzeit der freien Klasse auf dem Rechner *i* in diesem Fall nur von den Cache-Größen der Klasse *k* auf den verschiedenen Rechnern ab. Alle Werte für $LM_{l,i}$ ($l \neq k$) nehmen wir innerhalb dieser Berechnung als konstant an. Nutzen wir diese Annahmen aus und fassen wir alle konstanten Terme aus Geichung (6.7) zu der Konstanten $\Gamma_{0,i}$ zusammen, so können wir die Gleichung folgendermaßen umformen:

$$RT_{0,i}(LM_{k,1}, ...LM_{k,N}) = \left(-\alpha_{0,i} \right) \times LM_{k,i} + \left(-\beta_{0,i} \right) \times \sum_{j=1,j\neq i}^{N} LM_{k,j} + \Gamma_{0,i} \tag{6.8}$$

Berechnen wir nun analog zur Gleichung (6.4) die gewichtete Summe der lokalen Antwortzeiten und fassen alle Konstanten zusammen, so erhalten wir:

$$RT_0(LM_{k,1}, ..., LM_{k,N}) = \sum_{i=1}^{N}\left(\lambda_{0,i} \times RT_{0,i}(LM_{0,1}, ..., LM_{0,N})\right)$$
$$= \sum_{i=1}^{N}\mu_{0,i} \times LM_{k,i} + \nu_0 \qquad (6.9)$$

Besonders herauszustellen ist, daß in dieser Gleichung – im Gegensatz zu der Gleichung (6.4) – alle Gradienten $\mu_{0,i}$ größer als Null sind. Dadurch wächst die Antwortzeit der freien Klasse an, wenn wir einen beliebigen lokalen Cache-Bereich der Klasse *k* vergrößern. Diese Berechnung können wir an Hand der Abbildung 6.3 verdeutlichen. In dieser Abbildung ist die Antwortzeitkurve für die freie Klasse zusammen mit der – durch die Gleichung (6.9) – hergeleiteten Approximation dargestellt.

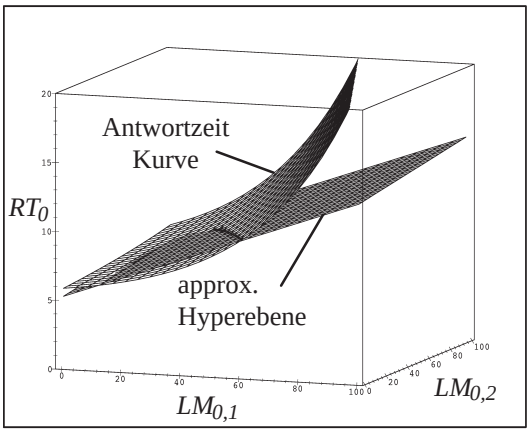


Abbildung 6.3: Approximation der Antwortzeitkurve für die freie Klasse durch eine Hyperebene

Fassen wir die verschiedenen Berechnungen in diesem Abschnitt zusammen, so können wir die gesuchte neue Cache-Partitionierung berechnen, indem wir das folgende *Lineare Progamm* mit den Variablen $\boldsymbol{LM_{k,i}}$ ($1 \le i \le N$) lösen:

Minimiere:

$$\sum_{i=1}^{N}\mu_{0,i} \times \boldsymbol{LM_{k,i}} + \nu_0 \qquad \text{(Gleichung 6.9)}$$

unter der Bedingung:

$$RT_k^{goal} = \sum_{i=1}^{N}\mu_{k,i} \times \boldsymbol{LM_{k,i}} + \nu_k \qquad \text{(Gleichung 6.5)}$$

und unter Berücksichtigung der Schranken:

$$0 \le \boldsymbol{LM_{k,i}} \le B_i - \sum_{l=1, l\neq k}^{K} LM_{l,i} \quad \text{für alle Rechner } i. \qquad \text{(Ungleichung 6.6)}$$

6.5 Online-Implementation

Nachdem wir in dem vorangegangenen Abschnitt den Algorithmus zur Berechnung einer neuen Cache-Partitionierung beschrieben haben, wollen wir in diesem Abschnitt darauf eingehen, wie wir die dort eingeführten Berechnungen online in unserem Prototyp aus Kapitel 4 durchführen können. Wir nehmen an, daß für jede Zielklasse auf jedem Rechner ein lokaler Agent existiert und daß zusätzlich für jede Zielklasse auf einem Rechner ein Koordinatorprozeß läuft. Um eine Lastbalancierung zu ermöglichen, dürfen die Koordinatorprozesse beliebig plaziert werden. Sogar die dynamische Migration von Koordinatorprozessen auf andere, weniger belastete Rechner ist prinzipiell möglich, solange die entsprechenden Agenten über diese Migration informiert werden. In der Abbildung 6.4 haben wir eine möglich Situation für ein System mit 4 Rechnern und 4 Klassen (3 Zielklassen und die freie Klasse) skizziert.

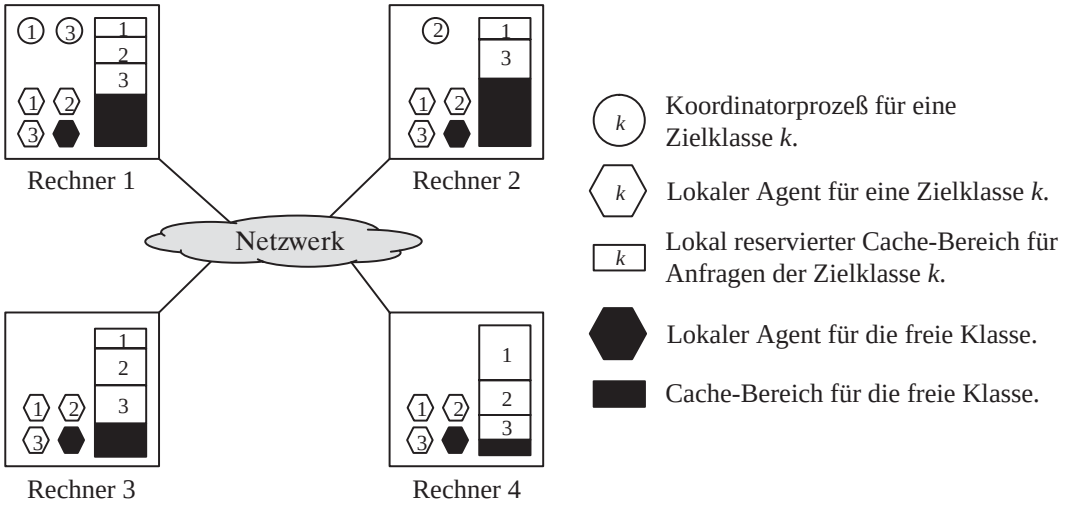


Abbildung 6.4: System mit 4 Rechnern und 3 Zielklassen

Abbildung 6.5 zeigt den prinzipiellen Ablauf unseres Algorithmus für ein zielorientiertes Caching. Die in dieser Abbildung gezeigten vier Phasen bilden zusammen einen Regelkreis, in dem jeweils kontrolliert wird, ob die vorgegebenen Ziele erfüllt werden. Im Falle einer Nichterfüllung wird durch eine Neuberechung der Cache-Partitionierung darauf reagiert. Um den Overhead für das Sammeln der Statistiken zu verringern, ist in unserer Implementation die Sammelphase in zwei Teile aufgespalten. Die erste Teilphase, die von den lokalen Agenten durchgeführt wird,

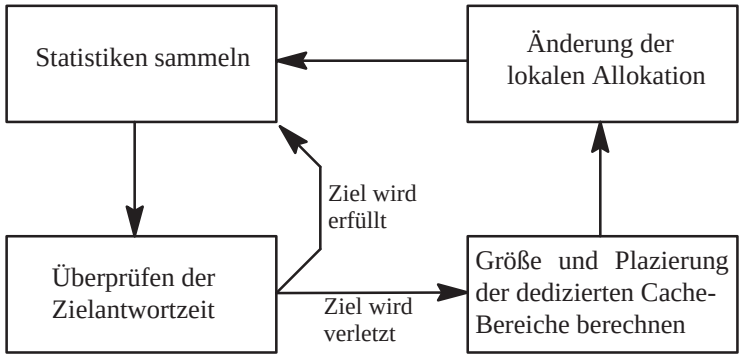


Abbildung 6.5: Regelkreis für zielorientiertes, verteiltes Caching

fängt geringfügige Schwankungen in den beobachteten Meßgrößen ab und reduziert dadurch den Nachrichtenverkehr auf dem Netzwerk. Im folgenden beschreiben wir detailliert die verschiedenen Phasen.

(a) Sammelphase auf dem Agenten
Jedesmal, wenn eine Anfrage der Klasse *k* auf dem entsprechenden Rechner gestartet wird, berechnet der Agent die Zwischenankunftszeit und nach der Beendigung der Anfrage die Antwortzeit und speichert diese Werte lokal ab. Um starke Schwankungen durch stochastisches Rauschen zu verhindern, merkt sich der Agent nicht nur den jeweils letzten Wert, sondern er führt eine exponentielle Glättung mit Hilfe der neuen Werte durch (siehe Abschnitt 3.1.4). Ändert sich trotz dieser Glättung einer dieser Werte signifikant, so informiert der Agent seinen Koordinator über die neuen Werte. Neben den Werten der entsprechenden Klasse werden auch die Ankunftsrate und die Antwortzeit der freien Klasse für den Optimierungsprozeß der Zielklasse benötigt. Daher müssen wir auch Änderungen dieser Werte an die Koordinatoren der Zielklassen senden. Zusätzlich müssen auch Veränderungen in der Größe des nicht-dedizierten Caches an die Koordinatoren gemeldet werden, da diese ebenfalls einen Einfluß auf die Optimierung besitzen.

Besonders erwähnt werden sollte, daß die Agenten nicht synchronisiert auf den verschiedenen Rechnern laufen müssen. Dies ist möglich, da der Koordinator die jeweils zuletzt empfangenen Werte seiner eigenen und der Agenten der freien Klasse lokal speichert.

(b) Sammelphase der Koordinatoren
Innerhalb dieser Phase wartet der Koordinator einer Klasse auf die Daten, die in der Phase (a) versendet werden. Empfängt der Koordinator eine Nachricht, so wird die in dieser Nachricht enthaltene Information entweder benutzt, um einen neuen Meßpunkt für den Approximationsprozeß zu generieren (wenn sich seit dem letzten Meßpunkt die Partitionierung geändert hat) oder um einen alten Meßpunkt zu aktualisieren (wenn sich nur die Antwortzeit für die aktuelle Partitionierung geändert hat). Im ersten Fall müssen wir sicherstellen, daß eine eindeutige Approximation der für den Optimierungsprozeß benötigten N-dimensionalen Hyperebene durch die vorhandenen Meßpunkte möglich ist. Nehmen wir an daß x_1 der jüngste Meßpunkt ist, so können wir dies gewährleisten, wenn wir jeweils die N+1 jüngsten Meßpunkte $\{x_1,...,x_{N+1}\}$ auswählen, für die die N Vektoren x_1-x_2,...,x_1-x_{N+1} linear unabhängig sind. Jeder Vektor x_i enthält dabei die lokalen Cache-Bereichsgrößen und die mittlere Gesamtantwortzeit der Klasse, für die der Koordinator zuständig ist.

Während wir nach einer einmaligen “Aufwärmphase” durch diese Methode immer sicherstellen können, daß wir genügend Information für den Optimierungsprozeß zur Verfügung haben – auch wenn zwischenzeitlich dedizierte Cache-Bereiche der entsprechenden Klasse gelöscht wurden –, fehlt uns während dieser Aufwärmphase die notwendige Information zur Berechnung einer neuen Partitionierung. In diesem Fall können wir die folgende einfache Methode benützen, um neue Meßpunkte zu “generieren”:

Es existiert erst ein Meßpunkt:
Erreicht eine Klasse *k* zum ersten Mal nicht ihre Zielantwortzeit, so wird auf einem Rechner (z.B. dem mit dem meisten nicht-dedizierten Hauptspeicher oder dem mit der größten Ankunftsrate für Anfragen der Klasse *k*) ein prozentueller Anteil des verfügbaren Caches für diese Klasse dediziert.

Es existieren n ($1<n<N+1$) Meßpunkte*:*
In diesem Fall berechnen wir zunächst die zu diesen Punkten korrespondierende (n-1)-dimensionale Hyperebene mit den Gradienten $\mu_{k,i}$ ($1 \leq i<n$) und der Konstanten ν_k. Diese Ebene erweitern wir nun auf eine n-dimensionale Ebene, indem wir $\mu_{k,j}$ (für ein beliebiges $j > n$) auf den maximalen Wert der berechneten Gradienten setzen. Eine sinnvolle Möglichkeit, diesen Rechner j zu bestimmen, besteht darin, wiederum eine der bereits oben angesprochenen Rangfolgen – entsprechend der Größe des nicht-dedizierten Speichers oder entsprechend der Ankunftsraten – zu berücksichtigen.

Diese Methode soll sicherstellen, daß für Rechner, die bisher noch nicht berücksichtigt wurden, neue Meßwerte generiert werden, ohne jedoch ein "Überschießen" in der Cache-Allokation zu verursachen. Dies wird dadurch erreicht, daß der Partitionierungsalgorithmus entsprechend der Wahl der Gradienten annimmt, daß bereits eine kleine Änderung auf einem Rechner j zu einer starken Verbesserung der Antwortzeit führt. Dadurch ist es sehr wahrscheinlich, daß auf dem Rechner j ein dedizierter Bereich alloziert wird. Gleichzeitig wird jedoch durch den großen Gradienten dieser Bereich ziemlich klein gewählt. Somit wird der Einfluß auf anderer Klassen – insbesondere auf die freie Klasse und auf die Klassen ohne dedizierte Cache-Bereiche – gering gehalten. Falls dieser initiale Anteil nicht ausreichend ist, um die Zielantwortzeit zu erreichen, kann der Algorithmus im nächsten Iterationsschritt einen zusätzlich gemessenen Meßpunkt berücksichtigen und sollte daher eine genauere Approximation der Hyperebene durchführen können. Für die freie Klasse brauchen wir keine analoge Berechnung durchzuführen, da eine neue Cache-Partitionierung einer beliebigen Zielklasse automatisch auch zu einem neuen Meßpunkt für die freie Klasse führt.

(c) Überprüfungsphase
Kommen die in der Phase (b) empfangenen Daten von einem Agenten der Klasse k, so müssen wir überprüfen, ob die aktuelle Antwortzeit die Zielvorgabe erfüllt. Mit Hilfe der Gleichung (6.4) können wir die gewichtete mittlere Antwortzeit berechnen. Falls eine Verletzung des Ziels vorliegt, gehen wir zur Phase (d); ansonsten wird der aktuelle Durchlauf des Regelkreises beendet und wir kehren wieder in die Sammelphase zurück.

(d) Berechnungsphase
Während dieser Phase bestimmt der Koordinator der Klasse k – entsprechend der Methode aus Abschnitt 6.4 – die optimale Partitionierung der lokalen Caches für die Klasse k. Dies umfaßt die Approximation der Hyperebene basierend auf den in der Phase (b) registrierten Meßwerten, gefolgt von der Minimierung entsprechend des vorgestellten Linearen Programms. Nachdem die neue Cache-Partitionierung für die Klasse k bestimmt ist, werden die neu berechneten Größen der Cache-Bereiche an die lokalen Agenten, die eine lokale Änderung durchführen müssen, gesendet.

(e) Allokationsphase
In der Allokationsphase empfangen die lokalen Agenten die Ausgabe des Koordinators aus der Phase (d) und führen die vorgeschriebenen Änderung aus. Wie wir in Abschnitt 6.4 gesehen haben, gehen wir bei der Berechnung davon aus, daß zu einem Zeitpunkt die Anpassung für höchstens eine einzige Klasse vorgenommen wird. In unserer Implementation würde dies bedeuten,

daß die Koordinatoren sich absprechen und die Berechnungen und Anpassungen strikt nacheinander erfolgen müßten. Da dies jedoch einen zusätzlichen Overhead und eine langsamere Anpassung an Änderungen bewirken würde, wollen wir eine solche synchrone Abarbeitung vermeiden. Durch die gleichzeitige Berechnungen für mehrere Klassen kann es dazu kommen, daß ein Agent, der auf einem lokalen Rechner einen dedizierten Bereich einer bestimmten Größe für eine Klasse k_1 allozieren soll, nicht mehr genügend nicht-dedizierten Speicher vorfindet, da bereits der Agent einer anderen Klasse k_2 diesen Speicher reserviert hat. In diesem Fall alloziert der lokale Agent der Klasse k_1 den maximal möglichen Speicher und informiert den Koordinator über die Differenz zwischen der geforderten und der wirklichen Größe. Daraufhin aktualisiert der Koordinator seine lokalen Informationen, führt jedoch keine weitergehenden Aktionen durch. Dies ist möglich, da der Regelkreis im Bedarfsfall noch einmal durchlaufen wird und er dabei die aktualisierte Information berücksichtigen kann.

Komplexität der Berechnungen

Nachdem wir die Aktionen innerhalb der einzelnen Phasen des Regelkreises beschrieben haben, untersuchen wir nun genauer die Komplexität der notwendigen Berechnungen. Wir beschränken uns bei dieser Untersuchung auf die Phasen (b) und (d), da die anderen Phasen nur triviale Berechnungen durchführen.

Da der Aufwand für die Berechnung der mittleren Antwortzeit ebenfalls vernachlässigt werden kann, bestimmt die Berechnung der neuesten “linear unabhängigen” Meßpunkte die Komplexität der Phase (b). Diese Berechnung können wir durchführen, indem wir das zu den Meßpunkten korrespondierende lineare Gleichungssystem auf Singularität überprüfen. Die einfachste Möglichkeit, diesen Test durchzuführen, bietet der Gauß-Algorithmus. Da sich bei jeder Berechnung das Gleichungssystem nur inkrementell ändert (ein neuer Meßwert ersetzt einen alten), benutzen wir eine Optimierung, die diese Eigenschaft ausnutzt, um die *Worst-Cast*-Laufzeit des Algorithmus von $\mathcal{O}(N^3)$ auf $\mathcal{O}(N^2)$ zu reduzieren [GMW91].

In der Phase (d) müssen wir zur Approximation der Hyperebene ein lineares Gleichungssytem lösen. Ähnlich wie in Phase (b) ist auch dies prinzipiell mit Hilfe des Gauß-Algorithmus möglich. Auch hier können wir davon profitieren, daß zwischen zwei Berechnungen jeweils nur geringfügige Änderungen am Gleichungssytem durchgeführt werden. Nutzen wir daher wiederum das Verfahren von [GWM91] aus, so ist auch hier die *Worst-Cast*-Laufzeit quadratisch in der Rechneranzahl.

Schließlich muß der Koordinator in der Phase (d) zur Bestimmung der optimalen Partitionierung noch das Linear Programm lösen, das wir am Ende von Abschnitt 6.4 aufgestellt haben. Hierfür haben wir eine Implementierung [BDS97] des *Simplex*-Algorithmus gewählt. Obwohl die *Worst-Case*-Laufzeit dieses Verfahrens exponentiell in der Anzahl der Variablen und der Bedingungen ist, wird in [Schr86] gezeigt, daß die mittlere Laufzeit linear ist.

Um neben der theoretischen Komplexität auch die reale Ausführungszeit zu bestimmen, haben wir Messungen auf einer Workstation des Typs *SUN Sparc 4* durchgeführt und die Ergebnisse in der Tabelle 6.1 aufgeführt. Die Messungen des Overheads für Berechnungen, bei denen entweder ein lineares Gleichungssystem gelöst oder auf Singularität geprüft werden muß, wurden sowohl mit dem optimierten als auch mit dem einfachen Gauß-Algorithmus (in Klammern) durchgeführt.

Anzahl der Rechner	5	10	20	30	40	50
Lineare Unabhängigkeit (Phase b)	0.14 (0.05)	0.2 (0.2)	0.7 (1.0)	2.4 (3.5)	2.8 (8.0)	4.2 (15.0)
Ebenenapproximation (Phase d)	0.24 (0.1)	0.6 (0.4)	2.7 (2.5)	5.5 (7.0)	11.1 (18.0)	14.8 (40.0)
Lineares Programm (Phase d)	0.6	0.8	1.1	1.5	1.8	2.4
Gesamt	0.75	1.4	4.3	9.4	15.7	21.4

Tabelle 6.1: Berechnungsaufwand in Millisekunden für die verschiedenen Phasen

In der Tabelle erkennen wir, daß sich erst ab einer gewissen Rechneranzahl die bessere asymptotische Laufzeit des optimierten Gauß-Algorithmus auch in der realen Ausführungszeit positiv bemerkbar macht. Daher wählen wir je nach Rechneranzahl die günstigere Methode aus. Dies haben wir bei der Berechnung der Gesamtzeit berücksichtigt. Insgesamt erkennen wir an Hand der Tabelle 6.1, daß der Aufwand auf dem Koordinator für die Berechnung einer neuen Cache-Partitionierung relativ gering ist. Zusätzlich weisen wir nochmals darauf hin, daß diese Phasen nur durchlaufen werden müssen, wenn ein signifikanter Wechsel in der aktuellen Last oder in den Zielantwortzeiten auftritt. Wenn alle Klassen ihre Zielantwortzeiten erfüllen, verursacht der Koordinator keinen zusätzlichen Mehraufwand. Weiterhin können wir von einer Verteilung der verschiedenen Koordinatoren auf die Rechner profitieren. Im Falle einer ungleichmäßigen CPU-Auslastung im System können zusätzlich einfache Methoden der Lastbalancierung – wie z.B. die Migration von Koordinatorprozessen – benutzt werden, um eine Überlastung eines Rechners zu vermeiden.

6.6 Integration der kostenbasierten Caching-Heuristik

Bis jetzt ist noch keine Festlegung auf eine spezielle Ersetzungsstrategie für die Cache-Manager erfolgt. In Abschnitt 6.3 haben wir lediglich gefordert, daß eine Vergrößerung eines lokalen, dedizierten Cache-Bereichs für eine Klasse *k* zu keiner Reduktion der Antwortzeit für diese Klasse führen darf. Diese Eigenschaft sollte von allen in Kapitel 3 vorgestellten Cache-Heuristiken erfüllt werden.

Betrachten wir z.B. unsere kostenbasierte Heuristik und nehmen wir an, daß ein dedizierter Bereich der Klasse *k* so vergrößert wird, daß ein weiteres Objekt in den lokalen Bereich des Rechners *i* eingefügt werden kann. Durch dieses neue Objekt wird auf jeden Fall die lokale Cache-Trefferrate auf dem Rechner *i* erhöht und damit die lokale Antwortzeit der Klasse *k* reduziert. Je nachdem, ob sich das Objekt schon vorher auch im Cache eines anderen Rechners befunden hat, wird auch die Cache-Trefferrate der anderen Rechner beeinflußt. Nehmen wir zuerst an, daß das Objekt sich noch nicht im aggregierten Cache befunden hat, so führt das Caching zusätzlich zu einer Erhöhung der nichtlokalen Trefferrate der anderen Rechner und unter der Voraussetzung, daß ein nichtlokaler Cache-Zugriff schneller als ein Plattenzugriff ist, führt auch dies zu einer Reduktion der Antwortzeit der Klasse *k*. Hat sich hingegen das Objekt bereits vorher im aggregierten Cache befunden, so ändert sich die Antwortzeit der anderen Rechner nicht. Insgesamt wird jedoch – durch die Verbesserung der lokalen Klassenantwortzeit– die gesamte Antwortzeit der Klasse *k* in jedem Fall verbessert. Ähnlich können wir vorgehen um zu zeigen, daß auch für die egoistisch oder die altruistisch agierenden Heuristiken die oben angeführte Bedingung gilt.

Obwohl wir also prinzipiell jede der eingeführten Caching-Heuristiken benutzen könnten, haben wir in Kapitel 5 doch gezeigt, daß wir die beste Ausnutzung des zur Verfügung stehenden Caches durch die kostenbasierte Heuristik erreichen. Daher stellen wir in diesem Abschnitt eine Möglichkeit vor, wie wir unsere kostenbasierte Caching-Heuristik für die Benutzung innerhalb des zielorientierten Partitionierungsalgorithmus abändern können.

Da unser zielorientierter Partitionierungsalgorithmus mehrere Cache-Manager pro Rechner erlaubt (einen für die freie Klasse und höchstens einen für jede Zielklasse), müssen wir die kostenbasierte Ersetzungsstrategie entsprechend anpassen. Um eine korrekte Rangfolge – sowohl innerhalb der nicht-dedizierten Cache-Bereiche als auch innerhalb der verschiedenen dedizierten Cache-Bereiche – gewährleisten zu können, müssen wir für jedes Objekt jeweils die verschiedenen Klassenhitzen und auch die akkumulierte Hitze über alle Zugriffe mitprotokollieren. Bei den lokalen Klassenhitzen ist jedoch nur die Berücksichtigung der Klassen notwendig, für die auf dem lokalen Rechner auch ein dedizierter Cache-Bereich existiert. Ebenso brauchen wir auch bei der Berechnung der globalen Hitze für jedes Objekt nur dann die klassenspezifische Hitze mitzuprotokollieren, wenn in dem Gesamtsystem mindestens auf einem Rechner ein dedizierter Bereich für diese Klasse vorhanden ist. Eine weitere Einschränkung des Overheads können wir dadurch erreichen, wenn wir die Hitzeinformation des Objekts *p* für eine Klasse *k* nur dann mitprotokollieren, wenn wenigstens eine Anfrage dieser Klasse auch wirklich auf das Objekt *p* zugreift. Da dies jedoch im allgemeinen nicht a priori bestimmt werden kann, benutzen wir die bereits in Abschnitt 3.1.1 vorgestellte Methode, nach der wir die Hitzeinformation dynamisch kreieren und löschen können.

In dem Algorithmus 6.1 haben wir den Pseudocode für die Abarbeitung eines einzelnen Zugriffs einer Anfrage der Klasse *k* auf ein Objekt *p* beschrieben. Die Funktion *lookup* wird benutzt, um im lokalen Katalog nach einem Eintrag für das entsprechende Objekt zu suchen. Falls sich dieses im Cache des lokalen Rechners befindet liefert die Funktion einen Zeiger auf das entsprechende Objekt, ansonsten den NULL-Zeiger. Wenn sich das Objekt nicht im lokalen Cache befindet, lädt *get_page* das Objekt entsprechend der Speicherhierarchie des verteilten Cachings entweder aus dem Cache eines anderen Rechners oder aber von Platte. Die Methode *cacheID* liefert die Identität des lokalen Cache-Bereichs, in dem sich das entsprechende Objekt befindet und die Methode *update_heat* wird benutzt, um die Informationen über die lokale Hitze zu aktualisieren. Diese Methode kann entweder für die akkumulierte oder für die klassenspezifische Hitze aufgerufen werden. Da das Verändern der Hitze eine Veränderung des Nutzens bewirkt, paßt die Methode *new_position* die Rangfolge innerhalb des Caches entsprechend des neuen Nutzens an. Der aktuelle Nutzen wird auch von den beiden *insert*-Methoden berücksichtigt, um das Objekt an der korrekten Stelle im Cache einzufügen. Der Unterschied zwischen *insert* und *insert** besteht darin, daß bei *insert** das Objekt nur dann eingefügt wird, wenn der Nutzen des neu einzufügenden Objekts größer als der der verdrängten Objekte ist; ansonsten wird das Objekt gelöscht.

```
Object* request (int p, int k)
begin
    object=lookup(p);                                   Nachschauen im lokalen Katalog
    if (object = = NULL)) then                          Objekt befindet sich nicht im lokalen Cache
        page = get_page(p);                             Objekt von nichtlokalem Cache oder von Platte laden
        if (cache_k ≠ NULL) then                        Es existiert ein dedizierter Cache für
            object.update_heat(k);                          die Klasse k
            object.update_heat(accumulated);
            victim = cache_k.insert(object);
            cache_No-Goal.insert*(victim);
        else if (cache_k = = NULL) then                 Es existiert kein dedizierter Cache für
            object.update_heat(accumulated);                die Klasse k
            cache_No-Goal.insert(object);
        endif;
    else                                                Objekt befindet sich im lokalen Cache
        if (object.cacheID() = = k) then                Objekt befindet sich im dedizierten Cache
            object.update_heat(k);                          der Klasse k
            object.update_heat(accumulated);
            cache_k.new_position(object);
        else if (object.cacheID() ≠ No-Goal) then       Objekt befindet sich in irgendeinem
            object.update_heat(accumulated);                dedizierten Cache
        else if (cache_k ≠ NULL) then                   Objekt befindet sich im Cache der freien
            object.update_heat(k);                          Klasse, und es existiert ein dedizierter
            object.update_heat(accumulated);                Cache für die Klasse k
            victim = cache_k.insert(object);
            cache_No-Goal.insert*(victim);
        else if (cache_k = = NULL) then                 Objekt befindet sich im Cache der freien
            object.update_heat(accumulated);                Klasse, und es existiert ein dedizierter
            cache_No-Goal.new_position(object);             Cache für die Klasse k
        endif;
    endif;
    return object;                                      Zeiger auf das Objekt zurückliefern
end;
```

Algorithmus 6.1: Abarbeitung eines Objektzugriffs

Kapitel 7

Ausblick

Zum Ausblick auf mögliche Fortsetzungen dieser Arbeit betrachten wir nochmals die Annahmen unserer kostenbasierten Heuristik für verteiltes Caching bezüglich der Netzwerkarchitektur. Hierzu diskutieren wir zuerst die notwendigen Änderungen für eine Verallgemeinerung unserer Heuristik auf beliebige Topologien, und anschließend untersuchen wir den Einfluß der räumlichen Netzwerkausdehnung auf den Einsatz von verteiltem Caching im allgemeinen. Nach der Diskussion der aus diesen Betrachtungen resultierenden Einschränkungen wollen wir noch zwei Erweiterungsmöglichkeiten vorstellen. Analog zu der Sicherstellung von Performance-Zielen, für die wir in Kapitel 6 eine entsprechende Methode beschrieben haben, skizzieren wir einen Ansatz der auch Anforderungen bezüglich der Verfügbarkeit von Objekten berücksichtigt, und schließlich betrachten wir die Möglichkeit des Vorladens (*Prefetching*) von Objekten.

Erweiterung auf allgemeinere Netzwerktopologien

Betrachten wir zunächst die Einschränkungen unseres Modells bezüglich der Netzwerkarchitektur. Bisher haben wir immer ein lokales Netzwerk mit einer Bus-Architektur angenommen. Um jedoch einen immer größeren Durchsatz zu erzielen werden auch bei lokalen Netzwerken in wachsendem Maße Technologien (z.B. sogenannte *Switches* bei *Fast-Ethernet* oder *ATM*) genutzt, bei denen mehrere Rechner innerhalb desselben LANs gleichzeitig miteinander kommunizieren können. In solchen Netzwerken kann es vorkommen, daß durch eine spezielle Last die Verbindung zwischen einem Rechner A und einem Rechner B permanent überlastet ist, während die Auslastung der Verbindung zwischen Rechner A und Rechner C sehr gering ist. Neben der Last können auch die Systemparameter zu unterschiedlichen Auslastungen innerhalb des LANs führen. So erlauben z.B. marktübliche Switches den gleichzeitigen Anschluß von Rechnern mit *Fast-Ethernet*- oder *Ethernet*-Verbindung. Dies kann dazu führen, daß zwischen den Rechnern A und B eine 100 MBit/s Verbindung besteht, während die Bandbreite zwischen den Rechnern A und C nur 10 MBit/s beträgt. Beide Faktoren führen dazu, daß im allgemeinen die Kommunikationskosten zwischen den verschiedenen Paaren von Rechnern nicht mehr für das gesamte System einheitlich sind. Da es unter diesen Voraussetzungen nicht mehr gerechtfertigt ist, alle Zugriffe auf den nichtlokalen Cache einheitlich zu betrachten, entfallen einige Vereinfachungen, die wir bei der Herleitung der Formeln für den Nutzen eingeführt haben. Die Berechnung des Nutzens für ein Objekt wird dadurch zwar schwieriger, jedoch nicht unmöglich. So müssen wir in diesem Fall bei den in Abschnitt 3.4 hergeleiteten Formeln zusätzlich die Kosten berücksichtigen, die aus den

unterschiedlichen Kommunikationskosten zwischen den Rechnern resultieren. Da die Kosten zwischen zwei anderen Rechnern nicht durch rein lokale Beobachtung bestimmt werden können, müssen wir eine explizite Verteilung der betreffenden Informationen im Netzwerk durchführen. Hier können wir jedoch ausnutzen, daß sich diese Kosten im allgemeinen nicht sprunghaft ändern, und daher können wir diese Werte asynchron und nach Möglichkeit durch Piggybacking zwischen den Rechnern propagieren. Im allgemeinen Fall ist es auch nicht mehr ausreichend, nur die lokale und die globalen Hitze zu kennen, sondern wir benötigen für die Berechnung des Nutzens die lokalen Hitzen auf allen Rechnern. Um diese Information mit geringem Mehraufwand zu verteilen, können wir das Protokoll zur Berechnung der globalen Hitze so verallgemeinern, daß wir auf dem Heimatrechner jeweils – an Stelle der globalen Hitze – alle lokalen Hitzen mitgeführen und schwellwertbasiert aktualisieren. Diese Änderungen erlauben uns, die kostenbasierte Heuristik für verteiltes Caching auch in Netzwerken beliebiger Topologie zu benutzen. Um jedoch den Overhead zu reduzieren, sollten die spezielle Topologie (z.B. eine hierarchische Struktur) und die daraus resultierenden Vereinfachungen bei der Herleitung der Kostenformeln berücksichtigt werden.

Einsatz in einem Weitbereichsnetzwerk

Unabhängig von der Topologie stellt sich weiterhin die Frage, ob wir auch die räumliche Einschränkung auf ein lokales Rechnernetzwerk aufheben können. Der Einsatz unserer Heuristik innerhalb eines Weitbereichsnetzwerk (WAN) könnte z.B. sinnvoll sein, um bei einem verteilten HTTP–Server das regional unterschiedliche Zugriffsverhalten auf den einzelnen Rechnern zu berücksichtigen. Im Gegensatz zu LANs spielt jedoch bei WANs nicht nur die Übertragungsgeschwindigkeit, sondern auch die Ausbreitungsgeschwindigkeit auf dem Netzwerk eine Rolle. So bewegen sich elektrische bzw. optische Signale mit ungefähr 200000 km/s über ein Kabel [Tane97]. Hieraus folgt, daß – unabhängig von der Übertragungsgeschwindigkeit, die durch die Netzwerktechnologie beeinflußt wird – pro 200 km die Latenzzeit für einen Netzzugriff um jeweils eine Millisekunde anwächst. Betrachten wir nun z.B. einen verteilten HTTP-Server, dessen einzelne Rechner sich in Hamburg, München und Saarbrücken befinden und die jeweils untereinander durch Hochgeschwindigkeitsstrecken miteinander verbunden sind. Wird nun auf dem Rechner in Saarbrücken ein Zugriff auf ein Objekt initiiert, das nicht lokal vorhanden ist, so muß die Anfrage an den Heimatrechner, z.B. in Hamburg weitergeleitet werden. Alleine die Signallaufzeit für die Übermittlung der Anfrage und der Antwort beträgt in diesem Szenario ca. 7 ms. Nehmen wir darüber hinaus an, daß sich das Objekt nur im Cache des Rechners in München befindet, so muß in Hamburg ein *Forwarding* erfolgen und wir erhalten dadurch eine reine Signallaufzeit von mehr als 10 ms. Berücksichtigen wir auch noch den zusätzlichen Overhead, der in einer realen Umgebung durch die Übertragungsgeschwindigkeit, die Signalverstärkung und die lokale Abarbeitung auf den verschiedenen Rechner entsteht, so ist die Zeit für einen Zugriff auf den nichtlokalen Cache größer als die für einen Plattenzugriff. Somit ist die Grundvoraussetzung für den Einsatz von verteiltem Caching im allgemeinen – und daher auch für unsere kostenbasierte Heuristik – in einem WAN nicht mehr gegeben.

Verfügbarkeitsgarantien

Die in dieser Arbeit vorgestellte Methode für verteiltes Caching ist dadurch motiviert, daß wir das Potential preisgünstiger Standardrechner innerhalb eines lokalen Netzwerks ausnutzen wollen,

um so ein leistungsfähiges Gesamtsystem zu erstellen. Neben der Performance spielt jedoch bei vielen Anwendungen auch die Zuverlässigkeit der Rechner und damit die Verfügbarkeit der Objekte eine wichtige Rolle. Dieser Gesichtspunkt wird in dem von uns betrachteten lokalen Netzwerk noch wichtiger, da die verschiedenen Rechner eines LANs im allgemeinen nicht so gut administriert werden wie z.B. ein einzelner dedizierter Datenbank-Server. Daher treten Rechnerausfälle bzw. Offline-Zeiten von Rechnern häufiger auf. Analog zu der Erweiterung unserer kostenbasierten Heuristik um Performance-Ziele sind daher auch Garantien bezüglich der Verfügbarkeit der im System gespeicherten Objekte wünschenswert. In [Ried98] wird unser Prototyp für verteiltes Caching um die zielorientierte, dynamische Replikation von Objekten auf die Platten verschiedener Rechner erweitert. Dabei wird jeweils versucht mit einer minimalen Anzahl von Kopien die geforderte Verfügbarkeit sicherzustellen. Neuberechnungen und eventuell daraus resultierende Umverteilungen der Kopien werden bei diesem Verfahren jeweils dann notwendig, wenn sich entweder die Last oder der Systemzustand ändert, also ein Rechner ausfällt oder wieder verfügbar wird.

Vorladen von Objekten

Eine weiter denkbare Erweiterung unserer kostenbasierten Heuristik, die wir ansprechen wollen, betrifft das Vorladen (*Prefetching*) von Objekten. Während unsere Heuristik vollständig reaktiv arbeitet, d.h. es wird erst dann ein Objekte geladen, wenn ein entsprechender Zugriff erfolgt ist, wird beim Vorladen versucht, ein Objekt bereits dann in den Cache zu laden, wenn in naher Zukunft eine Anwendung auf dieses Objekt (sehr wahrscheinlich) zugreift. Durch dieses Vorladen werden Verzögerungen durch das synchrone Lesen des Objektes von Platte potentiell vermieden, und dies kann zu einer weiteren Reduktion der mittleren Antwortzeit führen. Das Hauptproblem bei dem Vorladen besteht darin, zum aktuellen Zeitpunkt Rückschlüsse über das zukünftige Referenzverhalten zu erhalten. Während Anwendungen im allgemeinen keine entsprechenden Informationen liefern, kennt z.B. der Optimierer eines Datenbanksystems das Referenzmuster einer Anfrage schon, bevor die Anfrage selbst durchgeführt wird. Um auch in solchen Situationen eine optimale Performance zu erzielen, könnten Verfahren zur Abstimmung zwischen Caching und Vorladen für einzelne Rechner [CFKL95, PGG*95, KrWe97], mit unserer Heuristik für verteiltes Caching kombiniert werden.

Anhang A

Verwendete Bezeichnungen

Systemparameter

B_i	Größe des Caches auf Rechner *i*. Ist dieser Wert für alle Rechner identisch, so benutzen wir einfach *B*.
D_i	Größe der Festplatte an Rechner *i*. Ist dieser Wert für alle Rechner identisch, so benutzen wir einfach *D*.
M	Gesamtanzahl der Objekte im System
N	Anzahl der Rechner im System
s_p	Größe des Objektes *p*
$\bar{s}$	mittlere Objektgröße
X	Matrix der Cache-Allokation (nur für statische Heurstiken)
Y	Matrix der Platten-Allokation (nur für statische Heuristiken)

Lastparameter

δ	Ähnlichkeit im Zugriffsverhalten (*Deviation*)
ξ	Ähnlichkeit im Zugriffsverhalten (*Correlation*)
ψ	Schräge der Plattenallokation
η	Schräge der Rechneraktivität
θ	Schräge der Zugriffsverteilung
λ	Gesamtankunftsrate von Zugriffen im System.Diese Ankunftsrate kann durch die zusätzliche Angabe einer Ressource (z.B. Netzwerk oder Platte) und einer Zugriffsart (Lese- bzw. Schreibzugriff) spezifiziert werden.
λ_{loc}	Ankunftsrate von Zugriffen auf einen einzelnen Rechner. Analog zur Gesamtankunftsrate kann auch hier eine Einschränkung auf eine Ressource oder eine Zugriffsart erfolgen.
ν	Verhältnis der Datengröße zur aggregierten Cache-Größe
π	Schreibwahrscheinlichkeit
σ	Zyklische Verschiebung im Zugriffsverhalten
f	Matrix der lokalen Objekthitzen
f_{ip}	Lokale Hitze des Objektes *p* auf Rechner *i*.

Zugriffskosten

Je nach betrachtetem Modell (statisch oder dynamisch) kann bei den folgenden Bezeichnungen jeweils die Angabe der Rechner entfallen.

$c_{lc,i}$	Kosten für einen lokalen Cache-Zugriff auf Rechner *i*
$c_{rc,i,j}$	Kosten für einen nichtlokalen Cache-Zugriff von Rechner *i* auf Rechner *j*
$c_{lc,i}$	Kosten für einen lokalen Plattenzugriff auf Rechner *i*
$c_{rd,i,j}$	Kosten für einen nichtlokalen Plattenzugriff von Rechner *i* auf Rechner *j*

Spezifikation der Ressourcen

ϱ_{disk}	Auslastung der Festplatte
S_{disk}	Mittlere Gesamtbedienzeit der Festplatte
$S_{rot\_latency}$	Mittlere Rotationsverzögerung der Festplatte
S_{seek}	Mittlere Suchzeit der Festplatte
$S_{transfer}$	Mittlere Transferzeit der Festplatte
T_{cyl_seek}	Zeit für die Armbewegung über einen Zylinder
T_{full_seek}	Zeit für die Armbewegung über alle Zylinder
T_{rot}	Zeit für eine komplette Rotation der Festplatte
v_{disk}	Transfergeschwindigkeit der Festplatte
Z	Gesamtanzahl der Zylinder der Festplatte

ϱ_{net}	Auslastung des Netzwerks
S_{net}	Mittlere Bedienzeit des Netzwerks
v_{net}	Transfergeschwindigkeit des Netzwerks

Parameter für die Hilfsprotokolle

Θ_{local}	Lokaler Schwellwert für die Berechnung der globalen Temperatur
Θ_{global}	Globaler Schwellwert für die Berechnung der globalen Temperatur
γ	Gewichtungsfaktor bei der Methode der exponentiellen Gewichtung
w	Fenstergröße bei der Methode der minimalen Fehlerquadrate

Parameter für die zielorientierte Erweiterung

$\lambda_{k,i}$	Ankunftsrate von Operationen der Klasse *k* auf dem Rechner *i*
K	Anzahl der Zielklassen
$LM_{k,i}$	Größe des lokal dedizierten Caches für die Klasse *k* auf dem Rechner *i*
$RM_{k,i}$	Größe des dedizierter Caches für die Klasse *k* auf allen von Rechner *i* verschiedenen Rechnern
$RT_{k,i}$	Mittlere Antwortzeit der Operationen der Klasse *k* auf dem Rechner *i*
RT_k	Mittlere Antwortzeit der Operationen der Klasse *k* über allen Rechner
RT_k^{goal}	Zielantwortzeit der Klasse *k*

Anhang B

Die Komplexität des Verteilten-Caching-Problems

In diesem Anhang wollen wir beweisen, daß das Verteilte-Caching-Problem (VCP) mit variabel großen Objekten (vgl. Definition 2.2) $\mathcal{NP}$-vollständig ist. Dazu konstruieren wir zunächst eine Sprache, mit der wir zeigen können, daß VCP in $\mathcal{NP}$ liegt und anschließend führen wir eine Reduktion auf das PARTITION-Problem durch, für das die $\mathcal{NP}$-Vollständigkeit bereits gezeigt wurde.

B.1 Liegt VCP in $\mathcal{NP}$?

Für den Beweis, daß VCP in $\mathcal{NP}$ liegt, definieren wir in einem ersten Schritt die Sprache VCP', die alle Wörter enthält, die ein Verteiltes-Caching-Problem codieren, für das eine Allokation existiert, deren Kosten geringer als eine vorgegebene Schranke ist. Dieses Problem weicht von unserem ursprünglichen Problem dadurch ab, daß wir nun eine Allokation suchen, die billiger ist als eine gegebenen Schranke, anstatt daß wir die Allokationskosten minimieren. Lösen wir jedoch dieses modifizierte Problem, so können wir mit Hilfe eines polynomiellen Algorithmus das ursprüngliche Verteilte-Caching-Problem lösen [LePa81].

Übernehmen wir die Notation aus Anhang A, und nehmen wir an, daß die Funktion α, Vektoren und Matrizen in das unäre System codiert, so können wir die Sprache VCP' über dem Alphabet Σ={I,@,c} folgendermaßen definieren. (Das Zeichen 'c' wird von der Funktion α als Trennzeichen benötigt):

VCP = {I^N@I^M@I^B@$\alpha(s)$@$\alpha(f)$@I^{clc}@I^{crc}@I^{cld}@I^{crd}@I^b | Es existiert eine Allokation *X*, die alle Nebenbedingungen der Definition 2.2 erfüllt und deren Kosten geringer als *b* sind.}

Um zu beweisen, daß diese Sprache in $\mathcal{NP}$ liegt, konstruieren wir eine neue Sprache VCP'', die sich von VCP' darin unterscheidet, daß wir an jedes Wort ein sogenanntes Zertifikat anhängen. Ein Zertifikat stellt dabei eine codierte Lösung des betrachteten Problems dar.

VCP'' = {I^N@I^M@I^B@$\alpha(s)$@$\alpha(f)$@I^{clc}@I^{crc}@I^{cld}@I^{crd}@I^b\$$\alpha(X)$ | Die Allokation *X* erfüllt alle Nebenbedingungen und die Kosten der Allokation sind geringer als *b*.}

Entsprechend eines Ergebnisses aus [LePa81] liegt VCP' in $\mathcal{NP}$, wenn wir die folgenden zwei Eigenschaften der Sprache VCP'' beweisen können:

(a) *Jedes Wort in VCP' ist polynomiell balanciert, d.h. die Größe des Zertifikats ist polynomiell in der Größe des restlichen Wortes beschränkt.*
Da das Zertifikat aus einer unären Codierung einer bool'schen $N \times M$ Matrix besteht und da das restliche Wort unter anderem eine unäre Codierung einer $N \times M$ Matrix mit nicht-negativen Koeffizienten (die Matrix *f* der Objekthitzen) enthält, kann das Zertifikat – bei einer geeigneten Wahl der Funktion α (z.B. entsprechend [LePa81]) – nie mehr als doppelt so groß sein wie der Rest des Wortes.

(b) *Es muß eine deterministische Turing-Maschine existieren, die die Sprache VCP'' in polynomieller Zeit entscheidet.*
Diese Forderung bereitet ebenfalls keine Schwierigkeiten, da wir nur die Formel aus der Definition 2.2 entsprechend der gewählten Kostenfunktion auswerten müssen. Dies ist jedoch für die beiden in Kapitel 2 eingeführten Kostenfunktionen jeweils in der Zeit $\mathcal{O}(N^2M)$ möglich.

Nachdem wir somit bewiesen haben, daß VCP' in $\mathcal{NP}$ liegt, folgt – wie bereits oben angesprochen – daß auch VCP ein Element von $\mathcal{NP}$ ist.

B.2 $\mathcal{NP}$-Vollständigkeit von VCP

Nachdem wir wissen, daß VCP in $\mathcal{NP}$ liegt, beschreiben wir nun eine Beweis für die $\mathcal{NP}$-Vollständigkeit. Dazu reduzieren wir das PARTITION-Problem, für das in [GaJo79] die $\mathcal{NP}$-Vollständigkeit gezeigt wurde, auf unser VCP-Problem. Die Eingabegrößen des PARTITION-Problems sind ein Menge *Q* und ein Vektor *w*, der jeweils ein Gewicht w_k für jedes Element *k* der Menge *Q* enthält. Das Problem besteht nun darin, eine Teilmenge *Q'* von *Q* zu bestimmen, so daß die Summe der Gewichte der Elemente in *Q'* gleich der Summe der Gewichte der Menge *Q\Q'* ist.

Auch hier formen wir das Problem zunächst in ein Sprachenentscheidungsproblem um. Dabei benutzen wir das gleiche Alphabet und die gleiche Codierungsfunktion wie in Abschnitt B.1.

$$\textbf{PARTITION} = \{ I^Q @ \alpha(w) \mid \exists\, Q' \subset Q: \ \sum_{k \in Q'} w_k = \sum_{k \in Q \setminus Q'} w_k \}$$

Für diese Sprache müssen wir nun eine Funktion τ konstruieren, die jede Eingabe in polynomieller Zeit in eine entsprechende Eingabe für das Verteilte-Caching-Problem VCP transferiert. Diese Funktion überprüft zuerst ob die Summe der Gewichte w_k ungerade ist. Ist dies der Fall, so ist eine Partitionierung in zwei gleichgewichtige Mengen unmöglich und daher liefert τ ein beliebiges Wort (z.B. das leere Wort), das nicht Element von VCP ist. Ist die Summe der Gewichte gerade, so berechnet τ die folgende Transformation:

$\tau: \Sigma^* \rightarrow \Sigma^*$

$\tau\,(I^Q @ \alpha(w)) = I^2 @ I^{|Q|} @ I^B @ \alpha(w) @ \alpha(f) @ I^0 @ I^0 @ I^1 @ I^1 @ I^0$

wobei $B = 1/2 \sum_{k \in Q} w_k$

und $f_{ij} = 1 / |Q|$ für alle $i = 1,..,N$ und $j = 1,..,M$.

Informal können wir diese Transformation folgendermaßen beschreiben: Wir betrachten ein Verteiltes-Caching-Problem in dem wir 2 Rechner und $|Q|$ Objekte besitzen. Die Größen der Objekte werden entsprechend der Gewichte der Elemente des PARTITION-Problems gewählt und die Größe der lokalen Caches ist auf beiden Rechnern gleich. Zusammen entspricht die Größe des aggregierten Caches gerade der Summe der Objektgrößen. Die Zugriffshäufigkeit ist für jedes Objekte auf jedem Rechner konstant und ungleich Null. Schließlich wählen wir die Zugriffskosten auf den lokalen und den nichtlokalen Cache jeweils gleich Null und ein Zugriff auf eine Festplatte (lokal oder nichtlokal) kostet eine Zeiteinheit. Unter diesen Voraussetzungen suchen wir die Cache-Allokationen, die die Kosten Null besitzen.

Es ist einfach einzusehen, daß diese Funktion τ von einer Turing-Maschine in polynomieller Zeit berechnet werden kann. Daher müssen wir nur noch zeigen, daß eine Partitionierung der Menge Q in zwei gleichgewichtige Mengen genau dann möglich ist, wenn eine Allokation für das transformierte Problem existiert, welche die Kosten Null besitzt.

Da ein Plattenzugriff jeweils eine Zeiteinheit kostet, ist eine Allokation mit Kosten Null nur dann möglich, wenn alle Objekte entweder über einen Zugriff auf den lokalen oder den nichtlokalen Cache erreicht werden können. Da ferner die Cache-Kapazität der beiden Rechner zusammen genau gleich der Summe der Objektgrößen ist, ist eine Allokation, die alle Objekte im Cache hält, nur dann möglich, wenn jedes Objekt genau einmal im Cache vorkommt. Entsprechend der Transformation τ sind die Caches beider Rechner gleich groß. Da wir in VCP annehmen, daß jeweils nur komplette Objekte in einem lokalen Cache gehalten werden können, ist es nur dann möglich alle Objekte im aggregierten Cache zu halten, wenn die Menge der Objekte so aufgeteilt werden kann, daß die Summen der Objektgrößen in den lokalen Caches gleich sind. Wird daher eine Allokation gefunden, die die Kosten Null hat, muß auch eine Lösung des PARTITION-Problems existieren. Haben wir das Verteilte-Caching-Problem gelöst, so können wir eine Lösung des PARTITION-Problems bestimmen, indem wir in der Menge Q' all jene Elemente zusammenfassen, die zu den Objekten korrespondieren, die auf dem Rechner 1 alloziert sind. Da dies wiederum in polynomieller Zeit möglich ist, haben wir den Beweis für die $\mathcal{NP}$-Vollständigkeit vom VCP abgeschlossen.

Anhang C

Detaillierte Beschreibung des Callback-Locking-Protokolls

In diesem Anhang stellen wir die kompletten Übergangstabellen für die Synchonisationprozesse des Zugriffmanagers aus Abschnitt 4.1.3 vor. Anhang C.1 enthält die Übergangstabellen für den lokalen Synchronisationsprozeß (LSP) und Anhang C.2 die entsprechenden Tabelle für den globalen Synchronisationsprozeß (GSP).

C.1 Der lokale Synchronisationsprozeß

Die Tabelle C.1 beschreibt die Übergangsfunktion des LSP. Bei der Ankunft einer Nachricht wird in Abhängigkeit vom Nachrichtentyp und des aktuellen Ausgangszustands ein Zielzustand bestimmt. Zusätzlich können bei einem Übergang weitere Aktionen (z.B. das Versenden einer Nachricht) initiiert werden. Die erlaubten Nachrichtentypen korrespondieren zu den entsprechenden Einträgen bei der Beschreibung des LSP in Abschnitt 4.1.3. Der Ausgangszustand wird beschrieben durch die lokale Warteschlange (*Queue*), den aktuellen Sperrmodus (*Lock*) und durch die Gültigkeit des Objektes (*Valid*). In der lokalen Warteschlange können Lese- (*R*), Schreibzugriffe (*W*) oder Callback-Anfragen (*C*) auf eine Bearbeitung warten. Ein *'X'* kann für einen einzelnen Zugriff oder für eine beliebige – jedoch nicht leere – Kombination von Zugriffen stehen. Als Sperrmodi erlauben wir neben den in Abschnitt 4.1.3 beschriebenen Lese- (*R–Lock*), Schreib- (*W-Lock*) und Migrationssperren (*M-Lock*) auch noch sogenannte Intention-Sperren (*I-Lock*). Eine Intention-Sperre signalisiert, daß bereits ein Prozeß auf dieses Objekt zugreifen will, er jedoch noch keine der anderen Sperren erworben hat. Dies ist in unserem Protokoll notwendig, um Deadlocks zu verhindern. Ohne solche Intention-Locks kann es z.B. vorkommen, daß sich zwei gleichzeitig laufende Schreibzugriffe bei der Durchführung des Callbacks gegenseitig blockieren. Zur einfacheren Schreibweise erlauben wir auf Lesesperren ein Inkrement bzw. Dekrement. Dies bedeutet, daß die Anzahl der momentan lesenden Zugriffe auf diesem Objekt um eins erhöht bzw. erniedrigt wird. Ist die Anzahl der lesenden Zugriffe nach einem Dekrement gleich Null, so wird die Lesesperre freigegeben (*No-Lock*). Durch die lokale Gültigkeit beschreiben wir, ob sich das Objekt im lokalen Cache befindet (*true*) oder nicht (*false*). Die Einträge des Zielzustands sind analog zu denen des Ausgangszustands definiert.

Zur Vereinfachung der Darstellung führen wir die folgende Notation ein. Das Zeichen '*' benutzen wir als Platzhalter für einen beliebigen Wert, und durch das Zeichen '|' beschreiben wir die Konkatenation auf Listen. Leere Felder des Zielzustands bedeuten, daß keine Veränderungen gegenüber dem Ausgangszustand aufgetreten sind. Zur Bezeichnung der beteiligten Rechner benutzen wir *I* für den Initiatorrechner einer Anfrage und *H* für den Heimatrechner des entsprechenden Objekts.

Nachrichten-typ	**Ausgangszustand**			**Zielzustand**			**Aktionen**
	Queue	**Lock**	**Valid**	**Queue**	**Lock**	**Valid**	
Read-Req	∅	I-, W- oder M-Lock	*	R			
Read-Req	∅	R-Lock	false	R			
Read-Req	X	*	*	X|R			
Read-Req	∅	No-Lock	false		I-Lock		•*Read-Home* an GSP senden
Read-Req	∅	No- oder R-Lock	true		R-Lock++		•Prozeß aktivieren
Read-Unlock	*	R-Lock	true		R-Lock--		
Read-Reply	*	I-Lock	false		R-Lock++	true	•Objekt in Cache einfügen •Prozeß aktivieren •*Cache-Ins* an GSP senden
Read-Reply	*	R-Lock	true				•Prozeß aktivieren •*Read-Ack* an GSP senden
Read-Forward	*	*	false				•*Forward-Failed* an GSP senden
Read-Forward	*	≠ W-Lock	true				•*Read-Reply* an I senden
Write-Req	∅	≠ No-Lock	*	W			
Write-Req	X	*	*	X|W			
Write-Req	∅	No-Lock	*		I-Lock		•*Write-Home* an GSP senden
Write-Reply	*	I-Lock	false		W-Lock	true	•Objekt in Cache einfügen •Prozeß aktivieren
Write-Reply	*	W-Lock	true				•Prozeß aktivieren
Callback	∅	R-Lock	*	C			
Callback[14]	R^+|W|X	R-Lock	*	R^+|C|W|X			
Callback	*	No-, I- oder M-Lock	true			false	•Objekt aus Cache löschen •*Callback-Reply* an GSP senden
Callback	*	No-, I- oder M-Lock	false				•*Callback-Reply* an GSP senden
Write-Unlock (I≠H)[15]	*	W-Lock	true		No-Lock		•*Write-Free* an GSP senden •Objekt aus Cache löschen
Write-Unlock (I=H)[15]	*	W-Lock	true				•*Write-Free* an GSP senden

Tabelle C.1: Übergangstabelle für den lokalen Callback-Prozeß

14: Der Eintrag R^+ in der Spalte Queue steht stellvertretend für einen einzelnen oder für mehrere Lesezugriffe. Somit wird die neu ankommende Callback-Anfrage vor den ersten Schreibzugriff in der Warteschlange eingefügt. Existiert kein Schreibzugriff, so wird die Callback-Anfrage an Ende der Warteschlange angehängt. Diese Vorgehensweise ist notwendig, um zu vermeiden, daß Schreibzugriffe zu Deadlocks führen können.

15: Bei diesen beiden Fällen ist die Übergangstabelle nicht allein durch den Nachrichtentyp und den Ausgangszustand bestimmt, sondern es muß zusätzlich noch berücksichtigt werden, ob der Initiatorrechner gleich dem Heimatrechner ist.

In der Tabelle C.2 ist eine Erweiterung der lokalen Übergangstabelle gezeigt, die es erlaubt, auch Objektmigrationen zu berücksichtigen. Die Bezeichnungen der Spalten und die Notationen sind identisch zu denen in Tabelle C.1.

Nachrichten-typ	Ausgangszustand			Zielzustand			Aktionen
	Queue	Lock	Valid	Queue	Lock	Valid	
Migration-Data[16]	∅	No-Lock	false		M-Lock	true	•Falls bedingte Einfüge-operation Objekt in den Cache einfügt hat
					No-Lock	false	•Falls bedingte Einfüge-operation Objekt gelöscht hat
Migration-Data	SONST						•Empfangenes Objekt löschen
Migration-Ack	*	M-Lock	true		No-Lock		•*Cache-Ins* an GSP
Migration-Ack	SONST						•*Migration-Failed* an GSP
Migration-Rej	*	M-Lock	true		No-Lock	false	•Objekt aus Cache löschen
Migration-Rej	SONST						

Tabelle C.2: Erweiterung der lokalen Übergangstabelle für Migrationen

C.2 Der globale Synchronisationsprozeß

Nachdem wir in dem vorangegangenen Abschnitt die Übergangstabelle des LSP präsentiert haben, stellen wir nun die entsprechende Tabelle des GSP vor (Tabelle C.3). Auch hier wird auf Grund eines Nachrichtentyps und eines Ausgangszustands ein neuer Zielzustand bestimmt, und eventuell werden weitere Aktionen eingeleitet. Die Spalten *Queue* und *Lock* sind analog zu den entsprechenden Spalten des LSP, nur daß wir an Stelle der lokalen jetzt die globale Warteschlange betrachten, und daß wir hier nur Lese- und Schreibsperren benötigen. Zusätzlich zu diesen Spalten ist jeder Zustand auch von dem Inhalt der Owner-Liste (*Owner*) abhängig. Zur Aktualisierung der Owner-Liste und zur Bestimmung der korrekten Aktion ist es häufig notwendig, den Sender einer Nachricht zu kennen. Wir nehmen an, daß der Sender jeweils bekannt ist und bezeichnen ihn in der Tabelle mit *S*. Neben diesem Rechner benötigen wir auch hier den Initiatorrechner (*I*) und den Heimatrechner (*H*).

Abschließend stellen wir für den GSP ebenfalls eine Erweiterung vor, die Objekt-Migrationen berücksichtigt (Tabelle C.4). Da bei einer Migration im allgemeinen Fall drei Rechner beteiligt sein können, müssen wir neben dem Initiator der Migration (*I*) und dem Heimatrechner (*H*) auch den Zielrechner (*Z*) der Migration kennen. Da dieser bei unseren Lastverteilungsalgorithmen bereits zum Startzeitpunkt der Migration bekannt ist, können wir annehmen, daß diese Information jeweils mit der entsprechenden Nachricht übermittelt wird.

16: In diesem Fall ist neben dem Nachrichtentyp und dem Ausgangszustand auch das Ergebnis der bedingten Einfügeoperation zur eindeutigen Bestimmung des Zielzustands notwendig. Bei der bedingten Einfügeoperation wird das Objekt in Abhängigkeit von seinem eigenen Wert und dem Wert der Ersetzungsopfer entweder in den Cache eingefügt oder gelöscht (vgl. Abschnitt 3.5).

Nachrichten-typ	Ausgangszustand			Zielzustand			Aktionen
	Queue	Owner	Lock	Queue	Owner	Lock	
Read-Home	∅	∅	No-Lock			R-Lock	•Objekt lokal lesen •*Read-Reply* an I senden
Read-Home	∅	O≠∅	No- oder R-Lock			R-Lock++	a) H∈O\{S} : •Objekt lokal lesen •*Read-Reply* an I senden b) H∉O\{S} : *Read-Forward* an einen Owner
Read-Home	∅	*	W-Lock	R			
Read-Home	∅	∅	R-Lock	R			
Read-Home	X	*	*	X|R			
Read-Forward-Failed	*	O\{S}≠∅	R-Lock				a) H∈O\{S} : •Objekt lokal lesen •*Read-Reply* an I senden b) H∉O\{S} : •*Read-Forward* an einen anderen Owner
Read-Forward-Failed	*	O\{S}=∅	R-Lock				•Objekt lokal lesen •*Read-Reply* an I senden
Read-Ack	*	S∈O	R-Lock			R-Lock--	
Cache-Ins	*	S∉O	R-Lock		O∪{S}	R-Lock--	
Cache-Del	*	S∈O	No- oder R-Lock		O\{S}		•evtl. *Unikat*-Nachricht an verbleibenden Owner
Write-Home	∅	∅	No-Lock			W-Lock	•Objekt lokal lesen •*Read-Reply* an I senden
Write-Home	∅	O≠∅	No-Lock			W-Lock	•*Callback* an alle Owner Callback-Zähler = |O|
Write-Home	∅	*	≠ No-Lock	W			
Write-Home	X	*	*	X|W			
Callback-Reply	*	*	W-Lock				•Callback-Zähler erniedrigen •Falls Callback-Zähler = 0 – Objekt lokal lesen – *Read-Reply* an I senden
Write-Free	*	∅	W-Lock		{H}	No-Lock	•evtl. Objekt sichern

Tabelle C.3: Übergangstabelle für den globalen Callback-Prozeß

Nachrichtentyp	Ausgangszustand			Zielzustand			Aktionen
	Queue	Owner	Lock	Queue	Owner	Lock	
Migration-Req	∅	{I}	No-Lock		∅	R-Lock	*Migration-Ack* an Knoten Z
Migration-Req	SONST				O\{I}		*Migration-Rej* an Knoten Z
Migration-Failed	*	*	R-Lock			R-Lock--	

Tabelle C.4: Erweiterung der globalen Übergangstabelle für Migrationen

Literaturverzeichnis

[ABF93a] B. Awerbuch, Y. Bartal, A. Fiat: *Heat&Dump: Competitive Distributed Paging*, IEEE Symposium on Foundations of Computer Science, 1993

[ABF93b] B. Awerbuch, Y. Bartal, A. Fiat: *Competitive Distributed File Allocation*, ACM Symposium on Theory of Computing, 1993

[ACP*95] T.E. Anderson, C.E. Culler, D.A. Patterson, the NOW team: *A Case for NOW (Network of Workstations)*, IEEE Micro, Vol. 15 (1), February 1995

[ADN*95] T.E. Anderson, M.D. Dahlin, J.M.Neefe, D.A. Patterson, D.S. Roselli, R.Y. Wang: *Serverless Network File Systems*, 15th Symposium on Operating Systems Principles, 1995

[ASA*95] M. Abrams, C.R. Standridge, G. Abdulla, S. Williams, E.A. Fox: *Caching Proxies: Limitations and Potentials*, 4th International World Wide Web Conference, 1995

[BaSh85] A. Barak, A. Shiloh: *A Distributed Load-balancing Policy for a Multicomputer*, Software - Practice and Experience, Vol. 15 (9), September 1985

[BCDM93] K. Brown, M. Carey, D. DeWitt, M. Metha, J. Naughton: *Managing Memory to Meet Multiclass Workload Response Time Goals*, 19th International Conference on Very Large Databases, 1993

[BCF*95] N.J. Boden, D. Cohen, R.E. Felderman, A.E. Kulawik, Ch.L. Seitz, J.N. Seizovic, W.-K. Su: *Myrinet: A Gigabit-per-Second Local-Area Network*, IEEE Micro, No. 15 (1), February 1995

[BCL96] K. Brown, M. Carey, M. Livny: *Goal Oriented Buffer Management Revisited*, ACM SIGMOD Conference, 1996

[BDS97] M. Berkelaar, J. Dirks, H. Schwab: *lp_solve library Version 2.1 and documentation*, verfügbar über ftp.es.ele.tue.nl/pub/lp_solve, 1997

[Best95] A. Bestavros: *Demand-based Document Dissemination to Reduce Traffic and Balance Load in Distributed Information Systems*, 7th IEEE Symposium on Parallel and Distributed Processing, 1995

[Best96] A. Bestavros: *Speculative Data Dissemination and Service to Reduce Server Load, Network Traffic and Service Time in Distributed Information Systems*, 12th IEEE International Conference on Data Engineering, 1996

[BFR92] Y. Bartal, A. Fiat, Y. Rabani: *Competitive Algorithms for Distributed Data Management*, ACM Symposium on Theory of Computing, 1992

[Bjor92] R.D. Bjornson: *Linda on Distributed Memory Multiprocessors*, Ph.D. Thesis, Yale University, Department of Computer Science, November 1992.

[BMCL94] K. Brown, M. Mehta, M. Carey, M. Livny: *Towards Automated Performance Tuning for Complex Workloads*, 20th International Conference on Very Large Databases, 1994

[BNS69] L.A. Belady, R.A. Nelson, G.S. Shedler: *An Anomaly in Space-Time Characteristics of Certain Programs Running in a Paging Machine*, Communications of the ACM, Vol. 12, 1969

[BoDe83] H. Boral, D.J. DeWitt: *Database Machines: An Idea Whose Time Passed? A Critique of the Future of Database Machines*, 3rd International Workshop on Database Machines, 1983

[Brow95] K.P. Brown: *Goal Oriented Memory Allocation in Database Management Systems*, Ph.D. thesis, University of Wisconsin, Madison, 1995

[BrSe91] I.N. Bronstein, K.A. Semendjajew: *Taschenbuch der Mathematik*, Teubner Verlagsgesellschaft, 25. Auflage, 1991

[CABK88] G. Copeland, W. Alexander, E. Boughter, T. Keller: *Data Placement in Bubba*, ACM SIGMOD Conference, 1988

[CDN*95] A. Chankhunthod, P.B. Danzig, C. Neerdaels, M.F. Schwartz, K.J. Worrell: *A Hierarchical Internet Object Cache*, Technical Report 95-611, Department of Computer Science, University of Southern California, Los Angeles, 1995

[CFKL95] P. Cao, E.W. Felten, A.R. Karlin, K. Li: *A Study of Integrated Prefetching and Caching Strategies*, ACM SIGMETRICS Conference, 1995

[CFW*95] J.-Y. Chung, D. Ferguson, G. Wang, Ch. Nikolaou, J. Teng: *Goal Oriented Dynamic Buffer Pool Management for Database Systems*, International Conference on Engineering of Complex Computer Systems, 1995

[ChSe91] D.M. Choy, P.G. Selinger: *A Distributed Catalog for Heterogeneous Distributed Database Resources*, 1st International Conference on Parallel and Distributed Information Systems, 1991

[CoGu93] P. Corrigan, M. Gurry: *ORACLE Performance Tuning*, O'Reilly & Associates, 1993

[CoGr90] D. Comer and J. Griffioen: *A New Design for Distributed Systems: The Remote Memory Model*, Summer USENIX Conference, 1990

[CYD97] M. Colajanni, Ph.S. Yu, D.M. Dias: *Scheduling Algorithms for Distributed Web Servers*, 17th International Conference on Distributed Computing Systems, 1997

[DeGr92] D. DeWitt, J. Gray: *Parallel Database Systems: The Future of High Performance Database Systems*, Communications of the ACM, Vol.35 (6), June 1992

[DKMT96] D.M. Dias, W. Kish, R. Mukherjee, R. Tewari: *A Scalable and Highly Available Web Server*, COMPCON 1996: 41st IEEE Computer Society International Conference: Technologies for the Information Superhighway, 1996

[DWAP94] M.D. Dahlin, R.Y. Wang, Th.E. Anderson, D.A. Patterson: *Cooperative Caching: Using Remote Client Memory to Improve File System Performance*, 1st Symposium on Operating Systems Design and Implementation, 1994

[EBBV95] Th.v. Eicken, A. Basu, V. Buch, W. Vogels: *U-Net: A User-Level Network Interface for Parallel and Distributed Computing*, 15th ACM Symposium on Operating Systems Principles, 1995

[EGLT76] K.P. Eswaran, J.N.Gray, R.A. Lorie, I.L. Taiger: *The Notions of Consistency and Predicate Locks in a Database System*, Communications of the ACM, Vol.19 (11), November 1976

[EKK97] A. Eickler, A. Kemper, D. Kossmann: *Finding Data in the Neighborhood*, 23rd International Conference on Very Large Databases, 1997

[FeZa91] E.W. Felten, J.Zahorjan: *Issues in the Implementation of a Remote Memory Paging System*, Technical Report 91-03-09, University of Washington, 1991

[FCL92] M.J. Franklin, M.J. Carey, M. Livny: *Global Memory Management in Client-Server DBMS Architectures*, 18th International Conference on Very Large Databases, 1992

[FGND93] D. Ferguson, L. Georgiadis, Ch. Nikolaou, K. Davies: *Satisfying Response Time Goals in Transaction Processing Systems*, 2nd International Conference on Parallel and Distributed Information Systems, 1993

[FMP*95] M.J. Feeley, W.E. Morgan, F.H. Pighin, A.R. Karlin, H.M. Levy, Ch.A. Thekkath: *Implementing Global Memory Management in a Workstation Cluster*, 15th ACM Symposium on Operating Systems Principles, 1995

[Fran96] M.J. Franklin, *Client Data Caching*, Kluwer, 1996

[GaJo79] M.R. Garey, D.S. Johnson: *Computers and Intractability: A Guide to the Theory of NP-completeness*, W. H. Freeman and Co, New York, 1979

[GiGr97] M. Gillmann, W. Groß: *Devsim*, Dokumentation zum Fortgeschrittenenpraktikums, Fachbereich Informatik, Universität des Saarlandes, 1997

[GMW91] Ph.E. Gill, W. Murray, M.H. Wright: *Numerical Linear Algebra and Optimization*, Volume 1, Addison-Wesley, 1991

[GoHu97] G. Goldszmidt, G. Hunt: *NetDispatcher: A TCP Connection Router*, IBM Research Report RC20853, 1997

[Gray79] J. Gray: *Notes on Database Operating Systems*, in *Operating Systems: An Advanced Course*, R.Bayer, R.M. Graham, G.Seegmüller (Editor), Springer, 1979

[Gray95] J. Gray: *Super-Servers: Commodity Computer Clusters Pose a Software-Challenge*, in: Datenbanksysteme in Büro, Technik und Wissenschaft, G. Lausen (Hrsg.), 1995

[GrGr97] J. Gray, G. Graefe: *The Five-Minute Rule Ten Years Later, and Other Computer Storage Rules of Thumb*, SIGMOD Record, Vol. 26 (4), December 1997

[GrPu87] J. Gray, F. Putzolu: *The Five Minute Rule for Trading Memory for Disk Accesses and the 10 Byte Rule for Trading Memory for CPU Time*, ACM SIGMOD Conference, 1987

[GwSe95] J. Gwertzman, M. Seltzer: *An Analysis of Geographic Push-Caching*, Workshop on Hot Operating Systems. 1995

[GwSe96] J. Gwertzman, M.I. Seltzer: *World-Wide Web Cache Consistency*, USENIX Technical Conference, 1996

[HePa96] J.L. Hennessy, D.A. Patterson: *Computer Architecture: A Quantitative Approach*, Second Edition, Morgan Kaufmann, 1996

[HKM*88] J.H. Howard, M.J. Kazar, S.G. Menees, D.A. Nichols, M. Satyanarayanan, R.N. Sidebotham, M.J. West: *Scale and Performance in a Distributed File System*, ACM Transactions on Computer Systems Vol.6 (1), 1988

[HMP89] A.R. Hurson, L.L. Miller, S.H. Pakzad: *Tutorial on Parallel Architectures for Database Systems*, IEEE Computer Society Press, 1989

[HuWo93] Y. Huang, O. Wolfson: *A Competitive Dynamic Data Replication Algorithm*, 9th International Conference on Data Engineering, 1993

[Jain91] R. Jain: *The Art of Computer Systems Performance Analysis*, John Wiley & Sons, 1991

[KBM94] E. Katz, M. Butler, R. McGrath: *A Scalable HTTP Server: The NCSA Prototype*, Computer Networks and ISDN Systems, No. 27, 1994

[KePr92] D.A. Keim, E.S. Prwawirohardjo: *Datenbankmaschinen – Performanz durch Parallelität*, Reihe Informatik, BI Wissenschaftsverlag, 1992

[Knut73] D.E. Knuth: *The Art of Computer Programming, Volume 3: Sorting and Searching*, Addison-Wesley, 1973

[Köni98] A.-Ch. König: *Ersetzungsstrategien für verteilten Hauptspeicher unter Berücksichtigung von klassenorientierten Performance-Zielen*, Diplomarbeit, Fachbereich Informatik, Universität des Saarlandes, 1998

[KrWe97] A. Kraiß, G. Weikum: *Vertical Data Migration in Large Near-Line Document Archives Based on Markov-Chain Predictions*, 23rd International Conference on Very Large Databases, 1997:

[KrWi94] B. Kröll, P. Widmayer: *Distributing a Search Tree Among a Growing Number of Processors*, ACM SIGMOD Conference, Minneapolis, 1994.

[LePa81] H.R. Lewis, C.H. Papadimitriou: *Elements of the Theory of Computation*, Prentice-Hall, 1981

[Lind93] B.G. Lindsay: *Object Naming and Catalog Management for a Distributed Database Manager*, 2nd International Conference on Distributed Computing Systems, 1981

[LLM88] M.J. Litzkow, M. Livny, M.W. Mutka: *Condor: A Hunter of Idle Workstations*, 8th International Conference on Distributed Computers, 1988

[LNS94] W. Litwin, M.-A. Neimat, D. Schneider: *RP*: A Family of Order Preserving Scalable Distributed Data Structures*, 20th International Conference on Very Large Databases, 1994

[LuAl94] A. Luotonen, K. Altis: *World-Wide Web Proxies*, World Wide Web Conference, 1994

[LWY93] A. Leff, J.L. Wolf, Ph.S. Yu: *Replication Algorithms in a Remote Caching Architecture*, IEEE Transactions on Parallel and Distributed Systems, Vol. 4 (11), 1993

[LWY96] A. Leff, J.L. Wolf, Ph.S. Yu: *Efficient LRU-Based Buffering in a LAN Remote Caching Architecture*, IEEE Transactions on Parallel and Distributed Systems, Vol. 7 (2), 1996

[LYW91] A. Leff, Ph.S. Yu, J.L. Wolf: *Policies for Efficient Memory Utilization in a Remote Caching Architecture*, 1st International Conference on Parallel and Distributed Information Systems, 1991

[MaSe97] S. Manley, M. Seltzer: *Web Facts and Fantasy*, USENIX Symposium on Internet Technologies and Systems, 1997

[MDP95] E.P. Markatos, G. Dramitinos, K. Papachristos: *Implementation and Evaluation of a Remote Memory Pager*, Technical Report FORTH/ICS, 1995

[Micr97] Microsoft: *Two Commodity Scaleable Servers: a Billion Transactions per Day and the Terra-Server*, White Paper from the Business Systems Division of Microsoft, Verfügbar über www.microsoft.com, 1997

[MNU97] K. Mehlhorn, St. Näher, Ch. Uhrig: *LEDA: Library of Efficient Data types and Algorithms*, Verfügbar über ftp.mpi-sb.mpg.de, 1997

[MöWe92] A. Mönkeberg, G. Weikum: *Performance Evaluation of an Adaptive and Robust Load Control Method for the Avoidance of Data-Contention Thrashing*, 18th International Conference on Very Large Databases, 1992

[Mull93a] S. Mullender: *Distributed Systems, 2nd Edition*, Addision-Wesley, 1993

[Mull93b] C.S. Mullins: *DB2 Developer's Guide, DB2 Performance Techniques for Applications Programmers*, Sams Publishing, 1993

[Myri97] Myrinet Products: http://www.myri.com/myrinet/product_list.html, Dezember 1997

[Noon89] J. Noonan: *Automated Service Level Management and its supporting Technologies*, Mainframe Journal, 1989

[NWO88] M.N. Nelson, B.B. Welch, J.K. Ousterhout: *Caching in the Sprite Network File System*, ACM Transactions on Computer Systems, Vol. 6 (1), 1988

[OOW93] E.J. O'Neil, P.E. O'Neil, G. Weikum: *The LRU-K Page Replacement Algorithm for Database Disk Buffering*, ACM SIGMOD Conference, 1993

[PGG*95] H.R. Patterson, G.A. Gibson, E. Ginting, D. Stodolsky, J. Zelenka: *Informed Prefetching and Caching*, 15th ACM Symposium on Operating System Principles, 1995

[PLC95] S. Pakin, M. Lauria, A. Chien: *High Performance Messaging on Workstations: Illinois Fast Messages for Myrinet*, Supercomputing, 1995

[RAC97] St.H. Rodrigues, Th.E. Anderson, D.E. Culler: *High-Performance Local Area Communication With Fast Sockets*, 3rd Symposium on Operating System Design and Implementation. USENIX, 1997

[Rahm93] E. Rahm: *Empirical Performance Evaluation of Concurrency and Coherency Control Protocols for Database Sharing Systems*, ACM Transactions on Database Systems, Vol. 18 (2), June 1993

[Rahm94] E. Rahm: *Mehrrechner-Datenbanksysteme, Grundlagen der verteilten und parallelen Datenbankverarbeitung*, Addison-Wesley, 1994

[Rahm97] E. Rahm: *Goal-oriented Performance Control for Transaction Processing*, 9th ITG/GI Conference, 1997

[RFG*89] E. Rahm, D. Ferguson, L. Georgiadis, Ch. Nikolaou, G.–W. Su, M. Swanson, G. Wang: *Goal-oriented Workload Management in Locally Distributed Transaction Systems*, IBM Research Report RC 14712, T.J. Watson Research Center, 1989

[Ried98] M. Riedewald: *Methoden zur Sicherstellung von Verfügbarkeitszielen durch dynamische Objektreplikation*, Diplomarbeit, Fachbereich Informatik, Universität des Saarlandes, 1998

[RoSe92] R. Roth, T. Setz: *LiPS: A System for Distributed Processing on Workstations*, Technischer Bericht SFB 124, Universität des Saarlandes, 1992

[RuWi94] Ch. Ruemmler, J. Wilkes: *An Introduction to Disk Drive Modeling*, IEEE Computer 27 (3), 1994

[SaHa96] P. Sarkar, J. Hartman: *Efficient Cooperative Caching using Hints*, 2nd Symposium on Operating System Design and Implementation, 1996

[Schr86] A. Schrijver: *Theory of Linear and Integer Programming*, Wiley, 1986

[Schw96] H.D. Schwetman, *CSIM18 – The Simulation Engine*, Winter Simulation Conference, 1996

[SGDM94] V.S. Sunderam, G.A. Geist, J. Dongarra, R. Manchek: *The PVM Concurrent Computing System: Evolution, Experiences, and Trends*, Journal of Parallel Computing Vol.20 (4), 1994

[SiKö98] M. Sinnwell, A.Ch. König: *Managing Distributed Memory to Meet Multiclass Workload Response Time Goals*, submitted for publication, 1998

[SiWe97] M. Sinnwell, G. Weikum: *A Cost-Model-Based Online Method for Distributed Caching*, 13th International Conference on Data Engineering, 1997

[Stan97] O. Stanke: *Implementierung und Vergleich unterschiedlicher Verwaltungsverfahren für einen globalen Katalog in einem verteilten Datenbanksystem*, Diplomarbeit, Fachbereich Informatik, Universität des Saarlandes, 1997

[Stru98] M. Struwe: *Implementierung eines verteilten HTTP-Servers*, Diplomarbeit, Fachbereich Informatik, Universität des Saarlandes, 1998

[Sun88] Sun Microsystems Inc.: *Network Programming*, 1988

[SWZ94] P. Scheuermann, G. Weikum, P. Zabback: *Disk Cooling in Parallel Disk Systems*, IEEE Data Engineering Bulletin Vol.17 (3), 1994

[Tane92] A.S. Tanenbaum: *Modern Operating Systems*, Prentice-Hall, 1992

[Tane97] A.S. Tanenbaum: *Computernetzwerke, Dritte Auflage*, Prentice-Hall, 1997

[Tran97] Transtec: *Gesamtkatalog Unix und WindowsNT*, Herbst 1997

[VBW94] R. Vingralek, Y. Breitbart, G. Weikum: *Distributed File Organization with Scalable Cost/Performance*, ACM SIGMOD Conference, 1994

[Venk96] S. Venkataraman: *Global Memory Management for Multi-Server Database Systems*, Ph.D. thesis, University of Wisconsin, Madison, 1997

[VJV*97] G.M Voelker, H.A. Jamrozik, M.K.Vernon, H.M. Levy, E.D. Lazowska: *Managing Server Load in Global Memory Systems*, ACM SIGMETRICS Conference, 1997

[VLN95] S. Venkataraman, M. Livny, J.F. Naughton: *The Impact of Data Placement on Memory Management for Multi-Server OODBMS*, 11th International Conference on Data Engineering, 1995

[VLN97] S. Venkataraman, M. Livny, J.F. Naughton: *Memory Management for Scalable Web Servers*, 13th International Conference on Data Engineering, 1997

[VNL98] S. Venkataraman, J.F. Naughton, M. Livny: *Remote Load-Sensitive Caching for Multi-Server Database Systems*, 14th International Conference on Data Engineering, 1998

[WHMZ94] G. Weikum, Ch. Hasse, A. Moenkeberg, P. Zabback: *The COMFORT Automatic Tuning Project*, Information Systems, Vol. 19 (5), 1994

[WBE96] M. Welsh, A. Basu, Th. v. Eicken: *Low-Latency Communication over Fast Ethernet*, EuroPar96 Conference, 1996

[WBE97] M. Welsh, A. Basu, Th. v. Eicken: *ATM and Fast Ethernet Network Interfaces for User-level Communication*, 3rd International Symposium on High Performance Computer Architecture, 1997

[Zabb94] P. Zabback: *I/O-Paralelität in Datenbanksystemen*, Dissertation 10629, ETH Zürich, 1994

[Zenn97] Ch. Zenner: *NetClient: Eine detaillierte Ethernet-Simulation*, Dokumentation zum Fortgeschrittenenpraktikum, Fachbereich Informatik, Universität des Saarlandes, 1997

[Zipf49] G.K. Zipf: *Human Behavior and the Principles of Least Effort*, Addison-Wesley, 1949