

# Universität Stuttgart

## Fakultät Informatik

Diplomarbeit Nr. 2604

### **Caching und Prefetching for Efficient Read Access to Multidimensional Wave Propagation Data on Disk**

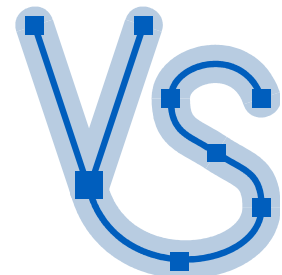
Philipp Häuser

Prüfer: Prof. Dr. K. Rothermel  
Betreuer: Dipl.-Inf. Steffen Maier  
Beginn am: 01.05.2007  
Beendet am: 31.10.2007  
CR-Nummer: D.4.2, H.3.3, C.2.1

Deutscher Titel: Caching und Prefetching mit dem Ziel des effizienten Lesezugriffs  
auf mehrdimensionale Wellenausbreitungsdaten von Festplatte



Institut für Parallele und  
Verteilte Systeme (IPVS)  
Abteilung Verteilte Systeme  
Universitätsstr. 38  
70569 Stuttgart





# Abstract

The NET (Network Emulation Testbed) project has established an emulation system for computer networks at the Distributed Systems department of the University of Stuttgart. One purpose of the emulation system is the realistic emulation of MANETs (mobile ad hoc networks), including node mobility and resulting communication channel quality between nodes. Since the prediction of wave propagation is not possible in real-time, precomputed attenuation values between all locations in the 2D simulation grid are used. Since the size of the wave propagation data may exceed the available main memory, it is stored on secondary storage and read from disk periodically.

In this work, caching methods that take advantages of previously read wave propagation data are developed and evaluated. Clustering is a technique used for the formation of larger blocks of data in the cache. It is explained and evaluated. For the special case of deterministic node mobility, the sequence of the queries to the cache is known beforehand. For this case, an implementation for optimal cache replacement is developed. For indeterministic node mobility, replacement heuristics are developed and analysed. The connection between node mobility patterns, query distances and cache hits and misses is explained. The results are presented through graphs showing the cache hit and miss rates. Some examples for the distribution of queries in the cache are presented. The feasibility of using application specific knowledge within the current NET framework is evaluated. Connections between the movement of mobile nodes and the resulting cache queries for wave propagation data are examined.

The results of this work include the following findings: For clustering, i.e. the grouping of data into blocks, the cluster size must be carefully selected. Large clusters increase the amount of cache memory wasted on unused data, whereas small clusters create unnecessary I/O overhead. A precomputed optimal replacement order can be used for emulation scenarios with deterministic node mobility. Application knowledge, such as the position of roads, is helpful for anticipating motion. Anticipating node movements can help optimising the cache replacement. The knowledge is layered: The scenario is at the top layer, the wave propagation data with only the grid coordinates at the bottom layer. Nonlinear mathematical interpolation of the mobile nodes on the lowest knowledge level is not exact enough to yield acceptable results. Other approaches are to collect statistical data about the distribution of the nodes in a scenario over several runs or to determine hot spots in the motion patterns. Simple triangular interpolation of the mobile node positions on the lowest knowledge layer also does not yield acceptable results.

The content of this thesis is written in German language.



# Inhaltsverzeichnis

<b>1</b>	<b>Einführung</b>	<b>1</b>
1.1	Motivation . . . . .	1
1.2	Network Emulation Testbed (Umgebung) . . . . .	2
1.3	Aufgabenstellung . . . . .	3
1.4	Struktur der Arbeit . . . . .	4
<b>2</b>	<b>Verwandte Arbeiten</b>	<b>5</b>
<b>3</b>	<b>Grundlagen</b>	<b>9</b>
3.1	Virtueller Speicher . . . . .	9
3.2	Seiteneretzungsalgorithmen . . . . .	9
3.2.1	Compulsory Misses und Conflict Misses . . . . .	10
3.2.2	Lokale und Globale Seiteneretzungsstrategien . . . . .	10
3.2.3	Online- und Offline-Algorithmen . . . . .	11
3.3	Kriterien für Seiteneretzungsalgorithmen . . . . .	11
3.4	Betrachtete Seiteneretzungsalgorithmen . . . . .	11
3.4.1	OPT/MIN (Optimal) . . . . .	12
3.4.2	RANDOM (Randomisiert) . . . . .	12
3.4.3	FIFO (First In First Out) . . . . .	12
3.4.4	FIFO Second Chance . . . . .	12
3.4.5	LRU (Least Recently Used) . . . . .	13
3.4.6	NRU (Not Recently Used) . . . . .	13
3.5	Knotenmobilität und Wellenausbreitung in NET . . . . .	14
3.5.1	Allgemeines . . . . .	14
3.5.2	Clustering . . . . .	15
3.5.3	Trajektorien und Traces . . . . .	17
3.6	Abarbeitung der Traces in CacheSim . . . . .	18
<b>4</b>	<b>Konzepte für Ersetzungsstrategien</b>	<b>21</b>
4.1	Deterministische Knotenbewegung . . . . .	21
4.1.1	OPT nach Belady . . . . .	21
4.1.2	OPT nach Mattson et al. . . . .	22
4.1.3	OPT nach Sugumar, Abraham . . . . .	27
4.1.4	Auswahl des OPT-Algorithmus . . . . .	29

4.2	Indeterministische Knotenbewegung . . . . .	29
4.2.1	RANDOM . . . . .	30
4.2.2	FIFO und Varianten . . . . .	31
4.2.3	LIL (Linear Interpolated Lookahead) . . . . .	33
4.2.4	Vorläufiger Vergleich der Ansätze . . . . .	38
4.2.5	Hotspot-Prefetching . . . . .	39
4.2.6	Lernendes Hotspot-Prefetching . . . . .	39
4.2.7	Aufteilung des Caches . . . . .	40
<b>5</b>	<b>Evaluation</b>	<b>41</b>
5.1	Simulationsframework für diese Arbeit . . . . .	41
5.2	Bezeichnung und Eigenschaften der verwendeten Traces . . . . .	43
5.2.1	Bezeichnung der Traces . . . . .	43
5.2.2	Ablauf der Simulation . . . . .	44
5.2.3	Eigenschaften der Traces . . . . .	44
5.2.4	Auswirkungen des Clusterings . . . . .	46
5.3	Deterministische Knotenbewegung . . . . .	51
5.3.1	Parameter . . . . .	51
5.3.2	Prototyp . . . . .	51
5.3.3	Bewertung . . . . .	51
5.4	Indeterministische Knotenbewegung . . . . .	52
5.4.1	Parameter . . . . .	52
5.4.2	Analyse . . . . .	53
5.4.3	Bewertung . . . . .	53
<b>6</b>	<b>Einbindung in die NET-Umgebung</b>	<b>57</b>
6.1	Schichtenprinzip . . . . .	57
6.2	Caching-Strategien . . . . .	57
6.3	OPT-Sequenzen . . . . .	58
<b>7</b>	<b>Zusammenfassung und Ausblick</b>	<b>59</b>
7.1	Zusammenfassung . . . . .	59
7.2	Schlußfolgerung . . . . .	61
7.3	Ausblick . . . . .	61
	<b>Literaturverzeichnis</b>	<b>63</b>

# Abbildungsverzeichnis

1.1	Architektur von NET . . . . .	2
1.2	Dämpfungsdaten in NET . . . . .	4
3.1	Darstellung des Simulations-Koordinatensystems mit Dämpfungswerten (markiert durch D) . . . . .	14
3.2	Darstellung eines Clusters mit Kantenlänge 3 . . . . .	16
3.3	Abarbeitung der Traces . . . . .	19
4.1	OPT nach Mattson et al. . . . .	23
4.2	Mögliche Datenstruktur für OPT . . . . .	26
4.3	Prinzip von OPT nach Sugumar, Abraham . . . . .	27
4.4	Mögliche Datenstruktur für FIFO, FIFO Second Chance, LRU . . . . .	32
4.5	Prinzip von LIL, Trajektorie . . . . .	34
4.6	Beispiel zu LIL . . . . .	37
5.1	Architektur der Komponenten für die Simulation . . . . .	45
5.2	TNR-Distanzen graph_walk-50-1-000-0 . . . . .	46
5.3	TNR-Distanzen random-50-1-000-0 . . . . .	47
5.4	Cache Hits graph_walk-50-1-000-0, Cluster Size 1 . . . . .	47
5.5	Cachegrösse und Cache Hits bei unterschiedlichen Cachegrößen, OPT . . . . .	48
5.6	Cachegrösse und Cache Hits bei unterschiedlichen Clustergrößen, FIFO . . . . .	49
5.7	Cachegrösse und Cache Hits bei unterschiedlichen Clustergrößen, FIFO, Detailansicht . . . . .	49
5.8	Mehrere Läufe von graph-walk-20-1-000, FIFO, im Vergleich . . . . .	50
5.9	Mehrere Läufe von graph-walk-20-1-000, FIFO, im Vergleich . . . . .	50
5.10	Cache Hits random-50-1-000-0, Cluster Size 1 . . . . .	54
5.11	Cache Hits random-50-1-000-0, Cluster Size 1 . . . . .	54
5.12	Cache Hits graph_walk-50-1-000-0, Cluster Size 1, Log. Achsen . . . . .	56
6.1	Schichtenprinzip der Simulation . . . . .	58



# Kapitel 1

## Einführung

### 1.1 Motivation

Mark Weiser prägte 1991 in [Wei91] den Begriff des „ubiquitären Computing“. Dieser Begriff steht für „Allgegenwart der Informationsverarbeitung“ und beschreibt die Integration von Informationstechnik in Alltagsprodukte, die insbesondere über Funktechnik miteinander kommunizieren.

Die Realisierung des „ubiquitären Computing“ ermöglicht neue Produkte und Dienste, die Probleme lösen und das Leben angenehmer machen können. Insbesondere am Arbeitsplatz, wo es oft um die Kommunikation und Koordination einer grösseren Anzahl von Kunden, Mitarbeitern, Lieferanten und Waren geht, wären große Steigerungen in der Effizienz der Abläufe denkbar.

„Ubiquitäres Computing“ benötigt kleine und leistungsfähige mobile Geräte, die miteinander kommunizieren, um die bereitgestellten Dienste zu erbringen. Es stellt sich nun die Frage, wie diese Kommunikation vonstatten gehen soll. Eine Realisierung mittels drahtgebundener LANs kommt bei den meisten Anwendungen wegen der Mobilität und mangels einer entsprechenden Infrastruktur vor Ort nicht in Frage. Eine Möglichkeit für die Kommunikation sind WLANs. Mittels zentralisierter fest installierter WLAN-Netzinfrastruktur lässt sich bei einigen Anwendungen die Konnektivität herstellen. Bei anderen Anwendungen ist es jedoch wünschenswert oder sogar unbedingt nötig, ohne zentrale Infrastruktur oder mit nur sehr wenigen fixen Komponenten auszukommen. Für diese Anwendungen bieten sich mobile ad-hoc Netzwerke (MANETs) an, hier werden über dynamische Routingprotokolle die anderen Netzteilnehmer in Kooperation mitbenutzt, um Daten zu empfangen und auch zu den richtigen Empfängern zu übertragen. Beispiele für solche Netzwerke finden sich heute schon im Automobilbereich für die Kommunikation von Fahrzeugen untereinander zum Informationsaustausch über aktuelle Verkehrsdaten, Unfälle, Fahrbanzustand und vieles mehr.

Bei MANET-Protokollen in der Entwicklung oder Erprobung besteht ein besonders großer Bedarf für den Test von Konzepten und Implementierungen. Ein Test unter realen Bedingungen mit physischen Knoten ist in der Praxis besonders in frühen Phasen der Protokollentwicklung oft nicht sinnvoll, bezahlbar oder machbar. Deshalb ist die Leistungsbewertung von Protokollen durch Simulation in diesem Bereich gefragt. Das Network Emulation Testbed (NET) Projekt ist hierfür ein Werkzeug.

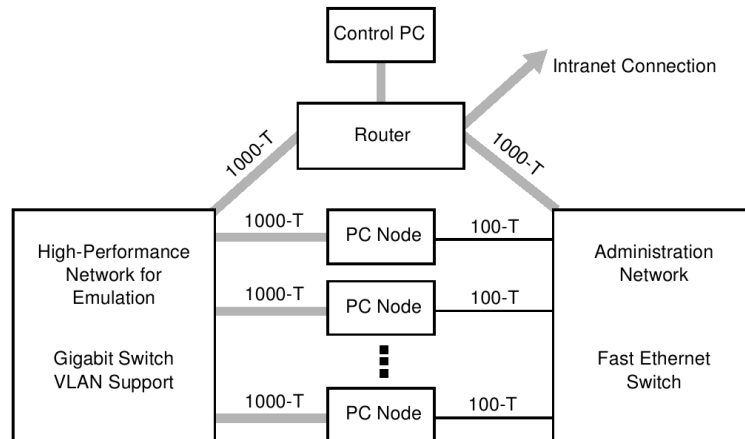


Abbildung 1.1: Architektur von NET

Im NET-Projekt existiert eine Datenbank mit Wellenausbreitungsdaten, aus der die Kanalqualitäten im Funkverkehr der Netzteilnehmer untereinander nachgeschlagen werden. Aus Performanzgründen ist diese Datenbank mit einem Cache versehen. Im Rahmen dieser Arbeit werden Ersetzungsstrategien für diesen Cache behandelt, um den Cache möglichst optimal zu nutzen. Es wird außerdem auf den Zusammenhang zwischen der Knotenmobilität und den resultierenden Cache-Anfragen sowie auf die Bildung von Datenblöcken der Ausbreitungsdaten eingegangen.

## 1.2 Network Emulation Testbed (Umgebung)

Diese Diplomarbeit findet im Rahmen des NET Projekts statt. Das NET Projekt ermöglicht den Test und die Leistungsbewertung von Kommunikationsprotokollen und verteilten Anwendungen unter anderem im Bereich der MANETs.

NET [HLR02] [SHB<sup>+</sup>03] besteht prinzipiell aus einem Cluster-System mit 64 Knoten („PC Nodes“), die mit einer flexiblen Netzwerktopologie verbunden sind (siehe Abbildung 1.1). Jeder Clusterrechner kann einen oder mehrere Knoten (z.B. aus dem Internet) repräsentieren, auf denen Protokolle und Anwendungen unverändert ausgeführt werden [MHR07]. Die Netzwerktopologie kann beliebig eingerichtet werden. Es können verschiedene Netzwerktypen emuliert werden, von WANs (Wide Area Networks) bis zu hochdynamischen mobilen ad-hoc Netzwerken. Paketverluste, Paketverzögerung und Bandbreite auf den emulierten Verbindungen werden auf den einzelnen Knoten festgelegt.

Für die Emulation der Knotenmobilität verwendet NET gespeicherte Daten über Straßen und Points of Interest, die von den mobilen Knoten nach festlegbaren Mustern besucht werden. Aus diesen Daten werden die Positionen der Knoten berechnet. Ein zentraler Kontrollrechner („Control PC“) steuert die Emulation und liefert Informationen zur Wellenausbreitung zwischen den Knoten in Form von Antworten auf entsprechende Anfragen.

Die Berechnung der Wellenausbreitung während eines Experimentes ist, auch bei Berücksichtigung nur eines Teils der Faktoren, in Echtzeit nicht möglich. Deshalb wurde die Dämp-

fung zwischen allen möglichen Knotenpositionen auf der festgelegten Fläche eines Stadtzentrums in NET vorausberechnet. Wegen der Größe der Wellenausbreitungsdaten sind diese auf einem Sekundärspeicher (d.h. Festplatte) abgelegt. Die Vorberechnung der Daten erfolgte wie in Abbildung 1.2 beschrieben.

Während jedem Experiment wird die Bewegung der Knoten simuliert, die Dämpfungsdaten für jedes Paar von Knoten werden periodisch von der Festplatte gelesen. Anhand der Wellenausbreitungsdaten können die Eigenschaften der Kanäle zwischen den Knoten festgelegt werden. Spezielle Kernelmodule auf den Emulationsrechnern sorgen dafür, dass die eingerichteten virtuellen Kanäle die passenden Eigenschaften auf ihren Schichten 2 und 3 bekommen, es wird die Latenz, Bandbreite und Fehlerrate der Kanäle festgelegt.

## 1.3 Aufgabenstellung

Um MANETs realistisch emulieren zu können, muss die Mobilität der Knoten und die resultierende Qualität der Kommunikationskanäle berücksichtigt werden.

Die Qualität von Funk-Kommunikationskanälen zwischen zwei Netzknoten wird von vielen Faktoren beeinflusst, unter anderem von:

- Entfernung der Kommunikationspartner voneinander
- Leistung des Senders, Empfindlichkeit des Empfängers
- Abschattung der Funksignale (Gebäude etc.)
- Reflexion der Signale an Oberflächen
- Streuung von Signalen
- Beugung von Signalen
- Kanalauslastung, Vorhandensein von anderen Netzknoten
- Hintergrundrauschen und Störungen

In dieser Arbeit sollen Methoden für Prefetching und Caching entwickelt und bewertet werden. Da die Gesamtmenge der Wellenausbreitungsdaten, die während eines Emulationslaufes gelesen wird, größer sein kann als der verfügbare Hauptspeicher, ist ein Cache-Ersetzungsalgorithmus notwendig. Der Algorithmus soll Anwendungswissen, d.h. die Bewegung der Knoten, benutzen, um zu bestimmen, welche Datenelemente in der Zukunft nochmals gebraucht werden könnten. Im Spezialfall deterministischer Knotenmobilität ist sogar eine optimale Ersetzungsreihenfolge bestimmbar, da die zukünftigen Cache-Anfragen im Voraus bekannt sind. Im allgemeinen Fall der nichtdeterministischen Knotenmobilität soll eine Ersetzungsheuristik entwickelt werden. Eine Bewertung soll ihre Effizienz im Vergleich zur optimalen Ersetzung zeigen.

Die Bewertungen können durch Simulation erfolgen. Für die genannten zwei Fälle der Mobilität soll ein Prototyp entwickelt werden, der die entsprechenden Ersetzungsalgorithmen enthält. Der Prototyp soll unter Linux laufen und die Seiten im Speicher sperren, um zu verhindern, dass das Betriebssystem Datenelemente im Cache auslagert.

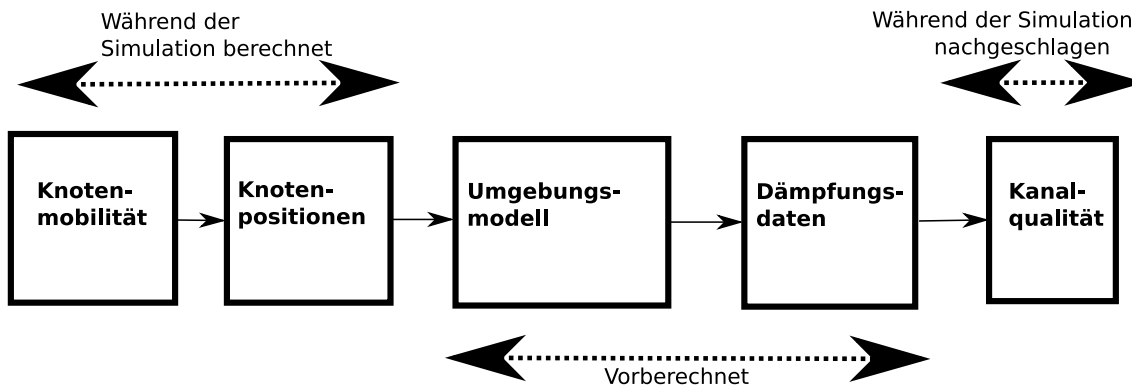


Abbildung 1.2: Dämpfungsdaten in NET

## 1.4 Struktur der Arbeit

Nach der Vorstellung der verwandten Arbeiten in Kapitel 2 werden in Kapitel 3 die Grundlagen dieser Arbeit beschrieben: Virtueller Speicher, Caches und Seitenersetzungsalgorithmen. Es wird außerdem auf die Besonderheit des Clusterings eingegangen. In Kapitel 4 werden zunächst einige Konzepte zur Bestimmung einer optimalen Offline-Knotenersetzungsstrategie beschrieben, d.h. die Bestimmung findet nach der Simulation bei Vorliegen aller Anfragen an den Cache statt. Anschließend werden die Möglichkeiten für möglichst gute Cache-Ersetzungsentscheidungen im Online-Fall erörtert. Im Offline-Fall findet die Bestimmung der Cache-Ersetzungen über eine Heuristik statt, ohne dass Daten über zukünftige Anfragen vorliegen. Es folgt die Evaluation in Kapitel 5. Zunächst wird der im Rahmen dieser Arbeit entstandene Simulator „CacheSim“ beschrieben. Der Simulator wendet festgelegte Ersetzungsstrategien bei seinen Läufen an. Außerdem bestimmt er einige Kennzahlen und die optimale Ersetzungsreihenfolge. Es wird jeweils der in der Implementierung gewählte Ansatz zur Problemlösung vorgestellt. Dann werden die Ergebnisse der Simulation der Ersetzungskonzepte im deterministischen und nichtdeterministischen Fall evaluiert, präsentiert und bewertet. In Kapitel 6 geht es um die mögliche Einbindung der Ansätze und Algorithmen in das vorhandene NET Framework. Eine Zusammenfassung und Schlussfolgerung findet sich im abschliessenden Kapitel 7.

# Kapitel 2

## Verwandte Arbeiten

Die Arbeiten [SW06] und [GWZ07] beschreiben das Radiowellen-Ausbreitungsmodell des Netzwerksimulators ns2.

[SW06] vergleicht die Simulationsergebnisse des ADOV-Routingprotokolls bei verschiedenen Ausbreitungsmodellen. Es wird ein Wellenausbreitungsmodell vorgestellt, das auf Basis von Raytracing funktioniert. In diesem Modell wird der Pfad der von jedem Sender emittierten Energie durch das Simulationsszenario berechnet. Die Daten werden in Form von Pfaden (d.h. Sequenzen von Vektoren) gespeichert statt als Werte an bestimmten Punkten. Werte von Punkten, an denen keine genaue Berechnung vorliegt, werden interpoliert. Fragen der Organisation der Daten auf Festplatte und Fragen des Caching werden in [SW06] und [GWZ07] nicht behandelt.

[Gri03] beschreibt die Berechnung von Funk-Wellenausbreitungsdaten für Mobilfunk in urbanen Gebieten und mögliche Vereinfachungen. Der Fokus der Arbeit liegt in der Verwendung von Echtzeit, hier wird deshalb ebenfalls nicht auf Caching und Datenorganisation eingegangen.

Viele Arbeiten zum Thema Caching befassen sich mit dem HTTP Protokoll im Zusammenhang mit Proxy-Caches. In HTTP ist ein „Cache-Control“-Header vorgesehen, der Informationen zum Caching enthält, wie z.B. „must-revalidate“, „no-cache“ oder „max-age“. [RFC 2616]. Aus dieser Information kann ein HTTP-Cache Anhaltspunkte für die Ersetzung von Daten im Cache ziehen. In der Praxis reichen diese Informationen jedoch nicht aus, um eine Ersetzungsentscheidung bestmöglichst festzulegen.

In Paper [PB03] werden verschiedene Ersetzungsstrategien bei Web-Caches behandelt. Anfragen an einen Web-Cache unterscheiden sich in ihrer Struktur deutlich von den im Rahmen dieser Arbeit vorkommenden Anfragen mit Wellenausbreitungsdaten. Werte, wie die auch in der vorliegenden Arbeit erhobenen TNR-Distanzen, können auch bei Web-Caches erfaßt werden, in [PB03] findet sich eine kurze Diskussion über den Zusammenhang zwischen nötiger Cachegröße und den vorkommenden TNR-Distanzen: Bei Web-Caches ist es mit den heutigen Preisen für Massenspeicher ohne Weiteres möglich, die Caches so groß zu dimensionieren, dass auch einfachste Ersetzungsstrategien zu einer sehr hohen Verwendung der gespeicherten Objekte im Cache führen. Die Autoren nennen keine Simulationsergebnisse oder Vergleichswerte unterschiedlicher Ersetzungsstrategien. Anwendungswissen wird nur begrenzt für Ersetzungsentscheidungen verwendet, einige der vorgestellten Strategien benutzen die zeitliche Verteilung

oder die Größe und Häufigkeit des Zugriffs auf bestimmte Objekte zur Klassifikation. Es wird kein Wissen aus den Nutzdaten extrahiert, im Fall eines Web-Caches könnten dies beispielsweise im Cache vorhandene Hyperlinks sein. In der vorliegenden Diplomarbeit wird auch geprüft, ob die Verwendung von Anwendungswissen über die Bewegung der mobilen Knoten bei den Ersetzungsentscheidungen der Dämpfungswerte bessere Ergebnisse erzielt. Die Problematik der Cache-Konsistenz wird in [PB03] angesprochen. In der vorliegenden Diplomarbeit ist dieser Aspekt wegen der statischen Dämpfungswerte nicht relevant.

Web-Caches haben die Eigenschaft, dass die zu cachenden Daten unterschiedlich groß sind und die Kosten ihrer Beschaffung (z.B. in Form von monetären Kosten oder Latenzen) unterschiedlich sind. Es sind somit Strategien wie GDS (Greedy-Dual Size) [CI] möglich, bei denen anhand der Objektgröße und Beschaffungskosten, versehen mit einem Mechanismus zur Alterung der Gewichte, Kandidaten für Ersetzungen gefunden werden. Beim Caching von Dämpfungswerten im Rahmen dieser Diplomarbeit gibt es die Attribute Größe und Kosten jedoch nicht. Dies verlangt eine genauere Betrachtung der verbleibenden Attribute, um eine sinnvolle Ordnung nach dem „Wert“ von Elementen im Cache zu ermöglichen.

In [WLRW04] wird der Traffic des Peer-to-Peer Filesharing Protokolls FastTrack analysiert, um u.a. Ersetzungsstrategien zu finden, mit denen Internet-Provider (ISPs) ihre eigenen Caches für FastTrack-Traffic betreiben können. FastTrack ermöglicht den Austausch von Dateien zwischen Nutzern. Durch Caching seitens des ISPs kann teure Bandbreite eingespart werden und die Transfargeschwindigkeit für die Nutzer gesteigert werden. Im FastTrack-Netzwerk tauschen, sehr vereinfacht gesagt, die Teilnehmer fehlende Teile des zu verteilenden Files blockweise miteinander aus, bis das komplette File vorliegt. Bei Caching für FastTrack gibt es eine Reihe von Möglichkeiten, wie ein Cache operieren kann. Die Granularität des Caches kann unterschiedlich sein: Es ist möglich, den Cache fileweise zu organisieren, genauso kann ein Cache auch auf Blockebene funktionieren, was auch das Caching von nur Teilen eines Files ermöglicht. Es werden im Paper verschiedene Cache-Ersetzungsstrategien wie LRU, MINS (Minimum Size) und GDS verglichen. Die Ergebnisse der Arbeit sind jedoch wegen der unterschiedlichen Eigenschaften des Anwendungsfalls nur zu einem kleinen Teil auf die Cache-Ersetzung im NET-Framework übertragbar.

In [Smi] wird tracebasierte Analyse („Trace-Driven Analysis“) verwendet, um die Performance von LRU-Disk-Caches anhand einiger aus dem Praxiseinsatz von IBM 360/370 Mainframes gewonnener Traces zu bestimmen. Bei der Analyse konnte in begrenztem Umfang Anwendungswissen verwendet werden, da die Trace-Einträge auch Informationen über den aufrufenden Prozeß und dessen Benutzer enthalten. So ist eine Klassifizierung nach Batchjobs, interaktiver Arbeit, Paging, Datenbankzugriffen, Zugriffen vom System u.a. möglich. Es wird auf die unterschiedlichen Eigenschaften der Klassen und das entsprechende Verhalten des Caches aufgrund der unterschiedlichen Lokalität der Zugriffe eingegangen. Der Einfluß der Blockgrößen im Cache auf den Erfolg des Caches wird diskutiert. One-Block Lookahead, eine einfache Prefetching-Methode, wird für die unterschiedlichen Klassen bewertet. Die Strategie „purge behind“, bei der nach dem Lesen eines Blocks der vorangegangene Block – sofern im Cache – sofort aus dem Cache gelöscht wird, wird vorgestellt und evaluiert. In [Smi] werden zahlreiche weitere Aspekte im Zusammenhang mit Disk-Caches behandelt, die jedoch für den Anwendungsfall des Caches für Dämpfungswerte nicht relevant sind. So wird genauer auf mögliche Positionen des Caches auf dem Pfad, den die Daten zurücklegen, eingegangen, sowie auf die

Verteilung der Zugriffe und Daten auf unterschiedliche Festplatten. Die Position des Caches ist in der derzeitigen NET-Architektur nicht variabel. Mit einer Verteilung der Zugriffe auf unterschiedliche Festplatten wird sich die Bereitstellung der Dämpfungswerte bei den beobachteten Zugriffsmustern nicht beschleunigen lassen, da der Engpaß in der Verarbeitung bei allen günstigen Clustergrößen in der Verarbeitung der Daten liegt. Eine Diskussion von anderen Ersetzungsstrategien als LRU findet nicht statt.

[DFJ<sup>+</sup>96] betrachtet clientseitiges Caching in einem Client-Server-Datenbanksystem. Es wird, als Neuigkeit gegenüber der üblicherweise ausgenutzten temporären oder räumlichen Lokalität, auch die semantische Lokalität von Elementen betrachtet. Die Anfragen werden variablenweise anhand ihrer prädikatenlogischen Bestandteile zerlegt. Dann werden in den verschiedenen Dimensionen der unterschiedlichen Variablen Wertebereiche gebildet. Der betrachtete Anwendungsfall unterscheidet sich jedoch sehr vom in dieser Arbeit betrachteten Cache mit Dämpfungswerten. Die Datensätze der Dämpfungswerte sind wesentlich kleiner als die in [DFJ<sup>+</sup>96] untersuchten Datenelemente. Es wäre voraussichtlich aufgrund des höheren Overheads bei Millionen von Dämpfungswerten nicht zu erwarten, dass die im Paper vorgestellte Strategie für den Anwendungsfall erfolversprechend ist.



# Kapitel 3

## Grundlagen

### 3.1 Virtueller Speicher

Die Grundidee eines virtuellen Speichers ist, dass der Programmcode und die Daten, die von einem Computer gleichzeitig in dessen RAM-Adressraum bearbeitet und adressiert werden können, größer sein dürfen als dessen tatsächlich vorhandener physischer RAM-Speicher.

Dazu wird ein großer virtueller Adressraum erstellt, das Betriebssystem bestimmt dann über die Abbildung von virtuellen Adressen auf physische Adressen die Position der abgelegten Informationen. Diese Abbildung wird meist von einer speziellen Einheit, der Memory Management Unit (MMU), erledigt. Der virtuelle Adressraum wird in kleine Einheiten (typischerweise zwischen einigen hundert Byte und einigen Kilobytes), sogenannte Seiten (engl. pages), aufgeteilt. Im physischen Speicher heißen die Einheiten Seitenrahmen oder Seitenkacheln bzw. page frames. Eine Seitentabelle hält für alle Seiten die dazugehörigen physischen Adressen.

Die Daten im virtuellen Adressraum können sich entweder eingelagert im schnellen Primärspeicher („Cache“, meist das RAM des Rechners) oder ausgelagert im langsameren, aber wesentlich größeren Sekundärspeicher (meist die Festplatte bzw. „Swap“) befinden. Das Betriebssystem kümmert sich um das Einlagern und Auslagern der richtigen Seiten. Ein present- bzw. absent-Bit für jeden Rahmen in der Seitentabelle ermöglicht es zu bestimmen, welche Rahmen gerade eingelagert und welche ausgelagert sind. Wird ein Seitenrahmen angefragt, der nicht in den Primärspeicher eingelagert ist, so wird ein Seitenfehler (page fault, cache miss) ausgelöst, der Rahmen wird dann eingelagert. Ist ein angefragter Seitenrahmen im Speicher enthalten, so spricht man von einem Seitentreffer (cache hit).

### 3.2 Seitenersetzungsalgorithmen

[Tan01] beschreibt verschiedene Seitenersetzungsalgorithmen – hier eine Auswahl der für diese Arbeit relevanten Algorithmen.

Wir gehen hier davon aus, dass jedes gelesene Datum auch in den Cache eingelagert wird, d.h. Zugriffe auch nur über den Cache möglich sind. Wann immer ein Cache verwendet wird, dessen Größe kleiner ist als die Gesamtgröße der zu fassenden Daten, stellt sich das Problem, welche Daten im Cache vorzuhalten sind. Die Problematik der Behandlung von während des

Betriebs des Caches modifizierten („dirty“) Elementen entfällt, da die Daten des im Rahmen dieser Arbeit untersuchten Caches während der Simulation nicht geändert werden. Mangels Änderungen können auch keine Inkonsistenzen zwischen Primär- und Sekundärspeicher auftreten.

Spätestens in dem Moment, in dem der Cache komplett mit gespeicherten Elementen gefüllt ist, und ein neues Element angefragt wird, das sich nicht im Cache befindet, muss entschieden werden, welches der vorhandenen Elemente ausgelagert (Push) wird, um das neue Element an seiner Stelle in den Cache einzulagern.

Dieses Problem tritt in den verschiedensten Anwendungsgebieten der Computertechnik auf, nicht nur bei der RAM-Verwaltung. Ein Beispiel ist der Festplattencache. Beim Festplattencache wird versucht, die durchschnittliche Zugriffszeit auf die langsame Magnetplatte (Sekundärspeicher) zu beschleunigen, indem ein schneller RAM-Speicher (Primärspeicher) zwischen den Anfrager (z.B. das Betriebssystem) und die Magnetplatte geschaltet wird. Die Wahl eines geeigneten Seitenersetzungsalgorithmus ist wichtig, damit möglichst günstige Entscheidungen bei der Seitenersetzung getroffen werden. Bei ungünstigen Entscheidungen wird ein häufiger Zugriff auf den Sekundärspeicher nötig, was den Zugriff wesentlich verlangsamt.

### 3.2.1 Compulsory Misses und Conflict Misses

„Cache Misses“ (Seitenfehler) lassen sich wie folgt klassifizieren: „Compulsory Misses“ („Unvermeidliche Seitenfehler“) sind Cache Misses, die auch bei einem Cache, der unendlich groß ist, auftreten würden. Jedes erstmalige Auftreten eines Elements erzeugt einen solchen Cache Miss, der dadurch unvermeidlich ist und unabhängig von der gewählten Ersetzungsstrategie und deren Erfolg auftritt.

Der Gegensatz zum „Compulsory Miss“ ist der „Conflict Miss“. Diese Art von Seitenfehlern tritt immer dann auf, wenn ein Miss entsteht bei einem Element, das in der Vergangenheit schon einmal im Cache war, aber inzwischen wieder gelöscht wurde und deshalb noch einmal geladen werden muss. Diese Art von Misses ist durch ausreichende Cache-Größe und natürlich durch die richtige Ersetzungsreihenfolge und -strategie vermeidbar.

### 3.2.2 Lokale und Globale Seitenersetzungsstrategien

Wenn in einem Computersystem mit mehreren laufenden Prozessen eine Seite ersetzt werden muss, so wählt ein lokaler Seitenersetzungsalgorithmus eine Seite desselben Prozesses für die Ersetzung. Nach einer globalen Strategie würde systemweit eine Seite für die Ersetzung gesucht. Hier wäre es also auch möglich, dass eine Seite eines anderen Prozesses ersetzt wird. Der Vorteil einer globalen Strategie ist, dass der vorhandene Cache allen Prozessen bei Bedarf zur Verfügung steht. Ein Nachteil ist, dass sich die Prozesse gegenseitig „stören“ können, z.B. würde ein Prozess, der auf immer andere große unzusammenhängende Speicherbereiche in kurzer Zeit zugreift, dazu führen, dass bei anderen Prozessen mehr page faults auftreten.

### 3.2.3 Online- und Offline-Algorithmen

Ist die Sequenz der Speicherzugriffe auch in die Zukunft bis zum Ende des Programmlaufes bekannt, so kann ein sogenannter Offline-Algorithmus das Problem der Entscheidung lösen, welche Seite auszulagern ist: Es wird die Seite ausgelagert, auf welche am weitesten in der Zukunft wieder zugegriffen wird. Es kann gezeigt werden, dass dieses Verfahren, in der Literatur OPT genannt [Bel66], optimale Entscheidungen trifft. Die entstandene Sequenz von Ersetzungsentscheidungen führt zu einer Minimierung der Zugriffe auf den Sekundärspeicher. Das OPT-Verfahren kann dazu verwendet werden, die untere Grenze der nötigen Zugriffe auf den Sekundärspeicher zu finden. Dadurch ist ein Vergleich und eine Bewertung von Algorithmen möglich.

In üblichen Computersystemen werden in der Praxis Online-Algorithmen für die Aus- und Einlagerungsentscheidungen verwendet. Diese Algorithmen zeichnen sich dadurch aus, dass ihnen zum Zeitpunkt der Entscheidungen keine Informationen über zukünftige Speicherzugriffe zur Verfügung stehen. Diese Situation ist in den meisten realen Computerprogrammen gegeben, dort kann der Computer zu einem bestimmten Zeitpunkt noch nicht vorhersehen, auf welche Seiten in Zukunft zugegriffen wird.

## 3.3 Kriterien für Seitenersetzungsalgorithmen

Als Kriterien für die Auswahl von Seitenersetzungsalgorithmen können folgende Aspekte gelten:

- Anzahl der Cache Misses

Meist ist das oberste Ziel einer Ersetzungsstrategie, die Anzahl der Cache Misses niedrig zu halten, deshalb ist dieser Wert oft die wichtigste Richtgröße. Als Vergleichswert kann der Wert einer OPT-Simulation dienen.

- Speicherverbrauch bzw. Platzkomplexität des Algorithmus

Die Algorithmen brauchen unterschiedlich viel Speicher für die nötigen Daten zur Verwaltung des Cache-Inhaltes. Hier ist ein Vergleich sinnvoll, da eingesparter Speicherplatz nutzbringend für andere Dinge – naheliegend ist eine Vergrößerung des Cache – verwendet werden kann.

- Rechenaufwand bzw. Zeitkomplexität des Algorithmus

Manche Algorithmen benötigen nur sehr geringen Rechenaufwand, andere haben eine sehr hohe Komplexität. Durch geschickte Implementierung, z.B. mit Hashtabellen, kann bei einigen Ansätzen der Rechenaufwand deutlich reduziert werden.

## 3.4 Betrachtete Seitenersetzungsalgorithmen

Die Seitenersetzungsalgorithmen unterscheiden sich in ihrem Verhalten, wenn der Cache-Speicher voll ist und ein vorhandenes Element im Cache zugunsten eines neu angefragten Elements überschrieben werden muss.

### 3.4.1 OPT/MIN (Optimal)

Der MIN Algorithmus verwendet das obengenannte OPT-Verfahren. Dieses ersetzt immer diejenigen Elemente im Cache, welche erst am weitesten in der Zukunft wieder verwendet werden. Es kann gezeigt werden, dass dieses Verfahren zur minimalen Anzahl von Seitenfehlern führt.

### 3.4.2 RANDOM (Randomisiert)

Der Random Algorithmus ersetzt ein zufällig ausgewähltes Element und setzt das neu aufzunehmende Element an dessen Stelle.

### 3.4.3 FIFO (First In First Out)

Das Kernstück dieses Algorithmus ist eine Warteschlange mit allen Seiten, die sich im Primärspeicher befinden. Seiten, die neu in den Primärspeicher aufgenommen werden, hängt der Algorithmus an den Kopf der Schlange an. Muss ein Element wegen Füllung des Caches aus diesem gelöscht werden, so wird immer das Element am Ende der Warteschlange freigegeben. Die Elemente rücken also vom Kopf zum Ende der Schlange durch, wo sie dann gelöscht werden. Dadurch bleibt jedes Element bei einer Größe des Caches von  $n$  Elementen genau  $n$  Löschungen lang abrufbar, weil es Stück für Stück mit jeder Ersetzung bis zum Ende durchrückt. Auch wenn ein Element während des Durchlaufens der Schlange aufgerufen wird, wird seine Position in der Liste nicht verändert.

### 3.4.4 FIFO Second Chance

Der FIFO-Algorithmus hat das Problem, dass auch häufig verwendete Elemente in der FIFO-Schlange aufrücken und bei Erreichen des Endes gelöscht werden. Der FIFO Second Chance Algorithmus ist eine Erweiterung von FIFO, um das Problem wie folgt abzumildern:

Jedes Element im Cache bekommt ein R (für Referenced) Bit, das bei Aufnahme in den Cache auf 0 gesetzt wird. Wird ein Element nach der Aufnahme in den Cache nochmals referenziert, während es im Cache ist, so wird das R-Bit auf 1 gesetzt.

Erreicht ein Element im Cache das Ende der Schlange und das R-Bit ist 0, so wurde das Element folglich seit seiner Aufnahme nicht wieder referenziert. Es wird gelöscht. Ist das R-Bit jedoch 1, so bekommt das Element eine „weitere Chance“: Das R-Bit wird auf 0 gesetzt und das Element an den Kopf der Schlange versetzt.

Diese Vorgehensweise führt dazu, dass jedes Element, welches während der  $n$  Ersetzungen (d.h.  $n$  Cache Misses) dauernden Aufenthaltsdauer im Cache nochmals referenziert wird, wieder an das Ende der Schlange gesetzt wird. Elemente, die folglich während des daran anschließenden Durchlaufens der Schlange nicht verwendet werden, werden am Ende der Schlange gelöscht.

Statt der Realisierung als Schlange wäre auch ein zusätzliches „letzte Referenzierung“-Datenfeld in jedem Element möglich, anhand dessen die Ordnung der Elemente stattfindet. Dieser Ansatz ist jedoch unter Umständen teurer.

### 3.4.5 LRU (Least Recently Used)

Die LRU-Strategie (Least Recently Used) geht davon aus, dass ein bisher häufig benutztes Element auch in Zukunft häufig verwendet werden wird. Für viele reale Anwendungen trifft diese Annahme zu, weshalb die Strategie meist recht hohe Cache-Hit Werte im Vergleich zu den bestmöglichen OPT-Werten liefert.

Der Ansatz ist relativ rechenintensiv: Jedes Element im Primärspeicher hat einen Zähler, der den Zeitpunkt (in Zeitschritten) seiner letzten Verwendung enthält. Bei allen Zugriffen auf Elemente wird der Zähler des zugegriffenen Elements aktualisiert. Ist der Primärspeicher voll, so wird immer das Element mit dem niedrigsten Zählerstand ersetzt, d.h. das Element, auf das am weitesten in der Vergangenheit zugegriffen wurde. Für LRU ist, wenn keine Spezialhardware vorhanden ist, eine nach letzter Zugriffszeit (also nach dem Zählerstand) geordnete Liste der Elemente nötig. Mittels dieser Liste kann das Element mit der am weitesten zurückliegenden Zugriffszeit effizient gefunden werden. Immer, wenn auf ein Element zugegriffen wird, erhält dieses den Listenplatz am Kopf der Schlange. Also wird diese Liste ständig verändert, was im Anwendungsfall des Caches für Dämpfungswerte jedoch trivial und effizient machbar ist.

Eine Annäherung des LRU-Algorithmus ist der Not Frequently Used Algorithmus. Hier besitzt jedes Cache-Element ein R-Bit, welches gesetzt ist, wenn eine Referenz auf das betreffende Element vorliegt, sowie einen Zähler. Bei jedem Zeitschritt, in dem eine Referenz auf das Element existiert, wird der Zähler um eins erhöht. Eine Alterung sorgt für regelmäßige Dekrementierung aller Zähler, damit der Algorithmus Elemente, die nur kurze Zeit sehr oft benutzt werden, nach einiger Zeit der Nichtnutzung wieder vergisst.

### 3.4.6 NRU (Not Recently Used)

Die NRU-Strategie (Not Recently Used) teilt die Elemente im Cache in vier Kategorien ein:

- Klasse 0: Nicht referenziert, nicht modifiziert
- Klasse 1: Nicht referenziert, modifiziert
- Klasse 2: Referenziert, nicht modifiziert
- Klasse 3: Referenziert, modifiziert

Wenn der Cache voll ist und ein Element ersetzt werden muss, ersetzt der NRU-Algorithmus zufällig ein Element in der niedrigsten Klasse, welche Elemente im Cache besitzt.

Der NRU-Algorithmus geht mit der Betrachtung von modifizierten Elementen über die Problemstellung dieser Arbeit hinaus. Da es bei den im Rahmen dieser Arbeit betrachteten Caches keine Modifikationen des Datenbestandes gibt, kann das Merkmal der Modifikation einfach ignoriert werden.

Damit vereinigen sich die Klassen 0 und 1 sowie 2 und 3, es wird nur noch zwischen referenzierten und nicht referenzierten Daten unterschieden.

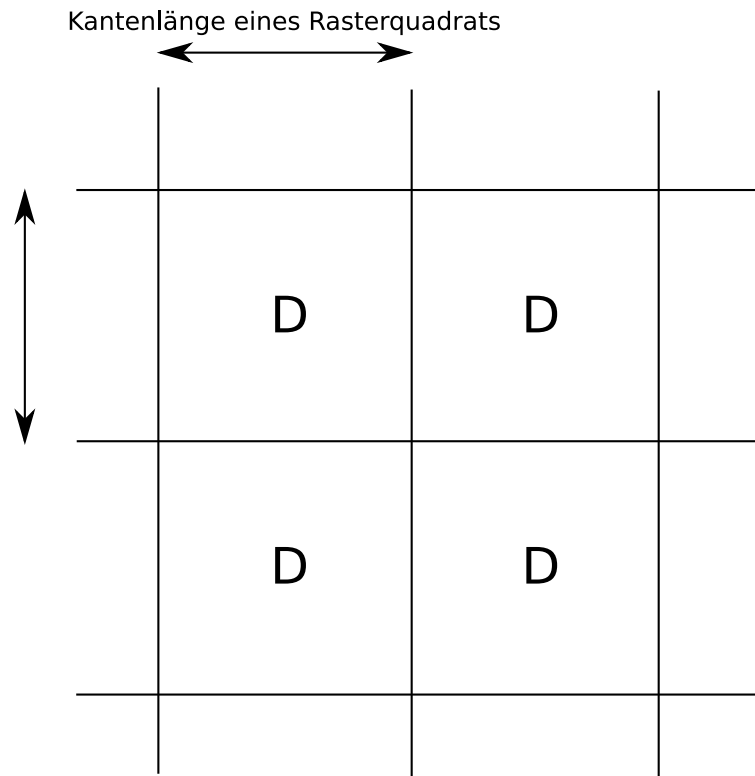


Abbildung 3.1: Darstellung des Simulations-Koordinatensystems mit Dämpfungswerten (markiert durch D)

## 3.5 Knotenmobilität und Wellenausbreitung in NET

### 3.5.1 Allgemeines

Die Koordinaten der Knoten sind in NET in den Eingabedaten als double-Zahlen (Gleitkomma-Darstellung) dargestellt. Für die Indexierung im Cache wird eine Abbildung der Positionen auf integer-Zahlen durchgeführt, diese diskretisierten Werte nennen wir Rasterkoordinaten. Die Simulation stellt Anfragen an den Cache, die von diesem mit den entsprechenden Dämpfungswerten beantwortet werden.

Die Dämpfungswerte existieren als Raster in festgelegtem Abstand, welches die Koordinatenachsen unterteilt. Alle Rasterquadrate haben eine festgelegte Kantenlänge. Jedem Rasterquadrat ist ein Dämpfungswert zugeordnet, dieser Dämpfungswert gilt für alle Punkte innerhalb des Rasterquadrats (siehe Abbildung 3.1). Für jeden dieser  $n^2$  Punkte in einem Feld von  $n \cdot n$  Punkten ist ein Dämpfungswert zu allen  $n^2$  anderen Punkten gespeichert. Der Datensatz der Dämpfungswerte besteht also aus  $n^4$  Werten. Ein Dämpfungswert ist durch vier Koordinaten festgelegt, jeweils x- und y- Werte von Sender- und Empfängerposition:

$$\text{Dämpfungswert} = (x_S, y_S, x_E, y_E).$$

Für die folgenden Darstellungen in den Diagrammen reduzieren wir die Koordinatenzahl auf zwei, um die Sachverhalte besser auf Papier darstellen zu können.

### 3.5.2 Clustering

Bei einer Auflösung von 1 Byte pro Dämpfungswert, einem Feld von  $3000 \times 3000$  Meter mit einem Dämpfungswert pro Quadrat von  $5 \times 5$  Meter ergibt sich eine Anzahl der Datensätze von ca. 129 Milliarden und damit eine Datenmenge auf dem Plattenspeicher von ca. 129 GB.<sup>1</sup>

Da jeder Dämpfungswert durch vier Koordinaten, nämlich die x- und y-Koordinaten von Sender- und Empfängerposition bestimmt wird, können die Dämpfungswerte als ein 4-dimensionaler Hypercube (d.h. ein Würfel mit vier Dimensionen) betrachtet werden. Beim Clustering wird der Simulationsraum in feste Blöcke von mehreren Dämpfungswerten aufgeteilt, der Hypercube also partitioniert. Jeder Cluster hat seinen Ankerpunkt beim jeweils geringsten Wert jeder seiner Koordinatenachsen, der noch im Cluster liegt, bildlich gesprochen also in einer der „Ecken“.

Die Cluster besitzen festgelegte Grenzen, diese können durch

$$\lfloor \text{Rasterkoordinaten} / \text{Clusterfaktor} \rfloor$$

berechnet werden. Auf diese Weise ist es auch nicht möglich, dass sich Cluster überlappen.

Cluster werden vom Cache als die kleinste adressierbare Einheit betrachtet. Im Spezialfall der Clustergröße 1 enthält jeder Cluster genau einen Dämpfungswert, die Kantenlänge beträgt also eine Rastereinheit.

Es stellt sich nun die Frage, welche Clustergröße optimal ist. Es sind zwei Extremfälle denkbar:

#### Clustergröße 1

Beträgt die Clustergröße 1, so wird jeder Dämpfungswert in einem eigenen Cluster geführt. Dieser Wert hat den Vorteil, dass die Dämpfungswerte einzeln adressierbar sind, es werden also niemals Daten im Cache gehalten, die eigentlich nicht gebraucht werden. Der Server mit den Dämpfungswerten liefert immer genau die benötigten Daten, jede Anfrage nach einem Dämpfungswert führt zu einem einzelnen Lesezugriff auf dem Server.

Abhängig von der verwendeten Hardware und Realisierung der Datenbank werden sehr viele Seek-Zugriffe auf der Festplatte stattfinden, um die einzelnen Daten anzuspringen. Viele der heutigen Datenbanken und Betriebssysteme liefern unter diesen Bedingungen verhältnismäßig schlechte Resultate, der Overhead für die Anfragen ist im Verhältnis sehr groß.

#### Clustergröße maximal

Ein pathologischer Extremfall wäre, die Clustergröße so groß zu machen, dass alle Dämpfungswerte des Szenario demselben Cluster zugewiesen werden, welcher das gesamte Gebiet des Szenario umfaßt. Dies würde dazu führen, dass nur dieser eine Datensatz als Einheit existiert, sinnvolles Management des Caches wäre allein schon wegen der Größe eines solchen Clusters nicht möglich.

---

<sup>1</sup>Wir verwenden hier die bei Datenträgern übliche Definition der Kapazität  $1 \text{ GB} = 10^9$  Bytes

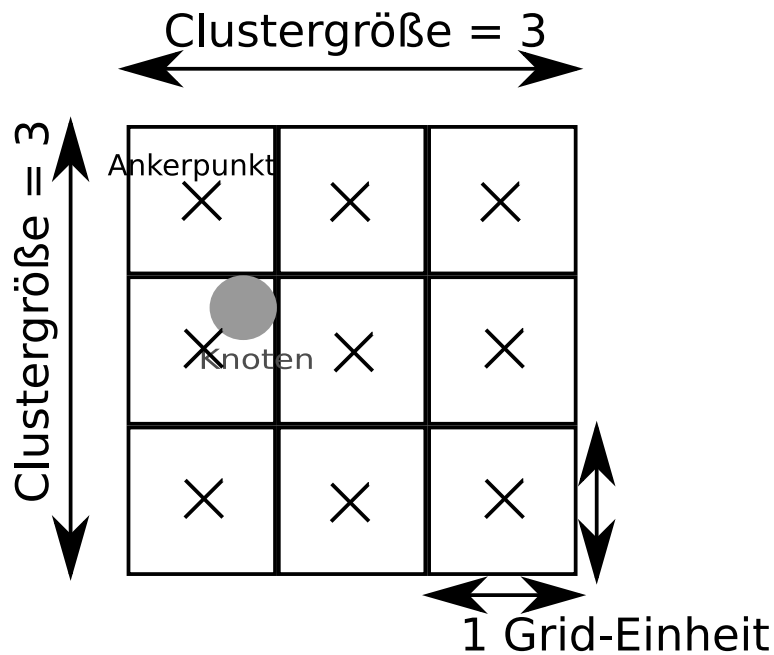


Abbildung 3.2: Darstellung eines Clusters mit Kantenlänge 3

### Wahl einer sinnvollen Clustergröße

Hohe Clustergrößen führen dazu, dass hohe Mengen von Dämpfungswerten auf einmal gelesen werden. Heutige Computersysteme kommen mit solchen Sequential-Read-Zugriffen gut zurecht und liefern folglich hohe Datenraten. Das Problem ist jedoch, dass beim Lesen von umfangreichen Clustern in der Simulation auch viele Werte gelesen und im Cache gehalten werden, die niemals benötigt werden. Die effektiv genutzte Cachegröße sinkt damit. Es muss also ein geeigneter Kompromiss zwischen vielen kleinen Seek-Zugriffen (kleine Cluster) und wenigen großen Sequential Read-Zugriffen (große Cluster) gefunden werden.

Abbildung 3.2 zeigt beispielhaft einen mobilen Knoten und seinen zugehörigen Cluster mit Kantenlänge 3. Da die Dämpfungswerte in Wahrheit nicht nur x- und y-Koordinaten besitzen, sondern die x- und y-Koordinaten von sowohl Sender- als auch Empfängerposition, ergibt sich tatsächlich ein 4-dimensionaler Hypercube. Ein Cluster der Kantenlänge  $k$  besitzt also  $k^4$  Dämpfungswerte.

Die optimale Clustergröße kann durch theoretische und empirische Analysen bestimmt werden. Dies soll in dieser Arbeit jedoch nicht behandelt werden. Eine entsprechende Analyse findet sich in [Dic07].

Wichtig für die Ausführungen in den folgenden Kapiteln ist die Tatsache, dass es bei Clustergrößen  $> 1$  oft vorkommt, dass Koordinaten unterschiedlicher Dämpfungswerte denselben Clustern zugeordnet werden. Bei der Prüfung, ob ein bestimmter Dämpfungswert im Cache ist, wird also in der Implementierung immer „ja“ zurückgegeben, wenn bei unverändertem Cache bereits für einen Dämpfungswert desselben Clusters die Antwort „ja“ gegeben worden ist. Bei Clustergröße 1 gleichen sich zwei Anfragen, wenn deren vier Koordinaten genau übereinstimmen.

### 3.5.3 Trajektorien und Traces

#### Trajektorien

Das hier verwendete Datenformat stammt vom Netzwerksimulator ns-2 [ns2]. Im NET-Framework wird jedem mobilen Knoten bei Beginn seiner Bewegung eine Trajektorie (siehe Abbildung 4.5) zugewiesen. Alle Positionen gelten relativ zum Nullpunkt des Koordinatensystems. Eine Trajektorie kann im Wesentlichen als Vektor zwischen Start- und Zielposition gesehen werden, der ab Zuweisung in der angegebenen Geschwindigkeit linear durchlaufen wird. Die Trajektorie enthält folgende Daten:

- Startzeitpunkt in Sekunden
- Startposition  $(x_s, y_s)$  des Knotens in Meter
- Zielposition  $(x_z, y_z)$  des Knotens in Meter
- Geschwindigkeit des Knotens in Meter pro Sekunde

Daraus berechenbare bzw. mitgeführte Daten:

- Momentane Position  $(x, y)$  des Knotens in Meter
- Abgelaufene Zeit in Sekunden, seit die Trajektorie gestartet ist

#### Traces

Ein Trace, oder Mobility Trace, ist eine nach Simulationszeit geordnete Liste von Trajektorien mit Angabe der jeweiligen Knoten. Jeder Trace beginnt mit den Anfangspositionen der Knoten. Mittels eines Traces kann also der Ablauf eines Szenarios mit Knotenbewegungen nachgespielt werden. Ein Beispiel für eine Trace-Datei findet sich in Abschnitt 5.1.

#### Generierung der Mobility Traces mit CanuMobiSim

CanuMobiSim ist ein durch Plug-In-Module erweiterbares Framework zur Modellierung von Knotenmobilität. Es wurde von der Forschungsgruppe CANU (Communication in Ad Hoc Networks for Ubiquitous Computing) [can] an der Universität Stuttgart erstellt. Im CanuMobiSim-Framework ist eine Reihe von Mobilitätsmodellen integriert. Außerdem existieren Parser für geographische Datenquellen in verschiedenen Formaten und ein Visualisierungsmodul. Wir verwenden CanuMobiSim in dieser Arbeit zur Erstellung der Mobility Traces. Folgende zwei Modelle für die Generierung der Simulationsdaten werden benutzt:

##### Random Walk

Das „Random Walk“-Modell weist jedem Knoten zu Beginn der Simulation zufällige Startkoordinaten im Simulationsgebiet zu. Den Knoten werden dann zufällige Zielkoordinaten zugewiesen. Hat ein Knoten sein Ziel erreicht, wird ihm ein weiteres Ziel zugewiesen. Die Knoten bewegen sich auf geraden Strecken direkt zu diesen Zielpunkten.

In der realen Welt ist es jedoch für mobile Knoten praktisch nicht möglich, ihr Ziel auf geradem Wege zu erreichen. Hindernisse wie Bäume, Mauern und Häuser verhindern dies. Außerdem bevorzugen Personen und Fahrzeuge üblicherweise vorhandene Wege und Straßen für ihre Fortbewegung, statt nach „Luftlinie“ zu navigieren. Bei „Random Walk“ gibt es wegen der Navigation nach Luftlinie praktisch keine Wegstrecken, die von mehreren Knoten gleichzeitig benutzt werden. Es kann eine Wartezeit festgelegt werden, in der die Knoten nach Erreichen eines Zielpunkts stillstehen, bevor sie sich zum nächsten Zielpunkt bewegen.

### Graph Walk

Im „Graph Walk“ Modell bewegen sich die Knoten zwischen festgelegten Zielpunkten auf festgelegten Wegen. „Graph Walk“ wählt für jeden Knoten zufällig Start- und Zielpunkte aus. Außerdem kann, wie bei Random Walk, die Geschwindigkeit der Knoten sowie eine Zeit festgelegt werden, welche die Knoten an den Zielpunkten verbringen, bevor sie den nächsten Zielpunkt anfahren. Bei dieser Art der Simulation bewegen sich Knoten folglich oft auf gemeinsamen Wegen.

## 3.6 Abarbeitung der Traces in CacheSim

Wir geben nun Positionen von Knoten als Vektoren von  $x$ - und  $y$ -Koordinaten an. Ein Knoten  $k$  hat die Position  $\vec{k} = (k_x, k_y)^T$ . Dämpfungswerte beziehen sich immer auf einen sendenden und einen empfangenden Knoten, für beide ist dabei deren Position festgelegt. Die Koordinaten des Dämpfungswerts einer Funkübertragung von Knoten  $a$  nach Knoten  $b$  an ihrer Position  $\vec{a}$  und  $\vec{b}$  sind charakterisiert durch  $\vec{d} = (a_x, a_y, b_x, b_y)^T$ .

Knoten ohne gesetzte Trajektorie stehen still. Die Knoten mit Trajektorien bewegen sich auf ihren Trajektorien von deren Startpunkt zum Zielpunkt bis zum Erreichen des Ziels oder bis eine weitere Trajektorie gesetzt wird. Im Folgenden bezeichnen wir den Startpunkt eines Knotens  $a$  als Vektor  $\vec{s}_a = (s_{a,x}, s_{a,y})^T$ , den Zielpunkt als  $\vec{z}_a = (z_{a,x}, z_{a,y})^T$ , analog dazu Knoten  $b$ :  $\vec{s}_b = (s_{b,x}, s_{b,y})^T$  und  $\vec{z}_b = (z_{b,x}, z_{b,y})^T$

Es muss auch berücksichtigt werden, dass nicht immer die Trajektorien aller Knoten gleichzeitig gesetzt werden. Es ist möglich, dass ein Knoten  $a$  gerade inmitten einer Trajektorie ist, während ein Knoten  $b$  gerade eine neue Trajektorie gesetzt bekommt. Mithilfe der ebenfalls gespeicherten Geschwindigkeit  $v$  des Knotens kann für beliebige Zeitschritte in der Zukunft bestimmt werden, wo sich der Knoten befindet, unter der Annahme, dass er sich weiterhin auf der derzeitigen Trajektorie bewegt.

In Abbildung 3.3 ist die Trajektorie von  $\vec{\delta}_{a,x}$  und  $\vec{\delta}_{a,y}$  entlang der Koordinatenachsen aufgespannt. Punkt  $(s_{a,x}, s_{a,y})$  bezeichnet die Position am Anfang der Trajektorie,  $(z_{a,x}, z_{a,y})$  die Position am Ende der Trajektorie. Die momentane Position  $(p_{a,x}, p_{a,y})$  wird in jedem Zeitschritt neu berechnet.  $t_{abgelaufen,a}$  gibt die bereits abgelaufene Zeit der Trajektorie des Knotens  $a$  an,  $v_a$  die Geschwindigkeit des Knotens.

Es gilt

$$\vec{\delta} = \vec{z} - \vec{s}$$

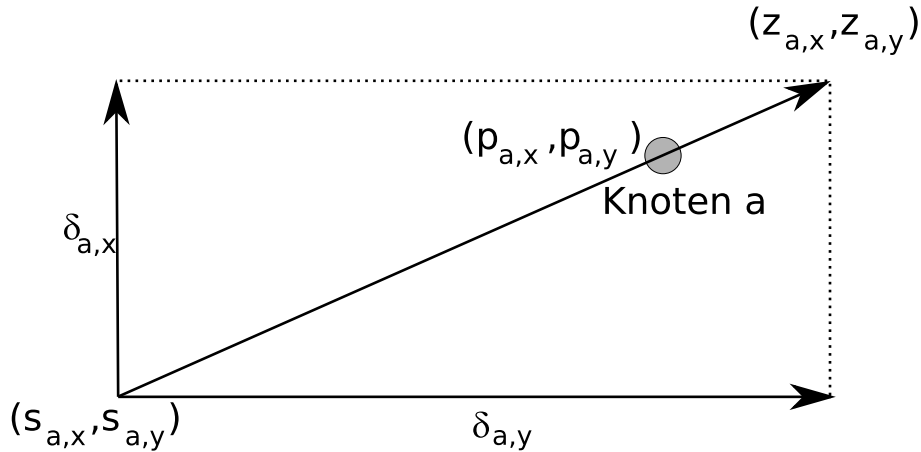


Abbildung 3.3: Abarbeitung der Traces

wobei sich die  $\vec{\delta}$ -Werte während des Durchlaufens der jeweiligen Trajektorie nicht ändern.

Die gesamte Zeit, die der Knoten  $a$  vom Start bis zum Ziel der Trajektorie benötigt, bezeichnen wir mit  $t_{ges,a}$ . Durch die Verwendung von Vektoren findet auch die nötige Normierung ohne weitere Rechenschritte statt.

$$t_{ges,a} = \frac{\sqrt{\delta_{x,a}^2 + \delta_{y,a}^2}}{v_a}$$

Die abgelaufene Zeit bezeichnen wir mit  $t_{abgelaufen,a}$ . Die berechneten Positionen des Knotens legen wir in  $p$  ab.

$$\vec{p}_a = \vec{s}_a + \frac{t_{abgelaufen,a}}{t_{ges,a}} \cdot \vec{\delta}_a$$

Es werden nun alle Zeitschritte abgearbeitet, dabei in jedem Zeitschritt  $t$  alle Trajektorien aktualisiert sowie für alle Knotenpaare  $(a, b)$  mit  $a \neq b$  die Koordinaten  $\vec{p}$ . Danach wird  $\vec{p}$  in Rasterkoordinaten umgewandelt, das resultierende 4-Tupel  $\vec{p} = (p_{a,x}, p_{a,y}, p_{b,x}, p_{b,y})^T$  bildet die jeweilige Anfrage an den Cache. CacheSim bildet aus den Traces intern die Liste der Anfragen an den Cache (im Folgenden Anfrageliste genannt) und spielt mit dieser die tatsächliche Belegung des Caches durch.



# Kapitel 4

## Konzepte für Ersetzungsstrategien

In diesem Kapitel wird der OPT-Algorithmus zur Bestimmung der optimalen Cache-Ersetzung nebst Implementierungsvarianten vorgestellt und eine der Varianten für den im Rahmen dieser Arbeit entstandenen Simulator „CacheSim“ ausgewählt. Eine Reihe von Cache-Ersetzungsstrategien werden sowohl für den Fall der deterministischen als auch für den Fall der indeterministischen Knotenmobilität vorgestellt und verglichen.

### 4.1 Deterministische Knotenbewegung

Wenn die Bewegung der Knoten in der Emulation im Voraus bekannt ist, ist auch die Sequenz der Anfragen an den Cache vollständig vor Beginn des Laufes des OPT-Algorithmus bekannt. Im Folgenden werden drei verschiedene Varianten der optimalen Ersetzungsstrategie OPT diskutiert. Die Ergebnisse können zum Vergleich verschiedener anderer Algorithmen benutzt werden.

#### 4.1.1 OPT nach Belady

In [Bel66] wird die Grundform des OPT-Algorithmus beschrieben. Wir stellen hier zum Verständnis für die folgenden Optimierungen das Prinzip vor, der Beweis der Optimalität findet sich in [MGST70].

Solange der Cache-Speicher (Größe:  $c$  Elemente) noch nicht gefüllt ist, können die Elemente an beliebigen freien Stellen des Caches abgelegt werden. Sobald jedoch nach Füllen des Caches ein neues Element geladen werden muß, beginnt eine Entscheidungsverzögerungszeit (decision delay), da zu diesem Zeitpunkt kein Block ein offensichtlicher Kandidat zum Löschen ist. Alle Blocks im Cache, die von jetzt ab nochmals angefragt werden (d.h. Wiederholungen), sind damit als Löschungskandidat disqualifiziert, da sie ja zeitnah benötigt werden. Wenn  $c - 1$  Blocks disqualifiziert worden sind, bleibt nur der letzte der Blöcke zum Löschen, die Entscheidung kann also eindeutig getroffen werden.

Oft gibt es jedoch zwischen zwei Anfragen, die zu Ersetzungen im Cache führen, nicht genügend Wiederholungen, um genügend Kandidaten zum Treffen der Löschungs-Entscheidung zu diesem Zeitpunkt zu haben. In diesem Fall muss die Entscheidung verzögert werden und die

weiteren Einträge müssen analysiert werden. Normalerweise sind mehrere Entscheidungen zur selben Zeit verzögert, weil jede neue Anfrage, die nicht im Cache ist, eine weitere Löschung nötig macht. Sofort, wenn  $c - 1$  Blocks disqualifiziert worden sind, kann eine Entscheidung über eine Löschung getroffen werden. Sind weitere offene Entscheidungen vorhanden, so kann möglicherweise auch von diesen eine Entscheidung getroffen werden. Die maximale Verzögerung reicht bis zum Ende der Anfragedaten, meist können jedoch früher Entscheidungen getroffen werden.

Ein kurzes Beispiel für den Ablauf des Algorithmus zeigt die folgende Tabelle:

Zeitpunkt	1	2	3	4	5	6	7	8	9	10	11
Anfrage	A	B	C	D	E	F	B	B	C	A	\$

Gleiche Buchstaben bedeuten gleicher Speicherbereich der Anfrage bzw. gleiche Koordinaten. Es wird die Anfragesequenz ABCDEFBCCA\$ in dieser Reihenfolge an den Cache gestellt. In der Tabelle sind die Anfragen der besseren Übersicht halber mit den Nummern 1 - 10 gekennzeichnet. Das Zeichen \$ markiert das Ende der Anfrage-Folge. Der Algorithmus liest ein Zeichen nach dem anderen zu den angegebenen Zeitpunkten ein, kann jedoch bei Bedarf auch auf zukünftige Zeichen zugreifen.

Die Cachegröße  $c$  sei für dieses Beispiel 3, der Cache ist am Anfang leer. Die Anfragen 1-3 können also einfach in den Cache geladen werden. Im Cache befinden sich die Anfragen ABC. Bei Anfrage Nummer 4 stellt sich die Frage, welches der Elemente im Cache zu ersetzen ist. Der Algorithmus forscht in der Sequenz der Anfragen schrittweise weiter, welche der drei Anfragen im Cache ABC sich wiederholen. Bei Punkt 7 wiederholt sich Anfrage B, danach wird bei Punkt 9 die Wiederholung von C festgestellt. Nun sind  $c - 1$  Elemente disqualifiziert (da diese ja gleich wieder gebraucht werden), es wird Element A gelöscht. Nun wird also Element D an seine Stelle geladen, der Cacheinhalt ist nun DBC. Bei Anfrage 5 (E) wird wiederum nachgeforscht: Anfrage B tritt zu Zeitpunkt 7 nochmals auf, Anfrage C zum Zeitpunkt 9. Damit ist D Löschungskandidat und wird durch E ersetzt. Der Cacheinhalt jetzt: EBC. Bei Anfrage 6 (F) wird wiederum geprüft: B kommt bei 7 wieder vor, C bei 9, also wird entschieden, E zu ersetzen. Cacheinhalt: FBC. Anfrage 7, 8 und 9 befinden sich im Cache, keine Ersetzung nötig. Für Anfrage 10 (A) am Ende der Sequenz könnte jedes der Elemente im Cache ersetzt werden, da keines von ihnen nochmals vorkommt.

Eine andere Sichtweise des „schrittweise in die Zukunft Weiterforschens“ im Algorithmus ist, die jeweiligen Ersetzungsentscheidungen zum Zeitpunkt ihres Einlesens als „verzögert“ zu sehen. Zum Zeitpunkt 4 direkt nach Eintreffen der Anfrage könnte man also auch sagen, die Entscheidung über die Ersetzung der Elemente im Cache wird verzögert. Sie kann im Beispiel erst getroffen werden, wenn sich zwei der drei Elemente ( $c - 1$ ) wiederholt haben, im Fall der Anfrage zum Zeitpunkt 4 also zum Zeitpunkt 9.

#### 4.1.2 OPT nach Mattson et al.

In [MGST70] wird ein Offline-Algorithmus zur Bestimmung der optimalen Ersetzungsreihenfolge beschrieben. Der Algorithmus benötigt als Eingabe sämtliche Anfragen an den Cache in der Reihenfolge, in der sie auftreten. Anders gesagt, die Vorausschau ist unendlich (bis zum

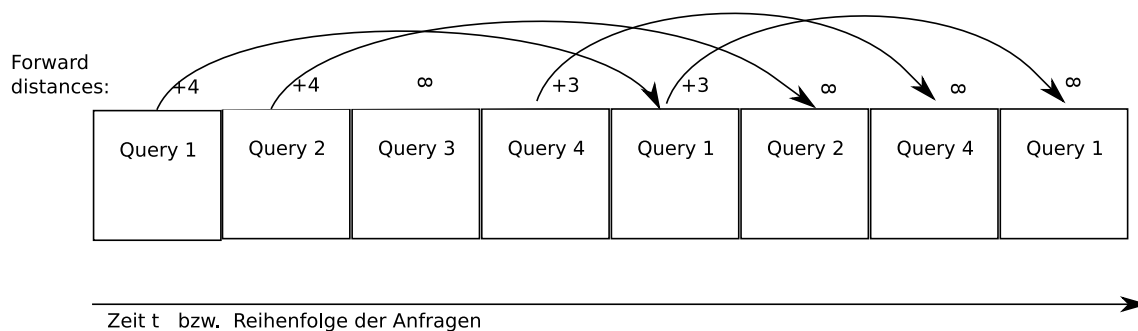


Abbildung 4.1: OPT nach Mattson et al.

Ende der Eingabe).

Es wird zunächst durch den Algorithmus von jeder vorkommenden Anfrage an den Cache die Vorwärtsdistanz („forward distance“, in einigen anderen Publikationen auch als „Time of Next Reference“, TNR, bestimmt [SA93]. Es handelt sich hierbei um die Anzahl der Schritte bis zum nochmaligen Auftreten derselben Anfrage. Kommt eine Anfrage zu einem Zeitpunkt im zukünftigen Verlauf der Anfragen nicht mehr vor, so ist ihre Vorwärtsdistanz an dieser Stelle unendlich.

Abbildung 4.1 zeigt die Funktionsweise des Algorithmus anhand einiger Anfragen („Queries“). In der Abbildung wird angenommen, dass die Simulation nach den dargestellten Anfragen beendet ist, also keine weiteren nachfolgen. Die erste Anfrage, hier „Query 1“ genannt, kommt vier Schritte später nochmals vor, deshalb ist als Vorwärtsdistanz 4 eingetragen. Nach dem zweiten Vorkommen tritt die Anfrage noch ein drittes Mal mit Distanz 3 auf. Wenn eine Anfrage zum letzten Mal auftritt, wird der Wert unendlich als Vorwärtsdistanz eingetragen.

Die Bestimmung dieser Werte kann durch einen einfachen Algorithmus geschehen, der aus zwei Teilen besteht. Teil 1 berechnet die Vorwärtsdistanzen. Teil 2 führt die eigentlichen Cache-Ersetzungen durch, spielt also die Ersetzungen konkret für einen Cache festgelegter Größe durch, wobei gemäß dem Prinzip des Algorithmus immer die Anfrage mit der höchsten Vorwärtsdistanz ersetzt wird. Der Algorithmus läuft zweimal durch die Anfrageliste – zuerst von hinten nach vorne in Teil 1, dann von vorne nach hinten in Teil 2.

### Teil 1

Erläuterungen:  $Q$  ist die Menge der Anfragen im Cache,  $a$  ist die aktuell bearbeitete Anfrage.  $w.d$  bezeichnet den zu einer Anfrage  $w$  gespeicherten Distanzeintrag. Zur Semantik (gilt auch für Teil 2): Bei der Zuweisung von Anfragen werden Referenzen übergeben, sodass z.B. bei

Zeile  $a.d = \infty$  in Algorithmus 1 der Distanzeintrag im Feld der Anfragen selbst geändert wird.

```

Q =  $\emptyset$  ;
Anfrageliste ablegen in Array Anfragen[];
index = maxindex(Anfragen);
while index  $\geq$  0 do
    a = Anfragen[index];
    if a  $\in$  Q then
        setze a.d auf das x.d für das gilt:  $x = a, x \in Q$ ;
        setze y.d auf index, es gilt:  $y = a, y \in Q$ ;
    else
        Q = Q  $\cup$  {a} ;
        a.d =  $\infty$  ;
    end
    index = index - 1;
end

```

**Algorithmus 1** : OPT-Algorithmus (Mattson et al.) Teil 1

Teil 1 des Algorithmus geht von hinten nach vorne durch das Array mit den nach der Anfragereihenfolge geordneten Anfragen durch. Kommt eine Anfrage erstmalig vor, so wird ihr Distanzeintrag im Array auf unendlich gesetzt, da dies ja, von vorne her gedacht, das letzte Auftreten dieser Anfrage ist. Die Anfrage wird in AnfrageMenge gespeichert, der Zeitpunkt des Auftretens im dortigen Distanzeintrag gespeichert. Beim weiteren Auftreten der Anfrage wird dieser gespeicherte Distanzeintrag in das Array an der aktuellen Stelle geschrieben, damit das vorher durchlaufene Auftreten dieses Elementes dokumentiert. Zum Abschluß wird die aktuelle Position wiederum in AnfrageMenge gespeichert, damit der nächste Durchlauf das richtige Element liefert. Nach dem Ablauf des Algorithmus steht für jede Anfrage in deren Distanzeintrag die Distanz zum nächsten Auftreten dieser Anfrage.

Der Algorithmus führt die while-Schleife für jede Anfrage an den Cache aus. In der Schleife muss im schlechtesten Fall die Anfragemenge komplett durchsucht werden. Um hier Rechenzeit zu sparen, wird die Anfragemenge mittels Hashing realisiert: Aus den Koordinaten der Anfrage wird durch eine Hashfunktion eine Speicherstelle (d.h. ein Hashbucket) gefunden. Jedes Hashbucket enthält eine Liste, in der die Anfragen mit demselben Hashfunktionswert abgelegt sind. Bei richtiger Wahl der Anzahl der Buckets, auf die per Hash abgebildet wird, gibt es kaum Kollisionen in den Buckets und die Listen enthalten je nur ein Element. ( Siehe Abbildung 4.2, Hashmap ).

Wir definieren  $n$  als die Anzahl der Anfragen. Je nach Realisierung, Qualität der Hashfunktion, Anzahl der Hashbuckets und nach der Art der Daten besitzt deshalb Teil 1 des Algorithmus zwischen  $O(n)$  und  $O(n^2)$  an Zeitkomplexität.

Zur Platzkomplexität (zusätzlich zu den Eingabedaten) läßt sich feststellen, dass der Algorithmus die Menge der Anfragen im Speicher halten muss. Jede Anfrage mit denselben Parametern existiert in der Anfragemenge nur einmal. Es gibt zwei pathologische Extremfälle: Im Falle ausschließlich gleicher Anfragen enthält die Anfragemenge 1 Element, im Fall komplett unterschiedlicher Anfragen (d.h. jede Anfrage kommt insgesamt nur einmal vor) enthält die Menge  $n$  Elemente. Die Platzkomplexität ergibt sich also zwischen  $O(1)$  und  $O(n)$ , wobei die Eingabedaten ebenfalls die Mächtigkeit  $n$  besitzen. Dies führt zu einer Gesamt-Platzkomplexität von

$O(2n)$ , was  $O(n)$  entspricht.

## Teil 2

Anfragen[] enthält die Anfragen aus Teil 1 des Algorithmus. GroesstesDistElement bezeichnet ein Element aus dem Cache, AktuelleAnfrage bezeichnet ein Element aus dem Strom der Anfragen, CacheSize die Größe des Caches.

```

CacheMisses = 0;
CacheHits = 0;
Cache =  $\emptyset$ ;
n = 0;
while  $n \leq \text{maxindex}(\text{Anfragen})$  do
  AktuelleAnfrage = Anfragen[index];
  if AktuelleAnfrage  $\in$  Cache then
    CacheIndex = i, sodass Cache[i] = AktuelleAnfrage;
    Cache[CacheIndex].d = Anfragen[n].d;
    CacheHits = CacheHits + 1;
  else
    if  $| \text{Cache} | < \text{CacheSize}$  then
      Cache = Cache  $\cup$  { AktuelleAnfrage };
    else
      GroesstesDistElement = j, sodass  $\forall$  Einträge  $e \in \text{Cache}$  gilt:  $j.d \geq e.d$ ;
      Cache = Cache  $\setminus$  { GroesstesDistElement };
      Cache = Cache  $\cup$  { AktuelleAnfrage };
      GroesstesDistElement.d = AktuelleAnfrage.d;
      CacheMisses = CacheMisses + 1;
    end
  end
  n = n + 1;
end

```

### Algorithmus 2 : OPT-Algorithmus (Mattson et al.) Teil 2

In Teil 2 des Algorithmus wird nun, wenn der Cache komplett mit Elementen gefüllt ist, immer dasjenige Element gelöscht und ersetzt, welches am weitesten in der Zukunft wieder benötigt wird. Es wird also das Element mit dem höchsten Distanzeintrag ersetzt. Der Distanzeintrags-Wert  $\infty$  bedeutet dabei, dass ein Element überhaupt nicht mehr gebraucht wird, es wird daher allen endlichen Werten bei der Ersetzung bevorzugt. Ersetzt werden auch Elemente, deren höchster Distanzeintrag in der Vergangenheit liegt, die also ebenfalls nicht mehr benötigt werden. Ist der Cache noch nicht voll, d.h. hat die Menge Cache noch nicht die Mächtigkeit CacheSize, so kann das betreffende Element einfach in den Cache aufgenommen werden.

Bei einfacher Implementierung muß der Cache pro Anfrage einmal komplett durchsucht werden. Bei diesem Suchlauf wird zum Einen festgestellt, ob das angefragte Element im Cache enthalten ist, zum Anderen wird bereits, für den Fall eines Cache Misses, der höchste Distanzeintrag festgestellt.

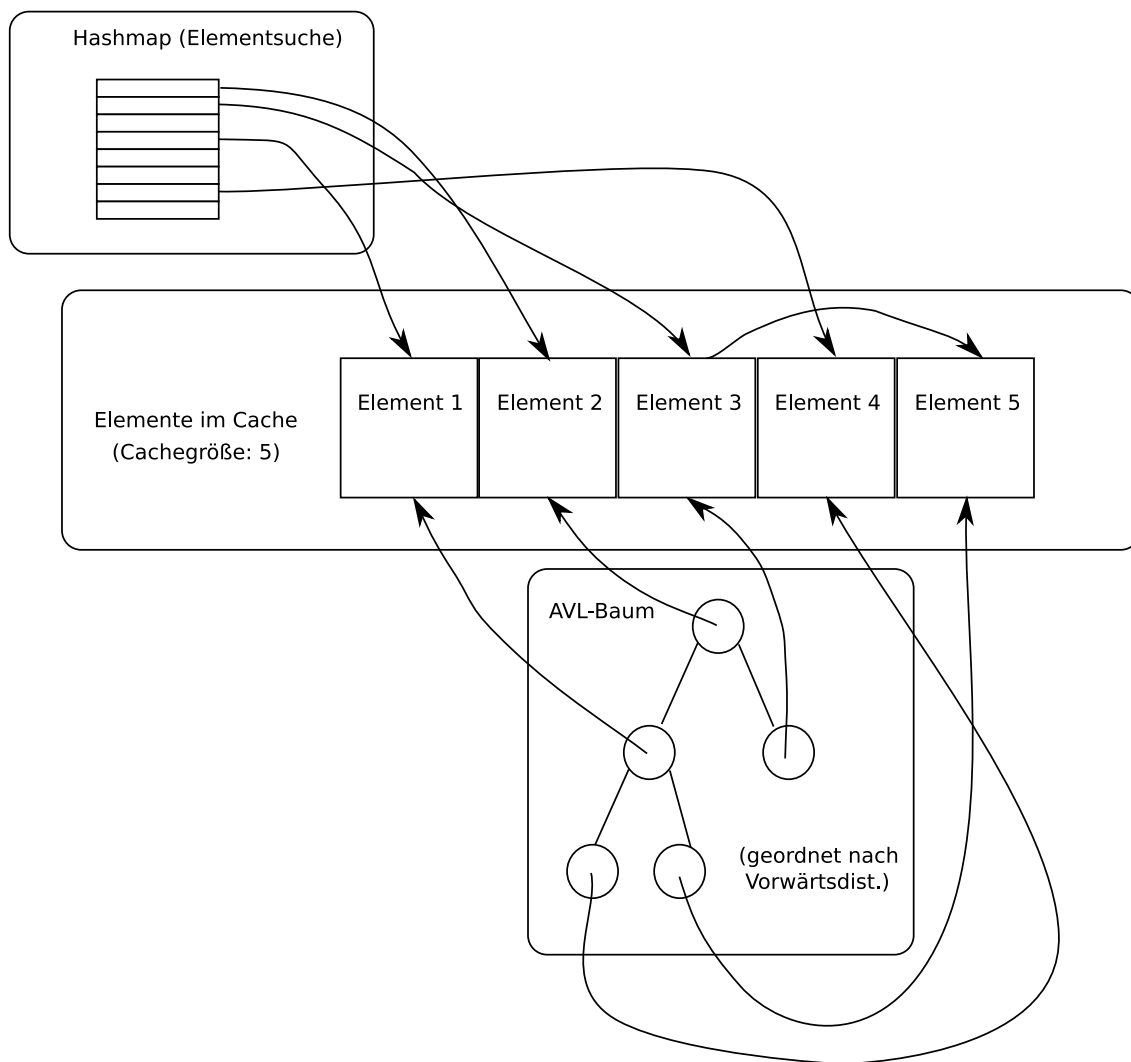


Abbildung 4.2: Mögliche Datenstruktur für OPT

Bei der Suche im Cache sind vielfältige Optimierungen möglich.

## Bäume

Denkbar ist beispielsweise, die Cache-Elemente als nach dem TNR-Wert geordneten Binärbaum oder AVL-Baum abzulegen. So kann immer schnell das größte Element gefunden werden. Der Zeitaufwand für das Einfügen, Suchen und Löschen wäre hier  $O(\log n)$ , was eine wesentliche Verbesserung darstellt. Es muß immer wieder festgestellt werden, ob sich ein bestimmtes Element im Baum befindet. Dies ist ohne komplettes Durchsuchen des Baumes möglich, wenn die Baumelemente analog zu Teil 1 als Hashfunktion verzeigert werden (siehe Abbildung 4.2). Die Cache-Elemente werden dabei jeweils über Zeiger im Hash und im Baum referenziert.

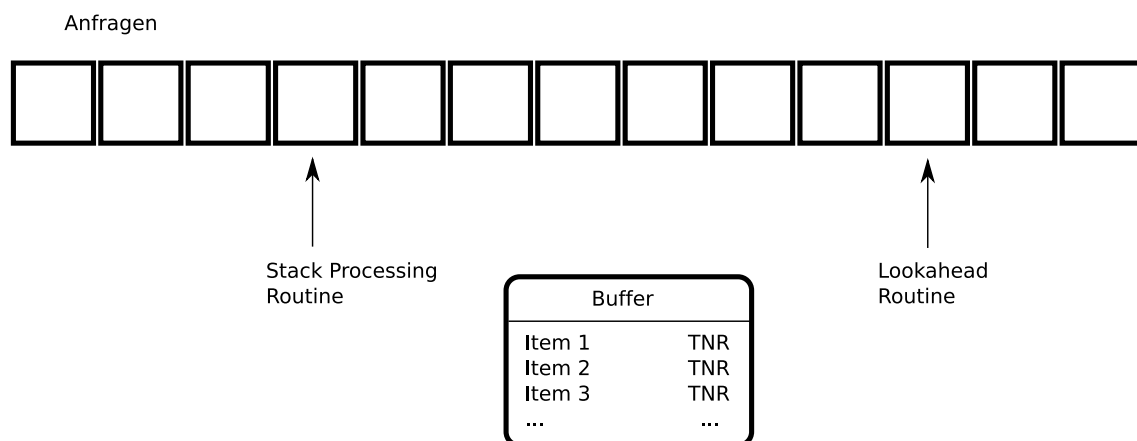


Abbildung 4.3: Prinzip von OPT nach Sugumar, Abraham

### Grouping

In Anlehnung an [SA93] (Algorithm Tree\_OPT) ist, alternativ zur Realisierung mittels eines Baums, folgende Optimierung bei der gezeigten Realisierung des OPT-Verfahrens nach Mattson et al. denkbar. Es geht hierbei um den Bereich der Abarbeitung der Anfrageliste (Teil 2), wenn also die TNR-Werte bereits von Teil 1 des Algorithmus vorliegen.

Der OPT-Algorithmus kennt die TNR-Distanzen jedes Cache-Eintrags. Wir teilen nun die Elemente in Gruppen auf, jede Gruppe enthält eine Liste von nach TNR-Distanz-Wert geordneten Elementen des Cache-Inhaltes eines bestimmten Bereichs von TNR-Distanz-Werten. Die Gruppen ihrerseits überlappen sich von ihren enthaltenen Wertebereichen her nicht, es findet also eine Partitionierung statt. Das Element mit dem niedrigsten Wert wird zuerst wieder vom Algorithmus benutzt, dies ist die bereits bekannte eigentliche Basis des Algorithmus. Wenn sich  $e$  Elemente in der ersten Gruppe (dies ist die Gruppe, welche die Elemente mit den kleinsten TNR-Werten enthält) befinden, so werden zwangsläufig in den nächsten  $e$  Ersetzungen nur Elemente dieser Gruppe ersetzt. Es ist also nicht nötig, alle Elemente des Caches zu durchsuchen, sondern nur die  $e$  Elemente in der ersten Gruppe. Nach diesen  $e$  Elementen muß die Gruppe neu geordnet werden, da nach jedem gefundenen Element in einer Gruppe dieses Element aus dieser Gruppe verschwinden könnte.

Soll ein neu in den Cache hinzukommendes Element in die Gruppen eingefügt werden, so muss im schlechtesten Fall das jeweils größte und das kleinste Element jeder Gruppe untersucht werden, um das Element der richtigen Gruppe zuzuordnen. Oft dürfte der Aufwand jedoch geringer sein.

Die Kosten einer Einfüge-/Löschoperation beim Gruppieren sind bei einer geprüften Gruppenanzahl  $g$  und Kosten einer Gruppen-Einfüge- bzw. Löschoperation  $k$ :  $O(g \cdot k)$ . Die Kosten einer Gruppen-Operation orientieren sich an  $O(\text{Gruppengröße})$ .

### 4.1.3 OPT nach Sugumar, Abraham

Im Folgenden bedeutet TNR, wie in Abschnitt 4.1.2 erwähnt, „Time of Next Reference“, welche zu einer Anfrage den Index des nächsten Auftretens einer gleichartigen Anfrage in der

Anfrageliste angibt.

In [SA93] wird beschrieben, wie sich die Bestimmung der OPT Ersetzungsreihenfolge mit endlicher Vorausschau durchführen lässt. In diesem Algorithmus hat das Datum mit der geringsten TNR die höchste Priorität. Er benutzt einen Puffer in Form einer Liste, in dem sich Tupel (Anfrage, TNR) ablegen lassen. Abbildung 4.3 zeigt das Prinzip des Algorithmus. Es laufen zwei Routinen: Lookahead und Stack Processing, beide operieren sequenziell vorwärts (d.h. in Reihenfolge des Auftretens der Anfragen) auf der Anfrageliste. Jede Anfrage hat ein Datenfeld für den nächsten Zugriff (TNR). Die beiden Routinen laufen abwechselnd, wobei die „Lookahead“ Routine der „Stack Processing“ Routine vorausläuft.

Der Algorithmus setzt an Stellen, wo die TNR unbekannt ist, zunächst den Wert „unbekannt“ im Puffer. Später, wenn dieser Wert bekannt wird, wird er entsprechend eingesetzt.

Die Lookahead Routine liest die Anfragen. Für jede Anfrage wird ein Tupel (Anfrage, nächstes Auftreten) an den Puffer angehängt. Per Hash Lookup wird nun geprüft, ob eine Anfrage mit diesen Koordinaten bereits auftrat. Falls nicht, so bleibt das Tupel auf „unbekannt“. Wurde eine solche Anfrage bereits in der Vergangenheit zum Zeitpunkt  $t$  durchgeführt, so wird geprüft, ob diese bereits von der Stack Processing Routine bearbeitet wurde. Falls nicht, kann der TNR-Wert der Anfrage bei  $t$  auf die aktuelle Zeit gesetzt werden. Im anderen Fall muß der Stack evtl. „repariert“ werden, und der richtige Wert für TNR eingefügt werden.

Die Stack Processing Routine wiederum entfernt Tupel vom Kopf der Liste im Puffer, sucht diese in der Anfrageliste und aktualisiert die betreffenden Werte. Anhand der TNR-Werte wird die Priorität der Anfragen bestimmt. Elemente mit TNR „unbekannt“ haben eine niedrigere Priorität als alle bekannten Werte. Die Anfrageliste wird nach der gerade bearbeiteten Anfrage durchsucht, der Algorithmus nimmt die Ersetzungen im Cache in jeder auftretenden Tiefe entsprechend vor.

Der Vergleich der TNR-Werte zweier Einträge im Puffer wird Interaktion genannt. Die Interaktionen legen damit fest, welche Ersetzungsentscheidungen im Cache getroffen werden. Unbekannt-bekannt-Interaktionen (d.h. Interaktionen zwischen einem Element, dessen TNR unbekannt ist und einem Element, dessen TNR bekannt ist) sind unkritisch. Das Unbekannt-Tupel hat zwangsläufig eine niedrigere Priorität, weil das Bekannt-Tupel früher in der Anfrageliste wieder auftritt. Somit wird in diesem Fall immer die richtige Ersetzungsentscheidung getroffen.

Ein Problem sind unbekannt-unbekannt-Interaktionen, wenn also ein Element mit unbekannter TNR mit einem Element mit ebenfalls unbekannter TNR verglichen wird. Hier ist nicht sicher, welches der beiden Elemente tatsächlich die kleinere Priorität hat. Der Algorithmus entscheidet sich willkürlich für eines der Elemente, eine evtl. daraus resultierende falsche Ersetzungsentscheidung muß später repariert werden. Um diese Reparatur durchführen zu können, wird ein sogenannter Interaction-Graph aufgebaut: Für jede willkürliche Entscheidung zwischen zwei Unbekannt-Elementen wird dem Graph eine Kante hinzugefügt. Anhand dieses Graphs kann die Stack-Repair-Routine in der Vergangenheit falsch durchgeführte Ersetzungen zwischen zwei Unbekannt-Elementen korrigieren, wenn eines der beiden Elemente bekannt wird.

Der verwendete maximale Lookahead (d.h. der maximale „Vorsprung“ der Lookahead-Routine gegenüber der Stack-Processing Routine) ist ein wichtiger Parameter des Algorithmus. Ist der Vorsprung zu klein, ergeben sich sehr viele unbekannt-unbekannt Interaktionen mit vie-

len Aufrufen der Stack-Repair-Routine. Ist der Lookahead zu groß, wird unnötig Speicherplatz verbraucht, außerdem kostet das Durchsuchen des bei großem Lookahead meist großem Puffer viel Zeit. Der optimale Wert für den Lookahead hängt von den individuellen Anfragen ab, hier sind kaum Voraussagen möglich. Der Aufwand für den Algorithmus beträgt prinzipiell  $O(n \cdot k)$ . Der Wert  $n$  hängt von der Anzahl der Anfragen ab,  $k$  hängt ab davon, wie oft die Stack-Repair-Routine aufgerufen wurde. Es sind, je nach individueller Anfrageliste und Parametern, sowohl kleine konstante Werte als auch große quadratische Werte bis zu  $n^2$  für  $k$  möglich.

#### 4.1.4 Auswahl des OPT-Algorithmus

Im Rahmen dieser Arbeit stellte sich die Frage, welche Art des OPT-Algorithmus zu implementieren ist. Da alle Varianten dasselbe Ergebnis liefern, ist es sinnvoll, die Auswahl auf nur eine Variante zu beschränken.

Der eben erwähnte Algorithmus von Sugumar und Abraham muss die Stack-Repair-Routine für jede falsch getroffene Ersetzungsentscheidung starten. Stack-Repair besitzt bei  $k$  Knoten im Puffer die Komplexität  $O(k^2)$ . Es hängt von der individuellen Anfrageliste ab, wie viele falsche Ersetzungen vorkommen. Bei vielen sehr großen TNR-Werten in der Sequenz der Zugriffe müssen, je nach auftretenden TNR-Distanzen, beim „Stack Repair“ teilweise sehr viele Entscheidungen revidiert werden.

Demnach ist beim Sugumar-Algorithmus bei ungünstiger Wahl der Parameter und ungünstiger Anfrageliste die Zeit-Komplexität bis zu  $O(n^3)$ , beim Mattson et al. Algorithmus jedoch nur  $O(n^2)$ . Zusätzlich ist beim Mattson et al. Algorithmus lediglich die Anzahl der Hash-Buckets als Parameter zu bestimmen. Eine größere Anzahl Hash-Buckets als nötig schadet der Laufzeit nicht und kostet nur wenig Speicher. Ein Vorteil des Algorithmus von Sugumar und Abraham ist prinzipiell, dass er schon parallel zum Aufbau der Anfrageliste ablaufen kann, da dieser nicht schon zu Beginn des Algorithmus vollständig gebraucht wird. Im Rahmen dieser Arbeit liegen jedoch immer vollständige Anfragelisten vor, sodass dieser Vorteil nicht ausgenutzt werden kann.

Bei der Auswahl des Algorithmus wurde davon ausgegangen, dass die durchschnittlichen TNR-Distanzen im Verhältnis zur Größe der Anfrageliste recht umfangreich sein werden, folglich oft „Stack Repair“ benötigt würde und deshalb der Algorithmus von Sugumar und Abraham wesentlich teurer wird im Vergleich zum Algorithmus von Mattson et al. Nach Abwägung der Vor- und Nachteile wurde der Algorithmus von Mattson et al. zur Bestimmung der OPT-Ersetzungsreihenfolge im Rahmen dieser Arbeit implementiert. Es ist wichtig, festzuhalten, dass je nach Struktur der Anfrageliste sowohl der eine als auch der andere Algorithmus mit weniger Rechenaufwand zu einem Ergebnis kommen kann.

## 4.2 Indeterministische Knotenbewegung

Ersetzungsstrategien für die nichtdeterministische Knotenbewegung zeichnen sich dadurch aus, dass ihnen keine Daten über zukünftige Knotenbewegungen zur Verfügung stehen und somit solche nicht genutzt werden können.

Eine wichtige Größe bei der Bestimmung der Treffer-Rate des Caches ist die Anzahl der „Compulsory Misses“. d.h. die Anzahl der Cache Misses, welche auch bei unendlich großem Cache auftreten würden, weil das jeweilige Element erstmalig auftritt. Die Anzahl der „Compulsory Misses“ gibt also an, wieviele unterschiedliche Anfragen an den Cache während des Emulationslaufs gemacht werden. Es leuchtet ein, dass „Compulsory Misses“ unvermeidlich sind, weil erst ab der zweiten Anfrage eines bestimmten Elements an den Cache überhaupt die Möglichkeit besteht, die Anzahl der vermeidbaren Seitenfehler („Conflict Misses“) zu drücken.

Im Folgenden werden drei Ersetzungsstrategien für nichtdeterministische Knotenbewegungen diskutiert: RANDOM, FIFO und Varianten, sowie LIL (Linear Interpolated Lookahead). In der Diskussion wird insbesondere auf den Zusammenhang zwischen Cachegröße, Knotenzahl und Cache Hit Rate (Erfolgsrate) des Caches eingegangen, damit der Anwender der Emulation durch geschickte Wahl der Parameter eine möglichst hohe Cache Hit Rate erzielen kann. Wir setzen in den folgenden Abschnitten die Cachegröße mit  $c$ , die Anzahl der „Compulsory Misses“ mit  $m$ , die Gesamtzahl der Anfragen in der Anfrageliste mit  $n$ . Folglich gilt immer  $m \leq n$ .

Die weiter unten vorgestellten Ansätze des Hotspot-Prefetching und der Cache-Aufteilung sind mit den für den Cache zur Verfügung stehenden Daten aufgrund der schichtartigen Architektur der Simulation im Rahmen dieser Arbeit nicht zu realisieren (siehe Abschnitt 6.1). Eine zukünftige Verwendung wäre jedoch bei entsprechender Erweiterung des Frameworks möglich.

### 4.2.1 RANDOM

Im diesem Abschnitt schätzen wir die Erfolgsrate einer Random-Ersetzungsstrategie ab (siehe 3.4.2). Es wird vereinfachend davon ausgegangen, dass gleiche Anfragen zufällig über die Anfrageliste verteilt sind.

Der Ablauf der Emulation ist in zwei Phasen unterteilt: Am Anfang der ersten Phase, wir nennen sie hier Füllphase, ist der Cache leer. Er füllt sich bei jeder Anfrage mit einem weiteren Element, welches aufgrund der Anfrage auch im Cache abgelegt wird. Nach  $c$  „Compulsory Misses“ ist der Cache voll. In der jetzt beginnenden zweiten Phase muss für jedes durch einen Miss neu hinzukommende Element entschieden werden, welches bisher im Cache befindliche Element dafür gelöscht wird. Bei der RANDOM-Variante wird in diesem Fall ein zufälliges Element gelöscht.

Während der Füllphase (also bei Anfrage  $a$  von 0 bis  $c - 1$ ) beträgt die Wahrscheinlichkeit  $p_1$  für einen Seitentreffer pro Zugriff (die bereits im Cache befindlichen Elemente setzen wir mit  $a$ , es gilt  $a < c$ ). Wir betrachten den durchschnittlichen Fall („average case“):

$$p_1 = \frac{a}{m}$$

Nach der Füllphase bildet sich die Wahrscheinlichkeit  $p_2$  für einen Cache-Treffer wie folgt pro Zugriff:

$$p_2 = \frac{c}{m}$$

Insgesamt ergibt sich also:

$$\text{Anzahl Compulsory Misses bei RANDOM: } \sum_{a=0}^{c-1} p_1 + \sum_c^n p_2 = \sum_{a=0}^{c-1} \frac{a}{m} + (n - c + 1) \cdot \frac{c}{m}$$

Der erste der beiden Summenterme beschreibt die Füllphase, der zweite Summenterm die weiteren Anfragen bis zum Ende des Laufes.

Der tatsächliche Wert der Compulsory Misses hängt stark von der Art der Anfrageliste ab – bei vielen Wiederholungen (z.B. viele Knoten laufen hintereinander auf demselben Pfad, Graph-Walk Mobilitätsmodell) ist die Anzahl der Compulsory Misses im Schnitt geringer als für den Fall, dass wenige Knoten isoliert voneinander auf unterschiedlichen Wegen durch das Simulationsareal laufen (Random-Waypoint Mobilitätsmodell).

## 4.2.2 FIFO und Varianten

### FIFO

Der grundlegende FIFO Algorithmus (siehe 3.4.3) hält die Elemente im Cache in einer Warteschlange. Bei jedem Hinzufügen eines Elementes nach einem Cache Miss rücken die anderen Elemente in der Schlange auf, d.h. ihrer Löschung entgegen. Zusätzlich verwenden wir (siehe Abbildung 4.4) eine Hashmap zum schnelleren Auffinden von Elementen, andernfalls müsste bei jeder Suchoperation die gesamte Warteschlange durchsucht werden.

Das Löschen eines Elementes aus der Schlange mit der Cachegröße  $c$ , das diese nach  $c$  Compulsory Misses komplett durchlaufen hat, bezeichnen wir hier als „Wraparound“. Beim einfachen FIFO-Algorithmus sind diese Compulsory Misses *nicht* mit den einfachen Misses gleichzusetzen! Nach einem Cache Miss ist das angefragte Element also im Cache, es bleibt in diesem während der nächsten  $(c - 1)$  Compulsory Cache Misses enthalten.

Wir gehen nun bei der Abschätzung vereinfachend davon aus, dass die Compulsory Cache Misses gleichmäßig über die Anfrageliste verteilt sind. Dies bedeutet für die durchschnittliche Verweildauer  $t_d$  (in Anfragen) eines Elementes im Cache nach dessen Auftreten:

$$t_d = \frac{n}{m} \cdot c$$

Durchschnittlich findet alle  $\frac{n}{m}$  Anfragen ein Compulsory Cache Miss statt.

Nehmen wir nun an, gleichartige Elemente wären gleichmäßig über die Anfrageliste verteilt. Bei drei Elementen A, B und C wäre eine gleichmäßige Reihenfolge ABCABCABC, nicht jedoch AABCBCABC. Bei ersterer Reihenfolge ist der Abstand der drei Elemente A, B und C jeweils 3. In einer gleichmäßigen Reihenfolge wiederholen sich die gleichartigen Anfragen alle  $\frac{n}{m}$  Elemente. Für Cachegrößen  $c < \frac{n}{m}$  würde es nur Cache Misses geben, solange sich alle Knoten an unterschiedlichen Orten befinden, welche auch unterschiedlichen Clustern zugewiesen werden. Beispielsweise gibt es bei der Reihenfolge ABCABCABC für Cachegrößen unter 3 nur Cache Misses.

FIFO profitiert also bei gleichmäßig auf die Anfrageliste (oder auf im Verhältnis zu  $c$  größere Abschnitte davon) verteilten Elementen davon, wenn es nicht mehr verschiedene sich wiederholende Elemente gibt als der Cache fassen kann. Umgekehrt argumentiert, ist es für

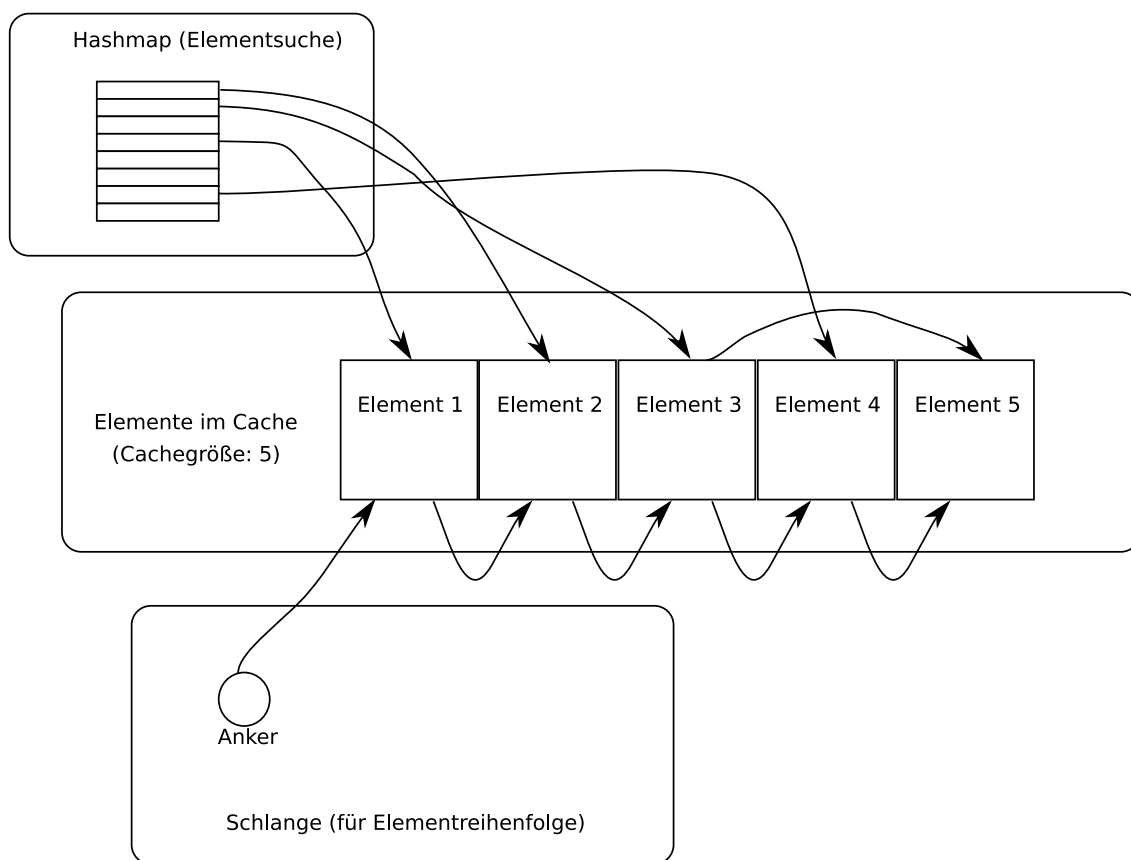


Abbildung 4.4: Mögliche Datenstruktur für FIFO, FIFO Second Chance, LRU

das Erzielen einer hohen Erfolgsrate für den Cache wichtig, dass die gleichmäßige Verteilung der Elemente möglichst oft in der Anfrageliste durchbrochen ist und der Cache groß genug ist, um von dieser Tatsache zu profitieren.

CanuMobiSim lässt zu jeder Zeiteinheit in der Simulation jeden Knoten alle anderen Knoten auf ihrer Position abfragen, damit generiert es die Anfrageliste. Bei  $k$  Knoten gibt es also pro Simulations-Zeiteinheit  $k^2 - k$  Anfragen. Wenn sich die Knoten immer an voneinander unterschiedlichen Positionen befinden, was vorwiegend der Fall ist, und sich auch nicht in denselben Clustern befinden, so entsteht eine Anfrageliste mit einer speziellen Struktur. Diese Struktur hat, wenn alle Knoten stationär sind, genau die am Beispiel ABCABCABC genannte Regelmäßigkeit und gleichmäßige Verteilung der Anfragen auf Teilstücke: Wenn ein Knoten A die Position eines Knoten B anfragt, so wird sich dieses i.d.R. in der nächsten Zeiteinheit – also  $k^2 - k$  Anfragen später – wiederholen.

Eine Abweichung hiervon entsteht lediglich, wenn Knoten sich in denselben Clustern befinden. In der Simulation ist dies durchaus möglich, beispielsweise an wichtigen Verkehrsknotenpunkten ist dies oft der Fall.

### FIFO Second Chance

FIFO Second Chance unterscheidet sich vom reinen FIFO dadurch, dass Elemente markiert werden, die während ihres Durchlaufens der FIFO-Schlange nochmals verwendet werden. Erreichen Elemente mit Markierung das Ende der Schlange, so wird die Markierung gelöscht und die Elemente nochmals für einen weiteren Durchlauf an den Kopf der Schlange angefügt.

Elemente, die während der bereits beim Abschnitt über FIFO bestimmten Verweildauer  $\frac{n}{m} \cdot c$  referenziert werden, werden also niemals gelöscht, vorausgesetzt, es sind genügend andere Elemente vorhanden, die nicht im letzten Durchlauf verwendet und damit ebenfalls markiert wurden. Werden alle im Cache enthaltenen Elemente verwendet, so degeneriert FIFO Second Chance zu reinem FIFO.

### LRU

Der LRU-Algorithmus (Least Recently Used) verwendet ebenfalls eine Schlange, in der alle Cache-Elemente aufgereiht sind. Wird ein Element im Cache verwendet, so wird es an den Kopf der Schlange versetzt. Je länger ein Element  $e$  also nicht benutzt wird, desto weiter nach hinten wird es in der Schlange verschoben. Mit jeder Verwendung eines anderen Elements wird das Element  $e$  in der Schlange weiter nach hinten durchgereicht. Soll ein neues Element in den vollen Cache aufgenommen werden, so wird das Element am Ende der Schlange gelöscht.

Im Gegensatz zu einigen der anderen Algorithmen muss der Cache bei LRU auch bei Lesezugriffen umgeordnet werden, da LRU aus der Verwendung von Elementen „lernt“ (das Wissen besteht in der Reihenfolge der Schlange). Die Idee dabei ist, dass gegenwärtig häufig verwendete Elemente voraussichtlich auch in der Zukunft häufig verwendet werden.

Die maximale Verweildauer eines Elementes  $e$ , welches nicht mehr verwendet wird, beträgt ab seiner letzten Verwendung zum Zeitpunkt  $t$  bei LRU  $c - 1$  Anfragen, wenn nach dem Zeitpunkt  $t$  die nächsten  $c - 1$  Anfragen alle verschieden und ungleich  $e$  sind. Mit jedem Vorkommen einer Anfrage ungleich  $e$ , die seit  $t$  nicht mehr vorkam, rückt  $e$  einen Schritt dem Ende der Schlange näher. Befindet sich Element  $e$  dann am Ende der Schlange, wird es beim Auftreten des nächsten nicht im Cache befindlichen Elements aus dem Cache entfernt.

### 4.2.3 LIL (Linear Interpolated Lookahead)

Der im Rahmen dieser Arbeit entstandene LIL-Algorithmus (Linear Interpolated Lookahead) versucht, durch lineare Interpolation der momentanen Bewegung eines jeden Knotens vorherzusagen, wo sich der Knoten in Zukunft befinden wird. Ein Lookahead-Wert gibt an, wie viele Zeitschritte in die Zukunft geschaut wird.

Die Dämpfungsdaten sind, wie bereits in Abschnitt 3.5.2 erwähnt, festgelegt durch die jeweiligen  $x$ - und  $y$ -Koordinaten von Sender und Empfänger. Die Dämpfungswerte befinden sich in einem 4D-Raum. Jeder Knoten hat seine eigene Datenstruktur. Zusätzlich zur momentanen Position des Knotens ist in der Datenstruktur auch der nächste Zielpunkt (d.h. der Zielpunkt der Trajektorie), der zurückgelegte Weg und die Geschwindigkeit des Knotens erfasst. Zur einfacheren Erklärung der Sachverhalte betrachten wir im Folgenden nur zwei der Dimensionen, damit eine Visualisierung auf Papier einfacher möglich ist.

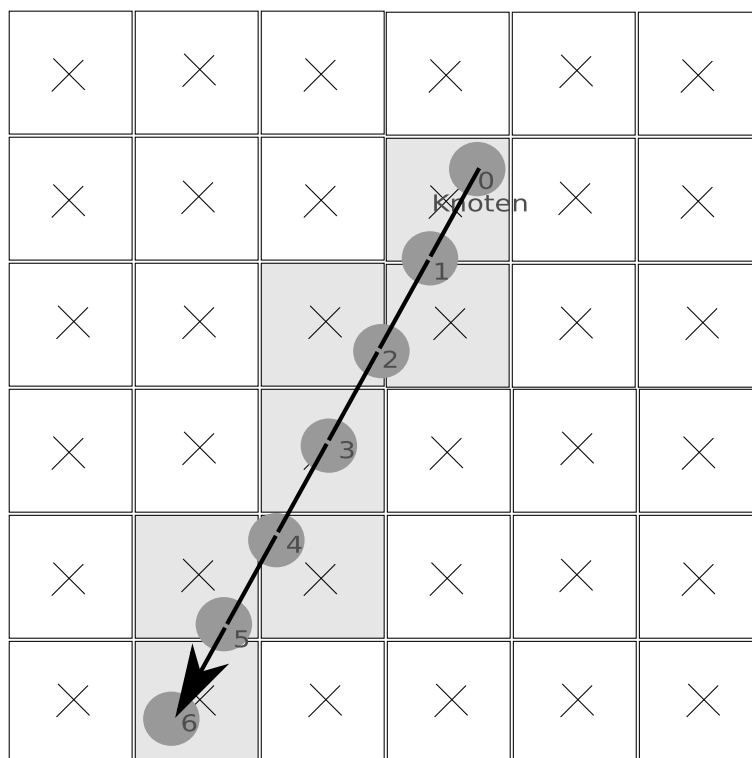


Abbildung 4.5: Prinzip von LIL, Trajektorie

Der in Abbildung 4.5 dargestellte Knoten befindet sich zur Zeit der Simulation an der mit 0 beschrifteten Position, er hat also unmittelbar vorher eine neue Trajektorie (d.h. Geschwindigkeit, 2D-Startkoordinaten, 2D-Zielkoordinaten) erhalten. Diese Trajektorie arbeitet er in den nächsten Zeitschritten solange ab bis eine neue Trajektorie gesetzt wird. Die Trajektorie ist in der Abbildung durch den Vektorpfeil dargestellt.

Es wird nun simuliert, an welchen Positionen sich der Knoten und alle seine Empfänger (d.h. alle anderen Knoten) in den nächsten  $h$  Zeitschritten befindet. Dieses  $h$  nennen wir den Vorausschau-Horizont, bis zu dem versucht wird, die Zukunft zu berechnen.

Die abgelaufene Zeit wird im Folgenden mit  $t$  bezeichnet, die Variable  $z$  wird verwendet, um die Zeitschritte von 1 bis  $h$  hochzuzählen. Die Menge der existierenden Knoten wird mit  $K$  bezeichnet, der Cache mit  $C$ . Die Verwendungszähler der Elemente im Cache werden mit  $u$  (bzw.  $u_M$  für ein Element  $M$ ) bezeichnet.

Für jede Anfrage wird nun folgender Algorithmus durchlaufen:

```

Knoten  $K_A$  = Knoten, der die Anfrage stellt;
for ( $z = 1; z \leq h \wedge \text{abgelaufeneZeit} + z \leq \text{Zielzeit}; z = z + 1$ ) do
  PosA = Knotenposition von  $K_A$  zum Zeitpunkt  $t + z$ ;
  for Alle existierenden Knoten  $K_B \in K \setminus \{K_A\}$  do
    PosB = Knotenposition von  $K_B$  zum Zeitpunkt  $t + z$ ;
    Anfrage  $Q = (X_{\text{PosA}}, Y_{\text{PosA}}, X_{\text{PosB}}, Y_{\text{PosB}})$ 
    if  $Q \in C$  then
      |  $u_Q = u_Q + 1$ ;
    end
  end
end
Cache-Element  $e$  mit  $u_e = \min\{u_j\}$  aus  $C$  löschen;
 $C = C \cup \{Q\}$ ;
 $u_C = 0$ ;

```

### Algorithmus 3 : LIL-Algorithmus

Der Algorithmus führt kein Prefetching durch. Elemente, die sich bereits im Cache befinden und voraussichtlich in der Zukunft innerhalb von  $z$  Schritten wieder benötigt werden, behält der Algorithmus jedoch bevorzugt im Cache, da ihr Verwendungszähler ja mit jeder Prüfung wächst.

Um eine größere Anzahl an möglichen Knotenkandidaten für die zukünftige Verwendung zu erhalten, werden die Verwendungszähler  $u_Q$  aller Knoten im Cache von Ablauf zu Ablauf des Algorithmus beibehalten. Nach jedem Zeitschritt der Emulation findet eine Verringerung der Verwendungszähler durch „Alterung“ statt:

$$u_Q = \alpha \cdot u_Q$$

Der Faktor  $\alpha$  ( $0 < \alpha < 1$ ) zur „Alterung“ sorgt dafür, dass Elemente, die eine Zeit lang sehr häufig verwendet wurden, dann aber nicht mehr vorkommen, zugunsten von anderen Elementen aus dem Cache entfernt werden. Je größer  $\alpha$  gewählt wurde, desto länger bleiben früher stark frequentierte Elemente im Vergleich zu aktuell stark frequentierten Elementen im Cache.

Der Algorithmus zählt, wie oft im Cache befindliche Elemente voraussichtlich in Zukunft noch gebraucht werden. Dabei wird bis zu  $h$  Zeiteinheiten in die Zukunft geschaut. Es wird die Annahme getroffen, dass die Trajektorie nicht geändert wird – dies ist jedoch in der Praxis nicht zutreffend, was einen Schwachpunkt dieses Ansatzes darstellt. Vor allem beim Durchlaufen von Kurven wird in der Praxis häufig eine neue Trajektorie gesetzt, um die Kurve anzunähern. Es ist zu erwarten, dass die Annäherung an eine Kurve durch lineare Interpolation zu großen Ungenauigkeiten und damit zu schlechten Voraussagen des LIL-Algorithmus führen wird.

Zu regelmäßigen Zeitpunkten setzt der Algorithmus alle LIL\_Usage Zähler auf 0, damit Elemente, die nur über einen gewissen Zeitraum oft verwendet werden – in späteren Zeiträumen jedoch nicht mehr – nach einiger Zeit wieder aus dem Cache verschwinden.

Kritisch ist die richtige Auswahl von  $h$ . Der Wert legt fest, wie weit der Algorithmus in die Zukunft schaut. Ist dieser Wert zu groß, so steigt der Rechenaufwand deutlich, da jeder Schritt ein Nachschauen im Cache nötig macht. Bei häufigen Richtungsänderungen der Knoten (d.h. Zuweisung von neuen Trajektorien) wird die Vorausschau falsche Ergebnisse liefern, da sie ja

immer nur von der derzeitigen Trajektorie ausgehen kann. Ändert sich die Trajektorie vor ihrem Ablauf, so werden Cache-Elemente bevorzugt, die in Wahrheit nicht mehr verwendet werden. Ist  $h$  zu klein, so wird möglicherweise ein Teil des Potenzials des Algorithmus verschenkt.

Wir untersuchen nun anhand des in Abbildung 4.6 dargestellten Beispiels, in welchen Knoten-Konstellationen der LIL-Algorithmus zu Verbesserungen führt. Dargestellt sind – zur Vereinfachung wiederum in 2D statt im 4D-Raum – fünf Trajektorien T1 bis T5 und sechs Knoten K1 bis K6. Die Knoten bewegen sich entlang der Trajektorie, auf der sie jeweils dargestellt sind, also K1 auf T1, K2 und K3 auf T2, K4 auf T3, K5 auf T4 und K6 auf T5.

Wir betrachten zunächst einen einfachen FIFO-Cache. Gehen wir nun davon aus, dass sehr viele weitere (hier nicht dargestellte und in anderen Regionen der Karte befindliche) Knoten und Trajektorien existieren. Außerdem hat der Cache bei  $n$  Knoten nur die Größe  $n^2$ , sodass durch die vielen Anfragen ein sehr hoher Anteil an Cache Misses erzeugt wird. In dieser Situation wird es überhaupt nur Cache Hits geben, wenn zwei oder mehr Knoten sich am selben Ort befinden, da dann dieselben Anfragen mehrmals während  $n^2$  Anfragen an den Cache gestellt werden.

LIL versucht, diese Tatsache auch für zukünftige Anfragen auszunutzen: Wenn sich ein Knoten an einem Punkt befindet, und sich ein weiterer Knoten voraussichtlich auch bald an diesem Punkt aufhalten wird, so wird das betreffende Element im Cache bevorzugt, d.h. das bevorzugte Element wird zuletzt gelöscht. Die beschriebene Situation tritt auf, wenn Knoten sich auf ihrem Weg kreuzen (z.B. Knoten K2 und K5) oder auf ähnlichen Trajektorien folgen (z.B. Knoten K2 und K3).

Nehmen wir nun an, nur  $k$  der  $n$  Knoten bewegten sich. Beispielsweise wird K2 voraussichtlich zum Zeitpunkt  $o$  an dem Ort sein, an dem sich K3 zum Zeitpunkt  $o - p$  befindet. Die Anfragen  $(X_{K3}, Y_{K3}, X_A, Y_A)$  zu allen anderen Knoten A werden also voraussichtlich bald in der Zukunft (in  $p$  Zeiteinheiten) nochmals auftreten. Deshalb werden diese Anfragen als bevorzugt markiert. Es handelt sich hier demnach nur um von K3 ausgehende  $2 \cdot (n - 1)$  Anfragen. Dies ist im Vergleich zur angenommenen Cachegröße  $n^2$  eine deutlich kleinere Anzahl von Anfragen.

Da sich auch die anderen der  $k$  Knoten bewegen, ergibt sich pro Zeiteinheit in der Simulation eine Anzahl von  $2 \cdot k \cdot (n - 1)$  Anfragen, die bevorzugt werden sollten, wobei gilt:  $0 \leq k \leq n$ .

$k$  lässt sich in der Praxis nur sehr grob abschätzen – CanuMobiSim ermöglicht es, eine Wartezeit für Knoten festzulegen, nachdem sie ein Ziel erreicht haben. Anhand dieser lässt sich ungefähr auf die Anzahl der im durchschnittlich gleichzeitig in Bewegung befindlichen Knoten schließen.

Ein Problem des Algorithmus ist, dass er auch falsche Entscheidungen treffen kann. K2 beispielsweise könnte sowohl auf T2 dem Knoten K3 folgen oder auch abbiegen und auf T5 dem Knoten K6 folgen. Ebenso könnte der Knoten unvermittelt einen komplett anderen Weg einschlagen, auf dem sich bisher noch kein anderer Knoten bewegt hat. Es muss jedoch zu jedem Zeitpunkt die Entscheidung getroffen werden, auf welchem Weg die Cache-Einträge reserviert werden sollen oder nicht. Diese Entscheidungen sind spekulativ. Fehlentscheidungen führen normalerweise zu schlechterer Cache-Performanz, da unnötige Elemente im Cache gehalten werden.

Im Rahmen dieser Arbeit wurde auch die Möglichkeit der Vorhersage von zukünftigen Knotenpositionen durch polynomielle Interpolation und kubische Splines geprüft. Bei diesem Ansatz wurden als Datenbasis alle aufgetretenen Positionen der Knoten in einem festgeleg-

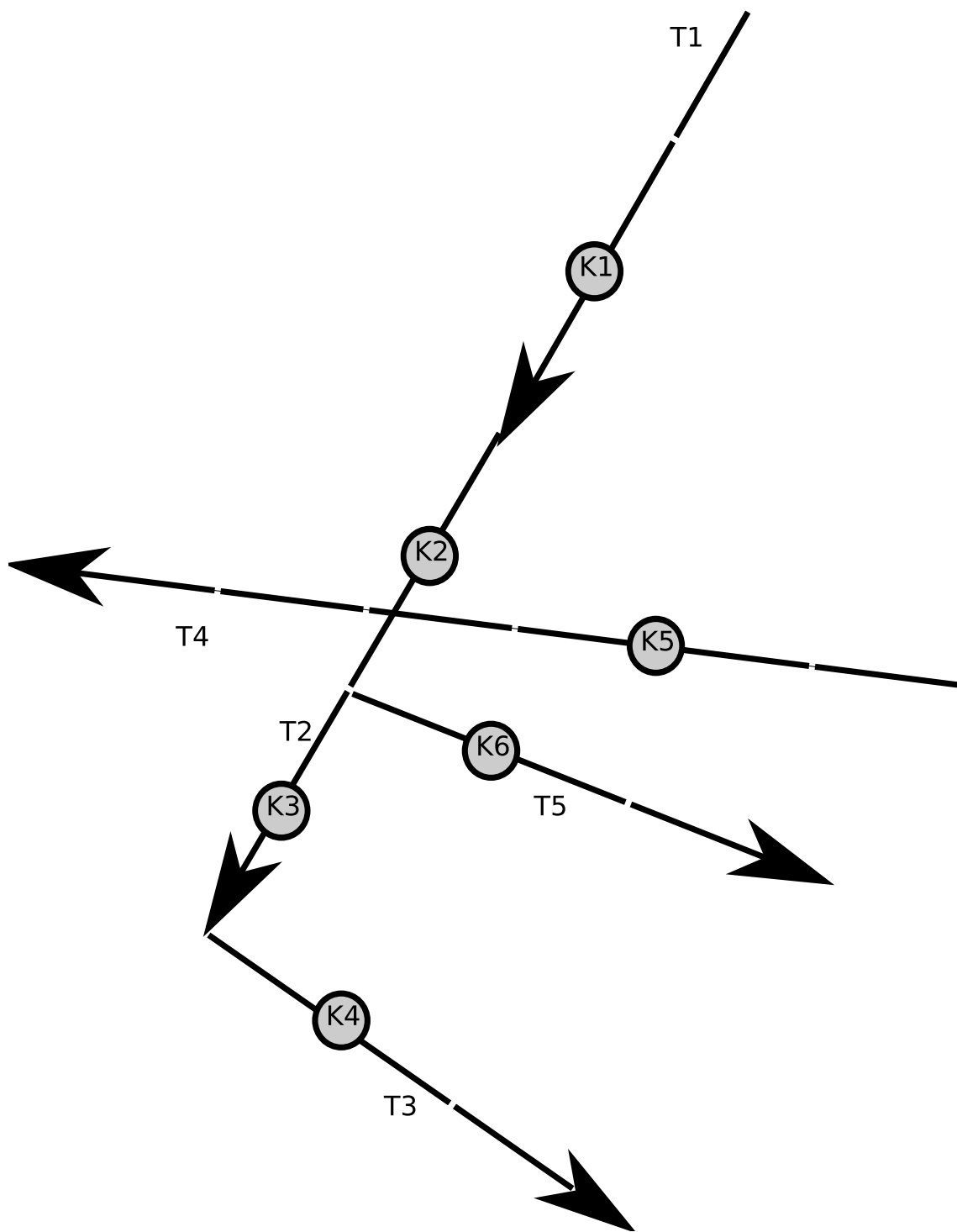


Abbildung 4.6: Beispiel zu LIL

ten Zeitraum in der Vergangenheit verwendet. Aus diesen Daten wurden, getrennt nach den Koordinatenachsen, die Funktionen der Knotenposition in Abhängigkeit von der Zeit aufgestellt. Die vorhandenen Daten sind hierbei die Stützpunkte der aufzustellenden Funktionen. Das Kalkül war, dass eine Vorhersage der Knotenposition durch das Einsetzen von zukünftigen Zeiten in die aufgestellte Funktionen möglich sein sollte. Hierbei sollte die Tatsache ausgenutzt werden, dass die Knoten oft in Zukunft ihre vorhandene Richtung beibehalten. Weder die Interpolation mit kubischen Splines noch die Bezier-Interpolation hat hierbei jedoch verwendbare Resultate geliefert. Außerhalb des bekannten Bereiches knicken die Kurven üblicherweise scharf ab, was nicht den hier anzutreffenden Bewegungsmustern entspricht. Bei höhergradigen Polynom-Interpolationen kommen außerdem auch im bekannten Bereich der Funktion sehr starke Schwankungen der Werte hinzu. Nach Betrachtung einiger Resultate wurde der Ansatz daher verworfen. Die Problematik von Kreuzungen, an denen mehrere betrachtete Knoten unterschiedliche Wege einschlagen, lässt sich mit dem Interpolationsansatz ebenfalls nicht lösen.

#### 4.2.4 Vorläufiger Vergleich der Ansätze

##### Vergleich der Erfolgsrate

Angenommen, alle jeweiligen Optimierungsansätze der Algorithmen greifen auch tatsächlich, so ist folgende Reihenfolge der Algorithmen, geordnet nach steigender Erfolgsrate, zu erwarten: FIFO, FIFO-Secondchance, LRU, LIL, OPT. Der RANDOM-Algorithmus lässt sich nicht ohne weiteres einordnen.

Die FIFO-Varianten FIFO, FIFO-Secondchance und LRU sind in dieser Reihenfolge nach steigender Anzahl der normal vorkommenden Umordnungen innerhalb des Caches (d.h. dessen Lösch-Warteschlange) geordnet. Ob diese Umordnungen auch tatsächlich Vorteile bringen, wird sich später zeigen.

Der LIL-Algorithmus verhält sich prinzipiell ähnlich wie die anderen FIFO-Varianten. Sein Erfolg oder Misserfolg wird jedoch von der Korrektheit der Vorhersagen zukünftiger Knotenpositionen bestimmt. Es ist zu erwarten, dass der Algorithmus in einigen Fällen besser, in anderen schlechter abschneidet.

##### Vergleich der nötigen Rechenleistung

In Bezug auf die benötigte Rechenleistung ist der LIL-Algorithmus der teuerste, da hier spekulativ für jede Kombination an Trajektorien Gleitkomma-Berechnungen durchgeführt werden. Der Aufwand für RANDOM ist bei Vorhandensein einer guten Pseudo-Zufallsfunktion eher gering, da keine Listen o.ä. umgehängt werden müssen.

Der OPT-Algorithmus hat je nach Realisierung unterschiedliche Komplexität (siehe Abschnitt 4.1.4), dürfte sich jedoch für die meisten Anwendungsfälle zwischen den FIFO-Varianten nebst RANDOM und den anderen Algorithmen befinden.

Der LRU-Algorithmus braucht geringfügig mehr Rechenleistung als die restlichen FIFO-Varianten, da für jeden Cache-Zugriff ein Element in der Liste umgeordnet werden muss. Bei FIFO-Secondchance kommen solche Umordnungen nur bei Cache Hits vor, bei FIFO überhaupt nicht. Insgesamt kann man sagen, dass die FIFO-Varianten sich in der Praxis nicht wesentlich bezüglich ihrer Kosten unterscheiden.

### 4.2.5 Hotspot-Prefetching

Hotspot-Prefetching und seine Varianten wurden im Rahmen dieser Diplomarbeit erdacht. Beim Hotspot-Prefetching ist der Cache in zwei Teile aufgeteilt: Einen Teil A, der nach einer beliebigen anderen Caching-Strategie anhand der Anfragen verwaltet wird, und einen Teil B, der nach vorher festgelegten Gesichtspunkten mit Daten gefüllt wird.

Der Algorithmus versucht die Dämpfungswerte rund um Punkte, die oft von Knoten besucht werden (sogenannte Hotspots), im Teil B des Caches zu halten. Dabei werden die Dämpfungswerte in einem festgelegten Radius rund um die Hotspots zu allen anderen Hotspots (ebenfalls mit deren Umkreis) im Cache gespeichert. Die Hotspots sind durch den Benutzer der Simulation im Voraus festgelegt worden. Das gute Funktionieren dieser Strategie hängt von der Korrektheit der Hotspot-Festlegungen ab.

Die Hotspots werden rund um Kreuzungen von Straßen festgelegt. Die Annahme ist hierbei, dass sich rund um die Kreuzungen statistisch gesehen öfter Knoten aufhalten als an anderen Orten. Wenn man davon ausgeht, dass die Knoten gleichmäßig auf alle Straßen verteilt sind und die Hotspots kreisförmig sind, so ergibt sich eine größere Straßenlänge pro Fläche in den Hotspots, die Annahme trifft in diesem Fall zu.

Prinzipiell wäre eine Festlegung der Hotspots auch auf Clusterebene möglich, hierbei müsste jedoch eine getrennte Auswertung der Straßen stattfinden, was die Architektur des Frameworks unnötig komplizieren würde.

### 4.2.6 Lernendes Hotspot-Prefetching

Beim lernenden Prefetching sammelt der Algorithmus während seiner Laufzeit Daten darüber, wo sich die Knoten befinden. Anhand der Verteilung der Aufenthaltsdauer der Knoten werden die Positionen der Hotspots und deren Radien festgelegt. Es wird dann Hotspot-Prefetching wie im letzten Abschnitt beschrieben durchgeführt. In regelmäßigen Abständen werden die definierten Hotspots aktualisiert und Teil B des Caches wird angepaßt.

Je mehr Wissen der Algorithmus über den Aufenthalt der Knoten bekommt, desto passender können die Hotspots gewählt werden. Für lernendes Hotspot-Prefetching sind die im Rahmen dieser Arbeit vorkommenden Simulationszeiträume von bis zu 900 Sekunden jedoch deutlich zu kurz, da nach so kurzer Zeit nur sehr wenige Knotenbesuche stattfinden und damit noch keine statistisch sinnvoll verwertbaren Informationen vorliegen können. Eine Lösung dieses Problems wäre, die in jedem Simulationslauf gewonnenen Informationen nach dem Durchlauf zu speichern. Die Auswertungen könnten dann laufend oder auch bei Überschreiten einer gewissen Datenmenge durchgeführt werden. Voraussetzung ist die richtige Zuordnung der unterschiedlichen Szenarien (d.h. die Lage der Straßen und Wegpunkte etc.) zu den gesammelten Knoten-Besuchsdaten, falls mehrere Szenarien verwendet werden. Dies könnte durch einen Hash-Wert über die Daten des Szenarios passieren, aus dem eine eindeutige Kennung für jedes Szenario gebildet wird.

Eine Erkennung von Hotspots könnte auch in einer vereinfachten Variante durch die Aufzeichnung der Orte, an denen Knoten stillstehen, mit der dazugehörigen gesamten Verweildauer der Knoten erfolgen.

### 4.2.7 Aufteilung des Caches

Ein Ansatz ist, den Cache in drei verschieden große Partitionen aufzuteilen. Eine Partition  $P_1$  speichert die Dämpfungswerte der Hotspots untereinander, eine weitere Partition  $P_2$  speichert Dämpfungswerte der Positionen des Straßennetzes untereinander, eine dritte Partition  $P_3$  speichert alle weiteren anfallenden Anfragen.

Jede der Partitionen hat ihre eigene Cache-Ersetzungsstrategie.

$P_1$  ist der Hotspot-Cache, er wird per Hotspot-Prefetching oder lernendem Hotspot-Prefetching verwaltet. Er enthält statisch die Dämpfungswerte zwischen den Hotspots. Bei der Anzahl von  $m$  Hotspots (der Einfachheit halber gehen wir hier von quadratischen Hotspots aus) der Kantenlänge  $k$  sind in jedem Hotspot also  $k^4$  Positionen möglich, zu allen anderen Hotspots sind dies also  $(m - 1) \cdot k^4$  Werte. Insgesamt hat der Hotspot-Cache  $P_1$  also  $m \cdot (m - 1) \cdot k^4$  Werte. Bei im Vergleich zur Gesamtgröße der Simulation geringen Hotspotradien und bei geringer Anzahl von Hotspots dürfte hier eine gute Ausnutzung der gespeicherten Daten durch Anfragen zu erreichen sein.

$P_2$  ist der Cache mit den Dämpfungswerten zwischen den Positionen der Straßen und Wege, er enthält wesentlich mehr Anfragen als der  $P_1$ -Teil. Es ist zu erwarten, dass ein Großteil der nicht von  $P_1$  erfassten Anfragen in die Kategorie  $P_2$  fällt. Wir teilen das Straßennetz für eine Abschätzung der Datenmenge in Teilstücke gleicher Länge auf. Bei einer Anzahl von  $l$  diskreten Punkten, die auf jedem Teilstück liegen, ergeben sich bei  $m$  Teilstücken insgesamt  $(l \cdot m)^2$  Dämpfungswerts-Datensätze für alle Teilstücke. Die Anzahl der zu speichernden Dämpfungswerte kann somit für kleine Werte von  $l$  gering gehalten werden. Wichtig für das Funktionieren des Algorithmus ist, dass die Punkte auf den Teilstücken im Abstand der Cluster gesetzt werden.

$P_3$  enthält alle Anfragen, die nicht in den ersten beiden Cache-Partitionen gespeichert werden.

Es wird erwartet, dass in den Cache-Partitionen  $P_1$  und  $P_2$  wesentlich höhere Trefferraten erreichbar sind als in  $P_3$ . Auch bei deutlicher Vergrößerung der ersten beiden Partitionen mit gleichzeitiger Vergrößerung von deren Parametern  $k$ ,  $l$  und  $m$  dürfte eine Erhöhung der Cache Hits möglich sein.

Beeinflussende Faktoren sind hierbei – nebst den genannten Parametern und der Cachegröße – die folgenden:

- Häufigkeit des Aufenthalts der Knoten an Hotspots
- Häufigkeit des Aufenthalts der Knoten auf den Straßen
- Fläche und Anzahl der Hotspots bzw. Länge und Anzahl der Straßen im Verhältnis zur Cachegröße

# Kapitel 5

## Evaluation

### 5.1 Simulationsframework für diese Arbeit

Im Rahmen dieser Arbeit entstand ein Simulationsframework, implementiert in der Programmiersprache C, welches sich „CacheSim“ nennt und folgende Funktionalitäten bietet:

- Einlesen der Mobility Traces und interne Generierung der Anfrageliste
- Bestimmung der optimalen Cache-Ersetzungssequenz durch eine Implementierung des OPT-Algorithmus
- Bestimmung der „Compulsory Misses“
- Erstellung eines Histogramms mit den auftretenden TNR-Distanzen (siehe auch Abschnitt 4.1.2)
- Simulation verschiedener Cache-Ersetzungsstrategien

#### Einlesen der Mobility Traces und Generierung der Anfrageliste

CacheSim erhält die Eingabe aus einer Datei, welche zu Beginn die Anfangsposition aller Knoten in der Simulation enthält. Danach folgen nach Bedarf Einträge mit der Änderung von Zielpunkten und Geschwindigkeit von Knoten.

Die Eingabe erfolgt zeilenweise in einer Textdatei. Hier ein Ausschnitt aus einer Beispieldatei:

```
$node_(0) set X_ 1398.0837480065454
$node_(0) set Y_ 649.0828697055036
$node_(0) set Z_ 0.0
$node_(1) set X_ 1107.0452177773038
$node_(1) set Y_ 1007.4587344860116
$node_(1) set Z_ 0.0
[...]
$ns_ at 0.0 "$node_(0) setdest 1386.3021613582894 660.1713706112856 0.086263545"
```

```

$ns_ at 0.0 "$node_(1) setdest 1119.5397628307035 1005.2642004319856 0.678752"
[...]
$ns_ at 22.555 "$node_(0) setdest 1215.2878480484867 1218.9150927753578 0.66984123"
$ns_ at 23.138 "$node_(1) setdest 1702.5946835373686 815.4365040406043 0.91078967"
[...]

```

Die \$node Zeilen dürfen nur am Anfang der Datei als durchgängiger Block vorkommen. Sie geben für jeden Knoten (die Knotennummer befindet sich in der Klammer) die anfängliche x- und y-Position an. Die dritte Koordinate wird derzeit nicht verwendet.

Die \$ns Zeilen geben an, zu welchen Zeiten (angegeben durch „at“ gefolgt vom Zeitpunkt als Float-Zahl) sich die Richtung und Geschwindigkeit von Knoten (die Knotennummer findet sich wiederum in der Klammer) ändern. Hinter dem Schlüsselwort setdest werden der neue Zielpunkt (x- und y-Koordinaten) sowie die neue Geschwindigkeit in m/s angegeben.

CacheSim arbeitet in diskreten Zeitschritten, standardmäßig ist ein Zeitschritt 1 Sekunde. Jede Sekunde (also zu den Zeitpunkten 0, 1, 2, ... Sekunden) wird von jedem Knoten anhand der Eingabedatei und der intern gehaltenen Bewegungsrichtung und Geschwindigkeit jedes Knotens bestimmt, an welcher Position er sich befindet. Die Positionen der Knoten werden an dieser Stelle in Metern auf der Karte (x- und y-Koordinaten, als Gleitkomma-Zahlen) festgehalten. Durch Ganzzahl-Division werden die Werte nun auf Koordinaten im Raster der Dämpfungswerte (als ganze Zahlen) umgewandelt.

Bei einem Clusterfaktor ungleich 1 wird dann der Ankerpunkt des Clusters (siehe Abbildung 3.2) zu jeder Anfrage um den Knoten herum bestimmt, mit diesen Koordinaten wird anschließend weiter simuliert. Damit ist das Clustering berücksichtigt. So wird simuliert, dass jeder Cluster alle Dämpfungswerte innerhalb des von ihm aufgespannten Hypercubes enthält.

Nun kann die Anfragesequenz generiert werden: Wie erwähnt kommuniziert jeder Knoten bei der gegebenen Problemstellung pro Zeitschritt einmal mit allen anderen Knoten. Eine Anfrage mit den Dämpfungswerten zu den Koordinaten  $(x_S, y_S, x_E, y_E)$  wird in einem Feld abgelegt. Die Anzahl der Anfragen bei  $n$  Knoten und  $r$  Zeitschritten beträgt  $n \cdot (n - 1) \cdot r$ .

### Bestimmung der optimalen Cache-Ersetzungssequenz

Nachdem die Sequenz der Anfragen an den Cache vorliegt, kann mit der Bestimmung der optimalen Ersetzung (OPT) begonnen werden. Eine Diskussion der Komplexitäten der verschiedenen OPT-Algorithmen befindet sich in Abschnitt 4.1.4. Es wurde der Algorithmus von Mattson et al. gewählt.

### Bestimmung der „Compulsory Misses“ und „Conflict Misses“

„Compulsory Misses“ können bei der OPT-Realisierung nach Mattson et al. einfach dadurch bestimmt werden, dass die Anzahl der Vorkommen des TNR-Werts  $\infty$  gezählt wird. Jede Anfrage bekommt in der Anfrageliste den TNR-Wert unendlich genau dann zugewiesen, wenn es das letzte Auftreten dieser Anfrage vor dem Ende der Anfrageliste ist. Dies trifft sowohl bei Anfragen zu, die es insgesamt nur einmal in der Anfrageliste gibt, als auch bei Anfragen, die mehrmals in der Anfrageliste vorkommen.

Aufgrund dieses Sachverhalts kann die Anzahl der unterschiedlichen Anfragen in der Anfrageliste einfach bestimmt werden. Jede unterschiedliche Anfrage erzeugt bei ihrem ersten Auftreten einen Compulsory Miss. Die tatsächlich in der Simulation mit einer bestimmten Cache-Größe vorgekommenen Seitenfehler abzüglich der „Compulsory Misses“ ergeben die Anzahl der „Conflict Misses“.

### Histogramm der TNR-Distanzen

Eine Sequenz von Anfragen ist auch charakterisiert durch die Abstände, welche gleiche Anfragen an den Cache untereinander haben. Die TNR-Distanz einer Anfrage zu einem Zeitpunkt  $t_1$ , die zum Zeitpunkt  $t_2$  wieder auftritt, ergibt sich also mit  $t_2 - t_1$ .

Je größer die TNR-Distanzen, desto schwieriger ist es in der Regel für einfache Cache-Algorithmen wie FIFO, Conflict Misses zu vermeiden. Es besteht ein Zusammenhang zwischen Cache-Größe und der Verteilung der Häufigkeit der unterschiedlichen TNR-Distanzen. Kommt beispielsweise eine Distanz  $d$  zwischen zwei Elementen in der Sequenz der Zugriffe  $h$  mal vor, so ist sicher, dass bei der FIFO-Strategie bei einer Cachegröße kleiner  $d$  mindestens  $h$  Cache Misses durch dieses Element erzeugt werden. Im Rahmen der Auswertung der Ergebnisse werden wir später in diesem Kapitel einige TNR-Histogramme präsentieren.

### Simulation der Ersetzungsstrategien

Die Simulation der Ersetzungsstrategien erfolgt durch Laden der Knoten-Anfangspositionen und Trajektorien aus dem Mobility-Trace-File. CacheSim spielt nun intern das Szenario durch und bestimmt für jeden Zeitschritt die Position aller Knoten. Aus diesen Positionsdaten können die Grid-Punkte bzw. Cluster bestimmt werden, daraus die Liste der Anfragen. Jetzt wird die Liste der Anfragen abgearbeitet, dabei der Inhalt des Caches mitgeführt und mit den zum jeweiligen Algorithmus passenden Zusatzinformationen und Daten ergänzt. Der OPT-Modus unterscheidet sich von den anderen Simulationen, hier wird aus den Anfragen im Verfahren nach Mattson et al. die optimale Ersetzungsreihenfolge generiert. Dabei kann die Anzahl der Compulsory Misses und Conflict Misses getrennt gezählt werden, bei den anderen Verfahren werden diese nur als Summe erfasst.

## 5.2 Bezeichnung und Eigenschaften der verwendeten Traces

### 5.2.1 Bezeichnung der Traces

Vor Beginn der eigentlichen Auswertung werden die von CanuMobiSim erstellten hier verwendeten Traces analysiert. Dies ermöglicht bereits Rückschlüsse auf zu erwartende Ergebnisse bei verschiedenen Cachegrößen und Hinweise, welche Teilaspekte der verschiedenen Ersetzungsstrategien besonders zu beachten sind. In dieser Arbeit werden Traces der zeitlichen Länge 900 Sekunden betrachtet.

Die Traces werden wie folgt bezeichnet:

**Traceart-Knotenanzahl-Maximalgeschwindigkeit-Pause[-Lauf].**

- Die Traceart gibt einen der in Abschnitt 3.5.3 beschriebenen Arten Graph Walk und Random Walk an.
- Die Knotenzahl gibt die Anzahl der Knoten in der Simulation an.
- Die Maximalgeschwindigkeit gibt die höchste für einen Knoten mögliche Geschwindigkeit in Metern pro Sekunde an. Jeder Knoten bekommt beim Start der Simulation eine zufällige Geschwindigkeit zwischen 0 und der Maximalgeschwindigkeit zugewiesen und behält diese bei Bewegung während der Simulation bei.
- Die Pause gibt an, bis zu wieviele Sekunden lang der Knoten stoppt, nachdem er einen der in der Simulation festgelegten Zielpunkte erreicht hat, bevor er seinen Weg zum nächsten Zielpunkt fortsetzt. Es wird pro Stop ein entsprechender Zufallswert ausgewählt.
- Der Lauf: Bei einigen der Traces wurden mit CanuMobiSim mehrmals Daten mit denselben Parametern Traceart, Knotenzahl, Maximalgeschwindigkeit und Pause generiert und ausgewertet. Wegen der Zufallskomponenten ergeben sich jeweils geringfügig andere Resultate. In diesem Fall sind die Läufe von 0 an aufsteigend nummeriert.

### 5.2.2 Ablauf der Simulation

Abbildung 5.1 zeigt das Zusammenspiel der verschiedenen Programme (abgerundete Kästen) und Daten (eckige Kästen) für die Cache-Simulation, der Datenfluss ist hierbei von unten nach oben.

Die Simulationsparameter werden von CanuMobiSim eingelesen, daraus generiert CanuMobiSim die Trajektorien, zugeordnet zu Knoten und Zeitpunkten. Diese Daten werden in eine Textdatei geschrieben. Entsprechende Shell-Skripte starten CacheSim mit den verschiedenen zu untersuchenden Parametern, Cachegrößen und Ersetzungsstrategien. Die Ergebnisse werden in Dateien geschrieben und von weiteren Skripten so zusammengefasst, dass sie mit Gnuplot zur Visualisierung in Graphen überführt werden können. Histogramme mit den TNR-Distanzen lassen sich als Nebenprodukt der OPT-Simulation ebenfalls mittels Gnuplot erzeugen.

Der Ablauf entspricht einer Art Schichtenarchitektur. Die niedrigeren Schichten (in dieser Abbildung weiter oben dargestellt) abstrahieren Details wie Knotennamen, Positionen etc. – je niedriger die Schicht, desto weniger Anwendungswissen steht zur Verfügung.

### 5.2.3 Eigenschaften der Traces

Wir betrachten bei der Analyse der Trace-Eigenschaften immer deren jeweilige von CacheSim erstellte Anfrageliste, diese entsteht aus den Traces. CacheSim ist deterministisch, die Ergebnisse sind also auch bei mehrmaligem Erstellen einer Anfrageliste aus einem bestimmten Trace immer dieselben.

Die TNR-Distanzen wurden mittels CacheSim bestimmt. Sie wurden in ein Histogramm mit logarithmischen Skalen eingefügt. Mit diesem Histogramm ist ein Rückschluss auf die nötige Cache-Größe sehr anschaulich möglich. Abbildungen 5.2 und 5.3 zeigen solche Graphen

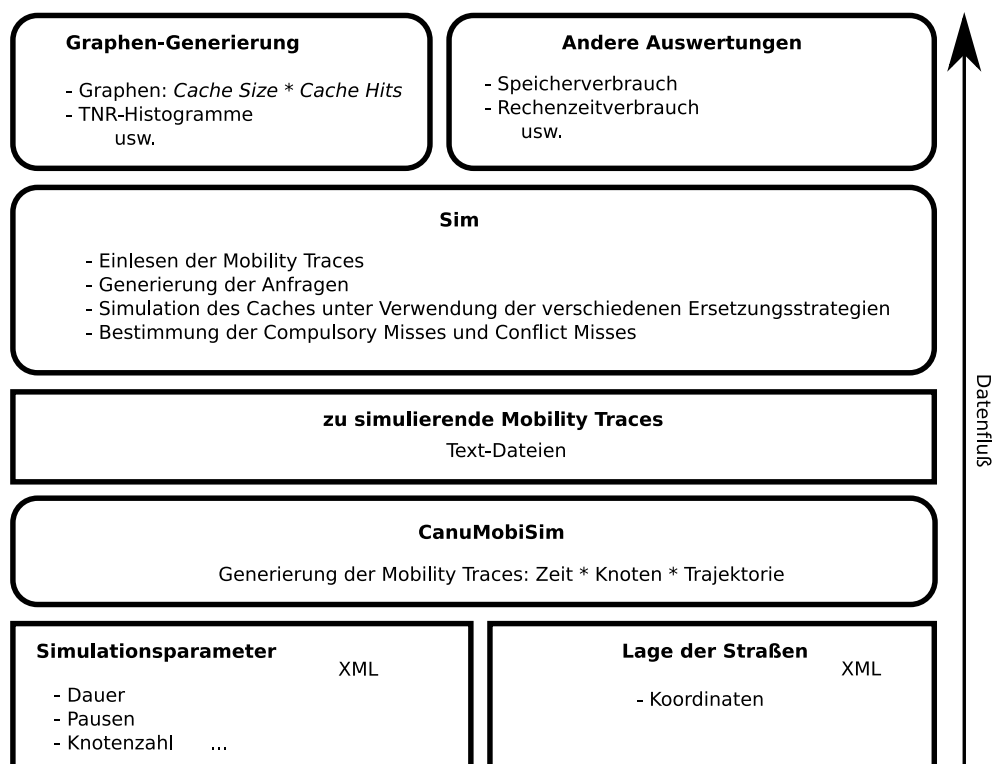


Abbildung 5.1: Architektur der Komponenten für die Simulation

für je ein Graph Walk und ein Random Walk Szenario. Auf der x-Achse ist die Weite der festgestellten TNR-Distanzen festgehalten, auf der y-Achse deren Häufigkeit. Somit sind alle vorkommenden Distanzen in einer Sequenz von Anfragen im Graph repräsentiert. Logarithmische Skalen auf beiden Achsen helfen, aussagekräftige Ergebnisse aus dem Graph abzulesen. Es handelt sich um eine Graph Walk Simulation mit 50 Knoten. Zu jeder Zeiteinheit werden also  $50^2 - 50 = 2450$  Anfragen an den Cache gestellt. Da sich ein Großteil der Knoten in der jeweils nächsten Zeiteinheit noch im selben Cluster befindet (wegen der geringen Geschwindigkeiten), wiederholen sich die Anfragen nach 2450 Anfragen. Deshalb hat die TNR-Distanz der weitaus meisten Anfragen diesen Wert, wie an einem deutlichen Peak in der logarithmischen Skala erkennbar ist. Geringere TNR-Distanzen treten auf, wenn Elemente sich paarweise am selben Ort bzw. im selben Cluster befinden. Es fällt auf, dass dies in Abbildung 5.2 häufig der Fall ist. Das ist nicht überraschend, da es sich bei der Abbildung um einen Graph Walk Trace handelt, bei dem die Knoten oft gemeinsame Wege benutzen. Auf diese Weise kommt es folglich oft dazu, dass sich Knoten im selben Cluster befinden. Bei den zufällig festgelegten Wegen der Knoten in einem Random Szenario kommt dies praktisch nie vor.

Größere TNR-Distanzen kommen vor, wenn Paare von Koordinaten zwischendurch längere Zeit nicht benötigt werden. Beim betrachteten Graph Walk Trace kommen wiederum sehr große TNR-Distanzen geringfügig häufiger vor als im Random Walk Szenario. Dies ist zum Teil dadurch zu erklären, dass Wege von Knoten gemeinsam benutzt wurden, jedoch mit größerem zeitlichen Abstand. Da zwei Anfragen nur übereinstimmen, wenn sowohl Sender- als auch Empfängerkoordinaten übereinstimmen, ist die Wahrscheinlichkeit für einen Cache Hit in

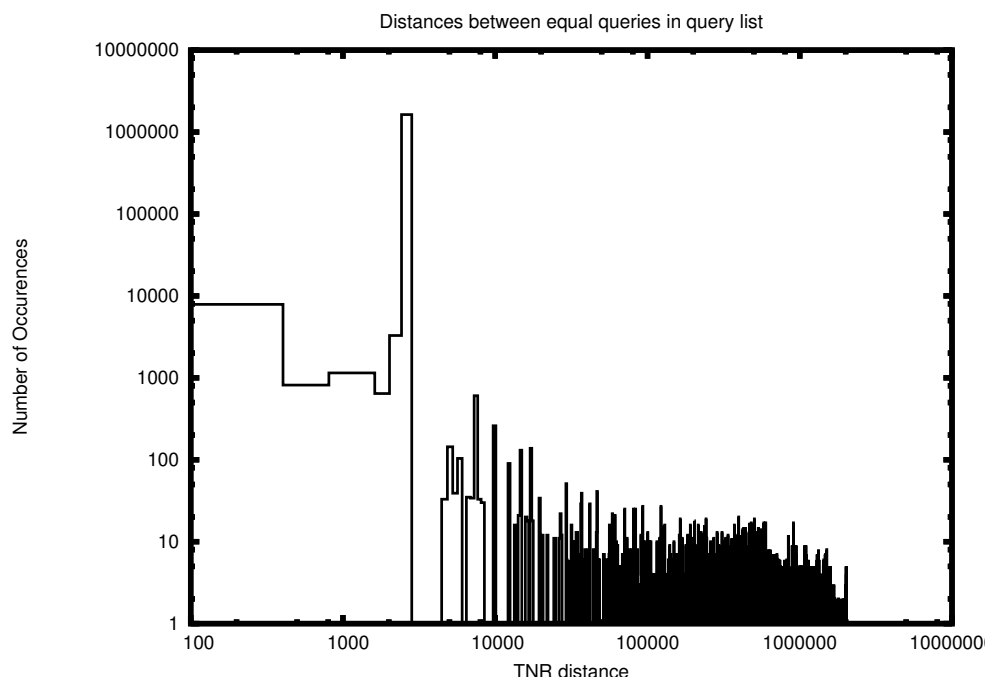


Abbildung 5.2: TNR-Distanzen graph\_walk-50-1-000-0

diesem Fall jedoch prinzipiell recht gering.

Abbildung 5.4 (lineare Achseneinteilungen) verdeutlicht, wie die FIFO-Strategie erst ab der Cachegröße von 2000 Elementen eine gute Hit Rate erzielt. OPT hingegen steigt bis zu dem Punkt 2450 weitgehend linear an.

#### 5.2.4 Auswirkungen des Clusterings

Clustering führt, wie in Abschnitt 3.5.2 beschrieben, bei Clusteringfaktoren  $k > 1$  dazu, dass 4D-Hypercubes der Kantenlänge des Clusteringfaktors an Dämpfungswerten (also  $k^4$  Werte pro Cluster) in den Cache gelesen werden. Dies hat den Vorteil, dass weniger Zugriffe auf den Sekundärspeicher stattfinden und die Anzahl der zu verwaltenden Datenelemente sich um den Faktor  $k^4$  verringert. Der Overhead wird dadurch reduziert.

Ein Nachteil des Clusterings ist jedoch, dass vermehrt nicht verwendete Daten in den Cache geladen werden. Abbildung 5.5 (logarithmische Einteilung auf der x-Achse) zeigt auf der y-Achse die Anzahl der Cache-Hits und auf der x-Achse die Cache-Größe in Dämpfungswerten. Es ist eine Simulation der OPT-Ersetzungsstrategie dargestellt. Bei einem Clusteringfaktor von 1 hat jedes Cache-Element 1 Dämpfungswert, bei Clusteringfaktor 24 hat jedes Cache-Element  $24^4 = 331.776$  Elemente. In der Abbildung ist erkennbar, dass bei Clustergröße 1 bereits mit einem Cache von ca. 2500 Elementen das Maximum der Cache Hits erreichbar ist. Die Bildung von größeren Clustern befördert offenbar sehr viele Daten in den Cache, die überhaupt nicht benötigt werden, denn erst bei einer Cachegröße von ca.  $10^9$  Dämpfungswerten wird das Maximum der Cache Hits erreicht, wenn die Daten mit Faktor 24 geclustert sind. Es findet also ein Tradeoff statt zwischen den Kosten der Verwaltung (die Minimierung verlangt große Cluster-

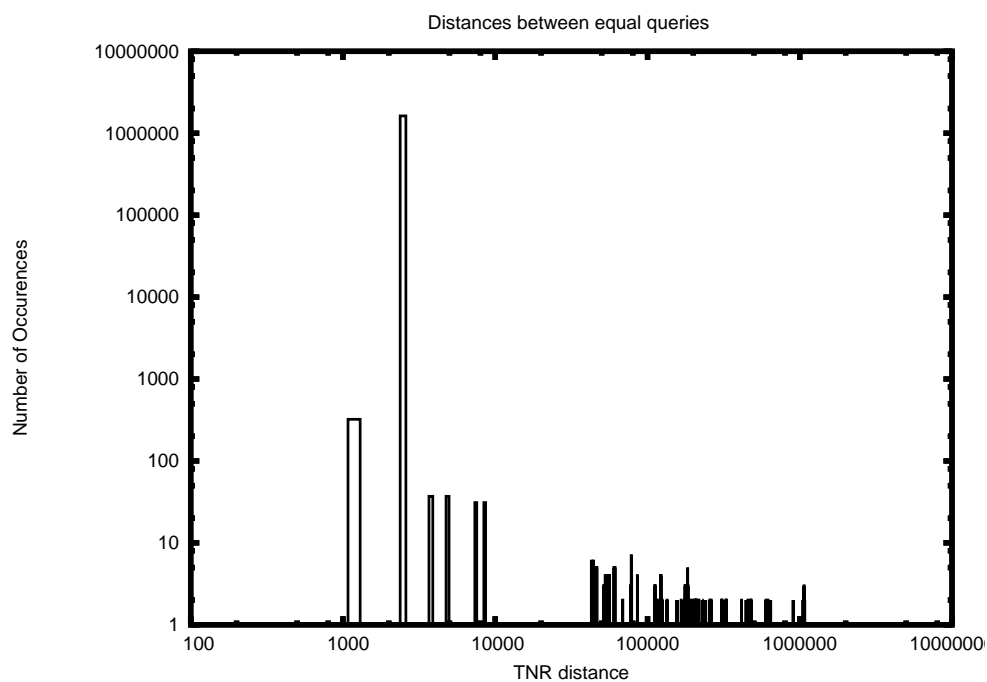


Abbildung 5.3: TNR-Distanzen random-50-1-000-0

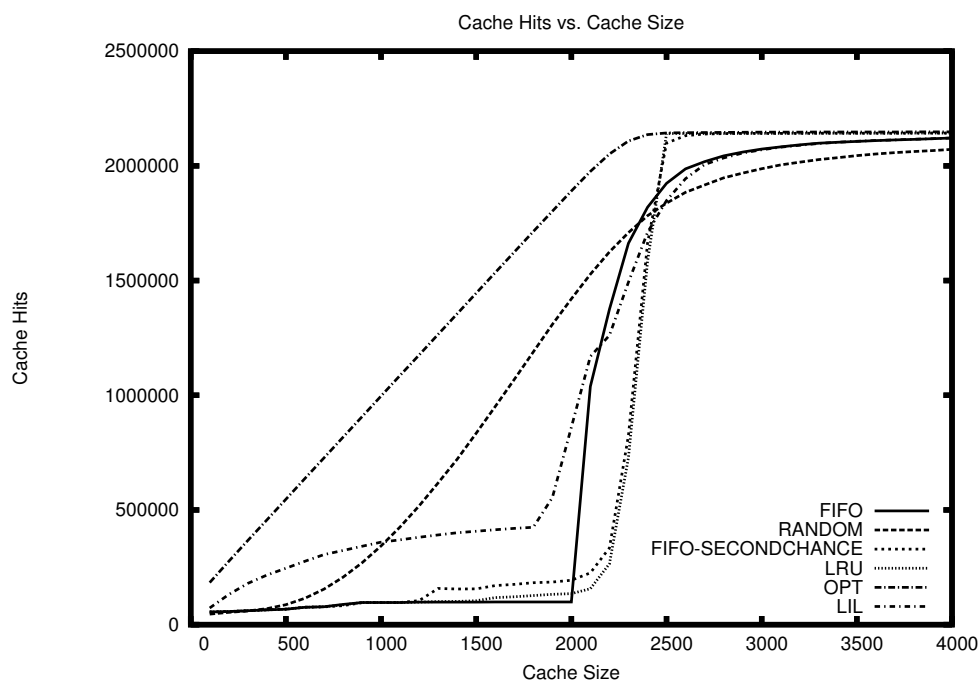


Abbildung 5.4: Cache Hits graph\_walk-50-1-000-0, Cluster Size 1

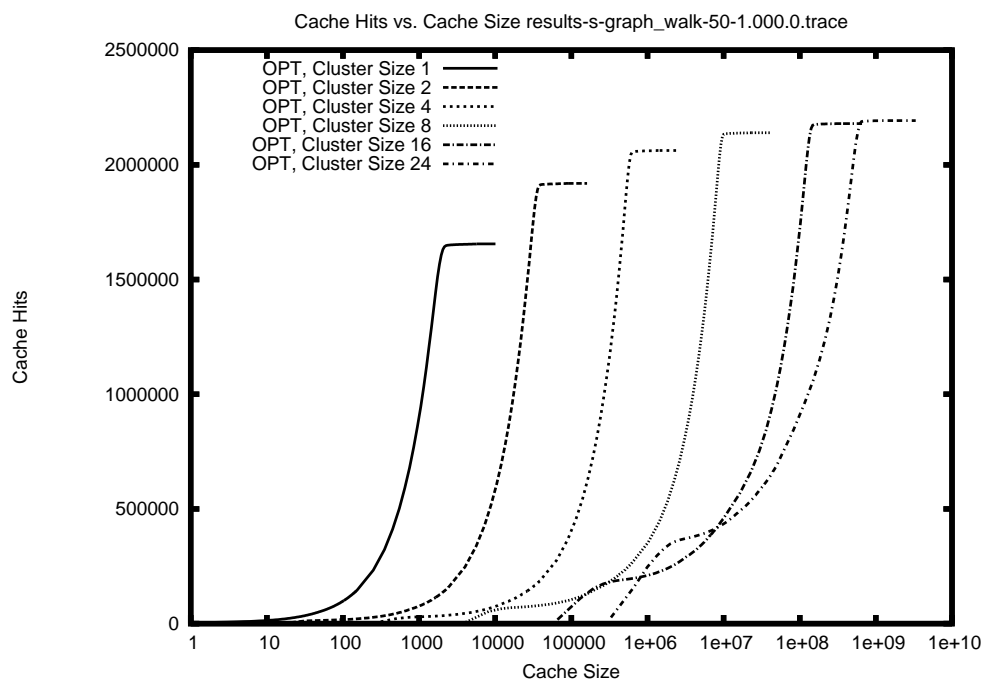


Abbildung 5.5: Cachegröße und Cache Hits bei unterschiedlichen Cachegrößen, OPT

größen) und dem Verschnitt im Cache (die Minimierung verlangt kleine Clustergrößen). Wie man der Abbildung leicht entnehmen kann, unterscheiden sich die angegebenen Clustergrößen 1, 2, 4, 8 und 16 jeweils um ungefähr eine Größenordnung bezüglich des nötigen Platzes. Die teilweise Überschneidung der Kurven für Clustergrößen 16 und 24 in Abbildung 5.5 zeigt, dass nicht immer die jeweils kleinere Clustergröße optimal sein muss.

In Abbildung 5.6 wird der Zusammenhang zwischen Cache-Hits und Cache-Größe für eine FIFO-Ersetzungsstrategie dargestellt. Die Detailansicht dieser Graphik in Abbildung 5.7 zeigt, dass es durchaus schwierig sein kann, den optimalen Clusterfaktor zu finden: Bei gleicher Cache-Größe liefert mal die eine, mal die andere Clustergröße eine größere Hit-Rate.

Anzumerken ist, dass in den Abbildungen 5.5 und 5.6 die jeweiligen Simulationen abgebrochen wurden, wenn mit steigender Cache-Größe das Maximum an Cache-Misses für jede Clustergröße erreicht wurde. Deshalb hören die betreffenden Linien nach der Annäherung an diesen Wert auf.

Für Abbildung 5.8 wurden mehrere Traces mit denselben Parametern von CanuMobiSim generiert und mit CacheSim simuliert. Es ist deutlich sichtbar, wie stark sich die Ergebnisse unterscheiden können, da CanuMobiSim bei jedem der Traces andere Zufallswerte für die Bewegungen der Knoten einsetzt. Zur Veranschaulichung der Unterschiede sind die Daten von Abbildung 5.8 auch nochmals in Abbildung 5.9 mit linearer Achseneinteilung dargestellt.

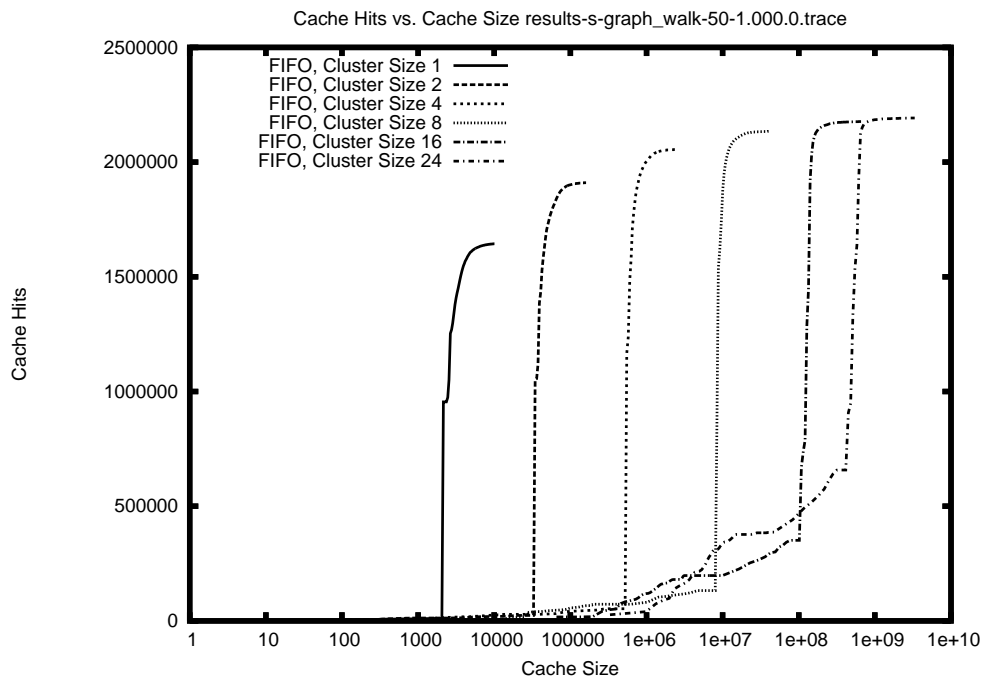


Abbildung 5.6: Cachegröße und Cache Hits bei unterschiedlichen Clustergrößen, FIFO

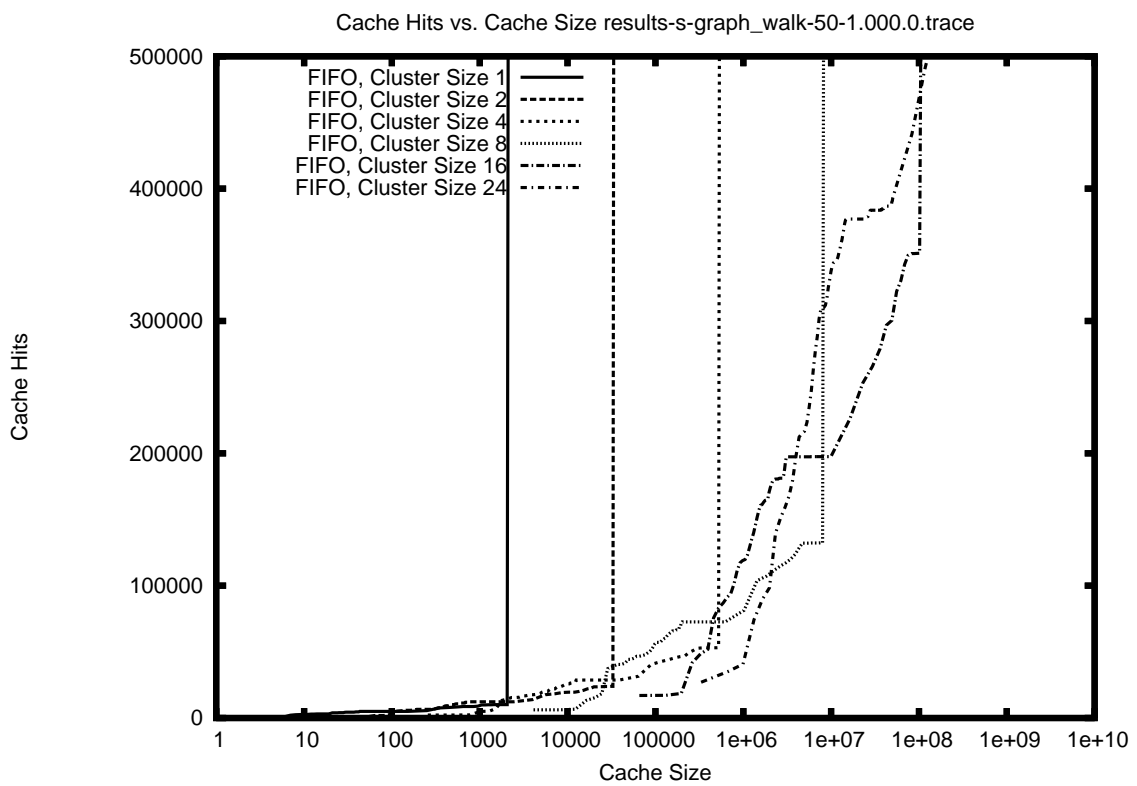


Abbildung 5.7: Cachegröße und Cache Hits bei unterschiedlichen Clustergrößen, FIFO, Detailansicht

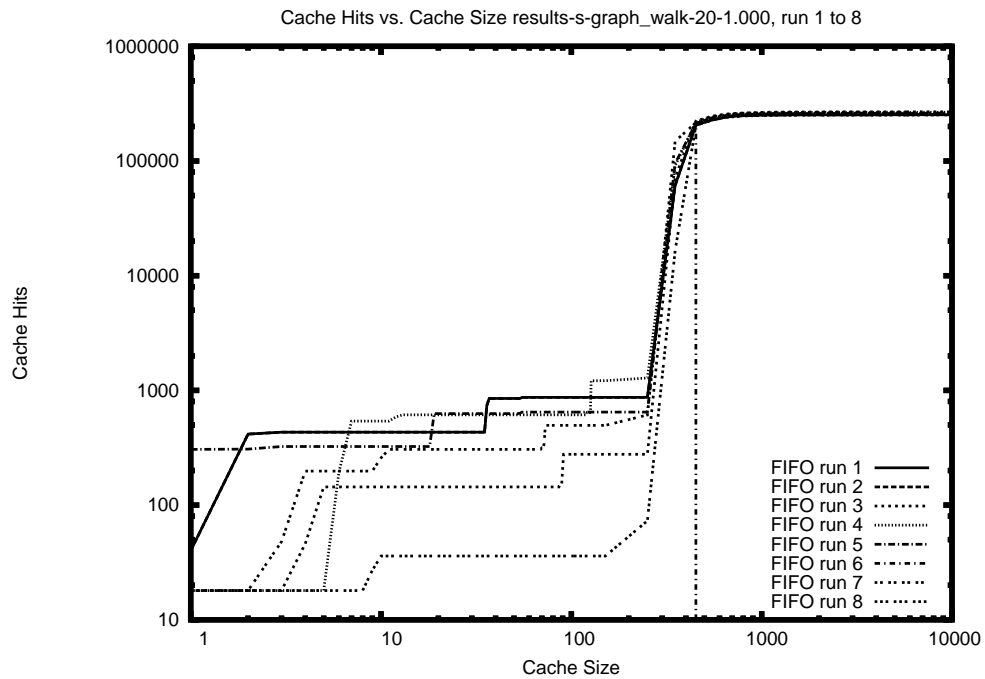


Abbildung 5.8: Mehrere Läufe von graph-walk-20-1-000, FIFO, im Vergleich

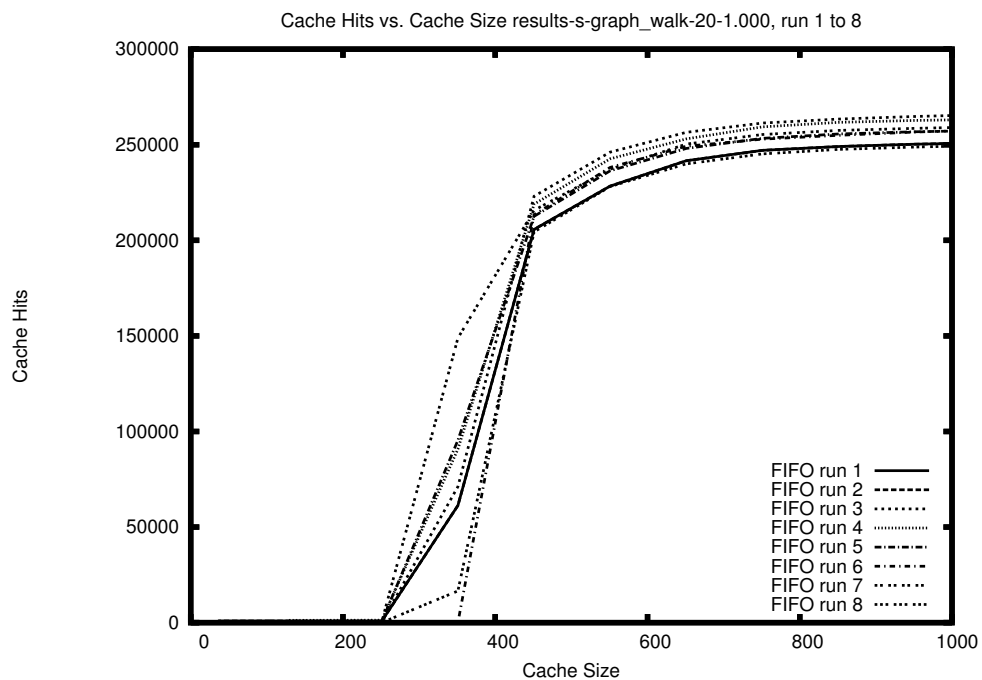


Abbildung 5.9: Mehrere Läufe von graph-walk-20-1-000, FIFO, im Vergleich

## 5.3 Deterministische Knotenbewegung

### 5.3.1 Parameter

Wie bereits ausgeführt, liegt bei der deterministischen Knotenbewegung der Trace und damit die Anfragesequenz an den Cache im Voraus vor. Damit kann die optimale Ersetzungssequenz einfach durch einen der vorgestellten OPT-Algorithmen im Cache bestimmt werden. Die variablen Parameter sind damit die Größe des Caches und die Größe der Cluster. Es existiert ein Wert für die Cachegröße, ab dessen Überschreitung kein weiterer Vorteil in Form einer noch geringeren Anzahl von Cache Misses mehr entsteht. Dieser Wert kann einfach bestimmt werden, bei dem hier implementierten OPT-Algorithmus von Mattson et al. erfolgt die Bestimmung durch Speichern des auftretenden Maximums der Anzahl von Elementen mit  $TNR \leq \infty$  in Menge  $Q$  in Teil 2 des Algorithmus. Die Prüfung muss beim vorliegenden Algorithmus nach jedem Durchlauf der While-Schleife erfolgen. Nicht mehr benötigte Elemente haben immer  $TNR = \infty$ , deshalb kann die festgestellte Anzahl nie durch unnötig gezählte Elemente größer als nötig sein.

### 5.3.2 Prototyp

In der konkreten Implementierung wurde die Suche nach Elementen in der Menge  $Q$  effizient mit Hashmaps realisiert, die Suche nach dem Element mit der geringsten TNR-Distanz im Cache jedoch durch lineare Suche durch den Cache. Aufgrund dieser ineffizienten Implementierung wird erwartet, dass dieser Teil des Algorithmus mit ansteigender Cache-Größe viel Rechenzeit benötigen wird. Eine deutliche Verbesserung der Laufzeit der Implementierung wäre durch die Verwendung eines Heaps möglich gewesen. Aus Zeitgründen, und weil die Laufzeit der Implementierung bei den überprüften Cachegrößen akzeptabel war, wurde auf eine heap-basierte Implementierung verzichtet.

Das Ergebnis des OPT-Algorithmus dient zum Vergleich der anderen Cache-Ersetzungsstrategien mit dem optimalen Ergebnis. Außerdem wäre die Nutzung der optimalen Ersetzungsinformation zur Implementierung von Prefetching bei der deterministischen Knotenmobilität möglich.

### 5.3.3 Bewertung

Für die Bestimmung der OPT-Performanz wurde der Algorithmus von Mattson et al. gewählt. Auf einem Rechner mit AMD Opteron 2800 MHz Prozessor braucht die Berechnung der OPT-Reihenfolge mit den Anfragen eines „Graph Walk“ Traces bei 100 Knoten, Clustergröße 1 und Cachegröße 10000 die Zeit von 5 Minuten. Davon entfallen nur 5 Sekunden auf Teil 1 des Algorithmus von Mattson et al. – der Rest der Rechenzeit von 4:55 Minuten wird durch Teil 2 des Algorithmus benötigt.

Bei einem Lauf mit Cachegröße 10 benötigt Teil 2 des Algorithmus nur noch 6 Sekunden. Dies bestätigt die Erwartung, dass die Suche nach dem Element mit dem geringsten TNR-Wert im Cache die meiste Zeit benötigt.

Die Erfolgsrate der OPT-Sequenzen ist erwartungsgemäß in allen Fällen den anderen Strategien überlegen. Der Vorsprung von OPT ist bei Cachegrößen kleiner als  $n \cdot n - n$  Elemente

mit Clusterfaktor 1 besonders deutlich.

Im folgenden Abschnitt wird für einen Trace der Vergleich zwischen der Gesamtzahl der Anfragen, den OPT-Misses für bestimmte Cachegrößen und den „Compulsory Misses“ durchgeführt. Dieser Vergleich hilft abzuschätzen in welchen Größenordnungen durch optimale Ersetzung bei den konkreten hier behandelten Daten Verbesserungen möglich sind. Zum Vergleich sind auch die Strategien FIFO und RANDOM mit aufgeführt. Die Tabelle in diesem Abschnitt bezieht sich auf Clustergröße 1.

Traceart: Graph Walk, Anzahl Knoten: 50, Maximalgeschwindigkeit: 1 m/s, Zeit: 900 Sekunden, Gesamtzahl der Anfragen: 2205000, Compulsory Misses: 549870.

Cachegröße	100	250	1050	1500	2000	2500	5800
OPT Misses	2107000	1972036	1259537	834842	573925	554286	549870
FIFO Misses	2199960	2199421	2195150	2194913	2194866	1154188	578867
RANDOM Misses	2200412	2199160	1998816	1716239	1443604	1234421	778365

Diese Tabelle zeigt ausgewählte Werte von Abbildung 5.4. Ab einer Cachegröße von 5800 Elementen kann der OPT-Algorithmus hier das theoretische Minimum an Cache Misses, welches gleich der Anzahl der unterschiedlichen Anfragen ist, erzielen. Deshalb wurden auch für diesen Wert die Ergebnisse angegeben. Die anderen Algorithmen verlangen einen wesentlich größeren Cache, um eine deutliche Reduktion der Cache Misses zu erreichen. Die Abweichung der dargestellten Werte von den in Abschnitt 5.2.3 aufgestellten theoretischen Richtwerten für FIFO beim Wert 2500 zeigen, dass die Knotenpositionen einer starken Veränderung von Zeiteinheit zu Zeiteinheit unterliegen.

Bei anderen Traces zeigen die Ergebnisse ein sehr ähnliches Muster mit im Verhältnis zueinander denselben Erfolgsraten, deshalb wurde auf eine genaue Auflistung der Ergebnisse weiterer Graphen verzichtet.

Zusammenfassend lässt sich sagen, dass immer, wenn es möglich ist, auf die OPT-Reihenfolge zurückgegriffen werden sollte.

## 5.4 Indeterministische Knotenbewegung

In diesem Abschnitt vergleichen wir die verschiedenen implementierten Ersetzungsstrategien OPT, RANDOM, FIFO, FIFO Secondchance, LRU und LIL.

### 5.4.1 Parameter

Verschiedene Traces können sich sehr stark unterscheiden. Deshalb ist es zweckmäßig, die Traces anhand ihrer Parameter Traceart, Knotenzahl, Simulationsdauer, Knotengeschwindigkeit und verwendete Landkarte zu klassifizieren und die Ergebnisse der so gebildeten Gruppen getrennt zu betrachten. Im Folgenden werden die Angaben in der Form wie in Abschnitt 5.2.1 beschrieben verwendet.

### 5.4.2 Analyse

Wir teilen die Ersetzungsalgorithmen für die folgenden Betrachtungen in Gruppen ein:

#### OPT

Die OPT-Gruppe enthält nur den Algorithmus OPT. Dieser Algorithmus hat von allen Algorithmen am meisten „Intelligenz“, da er immer optimale Ersetzungen vornimmt.

#### FIFO-ähnlich

Diese Gruppe enthält die Algorithmen FIFO, FIFO-SECONDCHANCE und LRU.

#### RANDOM

Die RANDOM-Gruppe enthält nur den Algorithmus RANDOM.

#### LIL

Die LIL-Gruppe enthält nur den Algorithmus LIL.

Wie wir bereits vermerkt haben, stellt der Algorithmus pro Zeitschritt bei  $n$  Knoten  $n \cdot n - n$  Anfragen in jeder Simulations-Zeiteinheit. In den meisten „Cachegröße gegen Cache Hits“ Graphen lassen sich (Clustergröße 1) zwei Bereiche auf der x-Achse mit den Cachegrößen feststellen: Von Cachegröße 1 bis Cachegröße  $c = c_{krit} = n \cdot n - n$  ist die RANDOM-Strategie in der Regel besser als die meisten anderen Strategien wie FIFO. Dies liegt daran, dass FIFO mit Cachegröße  $c < c_{krit}$  die sich zum Großteil alle  $c_{krit}$  Elemente wiederholenden Anfragen grundsätzlich aus dem Cache entfernt, bevor sie nochmals auftreten. RANDOM liefert hier schon „Zufallstreffer“.

Wird Cachegröße  $c_{krit}$  erreicht und überschritten, so funktionieren die FIFO-basierten Strategien besser, da sich wiederholende Anfragen vom jeweils letzten Zeitschritt noch im Cache befinden. Nun ist RANDOM im Nachteil, da zufällig immer auch noch benötigte Elemente ersetzt werden. Mit steigender Cachegröße nimmt diese Wahrscheinlichkeit jedoch ab.

### 5.4.3 Bewertung

Anhand einiger ausgewählter Diagramme (wir betrachten hier immer Clustergröße 1) werden die Simulationsergebnisse in diesem Abschnitt mit den unterschiedlichen Strategien dargelegt.

Die Ergebnisse des OPT-Algorithmus bilden die obere Grenze der gezählten Cache Hits für alle Simulationen. Der OPT Algorithmus ist aufgrund der nicht im Voraus verfügbaren Anfragen in diesem Fall nicht verwendbar. Er würde bei den meisten Traces eine um den Faktor 2 größere Anzahl an Cache Hits im Vergleich zu RANDOM unterhalb von  $c_{krit}$  erreichen. Oberhalb von  $c_{krit}$  ist der Vorteil der OPT-Strategie im Vergleich zu FIFO meist geringer. Unterhalb einer Cachegröße von  $c_{krit}$  sollte RANDOM verwendet werden, oberhalb von  $c_{krit}$  ist FIFO oder eine der Varianten die beste Wahl.

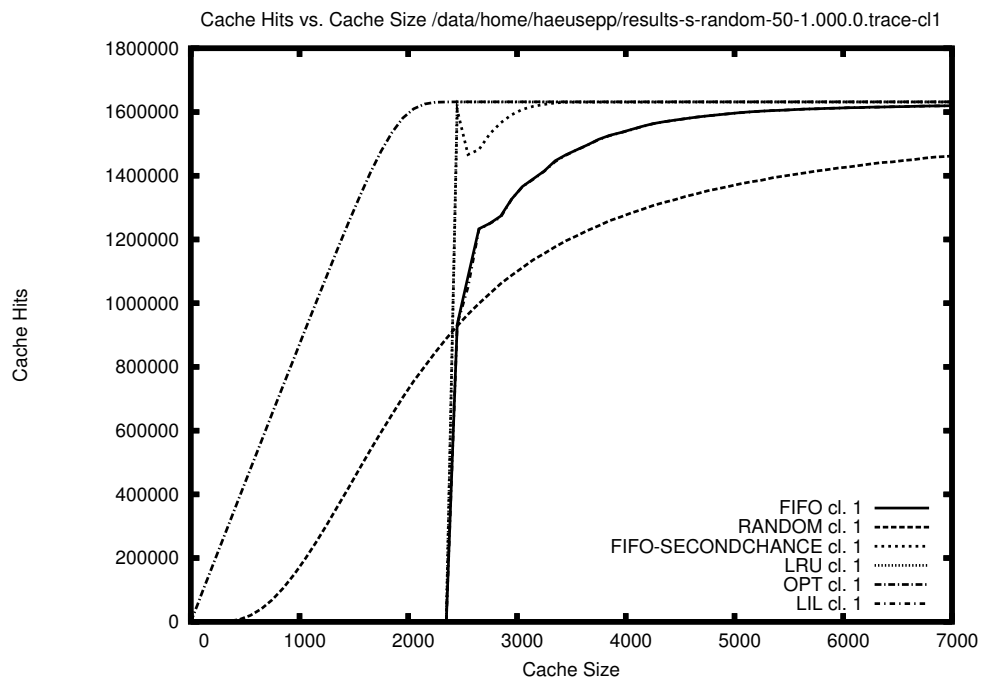


Abbildung 5.10: Cache Hits random-50-1-000-0, Cluster Size 1

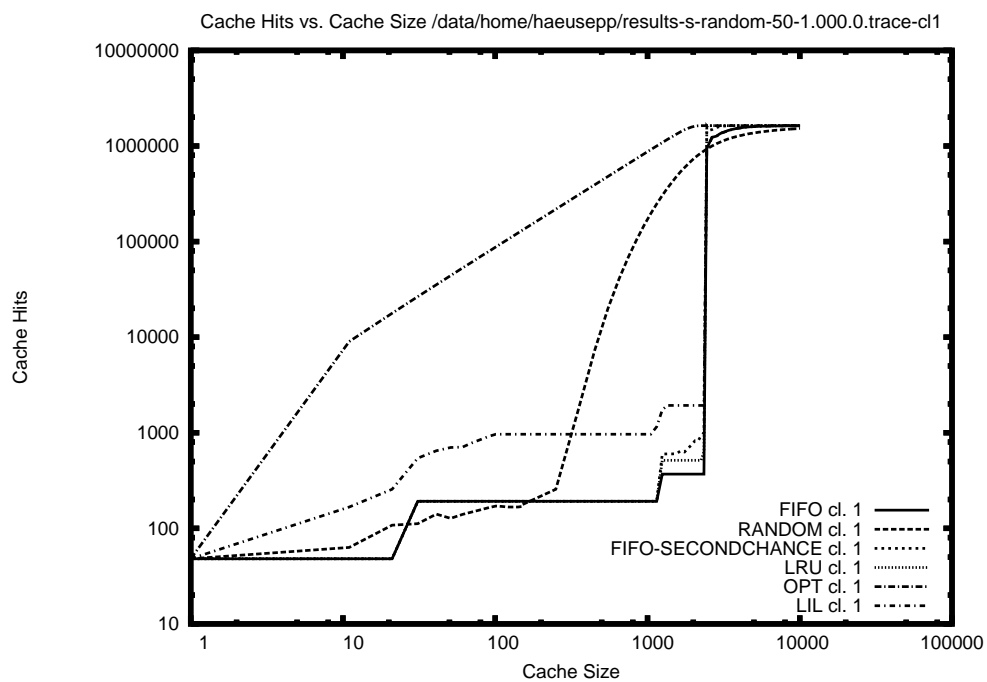


Abbildung 5.11: Cache Hits random-50-1-000-0, Cluster Size 1

### **RANDOM-50-1-0 Trace**

Die folgenden Betrachtungen treffen für alle anderen untersuchten Random Walk Traces zu. Zum Trace random-50-1-0 (Abbildungen 5.10 und 5.11) mit Knotenanzahl  $n = 50$  ist folgendes zur Cachegröße  $c$  anzumerken:

Die Änderung der Clustergrößen änderte das Verhältnis der Ergebnisse aller unterschiedlichen Ersetzungsstrategien zueinander nur sehr geringfügig (siehe Abschnitt 5.2.4).

FIFO liefert für  $c$  von 1 bis  $n$  nur sehr geringe Hit-Raten. Ab  $n$  bis unter  $n^2$  steigt die Anzahl der Cache Hits etwas an. In diesem Fall erzeugen zwei oder mehr Knoten, die sich an derselben Position (oder im selben Cluster) befinden, in derselben Zeiteinheit der Simulation Anfragen mit einer TNR-Distanz unter  $n^2$ . Davon kann der Cache profitieren. Die meisten unvermittelten Anstiege der FIFO-Erfolgsrate dürften aufgrund dieses Zusammenhangs von Knoten, die sich an derselben Position oder im selben Cluster befinden, ausgelöst werden. Bei Erreichen von  $n^2$  Knoten steigt die Erfolgsrate des Caches deutlich an, da alle Anfragen innerhalb einer Zeiteinheit, deren Knoten sich nicht bewegt haben, in der nächsten Zeiteinheit wiederholt werden. Abbildung 5.3 zeigt diese Häufung im TNR-Histogramm.

FIFO-SECONDCHANCE liefert nur in einigen Ausnahmefällen eine geringfügig bessere Ausbeute. Der Grund dafür ist darin zu suchen, dass die Strategie nur dann andere Ersetzungsentscheidungen als FIFO trifft, wenn ein Element während des erstmaligen Durchlaufens der Warteschlange nochmals auftritt und deshalb eine „zweite Chance“ bekommt. Aufgrund der großen Regelmäßigkeit der Anfragesequenz kommt dieses erneute Auftreten jedoch bei  $c < n^2$  fast nie vor, weder beim ersten noch beim zweiten Durchlauf. Für  $c \geq n^2$  wiederholen sich die Elemente bei stillstehenden Knoten ständig, eine weitere „Chance“ für ein Cache-Element nach einem Durchlauf durch die FIFO-Schlange macht hier keinen Unterschied.

Die Strategie LRU liefert praktisch dieselben Resultate wie FIFO. Bei einer Cachegröße von  $c < n^2$  führt der Algorithmus im Wesentlichen dieselben Ersetzungen aus wie FIFO. Aufgrund der sich zyklisch wiederholenden Anfragen mit sehr wenigen doppelten Verwendungen macht es keinen Unterschied, ob wie bei FIFO eine Warteschlange durchlaufen wird oder – bei LRU – die Anfragen anhand ihrer letzten Verwendung sortiert werden. Für  $c < n^2$  geben sich auch kaum Vorteile gegenüber FIFO bzw. FIFO-Secondchance.

Die RANDOM-Strategie liefert bei allen untersuchten Traces verschiedener Parameter für eine Cachegröße knapp unter  $n^2$  deutlich bessere Ergebnisse als die FIFO-Varianten.

Da die Knoten bei Random-Walk Traces meist keine gemeinsamen Wege benutzen, fällt LIL unter diesen Umständen faktisch auf eine FIFO-Strategie zurück. Die FIFO-Ergebnisse unterscheiden sich im Falle von Random Walk praktisch nicht von den LIL-Ergebnissen.

### **Graph\_Walk-50-1-0 Trace**

Es folgt nun eine Untersuchung des Trace graph\_walk-50-1-0, siehe Abbildungen 5.4 und 5.12.

Auch bei den Graph Walk Traces wurden die gefundenen Zusammenhänge bei allen untersuchten Traces festgestellt. Das Verhältnis der verschiedenen Ersetzungsstrategien zueinander änderte sich bei anderen Clustergrößen, wie bei den Random Traces, nur unwesentlich.

Die LIL-Strategie schneidet im Bereich  $c < n^2$  bei den untersuchten Graph Walk Traces besser ab als bei Random Walk. Dies liegt daran, dass hier wegen der gemeinsam von den

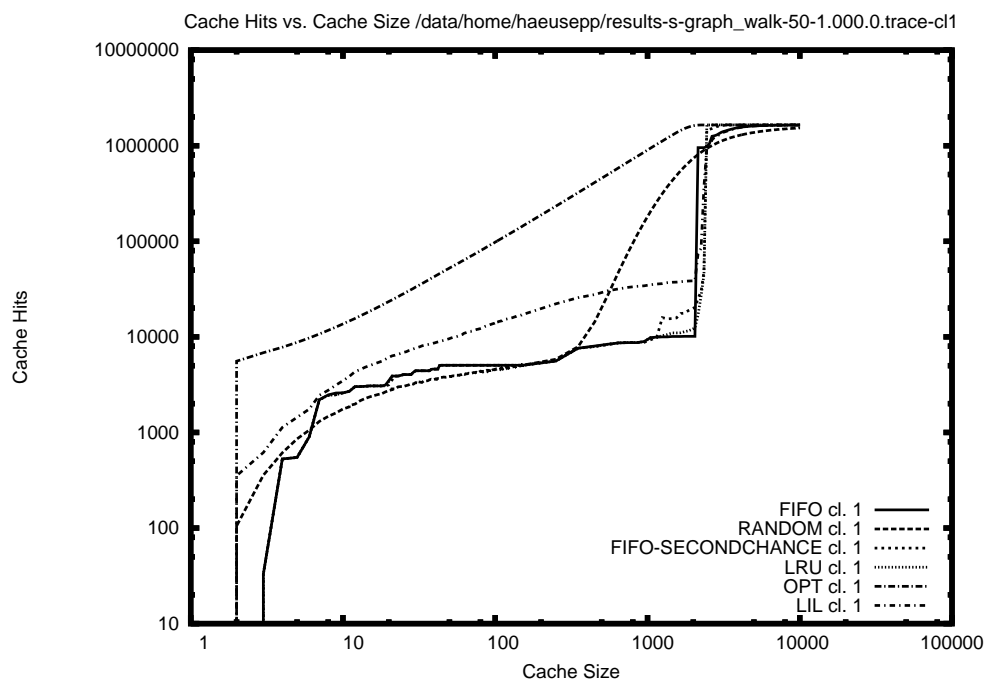


Abbildung 5.12: Cache Hits graph\_walk-50-1-000-0, Cluster Size 1, Log. Achsen

Knoten benutzten Wege tatsächlich viele Wiederholungen von Anfragen im Laufe des Traces stattfinden. Diese Wiederholungen können dann bei kleineren Caches in gewissem Maße ausgenutzt werden.

Ansonsten gibt es keine gravierenden Unterschiede bei den unterschiedlichen Strategien zwischen Graph Walk und Random Szenarien.

### Weitere Traces

Es wurden im Rahmen dieser Arbeit zahlreiche weitere Graph Walk und Random Walk Traces mit Knotenanzahlen von 10 bis 100 und verschiedenen Clustergrößen angefertigt, simuliert und bewertet. Es wurden jedoch stets dieselben bereits beschriebenen Zusammenhänge festgestellt. Auf einen Abdruck der Diagramme in dieser Arbeit wurde daher verzichtet.

# Kapitel 6

## Einbindung in die NET-Umgebung

### 6.1 Schichtenprinzip

In der NET-Umgebung ist der eigentliche Datenzugriff [Dic07] funktional vom Cache getrennt. Dies erscheint auch technisch sinnvoll, da durch die Modularisierung je nach Bedarf und Leistung ein Austausch des Codes für den Cache oder für die Datenablage möglich ist. So sind Änderungen wie z.B. in der Komprimierung oder an den Cache-Strategien mit wenig Aufwand möglich, da die Abhängigkeiten der Module gering sind.

Beim jetzigen Stand der Emulation existiert eine Trennung zwischen den Funktionen in einer Art Schichtenarchitektur, die in Abbildung 6.1 skizziert ist.

Die oberste Schicht hat das gesamte Anwendungswissen. Sie kennt die Existenz und Lage der Straßen sowie die Intentionen der mobilen Knoten. Die Simulationszeit sowie die nächsten Zielpunkte der Knoten sind bekannt. Am unteren Ende des Schichtenstapels gibt es nur noch abstrakte Anfragen an Elemente nach Koordinaten im Cache.

In der Schicht der einzelnen Cache-Anfragen – auf Cache-Ebene – werden keine Daten der höheren Schichten verarbeitet. Die Schichtenarchitektur hat den Vorteil, dass eine Implementierung unabhängig von den Anfragen erfolgen kann und die Implementierung auch austauschbar ist. Die Vorteile sind ähnlich wie bei den in Rechnernetzen bekannten Schichtenmodellen.

Ein Nachteil dieser Schichtenarchitektur ist, dass unten auf Cache-Ebene keinerlei geographische und zeitliche Daten oder Trajektorien-Daten mehr zur Verfügung stehen. Diese Tatsache macht die Implementierung einiger möglicher Cache-Ersetzungsstrategien problematisch, für diese müsste das Schichtenprinzip durchbrochen werden.

### 6.2 Caching-Strategien

Der Cache hat beim derzeitigen Stand von NET kein Wissen über Knoten, Trajektorien und Knotengeschwindigkeiten. Die einfachen Caching-Strategien (FIFO-Varianten und RANDOM) können problemlos in das NET-Framework eingebunden werden. Bei den vorgestellten komplexeren Strategien wäre eine Veränderung der Architektur bzw. die Schaffung von zusätzlichen Schnittstellen nötig.



Abbildung 6.1: Schichtenprinzip der Simulation

### 6.3 OPT-Sequenzen

Bei der deterministischen Knotenbewegung wird eine gespeicherte Anfragefolge „wiedergegeben“. Eine Möglichkeit, die optimale Ersetzungsreihenfolge im Cache sicherzustellen, bestünde darin, jeder Anfrage der Sequenz von Anfragen den nach dem vorherigen OPT-Lauf ermittelten optimalen Speicherplatz im Cache zuzuweisen. Eine andere Möglichkeit wäre, die TNR-Distanzen in der Folge von Anfragen zu jeder Anfrage mit zu speichern und den Cache-Prozeß während des Simulationslaufs das Element mit dem jeweils höchsten TNR-Wert finden zu lassen, um dorthin die Ersetzung durchzuführen.

Eine Verwendung der gewonnenen OPT-Sequenzen für im Voraus bekannte Simulationsläufe im NET Framework ist daher möglich. Der Server, der die Dämpfungswert-Anfragen beantwortet, müsste um die betreffende Funktionalität (TNR-Distanzen oder festgelegte Ersetzungen) erweitert werden. Die Zuordnung der OPT-Sequenzen zu den Szenarien könnte durch einen Hash-Wert über die Daten des Szenarios vorgenommen werden, aus dem eine eindeutige Kennung für jedes Szenario gebildet wird.

# Kapitel 7

## Zusammenfassung und Ausblick

### 7.1 Zusammenfassung

Das NET Framework ermöglicht die Emulation von Funkverbindungen zwischen mobilen Knoten in einem festgelegten geographischen Gebiet, auf diese Weise können neue Netzwerkprotokolle getestet werden. NET ist in einer Art Schichtenmodell aufgebaut: Auf der obersten Schicht werden die mobilen Knoten mit ihren geographischen Koordinaten betrachtet, außerdem die Straßen und Ziele, auf denen sich die Knoten zwischen Wegpunkten bewegen. Auf den untersten Schichten sind nur noch die Rasterkoordinaten der verschiedenen Orte in der Simulation sichtbar.

Der in dieser Arbeit betrachtete Cache enthält Dämpfungswerte zwischen den verschiedenen Rasterpositionen, der Cache soll die Ablaufgeschwindigkeit der Simulation erhöhen. Die Dämpfungswerte sind für die Emulation der Funkverbindungen nötig, sie werden durch Anfragen an den Cache ausgelesen. Die gesamten Dämpfungsdaten sind zu groß, um auf einmal in den Hauptspeicher derzeitiger Rechner zu passen.

Zentraler Bestandteil der Bewegung der Knoten sind Trajektorien. Trajektorien sind lineare Vektoren mit Geschwindigkeit und Richtung der Bewegung jedes Knotens. Sie sind in der Mitte des Schichtenmodells angesiedelt, mit ihrer Hilfe wird die Knotenbewegung den als Kurven vorliegenden Straßen angenähert.

Um die Arbeit des Caches zu simulieren, wurde im Rahmen dieser Arbeit das Programm „CacheSim“ entwickelt. „CacheSim“ erhält als Eingabe die Trajektorien der bewegten Knoten zu jedem Zeitpunkt und extrahiert daraus die Anfragen. Die Anfragen an den Cache werden durch „CacheSim“ aus der Bewegung der Knoten anhand der Trajektorien extrahiert, indem die Positionen aller Knoten zu jedem Zeitschritt bestimmt werden. Pro Zeiteinheit kommuniziert jeder Knoten einmal mit allen anderen Knoten. Zur Generierung der Anfragen an den Cache werden die Knotenpositionen durch „CacheSim“ von physischen Koordinaten in Rasterkoordinaten umgewandelt.

Ist der Cache nach einigen Anfragen voll, so muss vor dem Laden der nächsten Anfrage ein Ersetzungskandidat im Cache gefunden werden. Wird ein Datum in den Cache geladen, das noch nie angefragt wurde, so spricht man von einem „Compulsory Miss“, da auf jeden Fall auf den Sekundärspeicher zugegriffen werden muß. Wird das Datum später aus dem Cache gelöscht und dann wieder geladen, so spricht man beim zweiten Ladevorgang von einem

„Conflict Miss“.

Die Anfragen sind durch die x- und y-Rasterkoordinate von je Sender- und Empfängerposition charakterisiert. Die Position der Anfragen hat also vier Dimensionen, deshalb können die Anfragen als vierdimensionaler „Hypercube“ gesehen werden. Um die Anzahl der zu verwaltenden Datenelemente und damit den Overhead zu verringern, können mehrere dieser 4-Tupel zu Blöcken festlegbarer Kantenlänge im 4D-Raum, sogenannten Clustern, zusammengefaßt werden. Durch zu große Cluster wird Cache-Speicherplatz verschwendet, bei zu kleinen Clustern ist der Overhead für die Verwaltung der Daten zu groß. Die richtige Clustergröße muß daher ermittelt werden.

In dieser Arbeit wurde eine Analyse der bei einigen Läufen entstandenen Folgen von Anfragen an den Cache durchgeführt, hierbei wurden insbesondere die Distanzen zwischen gleichartigen Anfragen (TNR-Distanzen) und deren Abhängigkeit von der Bewegung der Knoten analysiert.

Die Performanz des Caches hängt stark von der verwendeten Ersetzungsstrategie ab. Es wurden einige Ansätze für die Ersetzung theoretisch und in der Simulation untersucht und nach Cachegrößen geordnet verglichen.

Beim OPT-Ansatz wird immer das Element ersetzt, welches am weitesten in der Zukunft wieder benötigt wird. Der Algorithmus benötigt die vollständige Kette von Anfragen im Voraus und erzeugt daraus die optimale Ersetzungsreihenfolge. Das Ergebnis von OPT dient als Referenz und kann auch im Fall der vorher feststehenden Knotenbewegung verwendet werden – dieser Fall stellt bei NET den Normalfall dar. OPT erzielt auch bei geringen Cachegrößen sehr gute Ergebnisse.

Die anderen Ersetzungsstrategien sind für den Fall gedacht, dass die Sequenz der Anfragen in der Simulation nicht vorher bekannt ist.

Die FIFO-Strategie verwaltet den Cache in Form einer Warteschlange. Neue Elemente werden am Kopf der Schlange eingefügt. Nach Aufnahme in einen Cache mit Größe  $c$  bleibt ein Element genau für die nächsten  $c - 1$  „Compulsory Misses“ im Cache enthalten, um am Ende der Schlange gelöscht zu werden. Bei FIFO-SECONDCHANCE wird das Element, falls es während des Durchlaufs der Warteschlange nochmals referenziert wurde, bei Erreichen des Endes der Schlange nochmals an den Kopf der Schlange gesetzt. LRU ersetzt immer das Element, welches am längsten nicht verwendet wurde. RANDOM ersetzt ein zufälliges Element aus dem Cache. Die Strategie LIL (Linear Interpolated Lookahead) ist im Rahmen dieser Diplomarbeit entstanden. LIL schätzt anhand der Trajektorien ab welche der Elemente im Cache in naher Zukunft wieder benötigt werden. Die so gefundenen Elemente werden bevorzugt im Cache gehalten. Es wird dabei also Wissen aus einer anderen Schicht der Simulation verwendet.

RANDOM liefert bei kleinen Cachegrößen ohne Anwendungswissen bessere Ergebnisse als die anderen Strategien. Die Varianten FIFO, FIFO-SECONDCHANCE und LRU verhalten sich recht ähnlich zueinander. Ab einer bestimmten Cachegröße fällt bei diesen Strategien die Anzahl der „Conflict Misses“ steil ab, in diesem Bereich liefert RANDOM schlechtere Ergebnisse als die anderen Strategien. Da die mobilen Knoten ihre Richtung oft ändern und damit von der Abschätzung abgewichen wird, liefert der LIL-Ansatz kaum weniger „Conflict Misses“ als FIFO, FIFO-SECONDCHANCE und LRU, benötigt jedoch wegen der nötigen Berechnungen für die lineare Interpolation erheblich mehr Rechenleistung. Ein weiterer Grund für das schlechte Abschneiden von LIL dürfte auch an der im Framework vorgenommenen Annähe-

rung durch lineare Vektoren an die größtenteils kurvigen Straßen zu suchen sein. Dieser Sachverhalt sorgt für häufige Richtungswechsel in den Trajektorien, woraufhin die Interpolation oft falsche Voraussagen trifft.

## 7.2 Schlußfolgerung

Für die Abschätzung der Cache-Erfolgsraten im Vorfeld der im Rahmen dieser Arbeit durchgeführten Cache-Simulation war die Analyse der TNR-Distanzen sowie der Anzahl der Knoten in der Simulation hilfreich.

Die FIFO-Varianten der Ersetzungsstrategien (FIFO, FIFO-SECONDCHANCE und LRU) und die RANDOM-Strategie liefern bei ausreichender Cachegröße gute Ergebnisse, die jedoch nicht an die Ergebnisse der OPT-Strategie herankommen.

Die Verwendung von Anwendungswissen ist wegen der schichtartigen Architektur von NET in der Anwendung mit dem tatsächlichen Framework problematisch. Allerdings liefert selbst der getestete einfache Algorithmus LIL mit Einbeziehung von begrenztem Anwendungswissen keine wesentlich besseren Ergebnisse als die FIFO-Varianten. Das Ergebnis von LIL rechtfertigt den Aufwand nicht, hier bringt die Durchbrechung des Schichtenprinzips keinen oder nur einen geringen Vorteil, da die Knoten zu oft ihre Trajektorien ändern.

Clustering hilft, viele Lesezugriffe mit kleinen Datenmengen an verschiedenen Stellen des Sekundärspeichers durch weniger Zugriffe mit größeren Datenmengen zu ersetzen. Wie erwartet, steigen die Datenmengen mit steigendem Clusterfaktor deutlich, der Anteil von nicht benutzten Daten im Cache nimmt dabei schnell zu. Clustering sollte auch im Zusammenhang mit einer möglichen Komprimierung der Dämpfungsdaten betrachtet werden.

RAM-Speicher wird laufend preiswerter. Im Bereich des Caching von HTTP-Daten im Internet werden daher relativ große Caches mit einfachen Ersetzungsstrategien verwendet. Ähnlich der üblichen Vorgehensweise beim Caching von HTTP-Daten ist es auch bei NET möglich, einen verhältnismäßig großen Cache zu verwenden. In dieser Arbeit hat sich herausgestellt, dass es bei den NET-Caches eine kritische Größe gibt, ab der auch die einfachsten Ersetzungsstrategien gut funktionieren. Diese kritische Größe liegt in allen untersuchten Fällen deutlich unter der Größe der gesamten Daten. Für den Fall, dass eine Durchbrechung des Schichtenprinzips nicht möglich ist, kann also auch bei heutigen Rechnern durch die Wahl eines genügend großen Caches der richtigen Größe die Performanz der Emulation deutlich gesteigert werden.

## 7.3 Ausblick

Die Optimierungs-Möglichkeiten der Cache-Ersetzung sind mit den Ergebnissen dieser Arbeit noch nicht ausgeschöpft. Das in Abschnitt 6.1 erwähnte Schichtenprinzip macht die Verwendung von Anwendungswissen beim derzeitigen Stand problematisch.

Die Ersetzungsstrategien LIL und das Hotspot-Prefetching bräuchten eine Durchbrechung des Schichtenprinzips, um dem Cache Anwendungsinformationen über Knoten, Knotengeschwindigkeiten und Trajektorien zu liefern. Sollte die Entscheidung getroffen werden, eine solche Durchbrechung zu realisieren, so wären diese Strategien verwendbar. Die Komplexität

der Simulation und der Kommunikationsaufwand würde durch eine solche Erweiterung steigen, da weitere Schnittstellen zwischen oberster und unterster Schicht zu schaffen wären. Ob eine Verbesserung der Cache-Performanz erreichbar ist, müsste durch weitere Experimente geklärt werden.

Weitere Analysen im Zusammenhang mit einer möglichen Komprimierung der Daten sollten durchgeführt werden, um mehr über den Zusammenhang zwischen Clustergröße und Performanz des Gesamtsystems zu erfahren.

# Literaturverzeichnis

- [Bel66] Laszlo A. Belady. A Study of Replacement Algorithms for Virtual-Storage Computer. *IBM Systems Journal* 5(2), pages 78 – 101, 1966.
- [can] CANU Mobility Simulation Environment (CanuMobiSim). In <http://canu.informatik.uni-stuttgart.de>, Universität Stuttgart.
- [CI] Pei Cao and Sandy Irani. Cost-aware www proxy caching algorithms. *Proceedings of the USENIX Symposium on Internet Technologies and Systems, Monterey, California, December 1997*.
- [DFJ<sup>+</sup>96] Shaul Dar, Michael J. Franklin, Bjorn T. Jonsson, Divesh Srivastava, and Michael Tan. Semantic data caching and replacement. pages 330 – 341, 1996.
- [Dic07] Björn Dick. Organizing Multidimensional Wave Propagation Data on Disk for Efficient Read Access. Studienarbeit Nr. 2099, Fakultät Informatik, Universität Stuttgart, 2007.
- [Gri03] Matthias Grimrath. Simulation von Funkwellenausbreitung im Gigahertzband durch Strahlverfolgung auf semi-diskreten 2D-Stadtmodellen. Diplomarbeit, Technische Universität Braunschweig, Institut für Computergraphik, 2003.
- [GWZ07] Mesut Günes, Martin Wenig, and Alexander Zimmermann. Improving manet simulation results - deploying realistic mobility and radio wave propagation models. *Proceedings of the 11st IEEE Symposium on Computers and Communications (ISCC'07)*, 2007.
- [HLR02] D. Herrscher, A. Leonhardi, and K. Rothermel. Modeling Computer Networks for Emulation. In *Proceedings of the 2002 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA 02)*, Las Vegas, pages 1725 – 1731, June 2002.
- [MGST70] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems Journal* 9(2), pages 78 – 117, 1970.
- [MHR07] S. Maier, D. Herrscher, and K. Rothermel. Experiences with Node Virtualization for Scalable Network Emulation. *Computer Communications* 30(5), pages 943 – 956, 2007.

- [ns2] The Network Simulator - ns-2. In <http://www.isi.edu/nsnam/ns/>, ns-2 Project.
- [PB03] Stefan Podlipnig and Laszlo Böszörményi. A survey of web cache replacement strategies. *ACM Computing Surveys*, Vol. 35, No.4, December 2004, pages 374 – 398, 2003.
- [SA93] Rabin A. Sugumar and Santosh G. Abraham. Multi-Configuration Simulation Algorithms for the Emulation of Computer Architecture Designs. Technical report, Advanced Computer Architecture Laboratory, University of Michigan, 1993. Pages 70 - 94.
- [SHB<sup>+</sup>03] I. Stepanov, J. Hähner, C. Becker, J. Tian, and K. Rothermel. A meta-model and framework for user mobility in mobile networks. *Proceedings of the 11th International Conference on Networking 2003 (ICON 2003)*, pages 231 – 238, 2003.
- [Smi] Alan Jay Smith. Disk cache – miss ratio analysis and design considerations. *ACM Transactions on Computer Systems*, Vol. 3, No. 3, August 1985.
- [SW06] Arne Schmitz and Martin Wenig. The effect of the radio wave propagation model in mobile ad hoc networks. *ACM MSWiM 2006*, pages 61 – 67, 2006.
- [Tan01] Andrew S. Tanenbaum. *Modern Operating Systems, 2nd Edition*, page 214ff. Prentice Hall, 2001.
- [Wei91] M. Weiser. The Computer for the 21st Century. *Scientific American* 265, No. 3, pages 94 – 104, 1991.
- [WLRW04] A. Wierzbicki, N. Leibowitz, M. Ripeanu, and R. Wozniak. Cache replacement policies revisited: The case of p2p traffic. *4th GP2P Workshop, Chicago, IL*, 2004.

Ich versichere, dass ich diese Arbeit selbständig verfasst und nur die angegebenen Hilfsmittel verwendet habe.

Stuttgart, den

---

Philipp Häuser