Endbericht
PG 503

# Xaver: Algorithm Engineering XXL


Teilnehmer

Karl Becker
Daniel Dombrowski
Stefan Kloe
Jochen Klump
Sebastian Land
Matthias Niewerth
Mathias Schwarzhoff

Betreuer

Prof. Dr. Petra Mutzel
Markus Chimani
Carsten Gutwenger
Karsten Klein

Lehrstuhl XI – Algorithm Engineering
Fakultät für Informatik
Universität Dortmund


WS 2006/2007 – SS 2007

# Inhaltsverzeichnis


**1 Einleitung** **4**
- 1.1 Lizenz . . . 6
  - 1.1.1 BSD Lizenz im Volltext . . . 6

**2 Organisation und Ablauf der Projektgruppe** **7**
- 2.1 Seminarphase . . . 7
- 2.2 Planung der Xaver-Library . . . 8
- 2.3 Implementierung der Xaver-Library . . . 8
- 2.4 Die Xaver Template Library . . . 8
- 2.5 Debuggen der Xaver-Library . . . 9
- 2.6 Die erste Anwendung für Xaver: Der Suffix-Array-Generator . . . 9
- 2.7 Dokumentation und Tools zur Teamarbeit . . . 10

**3 Die Seminarvorträge** **11**
- 3.1 A Survey of Techniques for Designing I/O-Efficient Algorithms . . . 11
- 3.2 Elementary Graph Algorithms in External Memory . . . 12
- 3.3 Computing Point-to-Point Shortest Paths from External Memory . . . 13
- 3.4 Engineering Multi-Level Overlay Graphs for Shortest Path Queries . . . 13
- 3.5 Suffix-Arrays . . . 14
- 3.6 Replacing Suffix Trees with Enhanced Suffix Arrays . . . 15
- 3.7 Lists Revisited: Cache Conscious STL Lists . . . 16

**4 Die Xaver-Library** **17**
- 4.1 Die externe Sicht auf Xaver . . . 17
  - 4.1.1 Welche Parameter lassen sich in Xaver variieren? . . . 17
  - 4.1.2 Die Modi aus Sicht des Benutzers . . . 18
  - 4.1.3 Welche Funktionen bietet Xaver dem Benutzer . . . 19
  - 4.1.4 Die Xaver Template Library . . . 20
- 4.2 Das Innenleben von Xaver . . . 20
- 4.3 Das Pointer-Konzept von Xaver . . . 20
  - 4.3.1 Implementierung als Templates . . . 21
  - 4.3.2 Dereferenzierung . . . 21
  - 4.3.3 Konstante Zeiger . . . 21
  - 4.3.4 Random-Access-Iteratoren . . . 21
  - 4.3.5 Weitere Operationen . . . 22
  - 4.3.6 FarPointer . . . 23
  - 4.3.7 NearPointer . . . 24
  - 4.3.8 BlockPointer . . . 25
- 4.4 Die interne Memory-Management-Unit . . . 25

4.4.1 Die Klasse InternalMMU . . . 25
4.4.2 Die Block-Tabelle . . . 28
4.4.3 Der Block . . . 28
4.4.4 Caching-Strategien . . . 30
4.5 Die externe Memory-Management-Unit . . . 31
4.5.1 Die Klasse ExternalMMU . . . 31
4.5.2 ExternalWorkerThread . . . 34
4.5.3 IO . . . 35
4.5.4 Logging . . . 37
4.6 Der Object-Controller . . . 38
4.6.1 Die Schnittstelle . . . 38
4.6.2 Der SimpleObjectController . . . 39
4.6.3 Der QuickObjectController . . . 41

**5 Die Xaver Template Library 45**
5.1 Fully Buffered Reader/Writer . . . 45
5.1.1 Beschreibung der Schnittstelle . . . 45
5.1.2 Die Funktionsweise . . . 45
5.2 Sortieren . . . 46
5.2.1 Beschreibung der Schnittstelle . . . 46
5.2.2 Funktionsweise des Sortieralgorithmus . . . 46
5.2.3 Laufzeit . . . 47
5.2.4 Optimierungen . . . 48
5.3 NaturalNumberSort . . . 48
5.3.1 Idee . . . 48
5.3.2 Realisierung . . . 48
5.3.3 Schnittstelle . . . 49
5.4 Vector . . . 49
5.4.1 Beschreibung der Schnittstelle . . . 49
5.4.2 Funktionsweise . . . 50
5.5 Liste . . . 50
5.5.1 Die Idee . . . 50
5.5.2 Die Umsetzung . . . 53
5.5.3 Das Interface von `List` . . . 56
5.5.4 Das Interface von `Iterator` . . . 58
5.6 Stack . . . 58
5.6.1 Beschreibung der Schnittstelle . . . 58
5.6.2 Die Funktionsweise . . . 59
5.7 Queue . . . 60
5.7.1 Beschreibung der Schnittstelle . . . 60
5.7.2 Die Funktionsweise . . . 60
5.8 PriorityQueue . . . 61
5.8.1 Funktionsweise . . . 61
5.8.2 Interface in der Xaver Template Library . . . 62
5.9 Baum . . . 62
5.9.1 Beschreibung der Schnittstelle . . . 62
5.9.2 Die Technik . . . 63
5.9.3 Die Klasse Map . . . 63

**6 Suffix-Arrays** **65**
6.1 Zielsetzung . . . 65
6.2 Interner Algorithmus . . . 65
6.3 Externer Algorithmus . . . 66
6.3.1 Erzeugen von Suffix-Arrays . . . 66
6.3.2 Generieren des Strings für den rekursiven Aufruf . . . 66
6.3.3 Erzeugen von $SA_0$ . . . 67
6.3.4 Zusammenfassen von $SA_{12}$ und $SA_0$ . . . 67
6.3.5 Rekursionsabbruch . . . 67
6.3.6 Semi-Interne Berechnung der Tripel . . . 67
6.3.7 Verkürzung des Strings . . . 68
6.4 Implementierung . . . 68
6.4.1 Beschreibung der Schnittstelle der Klasse `SuffixArray` . . . 69
6.4.2 Algorithmen zur Erstellung von Suffix-Arrays . . . 69
6.4.3 SuffixArrayGenerator . . . 69
6.4.4 SuffixArrayGeneratorNaiv . . . 70
6.5 Mögliche Verbesserungen . . . 70
6.5.1 Pipelining . . . 70
6.5.2 Effizientere Sortierverfahren . . . 71

**7 Vergleichsstudien** **72**
7.1 XAVER-Allocator . . . 72
7.2 List . . . 73
7.2.1 Test-Szenario . . . 73
7.2.2 Ergebnisse . . . 74
7.3 Tree . . . 84
7.3.1 Test-Szenario . . . 84
7.3.2 Durchführung und Ergebnisse . . . 84
7.4 Sortieren . . . 89
7.4.1 Test-Szenario . . . 89
7.4.2 Ergebnisse . . . 89
7.5 Suffix-Arrays . . . 94
7.5.1 Test-Szenario . . . 94
7.5.2 Ergebnisse . . . 94
7.6 Caching-Strategien . . . 97
7.6.1 Test-Szenario . . . 97
7.6.2 Durchführung und Ergebnisse . . . 97

**8 Fazit und Ausblick** **100**

# Kapitel 1

# Einleitung

Das noch sehr junge Forschungsgebiet des *Algorithm Engineering* befasst sich mit der Entwicklung und Evaluierung von Algorithmen und berücksichtigt insbesondere die Aspekte realer Rechnerarchitekturen. Während die klassische Algorithmik das sehr einfache von-Neumann Maschinenmodell verwendet und sich häufig mit der asymptotischen Worst-Case Analyse von Algorithmen begnügt, orientiert sich das Algorithm Engineering stärker an der Praxis:

- Moderne Prozessorarchitekturen sind komplex: Sie beinhalten Parallelismus (pipelining, branch prediction) und Multi-Level Caches.
- Der interne Speicher eines Computers ist begrenzt; wollen wir größere Datenmengen verarbeiten, dann müssen wir auf den sehr viel langsameren externen Speicher (Festplatten) ausweichen.
- Die in der Theorie „besten“ Algorithmen sind in vielen Fällen sehr komplex und daher schwierig und fehleranfällig zu implementieren; in der Praxis bevorzugt man daher einfache Algorithmen.
- In der Praxis gibt es häufig „typische“ Eingaben für einen Algorithmus und es ist sehr wichtig, dass ein Algorithmus für diese Eingaben schnell ist. Daher gibt es für viele Probleme Sammlungen von Eingabeinstanzen, so genannte Benchmark-Daten, mit denen Implementierungen von Algorithmen auf realen Rechnern evaluiert werden können. Auch sind konstante Faktoren, die bei der klassischen Laufzeitanalyse in der $O$-Notation verschwinden, in der Praxis sehr wohl von Bedeutung.

Im Rahmen dieser PG beschäftigen wir uns mit Algorithmen für große Datenmengen. Gerade in den letzten Jahren ist die Flut der zu verarbeitenden Daten stark gewachsen. So misst die NASA das Datenvolumen von Satellitenbildern in Petabytes ($10^{15}$ Bytes)! Auch in der Bioinformatik müssen große Mengen an Daten verarbeitet werden, die durch immer neue Methoden wie z.B. Genexpressionsanalyse produziert werden. Neue Screeningtechnologien erzeugen riesige Datenmengen mit mehr als 50 Millionen Werten pro Monat!

Für die Effizienz eines Algorithmus in der Praxis ist es wichtig, dass die Speicher-Hierarchie moderner Rechnerarchitekturen möglichst gut ausgenutzt wird. So ist der Zugriff auf den Cache-Speicher des Prozessors etwa 100-mal schneller als der Zugriff auf den Hauptspeicher, d.h. wir wollen möglichst wenige Cache Misses (Zugriff auf Daten, die noch nicht im Cache sind) verursachen. Sind andererseits die zu verarbeitenden Datenmengen so groß, dass auch der Hauptspeicher nicht ausreicht, dann müssen die Daten teilweise auf externen Speicher wie Festplatten ausgelagert

werden. Zugriffe auf den externen Speicher sind aber etwa 1000-mal langsamer als auf den Hauptspeicher; allerdings kann bei jedem Zugriff ein Block von konsekutiven Daten auf einmal übertragen werden. Für die Effizienz des Algorithmus ist es wichtig, die Anzahl dieser Block-Transfers (*IO-Zugriffe*) möglichst klein zu halten.

Wir konzentrieren uns im Rahmen dieser PG wegen der geringen Teilnehmerzahl auf ein algorithmisches Problem:

- *Sequenz-Suche in Protein-Datenbanken.* Dies ist ein typisches Problem der Bioinformatik, bei dem eine Vielzahl von als Zeichenketten codierten Proteinen durchsucht werden müssen. Die Datenmenge übersteigt die Größe des Hauptspeichers um ein Vielfaches; man löst dieses Problem durch externe Suffix-Arrays.

Das Ziel der PG ist die Entwicklung einer in C++ geschriebenen Plattform zur Implementierung und Evaluierung von Externspeicher-Algorithmen (*EM-Algorithmen, external memory*), sowie die Implementierung verschiedener Varianten zum Aufbau von Suffix-Arrays. Bei der Behandlung der algorithmischen Probleme implementiert die PG nicht nur bereits vorhandene Veröffentlichungen; ein besonderer Schwerpunkt liegt auch auf der Erarbeitung und Umsetzung eigener Ideen zur Verbesserung der Algorithmen! Die einzelnen Aufgabenblöcke sind wie folgt:

1. *Externspeicher-Plattform* XAVER (*eXternal-memory Algorithms in a Versatile EnviRonment*). Diese Plattform ist eine in C++ geschriebene Bibliothek, welche die Implementierung und Evaluierung von EM-Algorithmen unterstützt. Insbesondere regelt sie die Verwaltung des internen Speichers und den Zugriff auf den externen Speicher. EM-Algorithmen verwenden im Wesentlichen zwei Parameter: Die Größe $M$ des internen Speichers und die Blockgröße $B$ eines IO-Zugriffs; letzteres ist die Anzahl konsekutiver Bytes, die bei einem IO-Zugriff auf einmal in den internen Speicher übertragen werden. Entsprechend unterstützt die Plattform auch die Simulation verschiedener Werte für $M$ und $B$. Zwei Simulationsmodi sind dabei vorhanden:

   (a) Der interne *und* der externe Speicher werden im RAM des Computers gespeichert, d.h. $M + n \cdot B \leq$ RAM-Speicher. Dieser Modus erlaubt es, sehr einfach die Größe des internen Speichers zu variieren und den Einfluss dieser Größe auf den Algorithmus zu untersuchen. Desweiteren ist es so möglich schon mit kleinen Eingabeinstanzen, die eine geringer Laufzeit haben und eigentlich komplett in den internen Speicher passen, das Verhalten des Algroithmus bezüglich des externen Speichers zu untersuchen.

   (b) Der interne Speicher wird im RAM und der externe Speicher auf der Festplatte gespeichert. Diese Variante erlaubt also die Verarbeitung von Datenmengen, die nicht in den RAM des Computer passen; dadurch können die EM-Algorithmen dann auch für praktische Anwendungen genutzt werden.

2. *Grundlegende Datenstrukturen und Algorithmen.* Einige Datenstrukturen wie Stacks, Queues und Listen, sowie Algorithmen wie das externe Sortieren werden immer wieder in EM-Algorithmen verwendet und werden daher von der XAVER-Plattform zur Verfügung gestellt.

3. *Suffix-Arrays.* Zum Aufbau eines Suffix-Arrays existieren bereits verschiedene EM-Algorithmen, z.B.[5, 4]. Neben der Implementierung dieser Algorithmen sollen auch Verbesserungsmöglichkeiten, insbesondere im Kontext gegebener Benchmark-Daten aus der Bioinformatik, untersucht werden.

4. *Evaluierung der Algorithmen.* Die XAVER-Plattform erlaubt uns, die Performance von EM-Algorithmen experimentell zu evaluieren. Daher sollen verschiedene EM-Algorithmen zum Aufbau eines Suffix-Arrays und zur Berechnung von kürzesten Wegen in großen Graphen an Hand von praxisrelevanten Benchmark-Daten verglichen werden.

## 1.1 Lizenz

Die im Rahmen dieser Projektgruppe erstellte Software soll der Allgemeinheit als Open-Source zur Verfügung gestellt werden. Deshalb wird XAVER unter die 3-clause BSD Lizenz gestellt, um eine größtmögliche Flexibilität zu gewährleisten. Bei Software unter der BSD Lizenz ist es erlaubt, diese zu kopieren, zu verändern und zu verbreiten. Es ist nicht verpflichtend, den Quellcode von Änderungen zu veröffentlichen; außerdem ist die BSD Lizenz mit der GPL kompatibel.

### 1.1.1 BSD Lizenz im Volltext

# Kapitel 2

# Organisation und Ablauf der Projektgruppe

Der Ablauf der Projektgruppe lässt sich grob in fünf größere Abschnitte unterteilen. Zunächst fand in der *Seminarphase* eine allgemeine Einführung in das Thema statt. Dadurch hatten die Teilnehmer die Möglichkeit, sich bereits vor dem Beginn der tatsächlichen Gruppenarbeit kennen zu lernen. In der *Planungsphase* stellten dann alle Teilnehmer gemeinsam den Anforderungskatalog auf und erarbeiteten die grobe Struktur der XAVER-Bibliothek (siehe Kapitel 4). Danach spezifizierten kleinere Gruppen die einzelnen Komponenten. Die gleichen Gruppen setzten in der *Implementierungsphase* die einzelnen Komponenten um. Kleinere Änderungen wurden während den Teamsitzungen ins Design aufgenommen. Die *Debbugingphase* wurde mit der *Implementierung* der Standardbibliothek XAVER TEMPLATE LIBRARY(siehe Kaptiel 5) kombiniert, um die möglichen Synergieeffekte zu nutzen. Diese Bibliothek sollte die Standarddatenstrukturen und -algorithmen beinhalten, die auf den Einsatz mit XAVERoptimiert wurden. Da wir zum Einen für die Implementierung und das Debugging einige dieser Algorithmen deutlich mehr Zeit benötigten als vermutet, zum Anderen aber noch Unzulänglichkeiten in XAVER selbst aufgetreten waren, entschieden wir in der *Anwendungsphase* parallel zu arbeiten. Wir teilten uns in zwei Arbeitsgruppen mit jeweils parallelen Arbeitsfeldern:

- Die Weiterentwicklung und Pflege von XAVER und parallel dazu die Planung und Durchführung der Ergebnisanalysen.
- Die Fertigstellung der Standardbibliothek XAVER TEMPLATE LIBRARY sowie Fehlerbehebung in dieser und parallel dazu die Planung und Implementierung des geforderten Suffix-Array-Generators.

Während der Endphase kam die Weiterentwicklung des Zwischenberichts zum Endbericht für jeden einzelnen Teilnehmer hinzu.

## 2.1 Seminarphase

In den ersten zwei Wochen der Projektgruppe fand die Seminarphase statt. Jeder Teilnehmer musste den anderen Projektgruppenmitgliedern ein Thema vorstellen. Die Themen wurden von den Betreuern vorgegeben und den Teilnehmern zugelost. Ziel der Seminarphase war, dass alle Teilnehmer grundlegende Kenntnisse in speziell für die Projektgruppe relevanten Themenbereichen bekommen.

## 2.2 Planung der Xaver-Library

Bevor zur tatsächlichen Planung von Xaver übergegangen werden konnte, musste zunächst ein gemeinsames Verständnis für den Zweck und die vorgegebene Funktionalität von Xaver gefunden werden.

Die daraus abgeleiteten Anforderungen werden detailliert im Kapitel 4.1 beschrieben. Nachdem die Anforderungen abgeleitet wurden, musste ein Konzept gefunden werden, das diese erfüllt und flexibel genug ist, um verschiedene Ansätze testen zu können. Dieses Konzept wurde in Gruppenbesprechungen erarbeitet, damit alle Teilnehmer mit dem Design und der späteren Funktionsweise vertraut sind. Dieses Grundkonzept ist in Kapitel 4.2 näher erläutert. Teil dieses Grundkonzeptes war auch bereits ein komplettes Interface für Xaver und die einzelnen Komponenten. Auf dieser Basis konnte nun die Planung der konkreten Funktionsweisen der Komponenten beginnen.

Da das Besprechen der einzelnen Komponenten den Rahmen der gemeinsamen Sitzungen gesprengt hätte, wurden die Komponenten auf einzelne Gruppen aufgeteilt. Die Aufteilung der Personen in die Gruppen folgte hauptsächlich den zeitlichen Einschränkungen der einzelnen Personen. So wurden diejenigen Personen mit der größten Überschneidung in ihren Zeitfenstern in die gleiche Gruppe eingeteilt. Da Matthias und Daniel über die größten C++ Kenntnisse verfügten, beteiligten sie sich an unterschiedlichen Gruppen, damit möglichst viele Teilnehmer von ihnen lernen konnten. Dadurch ergab sich folgende Einteilung in die Gruppen:

- Die InternalMMU übernahmen Karl Becker und Daniel Dombrowski
- Die ExternalMMU übernahmen Jochen Klump, Mathias Schwarzhoff und Matthias Niewerth
- Den ObjectController übernahmen Stefan Kloe und Sebastian Land

Wöchentliche Treffen hielten dabei die gesamte Gruppe über die Fortschritte bei den Planungen auf dem Laufenden. Gleichzeitig konnten so nötige Änderungen an den Interfaces vorgenommen werden, da einige Fälle im Vorfeld nicht bedacht worden waren.

## 2.3 Implementierung der Xaver-Library

Nachdem die Planungen zu den einzelnen Komponenten abgeschlossen waren, wurde beraten, wie die Implementierung erfolgen sollte. Da die jeweiligen Planungsgruppenmitglieder am besten über die Konstruktion der Komponenten informiert waren, sollten sie auch die Implementierung übernehmen. So wurde dann gruppenintern diskutiert und entschieden, wer welche Klasse implementiert.

Um später die reibungslose Integration der verschiedenen Komponenten zu gewährleisten, sollte jede Gruppe ihre Klassen schon während der Implementierung mittels selbsterstellter Klassentests auf die vom Interface verlangte Funktionsweise testen.

## 2.4 Die Xaver Template Library

Da die Anforderungen und Interfaces der Xaver Template Library bereits durch die STL von C++ vorgegeben waren, waren größere Planungen weitestgehend überflüssig, nachdem sich die Gruppe darauf geeinigt hatte sich an diesem verbreiteten Standard zu orientieren.

Da jede Datenstruktur eine in sich geschlossene Komponente darstellt, schien es zweckmäßig, jeder Datenstruktur nur eine Person zuzuordnen. Dadurch sollten Reibungsverluste durch unnötige Kommunikation über die komplexen Schnittstellen innerhalb einer Datenstruktur vermieden werden.

Da einige Gruppenmitglieder bereits in der Seminarphase Vorträge über eine der Datenstrukturen gehalten hatten, bot es sich an, dass sie die entsprechende Datenstruktur implementieren. Die weiteren Datenstrukturen wurden dann auf die übrigen Gruppenmitglieder aufgeteilt:

- Karl Becker und Stefan Kloe übernahmen die Tree-basierte Implementierung einer Map
- Jochen Klump übernahm die Implementierung des Stacks und der Queue
- Mathias Schwarzhoff übernahm den Vector
- Sebastian Land übernahm die Liste
- Matthias Niewerth übernahm die Implementierung der PriorityQueue
- Daniel Dombrowski implementierte den Sortieralgorithmus

## 2.5 Debuggen der Xaver-Library

Neben den Klassentests der einzelnen Komponenten wurde auf ein weitergehendes Debugging verzichtet. Um dennoch eine effiziente Fehlerbehebung zu ermöglichen, wurde ein Fehlerverzeichnis im Wiki angelegt. Da zeitgleich die Xaver Template Library erstellt wurde, die damit die erste Anwendungsmöglichkeit für Xaverwar, sollten alle dabei gefunden XaverFehler in das Verzeichnis eingetragen werden, damit diese anschließend von den jeweiligen Verantwortlichen behoben werden konnten.

Der Hintergedanke war, dass man auf diese Art und Weise die Zeit für das Erstellen von expliziten Debugging Programmen verzichten kann, die später keinerlei Nutzen mehr haben. Da die Xaver Template Library hauptsächlich Datenstrukturen enthält, die extern liegende Datenmengen bewältigen, lag es nahe, ihre Funktion zu benutzten, um das korrekte Verwalten des Speichers durch Xaver zu testen. Wir vermuteten, dass dabei deutlich mehr Randfälle und Timingprobleme zu Tage treten, als bei manuell geschriebenen Testfällen, die naturbedingt keine so hohe Komplexität haben.

## 2.6 Die erste Anwendung für Xaver: Der Suffix-Array-Generator

Um die Praktikabilität von Xaver zu testen, sollte ein Suffix-Array-Generator auf Basis von Xaver implementiert werden. Nach ersten Literaturrecherchen entschied sich die damit betraute Gruppe dazu, den DC3 Algorithmus von Kärkkäinen und Sanders aus [14] zu implementieren. Da die beschriebene Version als interner Algorithmus optimiert wurde, waren zahlreiche Anpassungen nötig, um wahlfreien Zugriff auf den externen Daten zu vermeiden. Nach einer ausführlichen theoretischen Planungsphase konnte eine erste Version implementiert werden. Dem Algorithm-Engineering-Gedanken folgend wurde diese dann kontinuierlich optimiert.

## 2.7 Dokumentation und Tools zur Teamarbeit

Gerade für die Zusammenarbeit in der Gruppe und für die Sicherung unserer erarbeitenden Erkenntnisse war die Nutzung von Hilfsmitteln unerlässlich.

### Das Wiki

Das PG-Wiki auf den Seiten des LS11 diente uns seit Beginn der Projektgruppe als Wissenspeicher. Neben den Protokollen der wöchentlichen Gruppensitzungen wurden hier auch wichtige Spezifikationen und Entwürfe der einzelnen Teilgruppen hinterlegt. Somit waren immer alle Teilnehmer auch der anderen Gruppen auf dem gleichen Wissensstand.

Von großem Vorteil war das Wiki auch in technischen Fragen. Hier wurden alle Beschreibungen, die für die Installation und Nutzung der Teile unserer Entwicklungsumgebung nötig waren, hinterlegt.

### Programmieren im Team

Um gemeinsam die gleichen Sourcen nutzen zu können, haben wir uns für eine Quellcodeverwaltung mit SVN entschieden. Ebenfalls haben wir uns zu Beginn der Programmierphase auf Coding-Conventions geeinigt, die eine bessere Lesbarkeit des Quellcodes gewährleisten sollten. Wir haben hier insbesondere Wert auf eine einheitliche Namensgebung für Klassen und Variablen gelegt. Ein ebenfalls sehr wichtiger Punkt war die Kommentierung der Funktions-Signaturen im JavaDoc-Stil. Somit konnten wir mit Hilfe des Tools Doxygen eine übersichtliche Dokumentation der Schnittstellen generieren, die eine spätere Nutzung der Sourcen auch durch Andere ermöglicht.

# Kapitel 3

# Die Seminarvorträge

Im Folgenden finden sich kurze inhaltliche Zusammenfassungen der einzelnen Seminarvorträge.

Als Einstieg in die Seminarphase hielt Daniel Dombrowski einen einführenden Vortrag über wichtige Techniken bei der Implementierung von Externspeicher-Algorithmen. Einen Einblick in die Thematik von IO-Effizienten Graph-Algorithmen gab Stefan Kloe in seinem Vortrag. Die nächsten beiden Vorträge widmeten sich Methoden der Suche nach kürzesten Wegen in Graphen: Jochen Klump veranschaulichte in seinem Referat die Entwicklungsgeschichte der point-to-point Suche im externen Speicher. Das von Mathias Schwarzhoff behandelte Paper widmete sich hauptsächlich der Methode der Overlay-Graphen, um genau dieses Problem zu lösen.

Über das zweite Hauptthema der PG, die Suffix-Arrays referierten anschließend Matthias Niewerth und Karl Becker. Im ersten Vortrag ging es hauptsächlich darum, was Suffix-Arrays überhaupt sind, und welche Fragestellungen mit ihrer Hilfe bearbeitet werden können. Der zweite Vortrag beschreibt Methoden zur Abbildung von Suffix-Trees in Suffix-Arrays. Zum Vortrag von Silke Hochgräber über „Full-Text Indexes in External Memory“ sind uns leider keine Materialien geblieben, da Silke vorzeitig aus der PG ausgeschieden ist. Als letztes Thema behandelte Sebastian Land Cache Conscious STL Lists. Hier ging es insbesondere darum, die Anforderungen der STL-Schnittstelle an Listen unter Beachtung der Cache-Memory-Hierarchie effizient umzusetzen.

## 3.1 A Survey of Techniques for Designing I/O-Efficient Algorithms

In [15] stellen Anil Maheshwari und Norbert Zeh einige grundlegende Techniken für Externspeicheralgorithmen vor. Dabei werden sowohl algorithmische Konzepte vorgestellt als auch externspeicheroptimierte Datenstrukturen diskutiert. Der Seminarvortrag behandelte daraus einige ausgewählte Themen.

Ein sehr grundlegendes Konzept für Externspeicheralgorithmen ist das sogenannte Scanning-Paradigma. Dabei geht es darum, möglichst wahlfreien Zugriff auf extern gespeicherte Daten zu vermeiden und die Daten in Form eines Datenstroms zu lesen bzw. zu schreiben. Im Idealfall kann man durch Anwendung dieses Paradigmas an Stelle von $N$ IO-Operationen für $N$ Daten $N/B$ IO-Operationen erhalten, wenn die Blockgröße des externen Speichers $B$ ist.

Um das Scanning-Paradigma anwenden zu können, ist es häufig notwendig, die Daten zunächst in die richtige Reihenfolge zu bringen. Dazu wird ein effizienter, externer Sortieralgorithmus benötigt. Der vorgestellte Algorithmus basiert auf dem

Konzept des MergeSort und wurde auch so in der Xaver Template Library umgesetzt. Eine Beschreibung der konkreten Umsetzung findet sich in Kapitel 5.2.

Weiterhin wurden einige Eigenschaften und Konzepte erarbeitet, um Internspeicheralgorithmen für Graph-Probleme externspeichereffizient zu gestalten. Dabei wurde wesentliches Augenmerk auf Algorithmen gelegt, die greedy vorgehen, d.h. deren Berechnungen nur lokal auf kleinen Teilen des Graphen stattfinden.

Im letzten Abschnitt ging es um externspeichereffiziente Datenstrukturen für Graphen. Als einfachster Graph wurde eine (doppelt) verkettete Liste betrachtet. Weiterhin wurden Techniken für Bäume, 2-dimensionale Grids sowie planare Graphen vorgestellt. Dabei wurden für alle Graphtypen Konzepte erarbeitet, wie diese geschickt auf die Blöcke des externen Speichers aufgeteilt werden können. Auch wenn einige der Betrachtungen recht speziell waren, so lassen sie sich dennoch auf allgemeine, zeigerbasierte Datenstrukturen übertragen.

## 3.2 Elementary Graph Algorithms in External Memory

In [12] beschreiben Irit Katriel und Ulrich Meyer IO-effiziente Varianten grundlegender Algorithmen für die Graphen-Traversierung und das Auffinden von Zusammenhangskomponenten.

Der Breitensuch-Algorithmus von Munagala und Ranade [17] berechnet die BFS-Ebenen des Graphen durch die Betrachtung der Adjazenzlisten der Knoten benachbarter Ebenen. Neuere Algorithmen bauen auf diesem Verfahren auf. Mehlhorn und Meyer [16] verbesserten diesen Algorithmus durch eine Partitionierung des Graphen und das Vorladen weiterer Adjazenzlisten (Ausnutzen der IO-Zugriffe auf die Adjazenzlisten). Dieser Algorithmus lässt sich auch für die Berechnung von Zusammenhangskomponenten verwenden, indem er nach Beendigung mit eventuell vorhandenen bisher nicht besuchten Knoten neu gestartet wird. Zur Verbesserung der Laufzeit kann vor der Ausführung eine Knotenreduktion des Graphen [17] vorgenommen werden.

Der Single-Source-Shortest-Path-Algorithmus (SSSP) von Kumar und Schwabe [13] kann mittels IO-effizienten Tournament Trees als Variante des Dijkstra-Algorithmus [6] implementiert werden. Diese wurden entwickelt, um Priority Queues effizient im externen Speicher halten zu können. Während Tournament Trees vollständige binäre Bäume sind, verbessert die IO-effiziente Variante diesen Ansatz über das Ausnutzen der Blockgröße und Caching.

Der Buffered Repository Tree [3] wird benutzt, um auf ungerichteten Graphen bereits besuchte Knoten nachzuhalten, indem er jede Kante des Graphen zusammen mit einem Schlüssel extern speichert. Als Operationen bietet er das Einfügen eines Tupels (Kante, Schlüssel) und das Auslesen aller Kanten zu einem Schlüssel an.

Der Algorithmus zur Auffindung des minimalen Spannwaldes nach Jarnik und Prim [18] erstellt die einzelnen Spannbäume nacheinander. Dabei wird eine Priority Queue genutzt, die jeden Knoten mit dem Gewicht seiner leichtesten Kante zu dem aktuellen Spannbaum enthält. Da alle Gewichte ständig aktualisiert werden müssen, werden bei der Verbesserung von Arge et al. [2] Kanten an Stelle der Knoten zur Queue hinzugefügt und es wird nur noch betrachtet, ob der Zielknoten schon im Spannbaum enthalten ist. Wieder kann der Graph zur Vorbereitung durch eine Knotenreduktion verkleinert werden. Dazu wird jeder Knoten mit seinem nächsten Nachbarn verschmolzen.

Der Algorithmus von Tarjan und Vishkin [19] zur Berechnung von Zwei-Zusammenhangs-Komponenten erstellt einen abgeleiteten Graphen aus den Kanten des zu betrachtenden Graphen und verbindet dort alle Knoten, deren dargestellte Kan-

ten im Ursprungsgraphen auf einem einfachen Zyklus liegen. Der aufgebaute Graph kann mit Hilfe eines Spannbaums weiter reduziert werden und seine Zusammenhangs-Komponenten entsprechen den Zwei-Zusammenhangs-Komponenten des Ursprungsgraphen.

## 3.3 Computing Point-to-Point Shortest Paths from External Memory

In [8] präsentieren die Autoren Goldberg und Werneck einen IO-Effizienten Algorithmus für das *Point-to-Point-Shortest-Path* Problem. Hierbei wird auf einem gerichteten Graphen $G = (V, E)$ bei gegebenem Startort $s \in V$ und Zielort $t \in V$ der kürzeste Weg gesucht. Die Idee ist, den bekannten *Dijkstra*-Algorithmus so zu modifizieren, dass er sich frühzeitig intelligent in die richtige Richtung vorbewegt und unnötige Pfadkonstruktionen vermeidet.

Der Kern des resultierenden *ALT*-Algorithmus liegt darin, dass man für einen vorgegebenen Graphen $G$ versucht, eine möglichst gute Menge $V' \subset V$ von sogenannten *Landmarks* festzulegen. Für diese Knoten berechnet man einmalig die kürzesten Wege von und zu allen anderen Knoten. Da diese Menge immer noch zu groß ist, um sie komplett im Speicher zu halten, verwaltet man dynamisch zur Laufzeit eine Untermenge der *aktiven Landmarks*.

Betrachtet man nun zwei beliebige Punkte $v, w$ im Graphen, so lässt sich durch einfaches Aufzeichnen klar machen, dass die Distanz zwischen diesen Punkten für jeden Landmark $A$ nach unten durch

$$dist(v, w) \geq \max\{dist(v, A) - dist(w, A), dist(A, w) - dist(A, v)\}$$

abgeschätzt werden kann. Somit können mit Hilfe dieser unteren Schranken für die Länge des $s-t$-Pfades unnötige Pfadkonstruktionen frühzeitig unterbunden werden.

Im Paper werden nun zum einen Heuristiken vorgestellt, wie man für einen gegebenen Graphen eine möglichst „gute“ Menge von *Landmarks* finden kann, zum anderen werden Verfahren vorgestellt, mit deren Hilfe man online möglichst passende *Landmarks* aktiviert.

Als Hauptmethode im Zusammenhang der Landmarksuche wird *avoid* genannt. Wie der Name schon sagt, ist die Idee hinter dem *avoid*-Algorithmus, dass es vermieden werden soll, in ein Gebiet, das bereits „gut“ mit Landmarks abgedeckt ist, einen weiteren zu platzieren. Hierzu wird ausgehend von einem beliebigen Knoten $r$ im Graphen ein Shortest-Path-Baum zu allen anderen Knoten aufgebaut. Als neuer Landmark wird nun jener Knoten des Baumes gewählt, dessen Subgraph selbst keinen Landmark besitzt und der unter diesen in der Summe die bisher schlechteste Abschätzung durch die aktuellen Landmarks besitzt. Das Verfahren läuft dann iterativ weiter.

## 3.4 Engineering Multi-Level Overlay Graphs for Shortest Path Queries

In [11] werden Verfahren vorgestellt, um in einem Preprocessingschritt Datenstrukturen aufzubauen, die die Suche von kürzesten Wegen in Graphen beschleunigen. Ein Overlay Graph $G'$ zu einem gegebenen Graphen $G = (V, E)$ ist ein Graph mit Knotenmenge $S \subseteq V$, der gewisse Eigenschaften des Graphen $G$ aufrecht erhält. Für die hier behandelten Overlay Graphen gilt, dass ein kürzester Weg von $s$ nach $t$ in $G'$ die gleiche Länge hat wie in $G$ und dass die Kantenanzahl in $G'$ minimal ist. Mit einer Folge von Knotenteilmengen $V = S_0 \supseteq S_1 \supseteq \cdots \supseteq S_k$ erhält man

eine Menge von Overlay Graphen. Knoten $s$ und $t$ aus $G$ induzieren in den Overlay Graphen eine Kantenmenge, die für die Suche eines kürzesten Weges von $s$ nach $t$ ausreicht und die im Idealfall wesentlich kleiner als die Kantenmenge von $G$ ist.

Es werden verschiedene Kriterien zur Auswahl der Knoten für die Mengen $S_i$ vorgestellt.

- Random: Die Knoten werden zufällig gleichverteilt unter allen Knoten gewählt.
- Degree: Es werden die Knoten mit größtem Knotengrad gewählt.
- Percentage: Für einen Knoten $v$ ist der *percentage value* der Anteil der zu $v$ adjazenten Knoten, deren Grad kleiner ist als der von $v$, unter allen zu $v$ adjazenten Knoten. Die Knoten mit größtem percentage value werden gewählt.
- Core: Der *k-core* eines Graphen ist der größte Subgraph, so dass alle Knoten des Subgraphen einen Grad von mindestens $k$ haben. Die *core number* eines Knoten $v$ ist das größte $k$, so dass $v$ zum $k$-core des Graphen gehört. Die Knoten mit den größten core numbers werden gewählt.
- Closeness: Für einen Knoten $v$ ist *closeness* definiert als $c(v) = 1/\sum_{t \in V} d(v,t)$, wobei $d(v,t)$ die Entfernung zwischen $v$ und $t$ ist und $1/0$ als 0 definiert wird. Intuitiv gesehen hat ein Knoten mit großem closeness Wert kurze Entfernungen zu den meisten anderen Knoten. Es werden die Knoten mit den größten closeness Werten gewählt.
- Betweeness: Für einen Knoten $v$ ist *betweeness* definiert als $b(v) = \sum_{s,t \in V} \frac{\sigma(s,t|v)}{\sigma(s,t)}$, wobei $\sigma(s,t)$ die Anzahl der kürzesten Wege von $s$ nach $t$ und $\sigma(s,t|v)$ die Anzahl der kürzesten Wege von $s$ nach $t$, die $v$ als inneren Knoten enthalten, bezeichnet. Dabei sei $0/0$ als 0 definiert. Es werden die Knoten mit dem größten betweeness Wert gewählt.
- Betweeness Approximation: Die betweeness Werte werden approximiert, indem $s$ und $t$ aus der zufällig gewählten Menge $V' \subseteq V$ mit $|V'| = (\log n)/\varepsilon^2$ für eine Variable $\varepsilon$ gewählt werden. Die Wahrscheinlichkeit für einen Fehler größer als $\varepsilon n(n-2)$ ist höchstens $1/n$.

Diese und einige weitere Kriterien wurden exprimentell an verschiedenen Graphen untersucht. Den größten Speedup erreichte man mit betweeness und betweeness approximation, wobei sich letzteres deutlich schneller berechnen lässt und deshalb am vielversprechendsten erscheint.

## 3.5 Suffix-Arrays

Suffix-Arrays sind eine Datenstruktur, die effiziente Algorithmen auf Strings ermöglicht, z.B. die Suche nach Teilstrings. Dafür wird zusätzlich zu dem String ein Suffix-Array angelegt, das Zeiger auf alle Suffixe des Strings in geordneter Reihenfolge enthält. Da nur Zeiger auf die Suffixe angelegt werden, benötigt das Suffix-Array nur $\mathcal{O}(|S|)$ Speicherplatz, wenn S der String ist, zu dem das Suffix-Array gehört.

Im Skriptum zur Vorlesung Algorithmen auf Sequenzen [10] beschreibt Volker Heun die Funktionsweise von Suffix-Arrays, sowie mehrere (interne) Algorithmen zur Erstellung von Suffix-Arrays. Ferner zeigt er, wie man effizient in Suffix-Arrays suchen kann und wie man Algorithmen von Suffix-Trees auf Suffix-Arrays übertragen kann.

Von besonderem Interesse ist der vorgstellte Skew-Algorithmus[1] von Karkäinnen und Sanders. Dieser Algorithmus benutzt einen rekursiven Ansatz um Suffix-Arrays

[1] In neueren Veröffentlichungen auch DC3-Algorithmus genannt

in Linearzeit zu erstellen. Eine etwas ausführliche Beschreibung dieses Algorithmus ist im Kapitel über Suffix-Arrays (Kapitel 6) enthalten.

Wenn das Suffix-Array erstellt ist, kann man, da jeder Teilstring Präfix eines Suffixes ist, mittels binärer Suche Vorkommen von Teilstrings in Zeit $\mathcal{O}(|P|log|S|)$ finden, wenn P der gesuchte Teilstring ist.

Bei der einfachen binären Suche werden die Anfänge der Suffixe immer wieder mit dem Anfang von P verglichen. Dies kann verhindert werden, wenn man ausnutzt, dass Suffixe, die im Suffix-Array dicht beieinander liegen, häufig über lange gemeinsame Präfixe verfügen. Wenn man neben dem Suffix-Array noch eine Datenstruktur anlegt, die für je zwei Suffixe die Länge ihres längsten gemeinsamen Präfixes speichert, kann man die Suchzeit auf $\mathcal{O}(|P| + log|S|)$ verkürzen.

Die hierfür eigentlich benötigte quadratische Platz kann auf eine lineare Größe reduziert werden, wenn man nur die benötigten Einträge abspeichert. In der binären Suche werden grundsätzlich nur Suffixe, die an der linken oder rechten Intervallgrenze liegen, mit der Intervallmitte verglichen. Jede Position im Suffix-Array kann nur einmal Intervallmitte werden. Die Intervallgrenzen stehen dann eindeutig fest. Es reicht also aus, für jede Position die Längen der längsten gemeinsamen Präfixe mit den beiden Intervallgrenzen abzuspeichern.

## 3.6 Replacing Suffix Trees with Enhanced Suffix Arrays

In [1] stellen die Autoren Abouelhoda, Kurtz und Ohlebusch ein Verfahren vor, Algorithmen für Suffix-Trees auf Enhanced Suffix Arrays anzuwenden. Dies ist deswegen interessant, weil es für Suffix-Trees zwar eine Menge effizienter Algorithmen zur Stringverarbeitung gibt, durch den Platzverbrauch der Datenstruktur aber Grenzen in der Anwendung gesetzt sind. Enhanced Suffix Arrays dagegen sind relativ Speicherplatz-effizient. Um die Suffix-Tree-Algorithmen anwenden zu können, fügt man zum Suffix-Array zusätzliche Tabellen hinzu, so dass das Bottom-Up- und Top-Down-Traversieren des Baumes über dem Suffix-Array möglich wird.

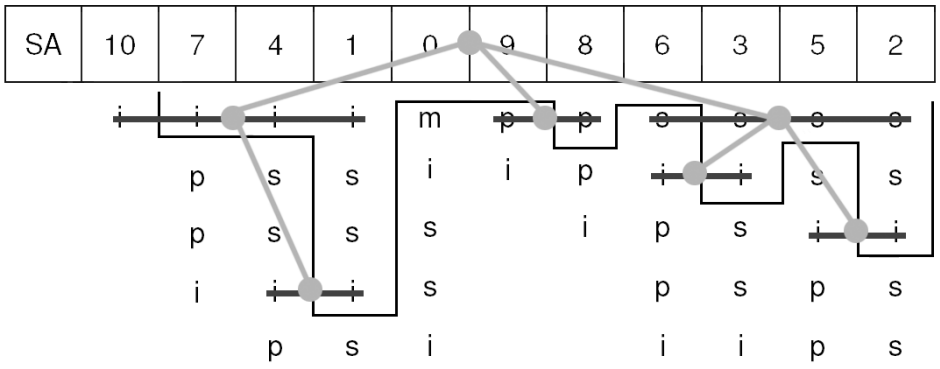


Abbildung 3.1: LCP-Intervall-Baum für Mississippi

Um die Bottom-Up-Traversierung zu ermöglichen, benötigt man die LCP-Tabelle (L), bei der an Position $i$ die gemeinsame Länge der Suffixe $SA[i-1]$ und $SA[i]$ steht. Die LCP-Tabelle ist in linearer Zeit zu erstellen und benötigt maximal $4n$ Bytes. Die LCP-Tabelle kann man nun anhand des Konzepts des LCP-Intervall-Baumes traversieren (siehe Abbildung 3.1), wobei ein LCP Intervall $[i, j]$ durch

$$L[i] < l;\ L[k] \geq l,\ \forall k \in [i+1, j];\ L[k] = l,\ \exists k \in [i+1, j];\ L[j+1] < l$$

definiert ist.

Um für die Top-Down-Traversierung nicht den Intervallbaum aufbauen zu müssen, benötigt man eine Tabelle, die die Elter-Kind-Beziehung der Intervalle (vgl. Abbildung 3.1) darstellt. Diese Funktion erfüllt die Child-Tabelle mit den drei Spalten Up, Down und Next. Up und Down verweisen vom rechten beziehungsweise linken Rand eines LCP-Intervalls auf die „Bruchstelle“ des Intervalls, also den linken Rand des rechten Kind-Intervalls. Next verweist auf das nächste Intervall auf der selben Ebene. Die Child-Tabelle hat eine Größe von $4(n + 1)$ Bytes und ist durch einen linearen Scan der LCP-Tabelle konstruierbar.

## 3.7 Lists Revisited: Cache Conscious STL Lists

In [7] wird eine Datenstruktur vorgestellt, die die STL-Anforderungen an eine Liste erfüllt und die für Speicherhierarchien moderner Rechner optimiert ist.

Um dies zu erreichen, verwendet sie “Bucket” genannte Datenbereiche, in denen einzelne Daten der Liste gespeichert werden. Die Buckets selbst werden in einer klassisch doppelt verlinkten Liste verwaltet. Durch diesen Trick wird eine erhöhte Lokalität der Daten erreicht, was die Verarbeitungsgeschwindigkeit in den vorgestellten Experimenten deutlich erhöht. Dieser Effekt tritt ein, da die Daten durch die Lokalität häufiger im Cache liegen.

Diese Lokalität muss allerdings mit einem Mehraufwand an Verwaltungsarbeit bezahlt werden, da sowohl die Daten innerhalb der Buckets verwaltet werden müssen, als auch Referrer genannte Elemente, die die Iteratoren auf den gerade aktuellen Speicherplatz eines Elementes verweisen. Die Referrer sind nötig, da sich durch Bucket-interne Verschiebungen oder Umordnungen der Speicherort der Daten verändern kann, dies aber die Iteratoren nicht ungültig machen darf.

Für die Organisation der Buckets nennen sie drei Varianten:

- In der *kontinuierlichen Organisation* werden die Daten in einem einfachen Array hintereinander in der Listenreihenfolge gehalten. Einfügen oder Löschen verursacht dann größere Verschiebungen.
- In der *kontinuierlichen Organisation mit Lücken* werden die Daten in einem einfachen Array in der Listenreihenfolge gehalten. Ein Feld markiert dabei, ob ein Array-Eintrag leer ist. Verschiebungen treten so seltener auf.
- In der *verlinkten Organisation* werden die Daten selbst in einer doppelt verlinkten Liste gehalten. Verschiebungen entfallen so vollständig für den Preis von größerem Overhead.

Für die Organisation der Referrer wird einmal die *field of pairs* Lösung vorgeschlagen, bei der jeder Dateneintrag einen Pointer auf seinen Referrer bekommt oder das Verwalten einer seperaten *referrer list*, die ihrerseits wieder eine doppelt verlinkte Liste ist.

# Kapitel 4

# Die Xaver-Library

XAVER ist eine C++ Bibliothek, die die Komplexität von Externspeicher-Implementierungen versteckt und dem Programmierer eine einfache Möglichkeit bietet, sich auf die wesentlichen Details im Algorithmus zu konzentrieren, ohne sich hierbei Gedanken darüber zu machen, wie er größere Datenmengen verwaltet.

## 4.1 Die externe Sicht auf Xaver

Die XAVER-Library verfolgt im Prinzip zwei Zielsetzungen:

1. Zum einen soll dem Programmierer eine Plattform zur Verfügung gestellt werden, mit deren Hilfe er reale externe Algorithmen entwickeln kann, ohne sich mit der Komplexität der Speicherverwaltung auseinander setzen zu müssen. Außerdem eröffnet XAVER die Möglichkeit mit Datenmengen zu arbeiten, deren Größe über den normalerweise adressierbaren Bereich hinaus geht (auf 32-Bit-Systemen in der Regel weniger als 4 GB).

2. Zum anderen soll XAVER die Möglichkeit bieten, externe Algorithmen auf kleinen Datenmengen komplett im internen Speicher zu simulieren. Dadurch sollen die Einflüsse konkreter Rechner-Peripherien auf die Güte eines Algorithmus eliminiert werden.

Bei beiden Varianten werden die signifikanten Speicherzugriffe direkt von der Plattform protokolliert. Somit liefert XAVER Informationen, die in wissenschaftliche Vergleichsstudien einzelner Algorithmenimplementierungen einfließen können und somit eine Analyse erleichtern.

### 4.1.1 Welche Parameter lassen sich in Xaver variieren?

Die XAVER-Plattform bietet die Möglichkeit, verschiedene systemspezifische Größen zu definieren, um Analysen für Algorithmen auf verschiedenen Architekturen durchzuführen. So sind die folgenden Werte variierbar:

- Größe des internen Speichers [Byte]
- Größe eines Blocks [Byte]

Neben diesen Größen, die als Parameter bei der Initialisierung anzugeben sind, bietet XAVER über die Datei `typedefs.h` die Möglichkeit, programminterne Typen zu definieren. Darüber lässt sich festlegen, welche Datentypen die XAVER-Library für die Indizierung und Verwaltung der einzelnen Blöcke benutzt.

- `BlockNo`: Typ für die Adressierung eines einzelnen Blocks (XAVER-Standard: `unsigned int`)
- `BlockCount`: Typ für die Angabe von Längen von Blocksequenzen (`unsigned int`)
- `BlockSize`: Typ für die Angabe von Blockgrößen (`unsigned int`)
- `Index`: Typ für die Indizierung einzelner Elemente eines externen Arrays (`long long int`)
- `IndexDiff`: Typ für die Beschreibung von Index-Differenzen in externen Arrays (`long long int`)
- `InternalMemoryUnit`: Typ für die Größe einer Einheit der internen Speicherverwaltung - sollte eine Größe von 1 Byte besitzen, da er als Typ für die kleinste adressierbare Einheit verwendet wird. (`char`)
- `InternalMemory`: Typ für Zeiger auf eine Position im internen Speicher - muss ein `internalMemoryUnit`-Pointer sein.
- `Locks`: Typ für den Datenbereich des Sperr-Zählers innerhalb eines Blockes (Durch Benutzung des Typs `unsigned char` können beispielsweise 255 Sperren auf einen Block gleichzeitig durchgeführt werden.)

Aus der Definition dieser Größen ergeben sich im Prinzip automatisch weitere Größen:

- ```
  const BlockNo NullBlockNo = std::numeric_limits<BlockNo>::max();
  const BlockPos NullBlockPos = std::numeric_limits<BlockPos>::max();
  ```
  Diese Werte stellen einen Null-Wert für Block-Nummern und Block-Positionen dar, analog zu dem von C-Pointern bekannten NULL.
- ```
  const Locks MaxLocks = std::numeric_limits<Locks>::max();
  ```
  Gibt die maximale Anzahl an Sperren für einen Block an.

Diese Größen sollten nicht anders definiert werden.

Die Einbindung der verschiedenen Caching-Strategien erfolgt ebenfalls über eine Typendefinition:

- `Caching` $\in$ {`CachingFIFO`, `CachingLRU`, `CachingLFU`, `CachingFIFO2ndChance`}

Genauso kann zwischen den beiden ObjectController-Varianten umgeschaltet werden:

- `ObjectController` $\in$ {`SimpleObjectController`, `QuickObjectController`}

Der `QuickObjectController` benötigt für die Definition seiner Alignierungsgrenzen zusätzlich noch die Typangabe `BitArray` (`unsigned int`)

### 4.1.2 Die Modi aus Sicht des Benutzers

Wie beschrieben, kann XAVER sowohl im Real- als auch im Simulationsmodus betrieben werden. Für den Benutzer wird dieser Unterschied durch die Plattform allerdings so gekapselt, dass die Bedienung in beiden Fällen absolut gleich gehalten wird. Somit funktioniert ein und der selbe Externspeicheralgorithmus sowohl mit der Simulation im RAM, als auch mit einer großen Datenmenge auf der Festplatte.

Der einzige Unterschied tritt an der Stelle in XAVER selbst auf, an der die nicht mehr in den internen Speicher passenden Daten ausgelagert werden müssen.

Abbildung 4.1 fasst die Kommunikation des Benutzers in Form eines Algorithmus mit der XAVER-Library zusammen. Auf die Details der einzelnen Verbindungen wird in den folgenden Kapiteln detailliert eingegangen.

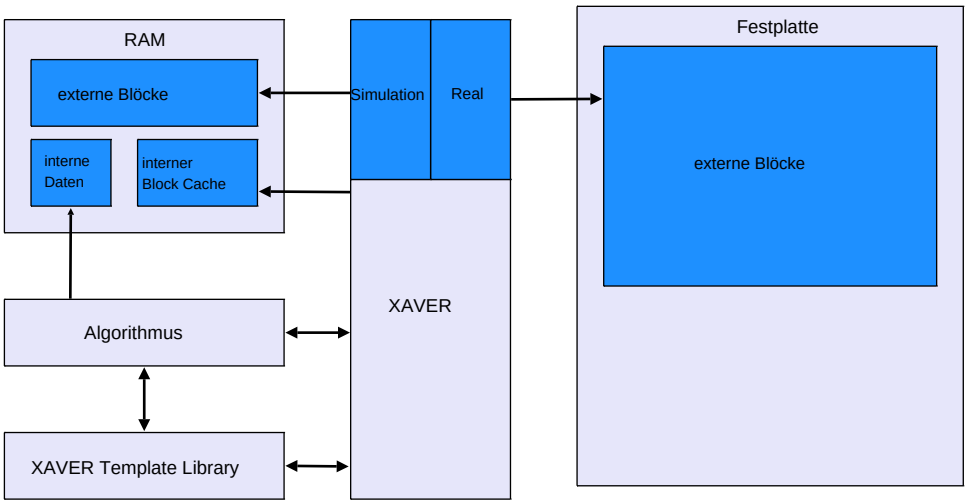


Abbildung 4.1: Die Kommunikation mit XAVER

### 4.1.3 Welche Funktionen bietet Xaver dem Benutzer

Da XAVER als eigenständige Externspeicherverwaltung fungiert, braucht der Programmierer sich selbst keine Gedanken mehr darum zu machen, wo die Daten auf der Festplatte abgelegt sind, wann sie wieder neu eingeladen werden und wie sie im RAM angeordnet werden.

Nach außen bietet XAVER im Groben die folgenden Methoden:

- **Anfordern einer Sequenz von Blöcken**
  Dieser Aufruf registriert im Prinzip eine bestimmte Anzahl von Blöcken in der Speicherverwaltung und gibt einen speziellen Zeiger auf den Beginn der Sequenz zurück. Der Programmierer kann mit diesem Zeiger genau so arbeiten, wie mit einem normalen C-Pointer. Wann und wie zwischenzeitlich diese Blöcke vom internen in den externen Speicherbereich wandern, bleibt für ihn verborgen, sofern er sich nicht explizit darum kümmert.

- **Freigeben einer Blocksequenz**

- **Priorisierung von Blöcken**
  Grundsätzlich haben alle von XAVER verwalteten Blöcke einen gleichen Rang. Sie werden also, wenn neuer interner Speicher benötigt wird, nach der von der Caching-Strategie angegebenen Reihenfolge aus dem internen Speicher verdrängt. Hier bietet XAVER allerdings die Möglichkeit, auf einen einzelnen Block ein explizites `lock` aufzurufen. Dieser Block wird dann definitiv solange nicht mehr aus dem internen Speicher verdrängt, bis die Sperre wieder aufgehoben wird.

- **Einzelne Objekte extern erzeugen**
  Neben der Möglichkeit, komplette Blöcke oder Blocksequenzen anzufordern, können einzelne Objekte über XAVER extern erzeugt werden. Über den zurückgegebenen FarPointer können die Objekte verwendet werden, als wären sie über den `new()`-Operator auf dem Heap angelegt worden.

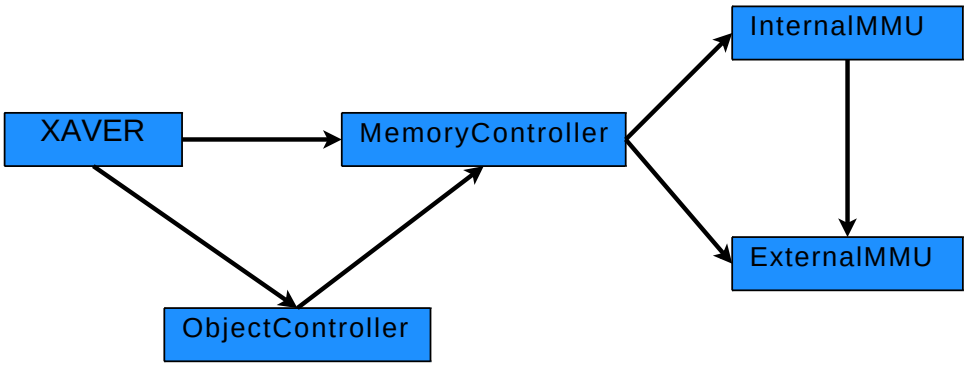


Abbildung 4.2: Innenleben von XAVER

### 4.1.4 Die Xaver Template Library

Zusammen mit XAVER wurde eine Bibliothek erstellt, die eine Auswahl von wichtigen Datenstrukturen und Algorithmen auf XAVER optimiert bereitstellt. So stehen analog zur STL beispielsweise Listen, Bäume, Vektoren, usw. zur Verfügung, die selbst auf XAVER aufsetzen. Ebenso ist ein externer MERGESORT-Algorithmus implementiert, der auf großen externen Datenmengen effizient arbeitet.

## 4.2 Das Innenleben von Xaver

In Abbildung 4.2 ist das Innenleben von XAVER beschrieben. Neben der Schnittstelle von XAVER, die nach außen sichtbar ist und die dem Programmierer die Möglichkeit bietet, Speicher anzufordern, gibt es intern vier wesentliche weitere Bereiche:

- Die **externe Memory Management Unit** ist dafür verantwortlich, die extern gespeicherten Blöcke zu verwalten und für den Transfer von Blöcken vom internen in den externen Speicher und vice versa zu sorgen.
- Die **interne Memory Management Unit** steuert die Verwaltung der Daten im RAM. Hier werden die verschiedenen Blockersetzungsstrategien eingebunden, wobei die Möglichkeit berücksichtigt wird, dass einige Blöcke explizit nicht ausgelagert werden sollen.
- Der **Memory-Controller** regelt das Zusammenspiel zwischen interner und externer MMU und leitet XAVER-Befehle an die entsprechende Verwaltungseinheit weiter.
- Der **Object-Controller** bietet die Möglichkeit, externen Speicherplatz sowohl für Arrays als auch für einzelne Objekte anzufordern. Der Object-Controller fordert dafür Blöcke bei XAVER an und versucht, die an ihn gestellten Anfragen nach Speicher möglichst geschickt auf diese Blöcke zu verteilen.

## 4.3 Das Pointer-Konzept von Xaver

Um es Algorithmen zu ermöglichen, den Externspeicher so transparent wie möglich zu nutzen, gibt es in XAVER Pointer, die in den Externspeicher zeigen und die neben der Funktionalität von normalen C-Pointern noch weitere externspeicherspezifische Funktionen bieten. Es gibt in XAVER neben den C-Pointern drei weitere Arten von Pointern:

- `FarPointer` zeigen auf eine beliebige Position im externen Speicher.
- `NearPointer` zeigen auf eine Position, die in dem Block liegt, in dem auch der entsprechende `NearPointer` liegt.
- `BlockPointer` zeigen auf einen ganzen Block.

### 4.3.1 Implementierung als Templates

Die `FarPointer` und `NearPointer` sind als Templates implementiert. Sie besitzen einen Template-Parameter T, der den Typ des Objekts spezifiziert, auf den der Pointer zeigt. Dies ermöglicht dem Compiler die übliche Typüberprüfung. Außerdem ist es für die Pointerarithmetik wichtig, zu wissen, wie groß die referenzierten Objekte sind. `BlockPointer` hingegen sind immer ungetypt, da sie immer auf den Block an sich zeigen.

### 4.3.2 Dereferenzierung

Die wichtigsten Operatoren von Pointern sind die Dereferenzierung (`*`) und der Pfeil-Operator (`->`). Der Pfeil-Operator muss einen C-Pointer auf das referenzierte Objekt zurückliefern.

Beim `FarPointer` und beim `BlockPointer` wird dieser C-Pointer durch eine Anfrage an den Memory-Controller bestimmt. Der Memory-Controller stellt dabei sicher, dass sich der Block, in dem sich das referenzierte Objekt befindet, im internen Speicher befindet. Danach rechnet er Blocknummer und die Position im Block in einen C-Pointer um. Ein `BlockPointer` liefert dabei immer einen C-Pointer auf die Basisadresse des Blocks im internen Speicher zurück.

Der `NearPointer` benutzt seine eigene Adresse, um diesen C-Pointer zu bestimmen. Der MemoryController liefert zu der Adresse des `NearPointer` die Basisadresse des Blocks, in dem der `NearPointer` liegt. Dazu muss der `NearPointer` dann nur noch die Position im Block addieren.

### 4.3.3 Konstante Zeiger

`FarPointer`, `NearPointer` und `BlockPointer` können analog zu C-Pointern auch als konstant definiert werden. Mit diesen kann genauso wie mit konstanten C-Pointern gearbeitet werden, d.h. das volle Spektrum der Pointer-Arithmetik steht zur Verfügung. Einzig das Ziel, auf das der Zeiger zeigt, kann nicht verändert werden.

Aus dieser Tatsache kann die Speicherverwaltung von XAVER einen Vorteil ziehen: Wird auf einen Block nur konstant zugegriffen, muss er nicht erneut in den externen Speicher geschrieben werden, wenn er aus dem internen Speicher verdrängt wird. Damit lassen sich durch konstante Zugriffe IO-Operationen sparen.

Da XAVER keine Möglichkeit hat, bei einer nicht-konstanten Dereferenzierung herauszufinden, ob diese nun schreibend oder lesend war, wird bei nicht-konstanten Dereferenzierungen immer davon ausgegangen, dass der Speicherinhalt verändert wurde. Es empfielt sich deshalb, für lesende Zugriffe immer explizit konstante Zeiger zu verwenden, da somit IO-Operationen gespart werden können.

### 4.3.4 Random-Access-Iteratoren

Alle drei Pointertypen unterstützen den vollen Funktionsumfang eines Random-Access-Iterators. Dieser besteht aus den folgenden Funktionen:

- Zuweisungen (`=`)

- Dereferenzierungen (`*`, `->`)
- Array-Zugriffe (`[]`)
- Kombinierte Zuweisungen (`+=`, `-=`)
- Addition und Subtraktion mit einem Indexwert (`+`, `-`)
- Differenzbildung von zwei Zeigern (`-`)
- Pre- und Postinkrement und -dekrement (`++`, `--`)
- Vergleichsoperatoren (`==`, `!=`, `<=`, `>=`, `<`, `>`)

Außerdem ist es möglich, die drei Pointertypen ineinander zu casten sowie den Typ des `FarPointer` bzw. `NearPointer` in einen anderen Typ umzuwandeln.

### 4.3.5 Weitere Operationen

Neben den Operatoren der Random-Access-Iteratoren stellen die Pointer weitere Funktionalität zur Verfügung. Diese sind im einzelnen:

#### Konstruktoren

- `inline FarPointer()`
  `inline NearPointer()`
  `inline BlockPointer()`
  Initialisiert den entsprechenden Pointer zu einem Null-Pointer.
- `inline FarPointer(BlockNo blockNo, BlockPos posInBlock)`
  `inline NearPointer(BlockPos posInBlock)`
  `inline BlockPointer(BlockNo blockNo)`
  Initialisiert den entsprechenden Pointer, so dass er auf die angegebene Position zeigt.
- `inline FarPointer(const FarPointer<T>& source)`
  `inline NearPointer(const NearPointer<T>& source)`
  `inline BlockPointer(const BlockPointer& source)`
  Initialisiert den entsprechenden Pointer, so dass er auf die gleiche Position wie der Pointer `source` zeigt.

#### Funktionen zur Externspeicherkontrolle

Alle Pointer besitzen Funktionen, um den Status des Blocks, auf den der Pointer zeigt, zu kontrollieren und auszulesen. Diese sind im einzelnen:

- `inline bool isBlockNew() const`
- `inline bool isBlockDirty() const`
- `inline bool isBlockLoading() const`
- `inline bool isBlockInternal() const`
- `inline bool isBlockLocked() const`
- `inline void flushBlock() const`
- `inline void discardBlock() const`

- `inline void lockBlock() const`
- `inline void unlockBlock() const`
- `inline void preloadBlock() const`
- `inline void unloadBlock() const`

All diese Funktionen werden direkt an den `MemoryController` weiter geleitet, der diese an die `InternalMMU` weiterleitet. Eine Beschreibung der Funktionen findet sich deshalb im Kapitel 4.4.

#### Weitere Funktionen

Darüber hinaus bieten alle Pointer die folgenden Funktionen an:

- `inline bool isNull() const`
  Liefert zurück, ob der Zeiger ein Null-Zeiger ist.
- `inline void setNull() const`
  Setzt den Zeiger auf Null.
- `inline BlockNo getBlockNo() const`
  Liefert die Block-Nummer des Blocks zurück, auf den der Zeiger zeigt (dies entspricht bei einem `NearPointer` dem Block, in dem der `NearPointer` liegt).
- `inline BlockPos getPosInBlock() const`
  Liefert die Position innerhalb des Blockes, auf die der Zeiger zeigt, zurück. Da `BlockPointer` immer auf ganze Blöcke zeigen, wird bei diesen hier immer der Wert `NullBlockPos` zurück geliefert.

### 4.3.6 FarPointer

`FarPointer` sind die wichtigsten Pointer in XAVER. Sie enthalten die Blocknummer und die Position innerhalb des Blocks als Attribute und adressieren damit eindeutig eine Position im externen Speicher.

#### Fallstricke

Bedingt durch die Verwendung des externen Speichers und die Art der Auswertung von Ausdrücken in C++ ist Folgendes bei der Verwendung von `FarPointer`n zu beachten. Betrachten wir dazu das folgende, auf den ersten Blick korrekt aussehende Stück Programm-Code:

```
FarPointer<int> x = Xaver::newExternalArray<int>(1);
FarPointer<int> y = Xaver::newExternalArray<int>(1);
*x = 42;
*y = *x;
```

Das Problem liegt hier in der vierten Zeile: C++ wertet zunächst die beiden Dereferenzierungen aus und führt dann die Zuweisung aus. Es gibt jedoch keine Gewähr, dass die Auswertung von `*y` nicht den Block, auf den `x` zeigt, aus dem Speicher verdrängt. Für diese Problematik sind folgende Lösungsmöglichkeiten vorhanden:

- Verwendung von maximal einer Dereferenzierung pro Ausdruck, also von `int xVal = *x; *y = xVal;` an Stelle von `*y = *x;`. Dies beseitigt zwar das Problem, führt aber auf Dauer zu recht unschönem Code.

- Verwendung einer Caching-Strategie, mit der das Problem nicht auftritt (z.B. LRU). Diese Lösung basiert natürlich darauf, dass der Code dann doch nicht mit einer anderen Caching-Strategie ausgeführt wird und ist daher nicht zu empfehlen.
- Verwendung von Sperren. Dies ist die zu empfehlende Lösung, da sie das Problem sicher löst. Dazu werden zunächst die Blöcke von `x` und `y` in den internen Speicher gesperrt, bevor die Zuweisung durchgeführt wird:

  ```
  x.lockBlock();
  y.lockBlock();
  *y = *x;
  y.unlockBlock();
  x.unlockBlock();
  ```

  Da ein Algorithmus idealerweise nicht nur auf einzelne Blockpositionen zugreift, sondern über die Blöcke iteriert, ist es ausreichend, die Sperren nur beim Block-Wechsel zu setzen bzw. zu lösen. Um hier das Rad nicht jedes mal neu erfinden zu müssen, wurde diese Funktionalität bereits in den Klassen `BufferedReader` und `BufferedWriter` implementiert.

**Optimierungen**

`FarPointer` sind ein sehr mächtiges Konzept, und auch wenn bereits sehr viel Optimierungen vorgenommen wurden, so ist klar, dass ein `FarPointer` dennoch langsamer ist als ein C-Pointer. Es empfielt sich von daher, `FarPointer` nur dann zu benutzen, wenn es wirklich erforderlich ist und ansonsten nach Möglichkeit nur C-Pointer zu verwenden. Dazu sperrt man zunächst den Block, auf den der `FarPointer` zeigt, im internen Speicher, damit sich die Basisadresse des Blocks im internen Speicher nicht mehr ändert. Mittels einer Dereferenzierung holt man sich nun diese Basisadresse als C-Pointer und arbeitet auf dieser, solange die Operationen sich nur auf diesen Block beziehen. Erst, wenn man auf den nächsten Block zugreifen möchte, löst man die Sperre, setzt den `FarPointer` auf den nächsten Block und beginnt für diesen das Prozedere von vorn. Das folgende Beispiel illustriert dies:

```
BlockSize objectsPerBlock = Xaver::getBlockSize() / sizeof(int);
FarPointer<int> x = Xaver::newExternalArray<int>(8 * objectsPerBlock);
for (int i = 0; i < 8; i++) {
    x.lockBlock()
    int* xCurrent = &*x;
    for (int j = 0; j < objectsPerBlock; j++) {
        xCurrent[j] = i * objectsPerBlock + j;
    }
    x.unlockBlock();
    x += objectsPerBlock;
}
```

### 4.3.7 NearPointer

`NearPointer` bieten gegenüber `FarPointern` den Vorteil, dass sie nur eine Position innerhalb eines Blocks, nicht jedoch die Nummer eines Blocks speichern, so dass sie in aller Regel nur die Hälfte des Speicherplatzes eines `FarPointer` belegen. Sie sind konzipiert worden, um Verweise innerhalb von Blöcken speicherplatzeffizient realisieren zu können.

Bedingt dadurch, dass die Nummer des Blocks nicht explizit gespeichert ist, muss bei der Dereferenzierung die Basisadresse des Blocks aus der Adresse des `NearPointer` berechnet werden. Diese Aufgabe übernimmt der MemoryController, der dies weiter an die InternalMMU deligiert.

Da das Ziel, auf das der `NearPointer` zeigt, immer innerhalb des Blocks liegt, in dem auch der `NearPointer` liegt, ergeben sich einige Dinge, die man bei der Verwendung von `NearPointern` beachten sollte:

- Kopiert man einen `NearPointer` in einen anderen Block, zeigt er auf die gleiche Position wie vorher, aber im neuen Block. Durch eine Kopie kann sich also das Ziel des Zeigers ändern. Innerhalb eines Blocks können `NearPointer` jedoch gefahrlos kopiert werden. Möchte man blockübergreifende Verweise realisieren, sollte man `FarPointer` verwenden.
- Liegt ein `NearPointer` in einem Speicherbereich, der nicht von XAVER verwaltet wird (z.B. der Stack oder Heap von C++), kann die Blocknummer des `NearPointer` nicht ermittelt werden. Es ist deshalb nicht möglich, den `NearPointer` dann zu dereferenzieren oder Operationen aufzurufen, die sich auf den Block beziehen, in dem der `NearPointer` liegt. Soll die Blockinformation behalten werden, muss der `NearPointer` zunächst in einen `FarPointer` gecastet werden.

### 4.3.8 BlockPointer

Die `BlockPointer` sind für Algorithmen interessant, die den inneren Aufbau eines Blocks kennen und nur die Referenz auf einen Block brauchen. Sie sind damit Speicherplatzeffizienter als `FarPointer`, da die Position innerhalb des Blocks nicht gespeichert werden muss. Bei der Verwendung von Pointerarithmetik ist zu beachten, dass sich diese beim `BlockPointer` auf ganze Blöcke und nicht wie beim `FarPointer` und `NearPointer` auf einzelne Positionen innerhalb der Blöcke bezieht. Ähnliches gilt für die Dereferenzierung: Diese liefert stets die Basisadresse des Blocks im internen Speicher.

## 4.4 Die interne Memory-Management-Unit

Die interne Memory-Management-Unit dient zur Verwaltung des internen Speichers. Sie besteht aus drei Klassen: Die Klasse `InternalMMU` stellt das Interface nach außen bereit. Der `BlockTabelle` verwaltet die Informationen über die Zustände der einzelnen Blöcke. Er verwendet dabei als Datenklasse die Klasse `Block`, in der jeweils der Zustand eines einzelnen Blocks verwaltet wird. Außerdem verwendet die `InternalMMU` eine der Caching-Klassen, die im Abschnitt 4.4.4 beschrieben sind, um zu entscheiden, nach welchen Strategien Blöcke zu ersetzen sind, wenn der interne Speicher komplett belegt ist und ein neuer Block hinein geladen werden soll.

### 4.4.1 Die Klasse InternalMMU

Die Aufgabe der `InternalMMU` besteht darin, den internen Speicher zu verwalten. Die wesentlichen Aufgaben im Überblick:

- Auflösen von externen in interne Adressen und zurück, das heißt also umwandeln einer Block-Nummer und einer Block-Position in einen C-Pointer beziehungsweise umgekehrt (letzteres erfordert natürlich, dass der C-Pointer auf eine Stelle im internen Speicher zeigt, an der wirklich ein Block liegt). Bei der Auflösung von externen in interne Adressen wird der Block gegebenenfalls in den internen Speicher geladen, sofern er dort nicht vorhanden ist.

- Bereitstellung von Funktionen, um Blöcke explizit in den internen Speicher zu laden oder aus dem internen Speicher zu entfernen, sowie um Blöcke zu verwerfen.
- Bereitstellung von Funktionen, um Blöcke in den internen Speicher zu sperren beziehungsweise deren Sperre wieder aufzuheben.
- Bereitstellung von Funktionen, um Statusinformationen über Blöcke abzufragen; z.B., ob ein Block im internen Speicher vorhanden ist.

Einige der Funktionen werden durch die `InternalMMU` direkt realisiert, andere werden an die oben genannten Klassen delegiert.

### Die Schnittstelle

Die `InternalMMU` stellt dem Nutzer folgende Operationen zur Verfügung:

- `static void init(BlockSize blockSize, BlockCount internalMemorySize, ExternalMMU* externalMMU)`
  Initialisiert die `InternalMMU` mit den entsprechenden Werten für die Blockgröße, interne Speichergröße und der entsprechenden `ExternalMMU`. Diese Funktion ist aufzurufen, bevor die `InternalMMU` benutzt werden kann.
- `static void destroy()`
  Gibt die internen Datenstrukturen der `InternalMMU` wieder frei.
- `static inline bool isBlockNew(BlockNo blockNo)`
  `static inline bool isBlockDirty(BlockNo blockNo)`
  `static inline bool isBlockLoading(BlockNo blockNo)`
  `static inline bool isBlockInternal(BlockNo blockNo)`
  `static inline bool isBlockLocked(BlockNo blockNo)`
  Diese Funktionen liefern die entsprechende Statusinformation über den angegebenen Block. Sie werden an den `BlockTabelle` deligiert, der diese dann an das zum Block gehörige Blockobjekt weiter leitet. Eine genaue Beschreibung der Funktionen findet sich deshalb im Abschnitt 4.4.3.
- `static void flushBlock(BlockNo blockNo)`
  Schreibt den angegebenen Block synchron in den externen Speicher. Der Block verbleibt auch im internen Speicher, wird dort jedoch als nicht verändert markiert, so dass er, wenn er aus dem internen Speicher verdrängt werden soll, nicht mehr in den externen Speicher geschrieben werden muss.
- `static inline void discardBlock(BlockNo blockNo)`
  Verwirft den angegebenen Block. Dies bedeutet, dass der Block sich wie ein neu allozierter Block verhält. Beim Zugriff auf diesen Block wird dieser folglich nicht aus dem externen Speicher geladen. Der Inhalt des Blocks ist solange undefiniert, bis ein definierter Inhalt in den Block geschrieben wird.
- `static inline InternalMemory getInternalAddress (BlockNo blockNo, BlockSize blockPos)`
  `static inline const InternalMemory getConstInternalAddress (BlockNo blockNo, BlockSize blockPos)`
  Liefern die Adresse der Position des angegebenen Blocks im internen Speicher. Wenn der angegebene Block sich nicht im internen Speicher befindet, wird der Block zunächst in den internen Speicher geladen. Falls kein freier Platz mehr im internen Speicher vorhanden ist, wird zunächst über die Caching-Strategie ein Block entfernt und dann der Block an die freie Position

geladen. Die Methode `getConstInternalAddress` ist für nur lesenden Zugriff vorgesehen, während `getInternalAddress` sowohl lesenden als auch schreibenden Zugriff ermöglicht. Letztere markiert den Block jedoch in jedem Fall als geändert, so dass er, wenn er aus dem internen Speicher entfernt werden soll, erst in den externen Speicher geschrieben werden muss. Deshalb sollte für lesenden Zugriff möglichst nur die erste Methode verwendet werden.

- `static inline BlockNo getBlockNo`
  `(const InternalMemory internalAddress)`
  `static inline BlockPos getPositionInBlock`
  `(const InternalMemory internalAddress)`
  Diese beiden Methoden stellen das entsprechende Gegenstück zu den beiden vorhergehenden Methoden dar, indem sie für eine Adresse des internen Speichers die dazugehörige Block-Nummer bzw. die Position im Block zurück liefern. Das korrekte Funktionieren dieser Methoden bedingt natürlich, dass die gegebene Adresse auch wirklich in den internen Speicher zeigt und an diese Stelle auch wirklich bereits ein Block geladen wurde.

- `static inline void lockBlock(BlockNo blockNo)`
  `static inline void unlockBlock(BlockNo blockNo)`
  Sperrt den angegebenen Block in den internen Speicher bzw. hebt die Sperre auf. Ein Block, der in den internen Speicher gesperrt ist, wird auf keinen Fall automatisch durch die Caching-Strategie aus dem internen Speicher entfernt. Wird `lockBlock` für einen Block aufgerufen, der nicht im internen Speicher ist, wird dieser zunächst in den internen Speicher geladen und dann gesperrt. Bei `unlockBlock` passiert in diesem Fall nichts. Es führt nicht zu Problemen, einen bereits gesperrten Block erneut zu sperren und dann wieder zu entsperren, da die Sperren intern über einen Zähler realisiert wurden.

- `static inline BlockCount getNumberOfLockedBlocks()`
  Liefert die Anzahl der derzeit in den internen Speicher gesperrte Blöcke zurück.

- `static inline void preloadBlock(BlockNo blockNo)`
  Lädt den angegebenen Block asynchron in den internen Speicher. Gegebenenfalls wird zunächst ein Block aus dem internen Speicher verdrängt, falls kein Platz mehr frei ist. Falls auf den Block zugegriffen wird, bevor der Ladevorgang abgeschlossen ist, wird zunächst abgewartet, bis der Ladevorgang beendet ist.

- `static void unloadBlock(BlockNo blockNo)`
  Markiert den Block als „demnächst günstig zu entfernen“. In der Regel wird die Caching-Strategie den Block dann als nächstes aus dem internen Speicher entfernen, wenn ein Block aus dem internen Speicher entfernt werden muss. Grundsätzlich besteht jedoch keine Garantie, dass der Block überhaupt irgendwann aus dem internen Speicher entfernt wird; vielmehr ist dies nur ein Hinweis, dass ein Entfernen des Blocks nicht schlimm ist.

- `static inline void addBlockSequence(BlockNo firstBlockNo,`
  `BlockCount length)`
  Teilt der `InternalMMU` mit, dass die Blöcke der angegebenen Blocksequenz angelegt wurden und nun zu verwalten sind, so dass die `InternalMMU` den Zustand der Blöcke initialisieren kann.

- `static inline void freeBlockSequence(BlockNo firstBlockNo,`
  `BlockCount length)`
  Teilt der `InternalMMU` mit, dass die Blöcke der angegebenen Blocksequenz

freigegeben wurden. Die `InternalMMU` wird Blöcke dieser Sequenz, die sich noch im internen Speicher befinden, freigeben, so dass diese wieder verwendet werden können.

### 4.4.2 Die Block-Tabelle

Um die Verwaltung der Zustände der Blöcke flexibel zu gestalten, verwendet die `InternalMMU` die Block-Tabelle. Die Block-Tabelle verwaltet die Zustände aller Blöcke und bietet entsprechende Funktionen zum Auslesen und Manipulieren dieser. Sie verwaltet alle Blöcke in einem einfachen Array, so dass sichergestellt ist, dass alle Operationen auf Blöcken in Zeit $\mathcal{O}(1)$ ablaufen. Da beim Vergrößern des Arrays die übliche Verdopplungsstrategie verwendet wird, läuft dies in amortisiert konstanter Zeit.

Während die Zugriffe auf diese Art sehr schnell sind, ist der wesentliche Nachteil der sehr hohe Speicherplatzbedarf des Arrays, der $\mathcal{O}(N/B)$ ist und damit vor allem linear in der Anzahl aller Blöcke ist. Da jedoch ein Block in der Regel nur 8 Byte für die Zustandsinformationen benötigt, ist dies durchaus noch vertretbar.

#### Die Schnittstelle der Block-Tabelle

Die meisten Methoden der Block-Tabelle leiten die entsprechenden Aufrufe einfach nur an das entsprechende Block-Objekt weiter. Auf diese Methoden wird hier nicht weiter eingegangen, da sich deren Beschreibung im Abschnitt 4.4.3 findet. Im folgenden nun die Methoden, die nicht an die Block-Objekte weiter deligiert werden:

- `static void init()`
  Initialisiert die Block-Tabelle, so dass sie benutzt werden kann.
- `static void destroy()`
  Gibt die internen Datenstrukturen der Block-Tabelle wieder frei.
- `static inline void addBlockSequence(BlockNo firstBlockNo, BlockCount length)`
  Teilt der Block-Tabelle mit, dass die Blöcke der angegebenen Blocksequenz angelegt wurden und nun zu verwalten sind, so dass die Block-Tabelle das Array auf die entsprechende Größe anpassen kann.
- `static inline void freeBlockSequence(BlockNo firstBlockNo, BlockCount length)`
  Teilt der Block-Tabelle mit, dass die Blöcke der angegebenen Blocksequenz freigegeben wurden.

### 4.4.3 Der Block

Das Block-Objekt verwaltet den Zustand eines Blocks. Die Charakteristika der Zustände eines Blocks sind:

- Intern/Extern: Falls ein Block im internen Speicher vorhanden ist, gilt er als intern, andernfalls als extern. Interne Blöcke kennen die Adresse, an der sie im internen Speicher stehen.
- Geändert: Blöcke im internen Speicher, die möglicherweise geändert wurden, werden entsprechend markiert, so dass nur möglicherweise geänderte Blöcke in den externen Speicher geschrieben werden, wenn sie aus dem internen Speicher verdrängt werden.

- Neu: Ein Block, der neu ist, hat noch keinen definierten Inhalt im externen Speicher. Wird auf einen neuen Block zugegriffen, wird er deshalb nicht in den internen Speicher geladen, sondern es wird nur markiert, dass er im internen Speicher liegt.
- Ladend: Der Block wird gerade vom externen in den internen Speicher geladen.
- Gesperrt: Ein interner Block, der gesperrt ist, darf nicht durch die Caching-Strategie verdrängt werden.

**Die Schnittstelle des Block-Objekts**

Um den Zustand des Block-Objekts auszulesen und zu manipulieren, werden folgende Methoden zur Verfügung gestellt:

- `inline bool isNew()`
  Liefert zurück, ob der Block neu ist. Ein Block ist neu, wenn er neu alloziert oder mittels `discard()` zurückgesetzt wurde. Der Inhalt eines neuen Blocks ist undefiniert, bis ein definierter Inhalt in den Block geschrieben wurde. Wird auf einen neuen Block zugegriffen, wird er nicht aus dem externen Speicher in den internen Speicher geladen, da es eben noch keinen definierten Inhalt für den Block gibt.
- `inline bool isDirty()`
  Liefert zurück, ob auf den Block bereits schreibend zugegriffen worden sein könnte. Blöcke, auf die möglicherweise schreibend zugegriffen wurde, müssen in den externen Speicher geschrieben werden, wenn sie aus dem internen Speicher verdrängt werden.
- `inline bool isLocked()`
  Liefert zurück, ob der Block derzeit in den internen Speicher gesperrt ist, d.h., ob der Zähler für die Anzahl der Sperren größer als 0 ist.
- `inline bool isInternal()`
  Liefert zurück, ob der Block sich derzeit fertig geladen im internen Speicher befindet.
- `inline bool isLoading()`
  Liefert zurück, ob für den Block derzeit ein Ladevorgang läuft.
- `inline void setLoading(InternalMemory address)`
  Markiert den Block als „wird gerade in den internen Speicher geladen“. Setzt außerdem die Adresse im internen Speicher, an die der Block geladen wird.
- `inline void setLoaded()`
  Markiert, dass der Ladevorgang für den Block abgeschlossen ist.
- `inline void setWriting()`
  Markiert den Block als „wird gerade in den externen Speicher geschrieben“. Falls der Block als möglicherweise geändert markiert ist, wird das entsprechende Flag zurückgesetzt. Außerdem wird dann der Block als nicht mehr neu markiert, da sein Inhalt dann ja im externen Speicher vorhanden ist.
- `inline void setWritten()`
  Markiert, dass der Schreibvorgang für den Block abgeschlossen ist.

- `inline void flush()`
  Teilt dem Block mit, dass er synchron in den externen Speicher geschrieben wurde, so dass der Inhalt im internen Speicher nun keine Änderungen mehr gegenüber dem Inhalt im externen Speicher enthält.
- `inline void discard()`
  Teilt dem Block mit, dass er verworfen wurde. Dies hat zur Auswirkung, dass er als neu angelegter Block markiert wird, der noch keinen Inhalt im externen Speicher hat und dessen Inhalt im internen Speicher noch nicht modifiziert wurde.
- `inline InternalMemory getInternalAddress()`
  `inline InternalMemory getConstInternalAddress()`
  Liefert die Adresse des Blocks im internen Speicher zurück oder NULL, falls der Block nicht im internen Speicher vorhanden ist. Bei der ersten Methode wird der Block außerdem als „möglicherweise geändert“ markiert.
- `inline void lock()`
  `inline void unlock()`
  Erhöht bzw. senkt den Zähler für die Anzahl der Sperren um 1.

### 4.4.4 Caching-Strategien

**Anforderungen**

Wir haben für Xaver die vier gängigsten Caching-Strategien realisiert: LRU, LFU, FIFO, FIFO-Second-Chance. Alle Implementierungen realisieren ein spezifisches Interface und unterstützen die folgenden Operationen:

- `static void init(BlockCount numberOfBlocks)`
  Initialisiert die Caching-Strategie für die angegebene Anzahl von Blöcken.
- `static inline void blockLoaded(BlockNo blockNumber)`
  Vermerkt, dass der übergebene Block gerade geladen wurde.
- `static inline void blockAccessed(BlockNo blockNumber)`
  Vermerkt, dass auf den übergebenen Block zugegriffen wurde.
- `static inline void blockUnloaded(BlockNo blockNumber)`
  Vermerkt, dass der übergebene Block zum Wegschreiben markiert wurde.
- `static inline BlockNo selectBlockToFree()`
  Gibt den Index des Blocks zurück, der laut Strategie als nächstes aus dem Speicher verdrängt werden soll.
- `static inline void blockLocked(BlockNo blockNumber)`
  Sperrt den angegeben Block. Gesperrte Blöcke werden bei einer Anfrage nach dem als nächstes zu entfernenden Block nicht zurückgeliefert.
- `static inline void blockUnlocked(BlockNo blockNumber)`
  Hebt die Sperre für den angegebenen Block wieder auf.
- `static inline BlockCount getNumberOfLockedBlocks()`
  Liefert die Anzahl der aktuell gesperrten Blöcke zurück.

Da die Caching-Strategien über jede einzelne Derefenzierung informiert werden müssen, haben wir insbesondere für `blockAccessed` eine konstante Laufzeit eingefordert.

Das Einbinden der einzelnen Strategien erfolgt über eine Typendefinition in `typedefs.h` und erfordert somit, die Xaver-Bibliothek neu zu erstellen.

**Einblick in die Realisierung**

Die Caching-Strategien LRU, FIFO und FIFO-Second-Chance verwenden intern ein Array, das eine doppelt-verkettete Liste repräsentiert, um so die konstante Laufzeit zu gewährleisten. Der vorderste Block des Arrays fungiert hier in der Sonderrolle des Dummies, und zeigt somit auf den ersten beziehungsweise letzten Block der Liste. Abbildung 4.3 verdeutlicht dies. Die einzelnen Strategien unterscheiden sich nun lediglich in den Ereignissen, die Änderungen an der Reihenfolge hervorrufen. So wird beispielsweise im LRU-Caching bei jedem Blockzugriff der entsprechende Block ans Ende der Liste befördert, wohingegen dies bei FIFO nur bei jedem Ladevorgang geschieht. Die Methode `selectBlockToFree` gibt nun in allen Varianten jeweils den vordersten Block der Liste zurück. Das Locking bewirkt im Prinzip das Entfernen des entsprechenden Blockes aus der verketteten Liste.

Da jede Operation durch das Umhängen von 8 Zeigern realisiert werden kann, wird die geforderte Laufzeit von $\mathcal{O}(1)$ erreicht.

LFU speichert in einem Array zu jedem Block im internen Speicher die Anzahl der Zugriffe und den Zustand. Bei `blockLoaded` wird der Zugriffszähler des geladenen Blockes auf eins gesetzt und der aller anderen halbiert. Bei `selectBlockToFree` wird ein Block mit minimalem Zugriffszähler gewählt. Deshalb ist die Laufzeit dieser beiden Methoden nicht durch eine Konstante sondern durch die Anzahl der Blöcke im internen Speicher nach oben beschränkt.

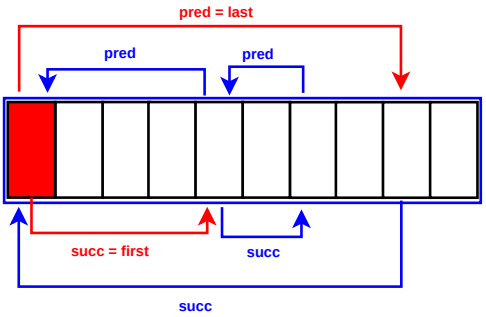


Abbildung 4.3: Prinzip der Caching-Strategien

Inwiefern sich ein Wechsel der Caching-Strategie auf den Programmfluss auswirkt, wird in Kapitel 7.6 an einigen Beispielen verdeutlicht.

## 4.5 Die externe Memory-Management-Unit

Die Verwaltung des Externspeichers unterteilt sich in drei verschiedene Klassen. Die Klasse `ExternalMMU` dient als Schnittstelle nach außen. Die `IO`-Klasse ist für den eigentlichen Datentransfer zwischen dem externen Speicher und den Daten im RAM verantwortlich. Die Klasse `ExternalWorkerThread` regelt die zeitliche Abfolge der Aufrufe an die `IO`-Klasse, so dass das Laden und Schreiben von Dateien asynchron vom sonstigen Programmfluss geschehen kann.

Abbildung 4.4 veranschaulicht die Zuständigkeiten innerhalb der externen Speicherverwaltung.

### 4.5.1 Die Klasse ExternalMMU

Die Hauptaufgabe der Klasse `ExternalMMU` besteht darin, die vom Programm allozierten Blocksequenzen zu verwalten. So muss bei jeder Lade- bzw. Schreib-Anfrage

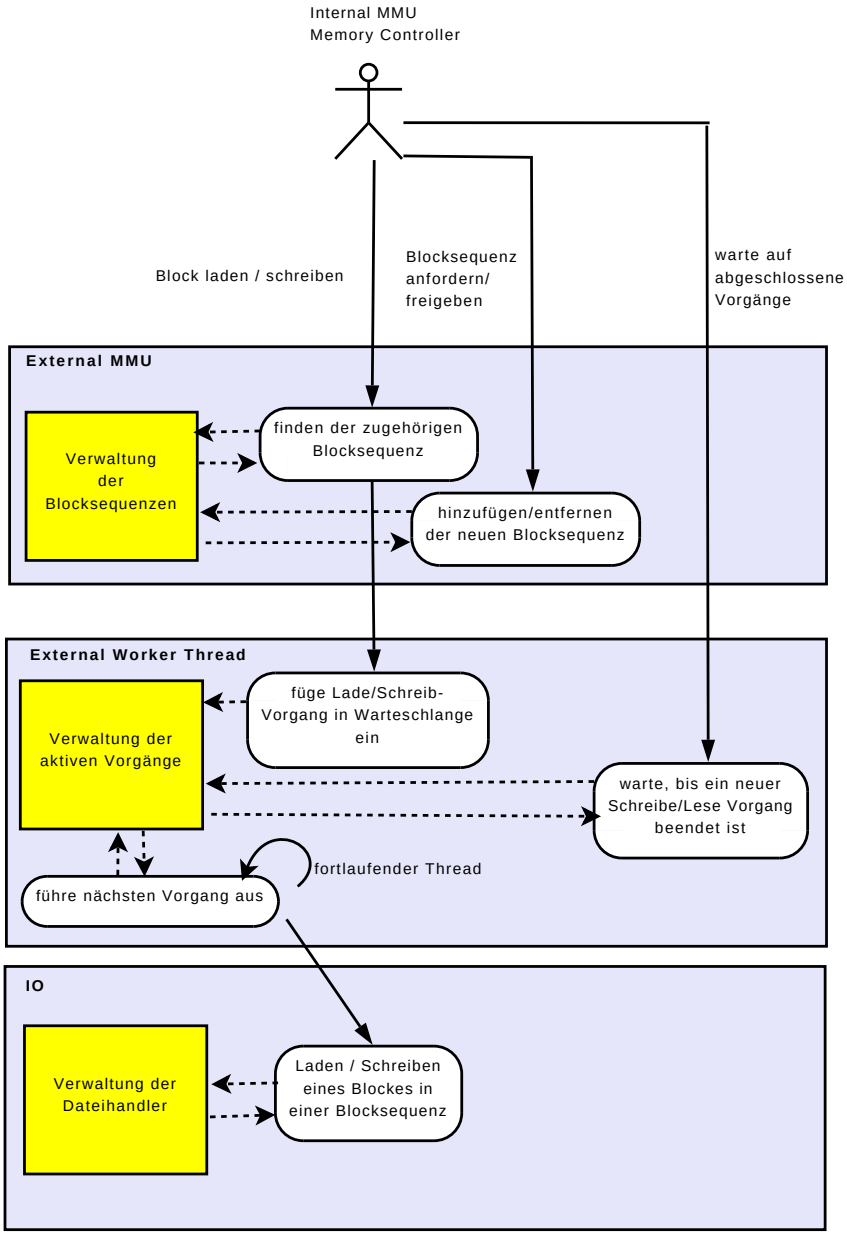


Abbildung 4.4: Die externe Memory-Management-Unit im Überblick

zu einem gegebenen Block die entsprechende Blocksequenz und die Position des Blockes in der Blocksequenz gefunden werden. Hierfür verwenden wir einen Vector, auf dem binär gesucht wird. Da die Verwaltung des internen Speichers durch die `InternalMMU` erfolgt, weiß die `ExternalMMU` nicht, wo ein bestimmter Block, der in den externen Speicher geschrieben werden soll, im internen Speicher liegt bzw. wohin ein bestimmter Block aus dem externen Speicher in den internen geladen werden soll. Deswegen werden bei den jeweiligen Methoden die internen Adressen als Paramter übergeben. Bei dem verwendeten Typ `InternalMemory` handelt es sich um einen Zeiger auf eine Stelle im Arbeitsspeicher. Von der `ExternalMMU` aus wird auch das Protokollieren der IO-Operationen angestoßen. An dieser Stelle werden allerdings konsekutive Vorgänge – also aufeinander folgende Anfragen, die zur gleichen Blocksequenz gehören – gesondert protokolliert, um hinterher einen Anhaltspunkt dafür zu haben, wie häufig zwischen Zugehörigkeitsbereichen von Daten hin und her gesprungen wurde.

### Die Schnittstelle

Die Klasse bietet nach außen folgende Methoden:

- `BlockNo allocateBlockSequence(BlockCount length)`
  Reserviert Speicher für eine Blocksequenz, deren Länge die kleinste Zweierpotenz ist, die gleich oder größer `length` ist, und gibt die eindeutige Nummer des ersten Blocks zurück.

- `BlockNo freeBlockSequence(BlockNo firstBlock)`
  Entfernt die durch ihren ersten Block definierte Blocksequenz.

- `void loadBlockAsync(BlockNo blockNumber, InternalMemory internalAddress)`
  Lädt einen Block in den internen Speicher. Die Methode kehrt sofort nach ihrem Aufruf wieder zurück. Bevor der Aufrufer auf den Block zugreifen möchte, muss er über `waitUntilLoaded()` sicher gehen, dass der Block wirklich komplett geladen wurde.

- `void loadBlockSync(BlockNo blockNumber, InternalMemory internalAddress)`
  Synchrone Variante einen Block zu laden.

- `void writeBlockAsync(BlockNo blockNumber, InternalMemory internalAddress)`
  Sichert den übergebenen Block an der entsprechenden Stelle im externen Speicher. Die Überprüfung, wann der Speichervorgang abgeschlossen ist, muss über `getCompletedWrites()` geschehen.

- `void writeBlockSync(BlockNo blockNumber, InternalMemory internalAddress)`
  Synchrone Variante einen Block zu sichern.

- `void replaceBlock(BlockNo writeBlockNumber, BlockNo loadBlockNumber, InternalMemory internalAddress)`
  Gleichzeitiges Wegschreiben eines alten Blocks mit darauf folgender Ersetzung durch einen anderen.

- `BlockCount getBlockSequenceLength( BlockNo firstBlock)`
  Gibt die Länge der durch ihren ersten Block definierten Blocksequenz zurück.

- `void waitUntilLoaded(BlockNo BlockNumber)`
  Dieser Aufruf blockiert so lange, bis der angegebene Block vollständig vom externen in den internen Speicher geladen wurde.

- `InternalMemory* getCompletedWrites()`
  Liefert die internen Adressen der Blöcke zurück, die bereits vollständig in den externen Speicher zurückgeschrieben wurden. Somit kann der interne Speicher anderweitig weiterverwendet werden.

Die Details darüber, was bei den einzelnen Aufrufen genau abläuft, werden in den folgenden Unterkapiteln erläutert.

### 4.5.2 ExternalWorkerThread

Die Klasse `ExternalWorkerThread` verwaltet eine Queue von Aufträgen und einen Thread, um das asynchrone Laden und Schreiben von Blöcken zu realisieren. Die asynchronen Lade- und Schreibmethoden dieser Klasse erzeugen neue Aufträge, legen diese in der Queue ab und kehren danach sofort zurück. Ein von dieser Klasse verwalteter Thread kümmert sich dann um die Abarbeitung der Aufträge in der Queue. Dieser Thread ruft dann synchrone Methoden der zugrunde liegenden Plattform auf, um die Daten hin und her zu kopieren. Die synchronen Lade- und Schreibmethoden der Klasse `ExternalWorkerThread` fügen den erzeugten Auftrag in die Queue hinter den Auftrag, der gerade bearbeitet wird, ein und warten bis der neu hinzugefügte Auftrag bearbeitet wurde.

Um die Threadfunktionalität zu realisieren wird die POSIX Thread API [9] verwendet. Da dieser Standard von vielen gängigen Unix-arigen Betriebssystemen unterstützt wird und sogar eine Windowsportierung existiert, wird dadurch die Portierung erleichtert. Da es sich dabei um eine C API handelt, wurden die benötigten Funktionen in den Klassen `Thread`, `Mutex` und `Condition` gekapselt, um eine möglichst objektorientierte Programmierung zu ermöglichen. Da bei der Programmierung mit Threads sehr leicht Deadlocks auftreten, wird nur ein `Mutex`-Objekt verwendet und dadurch die Komplexität gegenüber einem Ansatz mit mehreren `Mutex`-Objekten erheblich reduziert. Über diesen einen `Mutex` wird sowohl die Queue als auch die verschiedenen `Condition`-Objekte geschützt. Die `Condition`-Objekte dienen dazu, das Schlafenlegen und Aufwecken zu steuern.

Im Folgenden wird mit `WorkerThread` der Thread bezeichnet, der die Aufträge der Queue abarbeitet und mit `ManagerThread` der Thread, der die Aufträge in die Queue einfügt.

#### Arbeitsweise des `WorkerThread`

Der `WorkerThread` prüft, ob in der Queue noch Aufträge vorhanden sind. Falls ja, bearbeitet er den ersten Auftrag aus der Queue; ansonsten legt er sich schlafen. Wenn er einen asynchronen Ladeauftrag beendet hat, prüft er, ob der `ManagerThread` auf diesen Block wartet und weckt ihn gegebenenfalls auf. Wenn ein asynchroner Schreibauftrag beendet wurde, wird der `ManagerThread` aufgeweckt, falls er auf fertig werdende Schreibaufträge wartet. Wurde ein synchroner Auftrag beendet, wartet der `ManagerThread` in jedem Fall und wird geweckt.

#### Arbeitsweise des `ManagerThread`

Nach dem Einfügen eines Auftrages in die Queue wird der `WorkerThread` aufgeweckt, falls die Queue vorher leer war. Soll der `ManagerThread` auf das Laden eines Blockes warten, so prüft er zunächst, ob der korrespondierende Ladeauftrag noch in der Queue ist. Falls nicht, kehrt er sofort zurück, sonst legt er sich schlafen und kehrt

erst nach dem Aufwachen zurück. Soll der `ManagerThread` die internen Adressen geschriebener Blöcke zurück geben, so prüft er, ob schon Blöcke fertig geschrieben wurden. Falls ja, gibt er sie zurück. Gibt es keine und sind auch keine Schreibaufträge in der Queue, gibt er `NULL` zurück. Sonst legt er sich schlafen und gibt nach dem Aufwachen die Adressen zurück. Nach dem Einfügen eines synchronen Auftrags legt der `ManagerThread` sich in jedem Fall schlafen.

**Die Schnittstelle**

- `ExternalWorkerThread(IO* io)`
  Erzeugt einen neuen `ExternalWorkerThread`, der die `IO`-Klasse `io` verwendet.
- `void loadBlockAsync(ExternalPosition externalPosition, InternalMemory internalAddress)`
  Fügt einen neuen Auftrag, um einen Block vom externen in den internen Speicher zu laden, in die Queue ein und kehrt danach sofort zurück.
- `void loadBlockSync(ExternalPosition externalPosition, InternalMemory internalAddress)`
  Fügt einen neuen Auftrag, um einen Block vom externen in den internen Speicher zu laden, als nächsten zu bearbeitenden Auftrag in die Queue ein und wartet bis dessen Bearbeitung abgeschlossen ist.
- `void writeBlockAsync(ExternalPosition externalPosition, InternalMemory internalAddress)`
  Fügt einen neuen Auftrag, um einen Block vom internen in den externen Speicher zu schreiben, in die Queue ein und kehrt danach sofort zurück.
- `void writeBlockSync(ExternalPosition externalPosition, InternalMemory internalAddress)`
  Fügt einen neuen Auftrag, um einen Block vom internen in den externen Speicher zu schreiben, als nächsten zu bearbeitenden Auftrag in die Queue ein und wartet bis dessen Bearbeitung abgeschlossen ist.
- `void replaceBlock(ExternalPosition writeBlockExternalPosition, ExternalPosition loadBlockExternalPosition, InternalMemory internalAddress)`
  Fügt einen Auftrag, um einen Block aus dem internen Speicher in den externen Speicher zu schreiben und an die gleiche Stelle einen Block aus dem externen Speicher zu laden, in die Queue ein und kehrt danach sofort zurück.
- `void waitUntilLoaded(ExternalPosition externalPosition)`
  Die Funktion kehrt dann zurück, wenn der spezifizierte Block vollständig in den internen Speicher geladen wurde.
- `InternalMemory* getCompletedWrites()`
  Gibt ein `NULL`-terminiertes Array von internen Adressen, die zu Blöcken gehören, die komplett in den externen Speicher geschrieben wurden, zurück. Falls es noch keine fertigen Blöcke gibt, aber noch welche geschrieben werden, so wird gewartet, bis der erste Block fertig ist. Sonst wir `NULL` zurück gegeben.

### 4.5.3 IO

Die abstrakte Klasse `IO` definiert die Schnittstelle der Klassen, die den eigentlichen Datentransfer zwischen internem und externem Speicher durchführen:

- `loadBlock(ExternalPosition externalPosition, InternalMemory internalAddress)`
  Lädt einen Block von der Stelle `externalPosition` im externen Speicher an die Stelle `internalAddress` im internen Speicher.
- `writeBlock(ExternalPosition externalPosition, InternalMemory internalAddress)`
  Schreibt einen Block von der Stelle `internalAddress` im internen Speicher an die Stelle `externalPosition` im externen Speicher.
- `allocateBlockSequence(BlockNo firstBlock, BlockCount len)`
  Alloziert eine Blocksequenz der Länge `len`, deren erster Block die Nummer `firstBlock` hat.
- `freeBlockSequence(BlockNo firstBlock, BlockCount len)`
  Löscht die Blocksequenz, deren erster Block die Blocknummer `firstBlock` hat und deren Länge `len` beträgt.

Durch diese Kapselung kann man zwischen Real- und Simulationsmodus von XAVER einfach wechseln, indem die entsprechende `IO`-Klasse zur Initialisierung verwendet.

### IOFileBased

`IOFileBased` implementiert die IO Schnittstelle und verwendet Dateien als externen Speicher. Für Blocksequenzen, die eine gewisse Mindestlänge haben – standardmäßig drei – wird eine neue Datei angelegt, deren Namen gleich der Nummer des ersten Blockes der Sequenz ist. Alle kürzeren Blocksequenzen kommen in eine gemeinsame Pufferdatei, dabei werden sie stets hinten angefügt. Werden Blocksequenzen in der Pufferdatei wieder freigegeben, so wird der Platz nicht wiederverwendet. Eine `map` speichert zu jeder Anfangsnummer einer Sequenz deren Länge, den Filedeskriptor der zugehörigen Datei und den Offset des Anfangsblocks in der Datei. Letzterer ist für alle Sequenzen, die nicht in der Pufferdatei sind, null. Standardmäßig werden 100 Dateien gleichzeitig offen gehalten. Im Konstruktor kann optional ein anderer Wert übergeben werden. So kann es etwa sinnvoll sein, diesen Wert zu veringern, wenn das Programm, das XAVER verwendet, selbst viele Dateien gleichzeiteig öffnet. Die von `IOFileBased` geöffneten Dateien werden in einer Liste verwaltet.

Soll eine IO-Operation durchgeführt werden, wird zunächst in der `map` der passende Eintrag gesucht. Ist der Filedeskriptor nicht negativ, ist die Datei geöffnet und es kann nach der Berechnung des Offsets in der Datei direkt mit der IO-Operation begonnen werden. Andernfalls muss zuvor die Datei geöffnet werden. Solange die Anzahl der geöffneten Dateien größer gleich dem gewählten Grenzwert ist, oder das Öffnen der Datei wegen zu vieler offener Dateien scheitert, wird die vorderste Datei in der Liste geschlossen und der zugehörige Filediskriptor in der `map` auf -1 gesetzt. Nachdem dem das Öffnen der Datei erfolgreich war, wird der Filediskriptor in der `map` vermerkt und der eigentliche Kopiervorgang kann beginnen.

### IOMemoryBased

Diese Subklasse von `IO` dient dazu, den externen Speicher im internen Speicher zu simulieren und implementiert damit den Simulationsmodus. Die Blocksequenzen werden als Arrays der entsprechenden Größe im internen Speicher abgelegt. Mittels einer `map` wird zu jeder Anfangsnummer einer Blocksequenz die Anfangsadresse des Arrays zur Verwaltung gespeichert. Das Kopieren zwischen internem und externen Speicher geschieht mit der Funktion `memcpy` aus dem Standard C-Header `string.h`.

### 4.5.4 Logging

Die Protokoll-Einheit von XAVER ist ebenfalls in der EMMU angesiedelt, da hier alle Lade- und Speicher-Aufrufe für die Blöcke verwaltet werden.

Die Methoden der Klasse beschränken sich auf fünf wesentliche Vorgänge:

- `void reset()`
  Setzt alle internen Zähler des Loggings zurück.
- `void logLoadEvent(ExternalPosition block)`
  Protokolliert das Einladen des Blockes mit Index `block` in den internen Speicher.
- `void logWriteEvent(ExternalPosition block)`
  Protokolliert das Wegschreiben des Blockes mit Index `block` aus dem internen Speicher.
- `void logAllocateBlockSequenceEvent(CounterType length)`
  Protokolliert, dass eine Blocksequenz mit angegebener Länge alloziert wurde.
- `void logFreeBlockSequenceEvent(CounterType length)`
  Protokolliert, dass eine Blocksequenz der angegebenen Länge freigegeben wurde.

Mit diesen Funktionen kommt der Benutzer von XAVER nicht in Berührung, da sie automatisch im Hintergrund passieren. Über die Schnittstelle, die XAVER nach außen liefert, kann man allerdings zu jeder Zeit den aktuellen Stand des Protokolls erfragen:

- `void Xaver::showLoggingResults(std::ostream *stream)`
  Schreibt die aktuellen Ergebnisse des Protokolls in den angegeben Stream.
- `LoggingResults Xaver::getLoggingResults()`
  Liefert die aktuellen Ergebnisse in einer Struktur zurück.

Die folgenden Zähler werden von der Protokoll-Klasse verwaltet:

- Anzahl geladener Blöcke
  - insgesamt
  - Teilanzahl der konsekutiven Lese-Zugriffe innerhalb einer Blocksequenz
- Anzahl geschriebener Blöcke
  - insgesamt
  - Teilanzahl der konsekutiven Schreib-Anweisungen innerhalb einer Blocksequenz
- Anzahl allozierter Blocksequenzen
- Anzahl allozierter Blöcke
- Anzahl freigegebener Blocksequenzen
- Anzahl freigegebener Blöcke
- Maximale Anzahl gleichzeitig allozierter Blöcke
- gesamte Laufzeit
- Zeit für I/O-Befehle
- Zeit für die restlichen Befehle

## 4.6 Der Object-Controller

Der Objekt-Controller stellt das Verbindungsglied zwischen Standardalgorithmen und XAVER dar. Während die meisten angepassten Externspeicher-Algorithmen direkt auf den Blöcken operieren werden, können vorhandene Algorithmen mittels des ObjectControllers schnell an XAVER angepasst werden.

Dazu stellt der ObjectController Methoden bereit, die das einfache Erzeugen von Objekten im externen und damit von XAVER verwalteten Speicher ermöglichen. Dabei muss das Anwendungsprogramm der Speicherverwaltung keinerlei Beachtung schenken, sondern kann den kompletten von XAVER verwaltbaren Speicher nutzen um ihn mit einzelnen Objekten zu füllen.

### 4.6.1 Die Schnittstelle

Der ObjectController stellt folgende Methoden bereit, die XAVER kapselt, um sie dem Benutzer zur Verfügung zu stellen:

- `size_t getMaxObjectSize()`
  Gibt die maximale Größe eines Objekts in Byte zurück, das über den `ObjectController` angefordert werden kann.
- `FarPointer<T> allocateExternalMemory()`
  Gibt einen `FarPointer` vom Typ T auf einen freien Speicherbereich zurück, der an dem ausreichend Platz für ein Objekt des Typs T ist.
- `FarPointer<> allocateExternalMemory(size_t typesize)`
  Gibt einen typenlosen `FarPointer` auf einen freien Speicherbereich zurück, der mindestens `typesize` Bytes groß ist.
- `FarPointer<T> allocateExternalArray(Index size)`
  Gibt einen `FarPointer` vom Typ T auf den Beginn eines Arrays vom Typ T mit `size` Einträgen zurück.
- `FarPointer<> allocateExternalArray(Index size, size_t typesize)`
  Gibt einen typenlosen `FarPointer` auf den Beginn eines Arrays zurück. Es wird genug Speicherplatz für `size` Zellen der Größe `typesize` reserviert.
- `void freeExternalArray(FarPointer<T> pointer)`
  Gibt den Speicherplatz frei, der durch das Array belegt wurde, auf den der `FarPointer` zeigt.
- `void freeExternalArray(FarPointer<> pointer, size_t typesize)`
  Gibt den Speicherplatz frei, der durch das Array belegt wurde, auf den der `FarPointer` zeigt.
- `void freeExternalMemory(FarPointer<T> pointer)`
  Gibt den Speicherplatz frei, der für ein Objekt des Typs T an der Position, auf die der `FarPointer` zeigt, reserviert wurde.
- `void freeExternalMemory(FarPointer<> pointer, size_t typesize)`
  Gibt den Speicherplatz frei, der für ein Objekt mit der Größe `typesize` an der Position, auf die der `FarPointer` zeigt, reserviert wurde.

XAVER stellt zwei verschiedene Implementierungen des `ObjectController` bereit: Zum einen den `SimpleObjectController`, der unsere erste Implementierung des `ObjectController` war. Die Implementierung wurde dahingehend optimiert, eine möglichst gute Ausnutzung des Speichers zu erreichen, was allerdings dazu

führt, dass das Allozieren und Freigeben von Speicher recht langsam ist. Aus diesem Grund wurde der `QuickObjectController` entworfen, der wesentlich schneller als der `SimpleObjectController` ist. Allerdings kann man Fälle konstruieren, in denen die Speicherausnutzung des `QuickObjectControllers` bis zu einen Faktor 2 schlechter ist als die des `SimpleObjectControllers`.

### 4.6.2 Der SimpleObjectController

Während das Anlegen von Arrays durch das Allozieren einer Blocksequenz durch Xaver erfolgt, muss hierfür lediglich deren Länge berechnet werden. Hingegen erfordert die Verwaltung von einzelnen Objekte mehr Aufwand. Damit der `SimpleObjectController` auch diese Aufgabe erfüllen kann, muss er jederzeit den Zustand der von ihm verwalteten Blöcke kennen, also unterscheiden können, welcher Speicherbereich in den einzelnen Blöcken genutzt wird und welcher verfügbar ist. Wird ein neues Objekt angefordert, muss er mit dieser Information einen möglichst auf die Länge des Objektes passenden Bereich finden, um den Verschnitt gering zu halten. Bei der Zerstörung eines Objektes muss der entsprechende Speicher wieder freigegeben werden.

Da die nötige Information über die einzelnen Fragmente im Speicher und deren Zustände sehr umfangreich ist, muss die Speicherung ebenfalls im Externspeicher erfolgen. Um sowohl den für die Verwaltung benötigten Speicherplatz, als auch die Anzahl der zur Verwaltung benötigten IOs zu minimieren, verwendet der `SimpleObjectController` eine zweiteilige Heuristik, deren Komponenten im folgenden näher erläutert werden.

#### Die Umsetzung

Wird Speicherplatz angefordert, so wird zunächst im weiter unten beschriebenen Suchbaum nach einem Block gesucht, der eine ausreichend große freie Sequenz hat, um die Anfrage zu erfüllen. Dazu hält der Suchbaum die Information der Länge der maximalen freien Sequenz im Block. Existiert kein solcher Block, muss ein neuer Block von Xaver angefordert werden, der dann in die Verwaltung aufgenommen wird.

Durch die Anfrage wird im verwendeten Block eine freie Sequenz entweder komplett oder zumindest teilweise gefüllt. Ändert sich dadurch die Länge der maximalen freien Sequenz, muss der Suchbaum aktualisiert werden.

Diese Aktualisierung fällt auch an, wenn Speicherplatz freigegeben wird, der die maximale freie Sequenz in einem Block ändert.

#### Der Suchbaum

Um die Aktualisierungen möglichst schnell durchführen zu können, wird ein balancierter binärer Baum verwendet, der im internen Speicher angelegt wird. Dessen Knoten halten eine Größenangabe für die maximale freie Sequenz in den zugeordneten Blöcken und eine Liste dieser zugeordneten Blöcke. Der Baum wird nun auf Grundlage der Größenangaben aufgebaut, wodurch in logarithmischer Laufzeit die Liste von Blöcken gefunden werden kann, die eine freie maximale Sequenz haben, die mindestens der angeforderten Größe entspricht. Dabei wächst dieser Baum nicht mit zunehmender Blockanzahl, sondern nur mit zunehmender Anzahl verschiedener maximaler Sequenzen. Es kann also maximal so viele Knoten geben, wie die Größe eines Blocks in Byte.

Dazu verfügt der Baum, repräsentiert über seinen Wurzelknoten, über folgende Methoden. Die Wurzel wird dabei jedes mal mit übergeben, um einen rekursiven Durchlauf durch den Baum zu ermöglichen.

- `SpaceFragment consumeSpaceFragment(BinaryTreeNode* root, BlockSize len)`
  Sucht im Baum den kleinsten Eintrag der mindestens `len` groß ist. Bei Mehrdeutigkeiten wird der erste passende Block gewählt. Die Nummer eines entsprechenden Blockes wird dann zusammen mit der tatsächlichen Größe über das `SpaceFragment` zurückgegeben.

- `void addSpaceFragment(BinaryTreeNode* root, SpaceFragment* fragment)`
  Fügt einen Eintrag in den Baum ein. Die Größe der Sequenz und die Blocknummer wird dabei über das `SpaceFragment` übergeben.

- `BlockCount countFreeSpaceFragments(BinaryTreeNode* root, BlockSize len)`
  Sucht alle Einträge im Baum, die eine freie maximale Sequenz größer `len` haben und gibt ihre Anzahl zurück.

- `void updateSpaceFragment(BinaryTreeNode* root, SpaceFragment* oldFragment, SpaceFragment* newFragment)`
  Ersetzt den Eintrag, der von der Größe und der Blocknummer `oldFragment` entspricht durch einen korrekt einsortierten Eintrag, der `newFragment` entspricht. Dazu wird implizit `removeSpaceFragment` und `addSpaceFragment` aufgerufen.

- `void removeSpaceFragment(BinaryTreeNode* root, SpaceFragment* fragment)`
  Sucht genau den durch `fragment` beschriebenen Eintrag im Baum und entfernt ihn. Im Gegensatz zu `consumeSpaceFragment` wird hier der exakte Eintrag gesucht und nicht der erste passende gewählt. Dabei ist natürlich vorausgesetzt, dass ein exakt zu `fragment` passender Eintrag existiert.

### Die externen Blöcke

Da der Baum aus Speicherplatz-Gründen lediglich eine Information über den Zustand der Blöcke halten kann, ist es nötig, dass die Blöcke selbst die Information über die in ihnen enthaltenen freien Bereiche speichern. Dazu wird in jedem Block ein `NearPointer` an die niedrigste Adresse geschrieben. Dieser zeigt auf den ersten freien Speicherplatz im Block. Jeder freie Speicher beginnt mit einem `SpacePointer`, der beschreibt wie lang diese freie Sequenz ist und auf den `SpacePointer` der nächsten freien Sequenz zeigt. Dabei entspricht der `SpacePointer` prinzipiell einem `NearPointer` ohne Template-Eigenschaften, stattdessen verweist er immer nur auf einen weiteren `SpacePointer`. Zusätzlich verfügt er über ein Attribut, das die Länge des freien Speichers angibt. Diese Anordnung ist schematisch in Abbildung 4.5 dargestellt.

Indem auf diese Weise der nicht für Daten genutzte Speicherbereich zum Speichern der Verwaltungsinformationen genutzt wird, werden keinerlei zusätzlichen Blöcke benötigt. So muss beim Löschen oder Anlegen eines Objektes auch kein neuer Block nachgeladen werden, da die Änderungen in der Verwaltungsstruktur lokal im selben Block erfolgen. Da jede freie Sequenz von Bytes aber durch einen `SpacePointer` erfasst werden muss, dürfen nur freie Sequenzen entstehen, die mindestens der Länge des `SpacePointers` entsprechen. Bei der massenhaften Allozierung von sehr kleinen Datentypen wie `int` oder sogar `char` kommt es also zu einem großen Blow-Up des benötigten Speicherplatzes. Da solche Typen aber fast immer nur intern für kurze Verarbeitung benötigt werden und die Daten dann in größeren Objekten gespeichert werden, dürfte dieser Blow-Up in der Praxis nur sehr selten in bemerkbarem Umfang auftreten.

Wird nun eine Speichersequenz angefordert und hat der Baum wie oben beschrieben einen Block gefunden, wird die Liste der `SpacePointer` – mit dem `NearPointer` an der niedrigsten Blockadresse beginnend – durchlaufen. Sobald eine passende Sequenz mindestens der angeforderten Länge gefunden wurde, wird sie benutzt. Diese Sequenz kann höchstens besser passen als die maximale Sequenz, die ja bereits die kleinste maximale Sequenz aller Blöcke war. Zwar ist nicht garantiert, dass es in anderen Blöcken nicht genau passende Sequenzen gegeben hätte, doch würde das Lesen jedes Blockes einen IO-Zugriff je Block bedeuten, was das Gewicht des möglichen Verschnitts und der dadurch verursachten zusätzlichen IO-Zugriffe reduziert.

Wird eine Speichersequenz wieder freigegeben, wird zunächst ein `SpacePointer` an die niedrigste Adresse der Sequenz geschrieben, der die Länge der Sequenz beinhaltet. Dann findet eine Kontrolle statt, ob vielleicht eine direkt davor endende oder dahinter beginnende Sequenz existiert. Sollte eines oder beides der Fall sein, würden die Sequenzen vereint, ihre Längen summiert und dem ersten `SpacePointer` zugewiesen. Die beiden anderen `SpacePointer` würden dann aus der einfach verketteten Liste der `SpacePointer` entfernt.

Sowohl nach der Anforderung als auch der Freigabe einer Sequenz würde die neue maximale Sequenz gesucht werden, um mit diesem Wert den Block an die korrekte Position im Suchbaum zu verschieben.

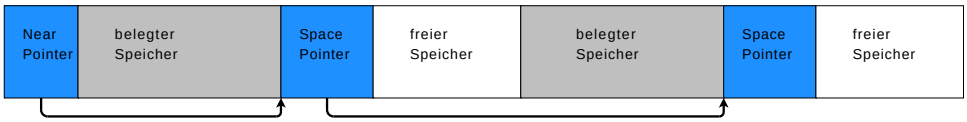


Abbildung 4.5: Aufteilung eines Blockes im `SimpleObject-Controller`

### 4.6.3 Der QuickObjectController

Der `QuickObjectController` wurde entwickelt, um das Allozieren und Freigeben gegenüber dem `SimpleObjectController` zu beschleunigen. Ein wesentlicher Unterschied gegenüber dem `SimpleObjectController` liegt darin, dass in einem Block immer nur Objekte gleicher Größe verwaltet werden. Durch diese Homogenität ist es möglich, viel schneller eine passende Stelle zu finden.

Kein Unterschied zwischen den beiden Implementierungen gibt es beim Allozieren von Arrays: Auch der `QuickObjectController` berechnet nur die Länge des Arrays und alloziert bei XAVER eine entsprechende Anzahl Blöcke.

#### Speicherorganisation innerhalb eines Blocks

Bevor nun auf die Auswahl, in welchem Block ein Objekt alloziert wird, eingegangen wird, soll zunächst die Organisation innerhalb eines Blocks dargestellt werden. Der schematische Aufbau innerhalb eines Blocks ist in Abbildung 4.6 dargestellt.

Der Inhalt innerhalb eines Blocks wird jeweils durch ein `BlockContent`-Objekt verwaltet, das zu Beginn des Blocks liegt. Darin enthalten sind Informationen, wie groß die in diesem Block verwalteten Objekte sind, wie viele Objekte bereits in dem Block als gelöscht markiert wurden, ein `NearPointer` auf die erste freie Stelle innerhalb des Blocks sowie ein Zeiger auf das den Block verwaltende `BlockInfo`-Objekt (die Beschreibung der `BlockInfo`-Objekte erfolgt im nächsten Unterabschnitt). Hinter dem `BlockContent`-Objekt liegt ein Bit-Array, in dem die Daten markiert werden, die bereits gelöscht wurden. Anschließend liegen die Nutzdaten in dem Block, wobei die freien Bereiche durch `NearPointer` miteinander verknüpft sind. Bedingt

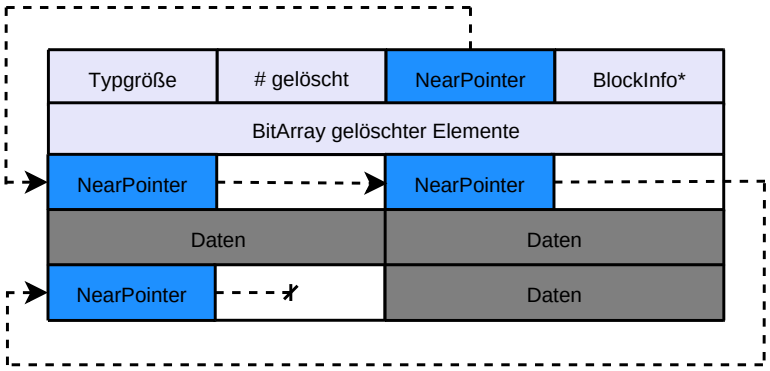


Abbildung 4.6: Aufteilung eines Blockes im `QuickObjectController`

dadurch belegt jeder durch den `QuickObjectController` verwaltete Typ mindestens die Größe eines `NearPointers`.

Im Folgenden werden nun die Funktionen des `BlockContent`-Objekts beschrieben. Dabei sind die Funktionen `allocateInBlock()`, `deallocateFromBlock()` und `reorganizeBlock()` die drei wesentlichen Operationen.

- `inline BlockContent(BlockSize typesize, BlockInfo* blockInfo)`
  Initialisiert das `BlockContent`-Objekt einschließlich des Blocks, in dem es liegt. Dabei wird auch die initiale Verzeigerung innerhalb des freien Speichers des Blocks hergestellt.
- `BlockSize getNoOfElements()`
  Liefert die Anzahl der Objekte zurück, die maximal in dem Block gespeichert werden können.
- `BlockSize getNoOfBitElements()`
  Liefert die Länge des BitArrays zurück.
- `BlockSize getNoOfDeletedFragments()`
  Liefert die Anzahl der in diesem Block als gelöscht markierten Objekte.
- `BlockInfo* getBlockInfo()`
  Liefert den Zeiger auf das `BlockInfo`-Objekt des Blocks zurück (oder NULL, falls kein solches Objekt existiert).
- `void setBlockInfo(BlockInfo* blockInfo)`
  Setzt den Zeiger auf das BlockInfo-Objekt des Blocks.
- `inline bool isFull()`
  Liefert zurück, ob es noch freie Speicherplätze in diesem Block gibt. Speicherplätze, die freigeben wurden, aber nur als gelöscht markiert sind, gelten nicht als frei.
- `inline BlockPos allocateInBlock()`
  Alloziert ein neues Objekt im Block und liefert dessen Position zurück. Dies benötigt nur Zeit $\mathcal{O}(1)$, da hierzu nur der Zeiger auf das erste freie Element geändert und dessen ursprünglicher Wert zurückgeliefert werden muss.
- `inline void deallocateFromBlock(BlockPos pos)`
  Markiert das Objekt an der angegebenen Stelle als gelöscht. Auch dies ist in Zeit $\mathcal{O}(1)$ möglich, da nur das zu dem Objekt gehörige Bit im Bit-Array gesetzt werden muss.

- `inline void reorganizeBlock()`
  Diese Funktion führt die als gelöscht markierten Bereiche wieder der Freispeicherverwaltung zu. Damit diese Methode korrekt arbeitet, darf es vorher keine freien Elemente mehr im Block geben. Die Methode prüft, welche Elemente als gelöscht markiert wurden und generiert aus diesen dann die Liste der freien Elemente. Da dazu ein kompletter Durchlauf durch alle Elemente notwendig ist, benötigt dies Zeit $\mathcal{O}$(maximale Anzahl der Elemente im Block).

### Verwaltung der Blöcke

Der `QuickObjectController` hält nun ein Array, in dem es für jede Typgröße, die er verarbeiten kann, einen Eintrag gibt. In diesem Eintrag befindet sich ein Zeiger auf eine doppelt verkettete Liste von Blöcken, in denen es noch benutzbaren Speicher gibt. Als Invariante gilt hier, dass es in jedem Block dieser Liste entweder noch freie Elemente gibt oder mindestens die Hälfte aller Elemente als gelöscht markiert ist. Die Liste besteht aus Objekten vom Typ `BlockInfo`, die eine sehr schmal gehaltene doppelt verkettete Liste mit `BlockPointern` als Datenelement beinhalten.

Wird nun ein neues Speicherstück angefordert, macht der `QuickObjectController` Folgendes:

- Falls es in der Liste keine Blöcke mehr gibt, wird ein neuer Block angefordert, initialisiert und in die Liste eingefügt.
- Falls der vorderste Block der Liste keinen freien Speicher mehr enthält, muss es gelöschte Elemente in diesem Block geben. Deshalb wird in diesem Fall `reorganizeBlock()` aufgerufen.
- Nun gibt es in der Liste an erster Stelle einen Block mit mindestens einem freien Element, so dass nun in diesem Block der Speicher angefordert wird.
- Falls danach kein freies Element mehr im Block vorhanden ist und mindestens die Hälfte aller Elemente in diesem Block als gelöscht markiert sind, wird der Block ans Ende der Liste geschoben.
- Falls nach dem Allozieren kein freies Element mehr in diesem Block vorhanden ist und weniger als die Hälfte aller Elemente als gelöscht markiert sind, wird der Block aus der Liste entfernt.

Bis auf den Aufruf von `reorganizeBlock()` laufen all diese Operationen in Zeit $\mathcal{O}(1)$. Der Aufruf von `reorganizeBlock()` erfolgt jedoch nur in Blöcken, in denen mindestens die Hälfte aller Elemente als gelöscht markiert sind. Damit amortisiert sich dieser Aufruf mit den Aufrufen des Allozierens dieser Elemente, so dass das Allozieren eines Elements in amortisierter Zeit von $\mathcal{O}(1)$ abläuft.

Die Entscheidung, einen Block ans Ende der Liste zu schieben, wenn er voll ist und mehr als die Hälfte der Elemente als gelöscht markiert sind, ist eine Heuristik, damit `reorganizeBlock()` erst möglichst spät aufgerufen wird, denn während der Block dann noch durch die ganze Liste läuft, werden möglicherweise weitere Elemente in diesem Block freigegeben. Die Heuristik optimiert daher auf Laufzeit, nicht jedoch auf Speicherausnutzung, da es für diese besser wäre, den Block vorne stehen zu lassen.

Um nun ein Speicherstück wieder freizugeben, wird Folgendes gemacht:

- Zunächst wird das Element innerhalb des Blocks als gelöscht markiert.
- Sind dadurch alle Elemente des Blocks als gelöscht markiert, wird der Block wieder freigegeben.

- Sind danach mehr als die Hälfte aller Elemente als gelöscht markiert, so wird der Block hinten in die Liste der Blöcke mit benutzbarem Speicher eingefügt.

Da alle diese drei Dinge in konstanter Zeit ausgeführt werden können, läuft auch das Löschen eines Elements in Zeit $\mathcal{O}(1)$.

### Optimierungen

Der `QuickObjectController` wurde sehr stark auf Laufzeit optimiert. Dennoch gibt es beim `QuickObjectController` noch Optimierungspotential, welches noch nicht umgesetzt wurde:

- Die Berechnung der maximalen Anzahl an Elementen in einem Block nimmt relativ viel Zeit in Anspruch und wird relativ oft benötigt. Es könnte deshalb von Vorteil sein, diese Werte einmal zu berechnen und abzuspeichern.
- Der `QuickObjectController` verwaltet für jede mögliche Typgröße einen Eintrag in seinem Array. Es würde zu einer wesentlichen Verkleinerung des Arrays führen, wenn man als Typgrößen nur Zweierpotenzen verwendet und dann immer auf diese aufrundet. Allerdings würde dies den Verschnitt weiter erhöhen.
- Eventuell könnte es sinnvoll sein, nicht mehr benötigte Blöcke nicht direkt wieder freizugeben, sondern erst einmal noch in einem Pool vorzuhalten. Ob dies jedoch zu einem wesentlichen Geschwindigkeitsvorteil führt, ist auszuprobieren, da Xaver für die Anforderungen einzelner Blöcke bereits intern eine Art Pool-System verwendet.

Auch wenn der experimentelle Vergleich der beiden `ObjectController` nicht systematisiert wurde, so wurde dennoch abgeschätzt, dass der Geschwindigkeitsvorteil des `QuickObjectController` ungefähr bei einem Faktor 2 bis 20 liegt, wohingegen die Speicherauslastung des `SimpleObjectControllers` um einen Faktor bis zu 2 besser ist.

# Kapitel 5

# Die Xaver Template Library

Die Xaver Template Library bietet eine Auswahl von Datenstrukturen und Algorithmen, die direkt auf der Externspeicherverwaltung aufsetzen. Wir haben uns bei den Schnittstellen weitestgehend an der STL orientiert, um eine einfache Bedienbarkeit der Komponenten zu erreichen.

Ziel der Bibliothek ist es, dass auch Algorithmen, die ohne explizite Betrachtung der Blöcke des externen Speichers arbeiten, unter Xaver lauffähig sind. Somit können diese ebenfalls für eine wissenschaftliche Vergleichsstudie mit den optimierten externen Algorithmen herangezogen werden (siehe Kapitel 7)·

## 5.1 Fully Buffered Reader/Writer

Die Klassen `FullyBufferedReader` und `FullyBufferedWriter` implementieren Einweg-Iteratoren für externe Arrays. Bei der Initialisierung wird festgelegt, ob vorwärts oder rückwärts iteriert wird.

### 5.1.1 Beschreibung der Schnittstelle

Die Klassen `FullyBufferedReader` und `FullyBufferedWriter`sind als Template-Klassen implementiert und bieteten das folgende Interface:

- `bool hasData()`
  Prüft, ob der Iterator vor der letzen Position steht.
- `T& readNext()`
  Gibt eine Referenz auf das nächste Element zurück. (nur reader)
- `writeNext(const T& value)`
  Schreibt den Wert value an die nächste Position

### 5.1.2 Die Funktionsweise

Der FullyBufferedReader(Writer) hält intern einen C-Pointer auf das aktuelle(nächste) Element, um Geschwindigkeitsnachteile durch häufiges Dereferenzieren von Far-Pointern zu vermeiden. Dafür wird jeweils der aktuelle Block gelockt.

Um IO-Wartezeiten möglichst zu vermeiden wird der jeweils nächste benötigte Block asynchron geladen. Bei der Initialisierung muss die Größe des Arrays übergeben werden, um zu verhindern, dass Blöcke hinter dem Ende des Arrays asynchron geladen werden.

## 5.2 Sortieren

Der in der XAVER TEMPLATE LIBRARY implementierte externe Sortieralgorithmus basiert auf dem in [15] vorgestellten MERGESORT. Um größtmögliche Flexibilität zu gewährleisten, werden für interne Berechnungen Datenstrukturen und Algorithmen aus der STL verwendet.

### 5.2.1 Beschreibung der Schnittstelle

Auf den Sortieralgorithmus kann über zwei verschiedene Funktionen zugegriffen werden:

- `template<class XRan>`
  `void sort (XRan begin, XRan end)`
- `template<class XRan, class Cmp>`
  `void sort (XRan begin, XRan end, Cmp cmp)`

  Die Beschreibung der Argumente im einzelnen:

- `XRan begin`
  `XaverRandomAccessIterator`, der auf das erste Element des zu sortierenden Arrays zeigt. Gültige `XaverRandomAccessIterator`en sind z.B. die Klasse `FarPointer` sowie die Iteratoren der Vector-Klasse aus der XAVER TEMPLATE LIBRARY.
- `XRan end`
  `XaverRandomAccessIterator`, der auf die Position nach dem letzten Element des zu sortierenden Arrays zeigt.
- `Cmp cmp`
  Zum Sortieren zu verwendende Ordnungsklasse. Die Funktionsweise ist analog zur STL. Wird keine Ordnungsklasse angegeben, wird die Klasse `less` aus der STL verwendet, welche wiederum `operator<` des verwendeten Datentyps benutzt.

### 5.2.2 Funktionsweise des Sortieralgorithmus

Das Sortieren besteht im Wesentlichen aus zwei Phasen: Zum einen aus der sogenannten Run-Formation-Phase und zum anderen aus der Merging-Phase. Sei im Folgenden stets $N$ die Länge des zu sortierenden Arrays, $B$ die Größe eines Blocks im externen Speicher sowie $M$ die Größe des internen Speichers (jeweils gemessen in der Anzahl der Elemente).

Um nicht nur das theoretisch nachgewiesene asymptotische Wachstum in Laufzeit und Anzahl der IO-Zugriffe zu erreichen, sondern auch in den $\mathcal{O}$-Notationen möglichst kleine Konstanten zu bekommen, wurden an dem in [15] vorgestellten Algorithmus noch kleinere Optimierungen vorgenommen, die Probleme beseitigen, die erst in der Praxis, nicht jedoch in der Theorie auftreten.

#### Die Run-Formation-Phase

In der Run-Formation-Phase wird das Array nicht in sortierte Runs der Länge $M$, sondern nur in sortierte Runs der Länge $M - B$ eingeteilt. Der Grund dafür ist folgender: In der theoretischen Beschreibung des Sortieralgorithmus ist man nur an der sortierten Folge interessiert; um nun in der konkreten Implementierung jedoch

konform mit dem Verhalten der STL zu sein, muss die sortierte Folge genau wieder dort liegen, wo auch die Ursprungsdaten lagen. In der Merging-Phase wandern die Daten nun immer zwischen dem ursprünglichen Datenbereich und einem temporären Datenbereich hin und her, bis die Folge sortiert ist. Dabei kann es natürlich passieren, dass die Daten am Ende an der falschen Stelle liegen. Eine naive Lösung dafür wäre, einfach die Daten umzukopieren, was aber einen kompletten Durchlauf des Arrays erfordert.

Die Run-Formation-Phase sorgt nun nicht nur dafür, dass die sortierten Runs gebildet werden, sondern auch, dass die Daten danach so platziert sind, dass sie nach der Merging-Phase wieder im ursprünglichen Datenbereich stehen. Durch eine geschickte Integration in die Run-Formation-Phase fallen so für das Platzieren der Daten im richtigen Datenbereich keine zusätzlichen IO-Zugriffe an. Wichtig ist dabei, dass die Run-Formation-Phase ihre Ergebnisse nach dem Bilden eines Runs in den korrekten externen Datenbereich zurück schreibt, wofür ein Block im internen Speicher als Puffer benötigt wird, so dass nur Runs der Länge $M-B$ erzeugt werden können.

Ein weiterer Aspekt ist die Benutzung des Sortier-Algorithmus' aus der STL. Wenn dieser den kompletten Run sortieren soll, muss er auf FarPointern arbeiten, da es zwischen den Blöcken gegebenenfalls Padding-Bytes gibt. Auch wenn die FarPointer recht stark optimiert wurden, so sind sie natürlich dennoch nicht so schnell wie normale C-Pointer. Um jedoch C-Pointer verwenden zu können, wird jeweils nur ein einzelner Block mit dem Sortier-Algorithmus der STL sortiert. Die Blöcke werden dann mit einem Merging-Step zum sortieren Run zusammengefasst. In diesem Schritt wird dann die Ausgabe in den korrekten Datenbereich geschrieben.

#### Die Merging-Phase

In der Merging-Phase werden die einzelnen Runs zu größeren Runs zusammengefügt. Dabei werden stets $M/B-1$ Runs zu einem neuen Run zusammengefasst. Dadurch entsteht eine optimale Auslastung des internen Speichers: Für jeden Run wird genau ein Block im internen Speicher gehalten. Außerdem gibt es einen Block im internen Speicher für die Ausgabe, so dass der interne Speicher voll ausgenutzt wird.

Die einzelnen Runs (bzw. deren vorderste Elemente) werden in einer Priority-Queue verwaltet, aus der stets das vorderste Element entfernt und in die Ausgabe geschrieben wird. Anschließend wird von diesem Run das nächste Element in die Priority-Queue aufgenommen. Sollte ein Run vollständig abgearbeitet sein, wird er nicht weiter berücksichtigt. Diese Prozedur wird solange wiederholt, bis die Priority-Queue leer ist – dann sind die Runs zusammengefügt. Da die Priority-Queue maximal $M/B-1$ Elemente enthält, wird hier eine interne Priority-Queue der STL eingesetzt.

Auch hier wird wieder an den FarPointern vorbei gearbeitet, indem der jeweils aktive Block eines Runs in den internen Speicher gesperrt wird und darin dann nur noch mit C-Pointern gearbeitet wird.

Das Zusammenfügen der Runs wird solange durchgeführt, bis nur noch ein einzelner, sortierter Run übrig bleibt, so dass dann die gesamte Folge sortiert ist. Dadurch, dass in der Run-Formation-Phase bereits betrachtet wurde, wo die sortierte Folge stehen soll, geht es nach der Merging-Phase genau auf, so dass die Daten genau wieder im ursprünglichen Datenbereich stehen.

### 5.2.3 Laufzeit

Der Sortieralgorithmus benötigt Laufzeit $\mathcal{O}(N \log N)$. Dabei wird jedoch vorausgesetzt, dass die Laufzeit des `sort`-Algorithmus der STL bei $n$ Elementen durch $\mathcal{O}(n \log n)$ beschränkt ist und das Einfügen und Entfernen in eine PriorityQueue mit

$k$ Elementen nicht mehr als Zeit $\mathcal{O}(\log k)$ benötigt. Sofern diese beiden Bedingungen nicht gelten, ist die Laufzeit möglicherweise schlechter.

Der Sortier-Algorithmus erreicht nicht nur die asymptotisch optimale Anzahl von $\mathcal{O}((N/B)\log_{M/B}(N/B))$ IO-Operationen, sondern hat auch in der $\mathcal{O}$-Notation eine sehr kleine Konstante. Da die Anzahl der IO-Zugriffe nur von der Länge der Daten, nicht aber den konkreten Inhalten abhängt, lässt sie sich recht einfach bestimmen. Die genaue Formel lautet:

$$\lceil N/B \rceil \cdot \lceil \log_{M/B-1}(\lceil N/B \rceil) \rceil$$

Für die Herleitung der Laufzeiten sowie der Motivation der Anpassungen des MergeSort sei auf den bereits oben genannten Artikel in [15] verwiesen.

### 5.2.4 Optimierungen

Der Sortier-Algorithmus wurde wie oben beschrieben bereits unter verschiedenen Aspekten optimiert; dabei denken wir, dass die wichtigsten Optimierungsmöglichkeiten ausgeschöpft wurden. Einzig die Tatsache, dass der Algorithmus keinen Gebrauch von asynchronen IO-Zugriffen macht, könnte noch Optimierungspotential bieten. Da jedoch dann statt $M-1$ nur $\lfloor (M-1)/2 \rfloor$ Runs in einem Schritt zusammengefasst werden können, könnte diese Maßnahme insgesamt zu einer Verschlechterung führen. Eine genauere Betrachtung wurde hier jedoch nicht durchgeführt.

## 5.3 NaturalNumberSort

### 5.3.1 Idee

Im Suffix Array Generator müssen häufig Arrays sortiert werden, wo die endgültige Position eines Elementes bereits vor der Sortierung bekannt ist. z.B. um aus einem Suffix Array das zugehörige Reverse Array zu erstellen.

Dieses ist ein eingeschränktes Sortierproblem und bei ersten Tests ist aufgefallen, dass der Mergesort aus der XTL relativ langsam ist. Insbesondere der interne Sortierschritt erfordert mehr Zeit als bei einem spezialisiertem Sortierverfahren nötig ist.

Daher kam die Idee auf für diesen Spezialfall einen eigenen Sortieralgorithmus zu implementieren.

Im internen Speicher lässt sich dieser Algorithmus mit einer einfachen Schleife realisieren. Zudem tritt dieses Problem deutlich seltener auf, da nicht, da die Reihenfolge der Daten im Arbeitsspeicher nicht so wichtig ist, wie die Reihenfolge der Daten im externen Speicher.

### 5.3.2 Realisierung

Der implementierte Algorithmus ist eine spezialisierte externe Variante des Quicksort. Es werden in jedem Rekursionsschritt O(M/B) Pivot-Elemente optimal gewählt. Dies ist möglich, da die sortierte Folge bekannt ist. Der Rekursionsabbruch erfolgt, sobald die Teilfolgen intern sortiert werden können. Die interne Sortierung ist trivial und schreibt jedes Objekt direkt beim Einlesen aus dem Externspeicher an die richtige Position im internen Speicher.

Zum Lesen und Schreiben der (Teil-)Folgen werden FullyBufferedReader/Writer benutzt.

### 5.3.3 Schnittstelle

Die Schnittstelle ist weitestgehend identisch mit derjenigen des Mergesorts. Die Unterschiede sind:

- Ein Comparator muss den Operator () derart überladen, dass er die absolute Position des Objekts im sortierten Array zurückgibt.
- Der Comparator wird zwingend benötigt und ist nicht mehr optional.

## 5.4 Vector

Die `Vector`-Klasse stellt mittels Templates ein dynamisches, externes Array zur Verfügung. Die API orientiert sich allerdings nicht nur an der `vector`-Klasse der STL, sondern nimmt auch Anleihen bei der `Vector`-Klasse des Java SE Development Kits. Es werden Iteratoren und Reverse Iteratoren bereitgestellt, die als Klassen innerhalb der `Vector`-Templateklasse implementiert wurden. Da die Klasse auf einem Array basiert, eignet sie sich weniger, um Elemente an anderen Stellen als dem Ende neu einzufügen oder zu entfernen. Bei solchen Anforderungen sollte auf eine verkettete Liste, wie sie ebenfalls durch die Xaver Template Library bereitgestellt wird, zurückgegriffen werden. Zur Indizierung der Elemente des Vectors wird der Typ `xaver::Index` benutzt, bei dem sich standardmäßig um einen `long long int` handelt.

### 5.4.1 Beschreibung der Schnittstelle

Im Folgenden sind die wichtigsten Funktionen der `Vector`-Klasse aufgelistet. Das vollständige Interface kann in der Xaver Template Library API Dokumentation eingesehen werden.

- `Vector()`
  Erzeugt einen neuen leeren Vector, dessen initiale Größe einen Block beträgt.
- `Vector(Index initialCapacity, Index capacityIncrement)`
  Erzeugt einen neuen Vector mit den spezifizierten Werten für initiale Größe und Vergrößerung.
- `T& at(Index index)`
  Gibt eine Referenz auf das Element an der spezifizierten Position zurück.
- `T& back()`
  Gibt eine Referenz auf das letzte Element zurück.
- `Iterator begin()`
  Gibt einen `Iterator` auf das erste Element zurück.
- `void clear()`
  Löscht alle Elemente aus dem Vector.
- `bool empty() const`
  Gibt genau dann `true` zurück, wenn der Vector leer ist.
- `Iterator end()`
  Gibt einen `Iterator` auf die Position hinter dem letzten Element zurück.
- `Iterator erase(Iterator loc)`
  Löscht das Element, auf das `loc` zeigt und gibt einen `Iterator` auf das Element dahinter zurück.

- `Iterator erase(Iterator start, Iterator end)`
  Löscht alle Elemente zwischen `start` und `end` (inklusive `start`, aber exklusive `end`).
- `void erase(Index index)`
  Löscht das Element an der spezifizierten Position.
- `T& front()`
  Gibt eine Referenz auf das erste Element zurück.
- `Iterator insert(Iterator loc, const T& val)`
  Fügt eine Kopie von `val` vor dem Element, auf das `loc` zeigt, ein und gibt einen `Iterator` auf das neu eingefügte Element zurück.
- `void insert(Index index, const T& val)`
  Fügt eine Kopie von `val` an der Position `index` ein. Ein eventuelles Element an dieser Position wird nach hinten verschoben.
- `void popBack()`
  Löscht das letzte Element.
- `void pushBack(const T& val)`
  Fügt eine Kopie von `val` an das Ende des Vectors an.
- `ReverseIterator rbegin()`
  Gibt einen `ReverseIterator` auf das letzte Element zurück.
- `ReverseIterator rend()`
  Gibt einen `ReverseIterator` auf das erste Element zurück.

### 5.4.2 Funktionsweise

Die `Vector`-Klasse verwaltet intern mit Hilfe von XAVER ein externes Array. Wird im Konstruktor keine initiale Größe angegeben, beträgt die initiale Größe einen Block. Sollte das Array zu klein werden, wird ein neues externes Array bei XAVER angefordert und der Inhalt in das neue Array kopiert. Die Größe des neuen Arrays ist dabei um den im Konstruktor angegeben Wert größer oder – falls keiner spezifiziert wurde bzw. der Wert Null ist – doppelt so groß wie das alte Array.

Beim Einfügen und Entfernen an anderen Stellen als dem Ende werden die nachfolgenden Elemente nach hinten bzw. vorne verschoben.

Der Arrayzugriffsoperator `[]` ist überladen, so dass die `Vector`-Klasse direkt wie ein Array benutzt werden kann.

Mit den Random-Access-Iteratoren `Iterator` und `ReverseIterator`, wie sie von `begin()` und `end()` bzw. von `rbegin()` und `rend()` zurück gegeben werden, kann man durch den Vector navigieren. Dadurch lässt sich die `Vector`-Klasse auch von allen Algorithmen verwenden, die nur mit Hilfe der STL-Iteratoren auf Datenstrukturen operieren. Die Iteratoren speichern intern einen Zeiger auf das `Vector`-Objekt, mit dem sie assoziiert sind, und den aktuellen Index, auf den sie zeigen.

## 5.5 Liste

### 5.5.1 Die Idee

Die Implementierung der Liste folgt im Groben den Ideen aus [7]. Da in diesem Paper jedoch die Optimierung auf Caches beschrieben wird, mussten einige Änderungen und Anpassungen erfolgen. Obwohl sich der Cache im Prinzip genauso zum

RAM verhält, wie der RAM zur Festplatte, unterscheiden sich die quantitativen Faktoren um einige Größenordnungen.

Die Liste nutzt dabei die von XAVER angebotenen Methoden, um sich externen Speicher blockweise zu reservieren und dort die Daten zu speichern. Im Folgenden wird die Funktionsweise erläutert, bevor auf die Komponenten der konkreten Implementierung eingegangen wird.

Ziel der Optimierung ist es, dass die Daten in Iterationsreihenfolge möglichst lokal gespeichert werden. Ohne diese Optimierung liegen schlimmstenfalls alle aufeinanderfolgenden Elemente in unterschiedlichen Blöcken, was ab einer gewissen Listengröße fast jedesmal zum Nachladen eines Blockes vom externen Speicher führt. Da die Verzögerung durch diese IO Operationen einem Vielfachen eines internen Zugriffs entspricht, zahlt sich auch ein Mehraufwand bei der Verwaltung der Daten aus, wenn dadurch IO Operationen vermieden werden.

Daher versucht die XTL-Liste, die Daten lokal in einem sogenannten Bucket zu halten, der genau die Größe eines Blockes einnimmt. Innerhalb dieses Buckets liegen die Daten nicht zwangsläufig in Iterationsreihenfolge, sondern sind selbst über je zwei Zeiger verkettet. Diese Vorgehensweise reduziert den Aufwand, wenn ein neues Element in der Mitte eingefügt wird. So kann einfach ein beliebiger freier Datenslot im Bucket verwendet werden, der dann durch Umhängen von Zeigern an der korrekten Stelle in der Liste eingebaut wird.

Soll ein Element in die Liste eingefügt werden, aber kein freier Datenslot mehr im entsprechenden Zielbucket vorhanden ist, so muss ein neuer Bucket angelegt werden. Dazu wird entweder ein neuer Block über XAVER alloziiert und initialisiert, oder es wird ein Bucket verwendet der zu einem früheren Zeitpunkt Teil der Liste gewesen ist und dann entfernt aber noch nicht verworfen wurde. Dieser Mechanismus wird im folgenden Kapitel „Umsetzung“ näher beschrieben. Der neue Bucket wird vor dem Zielbucket in die Liste gehängt. Ein Split-Vorgang verschiebt dann die Hälfte der Daten aus dem Zielbucket in den neuen, so dass beide Buckets nach Abschluss der Prozedur zur Hälfte gefüllt sind. Da aber in beiden Buckets nun freie Datenslots zur Verfügung stehen, kann das Element eingefügt werden, unbeachtet der Tatsache, in welchem der beiden Buckets sich die Listenposition befindet, an der es eingefügt werden soll.

Wird ein Element aus der Liste gelöscht, wird nicht nur sein Datenslot zur Wiederverwendung freigegeben, sondern auch getestet, ob der Bucket zu leer geworden ist. Dahinter steckt der Gedanke, dass es sich nicht lohnt, für jede Iteration einen kompletten Block zu laden, wenn darin nur wenige Daten gespeichert sind. Endet ein solcher Test mit der Feststellung, dass der Block leer und sein Nachbarblock relativ leer ist, dann werden die Elemente unter Beibehaltung der Reihenfolge in den Nachbarblock verschoben und der Block selbst aus der Liste entfernt.

Um ein Element der Liste zu adressieren, werden Iteratoren verwendet. Das übliche Konzept statischer Zeiger greift hier nicht, da sich sowohl durch Split-Vorgänge, als auch Vereinigungen die absoluten Adressen der Elemente ändern können. Zur Lösung wird eine Zwischenschicht eingeführt, die Referrer. Die Adressen der einzelnen Referrer bleiben unter allen Umständen stabil und führen jeweils einen aktuellen Verweis auf die Speicherposition des assoziierten Elements. Um Speicher zu sparen, kennen Referrer die Anzahl an Iteratoren, die derzeit auf sie verweisen und können sich selbst wieder freigeben, sobald kein solcher Iterator mehr existiert. Die einzelnen Listenelemente wissen, ob für sie bereits ein Referrer existiert und kennen seine Adresse. So können sich die Iteratoren an bereits existierenden Referrern anmelden. Ändert sich nun die Speicherposition eines Listenelementes, wird ein eventuell existierender Referrer darüber informiert und alle Iteratoren verweisen weiterhin auf das korrekte Objekt an nun geänderter Adresse.

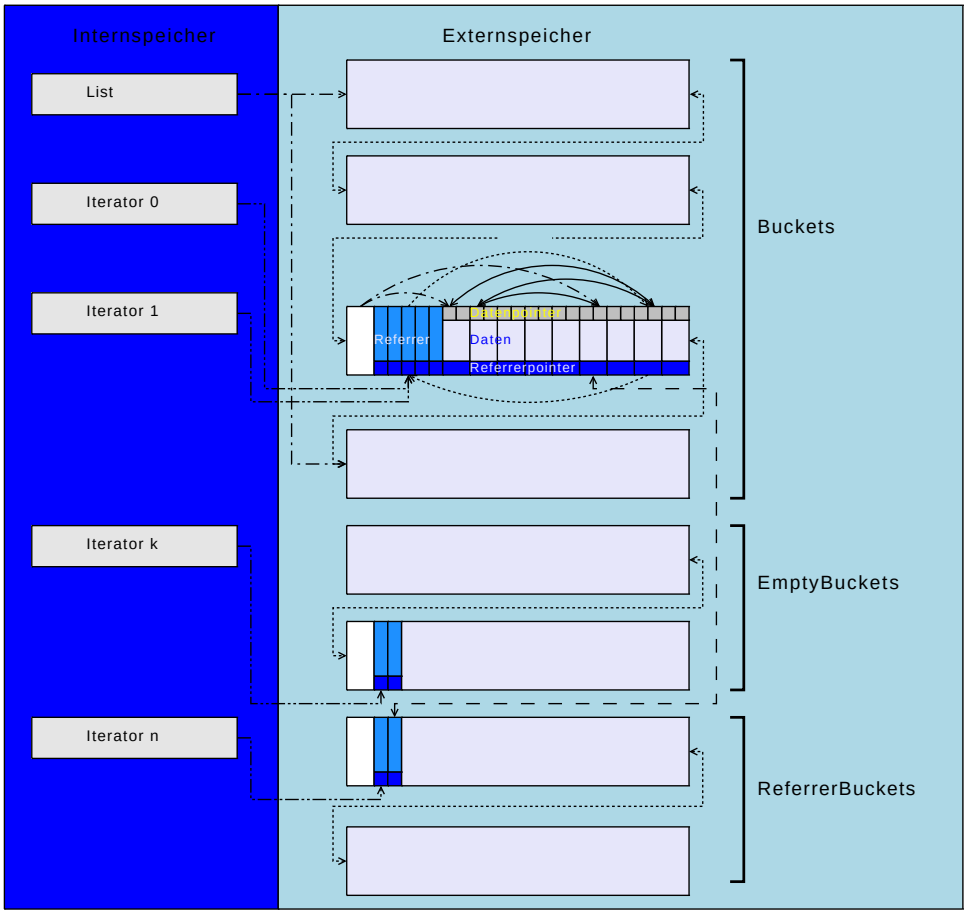


Abbildung 5.1: Schematischer Aufbau einer Liste

### 5.5.2 Die Umsetzung

Die Implementierung setzt sich aus verschiedenen Klassen zusammen, deren Zusammenhang in Abbildung 5.1 deutlich gemacht wird. Auf die Details der einzelnen Objekte und deren Interaktion werden wir im Folgenden eingehen.

#### List

Das zentrale Element der Liste ist das `List`-Objekt. Sämtliche Operationen auf der Liste, die Iteration über die Liste ausgeschlossen, werden über dieses Objekt angestoßen. Es verfügt daher über die oben genannten Methoden und Funktionen. Um diese Methoden zu realisieren, speichert das Objekt folgende Informationen:

- Die Anzahl der Elemente in der Liste
- Die Anzahl der Datenslots pro Bucket
- Die Anzahl der Referrer pro Bucket
- Einen Pointer auf den ersten und den letzten Bucket der Liste
- Einen Pointer auf den ersten freien und den letzten freien Bucket
- Einen Pointer auf den ersten ReferrerBucket

Über diese Informationen lassen sich alle benötigten Informationen beschaffen, um die Operationen durchzuführen.

Das Einfügen und Herausnehmen von Buckets in bzw. aus der Liste wird über den Zeiger auf den ersten und letzten freien Bucket realisiert. Dieser zeigt auf das erste Element einer einfach verketteten Liste von freien Buckets. Diese ist zum Initialisierungszeitpunkt der Liste leer. Läuft aber nun ein Bucket leer, weil er zum Beispiel mit seinem Nachbarbucket vereinigt wird, kann er als neues erstes Element an diese Liste gehängt werden. Dies geschieht aber nur, falls noch Referrer im Bucket gespeichert sind, denn diese müssen unbedingt erhalten bleiben. Andernfalls würde der ganze Block freigegeben werden.

Wird ein leerer Bucket benötigt, wird zunächst geprüft, ob sich in dieser Liste Buckets befinden. Dabei werden Buckets, die inzwischen keine Referrer mehr enthalten, freigegeben. Existiert nach der Prüfung noch ein Bucket in der Liste, wird er einem neu von XAVER angeforderten Block vorgezogen.

Durch die Verwaltung der leeren Buckets in einer Liste können die für die Listenfunktionalität sowieso nötigen Eigenschaften der Buckets genutzt werden, um die Freispeicherverwaltung zu realisieren. Daher wird kein zusätzlicher Speicherplatz benötigt.

Die `sort()`-Funktion verwendet einen MERGESORT Algorithmus, um möglichst wenig IO-Last zu erzeugen. Im ersten Schritt werden alle einzelnen Buckets sortiert und danach als sortierte Sequenz betrachtet. Passen $M$ Blöcke in den Hauptspeicher, werden $M - 2$ Sequenzen mittels einer internen PriorityQueue in einem Merging-Step sortiert. Dabei entstehen neue sortierte Sequenzen von der Länge der Summe der jeweils bearbeiteten Sequenzen. Diese werden dann rekursiv weiter zusammengefasst, bis die Liste eine einzige sortierte Sequenz ist.

#### Bucket

Die zentrale Datenstruktur sind die `Buckets`. Sie werden doppelt verlinkt und bilden somit Abschnitte der Liste. Sie verfügen über folgende Informationen:

- Die Anzahl der benutzten DatenSlots

- Einen Pointer auf ihren Nachfolger- und Vorgängerbucket
- Einen NearPointer auf den ersten und letzten Datenslot im Bucket
- Einen NearPointer auf den ersten freien Datenslot im Bucket
- Einen NearPointer auf den ersten Referrer im Bucket
- Einen NearPointer auf den ersten freien ReferrerSlot im Bucket

Für diese Angaben wird der in Abbildung 5.1 gezeigte Speicherplatz in den Buckets benötigt. Dieser Speicherplatz stellt Overhead gegenüber einer herkömmlichen Liste dar.

Mittels der Anzahl der benutzten Datenslots läßt sich schnell prüfen, ob ein Bucket mit einem anderen vereinigt werden kann.

Die Pointer auf den Vorgänger- und Nachfolgerbucket ermöglichen das bidirektionale Iterieren über die Liste. Wird der Bucket als Vorgänger bzw. als Nachfolger adressiert, kann jeweils über den Verweis auf den letzten bzw. ersten Datenslot das nächste Element in Iterationsreihenfolge adressiert werden.

Das Einfügen und Löschen von Elementen wird über den Zeiger auf den ersten freien Datenslot im Bucket realisiert. Dieser zeigt auf das erste Element einer einfach verketteten Liste von freien Datenslots. Wird nun ein Datenslot gefüllt, wird der erste aus dieser Liste entfernt und der Nachfolger wird neues erstes Element. Wird ein Datenslot geleert, weil ein Element gelöscht wurde, wird es als neues erstes Element an diese Liste gehängt. Durch die Verwaltung der leeren Datenslots in einer Liste können die für die Listenfunktionalität sowieso nötigen Eigenschaften der Datenslots genutzt werden, um die Freispeicherverwaltung zu realisieren. Daher wird kein zusätzlicher Speicherplatz benötigt.

Beim Iterieren wird für jeden Datenslot ein Referrer benötigt. Um nun keinen zusätzlichen Block in den Speicher laden zu müssen, verfügt der Bucket über einen kleinen Buffer für Referrer. Damit bleiben alle Datenzugriffe lokal, falls nur wenige Iteratoren gleichzeitig benutzt werden. Da dieser Buffer in der Implementierung dem ReferrerBucket folgt, werden nähere Informationen zur Umsetzung dort gegeben.

### DataSlot

Die Container für die einzelnen Listenelemente stellen die `DataSlot`-Objekte dar. Neben den Daten selbst speichern sie noch folgende Information:

- Einen NearPointer auf den vorhergehenden und den nachfolgenden DataSlot
- Einen FarPointer auf den zugehörigen Referrer

Während die Buckets über FarPointer verkettete Listenabschnitte bilden, sind nun die DataSlots die Elemente in diesen Abschnitten. Über die NearPointer wird die Iterationsreihenfolge innerhalb der Buckets festgelegt. Die Verwendung von NearPointern erspart hier massiv Speicherplatz, da als Datentyp der kleinstmögliche ausreicht, der die gesamte Blockgröße adressieren kann.

Der FarPointer zeigt jeweils auf einen Referrer, falls einer existiert, sonst ist er `NULL`. Ändert eine Methode die Speicheradresse eines DataSlots, bzw. seines Inhaltes, kann der entsprechende Referrer über diesen Pointer über die Verschiebung informiert werden, so dass er wieder auf die korrekte Adresse verweisen kann.

### Iterator

Für den Zugriff auf die Liste sollten in erster Linie Iteratoren verwendet werden. Deren Funktionalität übernehmen `Iterator` Objekte. Sie speichern folgende Information:

- Einen internen Pointer auf die Liste
- Einen FarPointer auf den aktuellen Referrer

Der FarPointer speichert dabei indirekt die aktuelle Position innerhalb der Liste, indem er auf einen Referrer zeigt. Die Referrer halten dabei wie in Abbildung 5.1 gezeigt immer den Kontakt zu ihren Datenslots, so dass die Position der Iteratoren invariant unter Sortierung und anderen Operationen bleiben, solange das aktuelle Objekt nicht entfernt wird.

Um einen Iterationsschritt zu vollführen, wird der Iterator am Referrer die Position des aktuellen DataSlots erfragen und von diesem den Nachfolger erhalten. Dann prüft er, ob dieser Nachfolger bereits einen Referrer besitzt. Falls dem so ist, wird er sich bei diesem Referrer registrieren, falls nicht, muss ein neuer Referrer erzeugt werden. Dazu fragt er zunächst am lokalen Bucket des DataSlots nach, ob noch ein Referrer gespeichert werden kann. Falls der lokale Buffer bereits voll ist, wird eine Anfrage an die Liste gestartet, um die Adresse des globalen Referrer-Speichers zu bekommen. Dort wird der neue Referrer dann erzeugt und mit dem Datenslot bidirektional verknüpft. Anschließend registriert sich der Iterator an diesem Referrer und gibt einen FarPointer auf die Daten zurück.

Bevor ein solcher Iterationsschritt ausgeführt wird, sollte zunächst mit `hasNext` geprüft werden, ob überhaupt ein nächstes Element existiert. Diese Methode prüft dazu den Zeiger auf das nächste Element auf `NULL`.

### Referrer

Die zentralen Elemente für die korrekte Funktionsweise der Iteratoren stellen die `Referrer`-Objekte da. Sie besitzen folgende Eigenschaften:

- Einen Zähler für die Iteratoren, die aktuell am Referrer angemeldet sind
- Einen FarPointer auf den zugeordneten Datenslot in der Liste
- Einen NearPointer auf den nächsten Referrer im aktuellen Container

Wie beim `Iterator` bereits beschrieben, melden sich die Iteratoren am Referrer an und ab, damit dieser feststellen kann, wann er nicht mehr notwendig ist und sich dann selbst freigeben kann. Dazu hält er die Nummer der aktuell angemeldeten Iteratoren, fällt diese auf Null, gibt er sich wieder frei.

Der FarPointer hält unter allen Umständen die Verbindung zum Element in der Liste, indem bei jeder Operation, die die Adresse des Elementes verändert, der Zeiger neu gesetzt wird. Der NearPointer dient nun der Freispeicherverwaltung. Ist der Referrer unbenutzt, ist er Teil einer einfach verketteten Liste von freien Referrern, wird ein neuer Referrer angelegt, so wird der erste Referrer dieser Liste verwendet, um ihn zu speichern. Wird ein Referrer freigegeben, wird er neuer Kopf dieser Liste. Dabei sind diese Listen nicht global, sondern jeweils beschränkt auf den aktuellen `ReferrerContainer`.

### ReferrerContainer

Die Struktur, die die Referrer speichert, heißt `ReferrerContainer`. Sie stellt Informationen und Methoden zur Verfügung um die Referrer zu speichern und den freien Platz für neue Referrer zu verwalten. Dazu werden folgende Informationen gespeichert:

- Einen FarPointer auf den ersten leeren Referrer

Dieser Zeiger zeigt auf das erste Element der Liste von freien Referrern, die wie oben beschrieben über die NearPointer in den Referrern realisiert wird.

Es gibt verschiedene `ReferrerContainer`: zum einen existieren dedizierte `ReferrerBuckets`, die einen kompletten Block einnehmen, aber nur Referrer speichern, zum anderen können die Referrer auch in den `Buckets` gespeichert werden.

Die `ReferrerBuckets` stellen den globalen Speicher für Referrer dar, der benutzt wird, wenn der lokale Buffer in den Buckets voll gelaufen ist. Die `ReferrerBuckets` sind in einer einfach verketteten Liste aneinander gehängt, so dass beliebig viele verwaltet werden können. Ist der erste ReferrerBucket ebenfalls voll, wird ein zweiter angehängt, der dann gefüllt wird. Da dieser als neuer Anfang der Liste angehängt wird, müssen nie mehr ReferrerBuckets durchlaufen werden, falls viele volle existieren.

Es darf kein ReferrerContainer gelöscht werden, der noch Referrer enthält, da sonst die Iterationsreihenfolge nicht mehr deterministisch wäre. Daher kann es schlimmstenfalls vorkommen, daß jeder ReferrerBucket nur noch einen Referrer enthält. Da diese Situation in der Praxis aber so gut wie nie auftauchen wird, da dafür unzählige Iteratoren nötig wären, von denen anschließend die richtigen zufällig wieder gelöscht werden müssten, ist es vertretbar hier keine weitere Optimierung vorzunehmen.

### 5.5.3 Das Interface von `List`

Die Liste der XTL implementiert eine Teilmenge der STL Methoden und erfüllt für sie die Anforderungen der STL. Folgende Methoden werden unterstützt:

- `bool isEmpty()`
  Mit `isEmpty` lässt sich herausfinden, ob die Liste Elemente enthält. Sie liefert `true` zurück, falls ja, `false` sonst.

- `int size()`
  `size` gibt die Anzahl der Elemente der Liste zurück.

- `void pushFront(const T& object)`
  `pushFront` fügt das Element `object` an den Anfang der Liste ein. Falls der erste Bucket der Liste gefüllt ist, wird ein neuer Bucket angelegt und an den Anfang gehängt. Dann werden die Daten zu gleichen Teilen auf beide Buckets verteilt und das neue Element an den Anfang der Liste eingehängt.

- `void pushBack(const T& object)`
  `pushBack` fügt das Element `object` am Ende der Liste ein. Falls der letzte Bucket der Liste gefüllt ist, wird ein neuer Bucket angelegt und vor den letzten Bucket der Liste eingehängt. Dann werden die Daten zu gleichen Teilen auf beide Buckets verteilt und das Element ans Ende der Liste gehangen.

- `void densePushFront(const T& object)`
  `densePushFront` fügt das Element `object` an den Anfang der Liste ein. Falls der erste Bucket der Liste gefüllt ist, wird ein neuer Bucket angelegt und als erster Bucket in die Liste eingehängt. Es finden allerdings keine Datenverteilungen statt, es wird nur das neue Element eingefügt. Nach der Operation können also Buckets entstehen die nur ein Element enthalten.
  Dieses Vorgehen wird nicht empfohlen wenn die Liste über Iteratoren manipuliert werden soll, kann aber sinnvoll sein, wenn einmal massiv Daten eingefügt werden, und danach die Struktur der Liste unverändert bleibt.

- `void densePushBack(const T& object)`
  `densePushBack` fügt das Element `object` am Ende der Liste ein. Falls der letzte Bucket der Liste gefüllt ist, wird ein neuer Bucket angelegt und als letzter

Bucket in die Liste eingehängt. Es finden allerdings keine Datenverteilungen statt, es wird nur das neue Element eingefügt. Nach der Operation können also Buckets entstehen, die nur ein Element enthalten.
Dieses Vorgehen wird nicht empfohlen, wenn die Liste über Iteratoren manipuliert werden soll, es kann aber sinnvoll sein, wenn einmal massiv Daten eingefügt werden, und danach die Struktur der Liste unverändert bleibt.

- `void popFront()`
  Diese Methode entfernt das erste Element aus der Liste.
  Bei dieser Operation kann es zu einer Vereinigung von Buckets kommen, falls aus einem Bucket gelöscht wurde, der anschließend nur noch zu 1/4 gefüllt ist, und ein benachbarter Bucket existiert, der nach der Vereinigung nur zu 4/5 gefüllt wäre.

- `void popBack()`
  `popBack` entfernt das letzte Element aus der Liste.
  Bei dieser Operation kann es zu einer Vereinigung von Buckets kommen, falls aus einem Bucket gelöscht wurde, der anschließend nur noch zu 1/4 gefüllt ist, und ein benachbarter Bucket existiert, der nach der Vereinigung nur zu 4/5 gefüllt wäre.

- `Iterator<T> insert(Iterator<T> position, const T& object)`
  Mittels `insert` läßt sich ein Element `object` an einer beliebigen Stelle innerhalb der Liste einfügen. Dabei wird das Element vor der aktuellen Position eingefügt, auf die der Iterator `position` zeigt.
  Hat der Bucket, der die aktuelle Position enthält, keinen freien Speicher mehr, wird ein neuer Bucket angelegt und die Hälfte der Daten in den neuen Bucket verschoben. Dabei kann auch die aktuelle Position in einen anderen Bucket verschoben werden. Danach wird dann das neue Element eingefügt.

- `Iterator<T> remove(Iterator<T>& position)`
  `remove` entfernt das Element, auf das der Iterator `position` verweist, aus der Liste. Der Iterator zeigt anschließend auf das Nachfolgedatum in der Liste oder ist `invalid` falls keines existiert.
  Bei dieser Operation kann es zu einer Vereinigung von Buckets kommen, falls aus einem Bucket gelöscht wurde, der anschließend nur noch zu 1/4 gefüllt ist, und ein benachbarter Bucket existiert, der nach der Vereinigung nur zu 4/5 gefüllt wäre.

- `void remove(T& value)`
  Mittels `remove` werden alle Elemente entfernt, für die der Operator `operator== true` für `value` liefert.
  Bei dieser Operation kann es zu einer Vereinigung von Buckets kommen, falls aus einem Bucket gelöscht wurde, der anschließend nur noch zu 1/4 gefüllt ist, und ein benachbarter Bucket existiert, der nach der Vereinigung nur zu 4/5 gefüllt wäre.

- `void sort()`
  `sort` sortiert die Liste. Dazu benutzt es eine modifizierte MergeSort Variante, die zunächst alle einzelnenen Buckets sortiert und anschließend so viele Buckets wie möglich in den Hauptspeicher lädt. Diese werden dann zu einer Sequenz zusammengefasst. Passen die Daten nicht komplett in den Hauptspeicher, und gibt es mehrere solcher Sequenzen, so werden die Sequenzen rekursiv zusammengefasst.

- `void clear()`
  `clear` löscht alle Inhalte aus der Liste. Alle existierenden Iteratoren werden

damit `invalid` und dürfen nicht mehr benutzt werden. Allozierte Blöcke werden wieder freigegeben.

- `void unique()`
  `unique` löscht alle Vorkommen bis auf das erste von nacheinander vorkommenden gleichen Elementen. Um die Gleichheit zu überprüfen wird der Operator `operator==` verwendet, der von dem Typen korrekt implementiert sein muß.

- `Iterator<T> iterator()`
  `iterator` gibt einen Iterator zurück, der auf den Anfang der Liste zeigt.

### 5.5.4 Das Interface von `Iterator`

Das Interface des Iterators stellt alle benötigten Methoden zur Verfügung, um komfortabel durch die Liste zu iterieren:

- `Iterator(List<T>* list)`
  Über diesen Konstruktor läßt sich ein neuer Iterator erzeugen. Dieser zeigt auf das erste Element der Liste.

- `bool hasNext()`
  `hasNext` liefert `true` zurück, falls ein weiteres Element in der Liste folgt, sonst `false`.

- `bool hasPrevious()`
  `hasPrevious` liefert `true` zurück, falls ein vorhergehendes Element in der Liste existiert, sonst `false`.

- `xaver::FarPointer<T> next()`
  `next` liefert einen FarPointer auf das nächste Element in der Liste zurück, falls noch ein Weiteres existiert. Vor dem Aufruf dieser Methode ist mit `hasNext` zu prüfen, ob ein weiteres existiert.

- `xaver::FarPointer<T> previous()`
  `next` liefert einen FarPointer auf das vorhergehende Element in der Liste zurück, falls noch eins existiert. Vor dem Aufruf dieser Methode ist mit `hasPrevious` zu prüfen, ob ein vorhergehendes existiert.

## 5.6 Stack

Die Klasse `Stack` liefert nach außen eine Schnittstelle, die die Bedienung eines extern verwalteten Stacks ermöglicht. Das Stack-Objekt kommuniziert hierbei mit der statischen XAVER-Klasse, um Speicher für die zu verwaltenden Objekte zu reservieren.

### 5.6.1 Beschreibung der Schnittstelle

Der Stack ist als Template-Klasse implementiert und kann somit Objekte beliebigen Typs speichern. Die folgenden Methoden werden unterstützt:

- `bool empty()`
  Liefert genau dann wahr, wenn der Stack kein Element enthält.

- `size_t size()`
  Liefert die Anzahl der Elemente im Stack zurück.

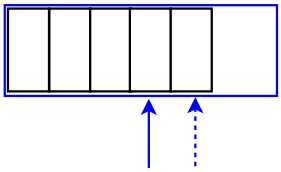

Abbildung 5.2: Löschen eines Elementes

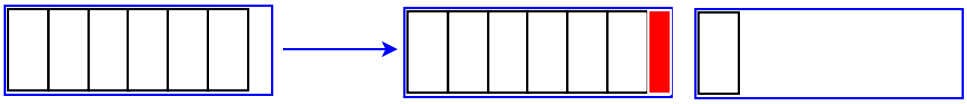

Abbildung 5.3: Einfügen bei vollem Block

- `T& top()`
  Liefert eine Kopie des obersten Elementes des Stacks.
- `void push(const T& element)`
  Packt das übergebene Objekt als Kopie auf den Stack.
- `void pop()`
  Entfernt das oberste Element vom Stack.

### 5.6.2 Die Funktionsweise

Der Stack besteht intern aus einer Folge von Blöcken. Hierbei wächst die Größe des Stacks dynamisch mit jedem neu hinzugefügten Objekt. Es wird immer dann ein neuer Block bei XAVER angefordert, wenn alle bisherigen Blöcke belegt sind. Ein leerer Block dagegen wird erst freigegeben, wenn sein Vorgänger nur noch bis zur Hälfte gefüllt ist.

#### Hinzufügen des ersten Elementes

Es wird ein neuer Block bei XAVER angefordert und das übergebene Element an die erste Stelle des Blocks kopiert.

#### Methoden ohne Block-Allozierung

Die meisten `push`- und `pop`-Operationen arbeiten innerhalb eines Blockes. Hierzu wird intern ein Zähler verwaltet, der immer auf die nächste freie Position im Block zeigt. Das Entfernen eines Elementes führt somit zu einer einfachen Verringerung des Zählers für den nächsten Schritt. (siehe Abbildung 5.2)

#### Hinzufügen mit Block-Assozierung

Sobald das hinzuzufügende Element nicht mehr in den aktuellen Block passt, wird bei XAVER ein neuer Block alloziert, in dem das übergebene Objekt abgelegt wird (siehe Abbildung 5.3). Da Elemente nicht zwischen zwei Blöcken aufgeteilt werden, hat die Struktur eine innere Fragmentierung von

$$\text{size(T)} \mod \text{blockSize}$$

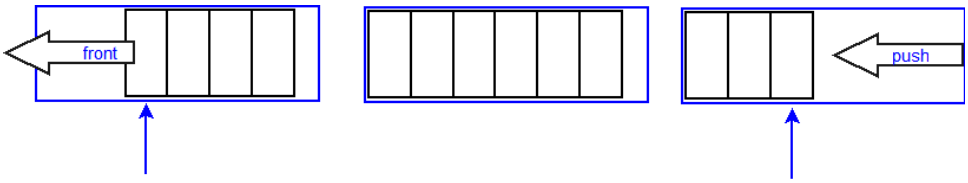


Abbildung 5.4: Struktur der Queue

**Entfernen eines Elementes mit Block-Freigabe**

Sobald das letzte Element eines Blockes entfernt wird, springt der Zähler automatisch auf den vorherigen Block im Stack. Der hierdurch freigewordene Block wird allerdings nicht sofort wieder aus der Speicherverwaltung entfernt, da ansonsten eine ungünstige `Push-Pop`-Kombination genau am Blockende auf jeden Fall eine IO-Operation benötigen würde. Der freigegebene Block wird solange als potentieller nächster Block vorgehalten, bis der Vorgänger nur noch höchstens zur Hälfte mit Elementen gefüllt ist.

## 5.7 Queue

Die Klasse `Queue` implementiert eine Warteschlange, die Objekte nach und nach aufnehmen kann, welche dann nach dem FIFO-Prinzip wieder ausgelesen werden können.

### 5.7.1 Beschreibung der Schnittstelle

Die Queue ist als Template-Klasse implementiert und bietet das folgende Interface:

- `bool empty()`
  Prüft, ob die Queue leer ist.
- `size_t size()`
  Liefert die Anzahl der Elemente in der Queue.
- `T& front()`
  Liefert eine Kopie des vordersten Elementes in der Queue.
- `T& back()`
  Liefert eine Kopie des letzten Elementes in der Queue.
- `void push(const T& element)`
  Fügt das übergebene Objekt hinten an die Queue an.
- `void pop()`
  Entfernt das vorderste Element aus der Queue.

### 5.7.2 Die Funktionsweise

Die innere Struktur der `Queue` ähnelt sehr stark jener des in Kapitel 5.6 beschriebenen `Stacks`. Allerdings wächst die Queue durch `push`-Aufrufe am hinteren Ende und verkürzt sich durch `pop` am Anfang. (siehe Abbildung 5.4)

#### Hinzufügen am Ende der Queue

Das neue Element wird in den letzten Block der Queue eingefügt. Passt das Objekt nicht mehr komplett in den Block, so reserviert sich die Datenstruktur bei XAVER automatisch einen neuen Block, hängt diesen komplett an die Queue an und fügt das einzufügende Objekt dort ein.

#### Entfernen des vordersten Elements

Im Normalfall wird einfach das vorderste Element des ersten Blocks gelöscht und der Zeiger um eine Position weiter nach hinten gesetzt. Wird hierdurch allerdings der erste Block leer, so wird dieser direkt bei XAVER aus der Speicherverwaltung entfernt. Ein Zwischenspeichern dieses Blockes, wie es beim Stack aus Effizienzgründen gemacht wird, ist hier nicht notwendig, da bei der Queue von vorne keine neuen Elemente eingefügt werden können.

## 5.8 PriorityQueue

### 5.8.1 Funktionsweise

Eine PriorityQueue ist eine abstrakte Datenstruktur, die folgende Operationen auf einer total geordneten Menge unterstützt:

- `getMin`: Ausgabe des kleinsten Elementes
- `delMin`: Entfernen des kleinsten Elementes
- `insert`: Einfügen eines neuen Elementes
- `remove`: ein Element entfernen (optional)
- `decreaseKey`: den Schlüsselwert eines Elementes verringern (optional)

Im internen Speicher werden Priority-Queues meist mit Hilfe von Heaps realisiert. Diese sind aber nicht IO-Effizient. Deswegen benutzt die Implementierung von PriorityQueues, die in XAVER verwendet wird, externe Array-Heaps als Datenstruktur.

Die wesentliche Idee hinter den externen Array-Heaps ist, Daten die nicht in den internen Speicher passen, sortiert in den externen Speicher auszulagern. Daten im internen Speicher werden weiterhin in einem Heap gespeichert. Dies verknüpft die Geschwindigkeit von Heaps im internen Speicher (kein vollständiges Sortieren notwendig) mit der IO-Effizienz sortierter Arrays.

Der interne Heap benutzt den halben internen Speicher. Ein neues Element der Priority-Queue wird grundsätzlich in den internen Heap eingefügt. Sollte dieser vollständig gefüllt sein, so wird der komplette Heap in sortierter Reihenfolge in ein neues externes Array geschrieben und geleert.

Um IO-Zugriffe zu minimieren wird von jedem so erstellten externen Array jeweils der Block mit dem kleinsten Element im internen Speicher gehalten. Das jeweils kleinste Element von jedem externen Array wird zusätzlich in einen weiteren internen Heap eingefügt, um das Auffinden des global kleinsten Elementes effizient zu ermöglichen.

Sollte der interne Speicher nicht mehr ausreichen, um für jedes externe Array einen Block intern vorzuhalten, werden alle externen Arrays zu einen neuem Array zusammengefasst. Dafür wird die gesamte Priority-Queue in sortierter Reihenfolge ausgelesen.

### 5.8.2 Interface in der Xaver Template Library

Die Priority-Queue ist als Template-Klasse implementiert und hat das folgende Interface:

- `bool isEmpty()`
  Prüft, ob die Priority-Queue leer ist.
- `size_type size()`
  Liefert die Anzahl der Elemente in der Priority-Queue.
- `T& top()`
  Liefert eine Referenz auf das kleinste Element in der Priority-Queue.
- `void push(const T& element)`
  Fügt das übergebene Objekt in die Priority-Queue ein.
- `void pop()`
  Entfernt das kleinste Element aus der Priority-Queue.

## 5.9 Baum

Die Baum-Datenstruktur bildet einen B-Baum auf das Xaver-Framework ab. Ein B-Baum ist ein balancierter Baum, der in jedem Knoten mehrere Elemente halten kann. Dadurch kann jeder Knoten bis maximal |Elemente| + 1 Kinder haben. Die Ausgeglichenheit und hohe Elementanzahl in den einzelnen Knoten lässt einen flachen Baum entstehen, was besonders im Hinblick auf Externspeicher interessant ist.

### 5.9.1 Beschreibung der Schnittstelle

Der Benutzer arbeitet mit der Klasse `BTree`, die im internen Speicher liegt und die komplette Funktionalität des Baumes zugänglich macht. Im Folgenden sind die wichtigsten Funktionen aufgelistet. Das vollständige Interface kann in der Xaver Template Library API Dokumentation eingesehen werden.

- `void insert(const T& obj)`
  Fügt ein Element in den Baum ein.
- `void remove(const T& obj)`
  Löscht ein Element aus dem Baum.
- `bool contains(const T& obj)`
  Prüft, ob das gegebene Element im Baum existiert.
- `T get(const T& obj)`
  Liefert das Element zurück.
- `bool empty()`
  Prüft, ob der Baum überhaupt Elemente enthält.
- `string printTree()`
  Erstellt einen String, der den Inhalt des Baumes enthält.

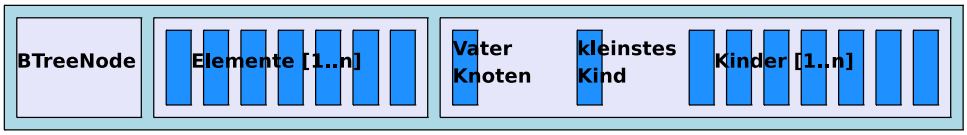


Abbildung 5.5: Aufbau eines Baumknotens

### 5.9.2 Die Technik

Die grundlegende Idee der Baum-Implementierung ist das Ausnutzen kompletter Blöcke als Baumknoten. Ein Block wird partitioniert, um eine Ordnungs-Datenstruktur (`BTreeNode`), ein Array für die Elemente und eines für die Zeiger auf Vater- und Kinder-Baumknoten zu halten. Die Zeiger sind als `FarPointer` realisiert, deren Dereferenzierung über XAVER zum gewünschten Block (in diesem Fall Baumknoten) führt. Ein Zeiger an Position $i$ im Array verweist dabei auf das Kind, das den nächstgrößeren Wertebereich verglichen mit dem Element an Position $i$ und den insgesamt kleineren Wertebereich bezogen auf das Element an der $(i+1)$-ten Position hat. Der Vaterknoten und der Knoten mit einem kleineren Wertebereich als alle in diesem Knoten enthaltenen Elemente werden durch diese Vereinbarung nicht abgedeckt und entsprechend einzeln behandelt (beide liegen zwar auch im Zeiger-Array, eine komplizierte Referenzierung darüber würde allerdings die Lesbarkeit des Quelltextes verringern). Wenn ein Block neu hinzugefügt wird, wird an seinem Anfang ein Objekt der Klasse `BTreeNode` erzeugt, welches den Füllstand des Blockes verwaltet und die Arrays über `NearPointer` zugänglich macht. Der komplette Aufbau ist in Abbildung 5.5 schematisch dargestellt.

Das Einfügen eines neuen Elementes findet auf der Ebene der Blätter statt. Dazu wird erst das Blatt mit dem entsprechenden Wertebereich gesucht und das Element diesem Knoten (Block) hinzugefügt. Sobald ein Element entfernt werden soll, muss zuerst überprüft werden, ob dieses Element ein Kind besitzt. In diesem Fall wird das Element durch das kleinste Element in dem darunter liegenden Teilbaum ersetzt, ansonsten kann es einfach gelöscht werden.

Um die Ausgeglichenheit des Baumes zu sichern, existieren Randbedingungen bezüglich des Füllstandes der Baumknoten. Sobald ein Knoten weniger als zur Hälfte gefüllt ist, wird ein Geschwisterknoten ausgewählt. Falls dieser genug Elemente enthält, findet ein Austausch statt. Der aufzufüllenden Knoten erhält ein Element aus dem Vaterknoten, während der Geschwisterknoten ein Element an den Vaterknoten abgibt. Falls der Geschwisterknoten jedoch selber nur zur Hälfte gefüllt ist, werden sowohl das passende Element im Vater als auch alle Elemente des Geschwisterknotens in den aufzufüllenden Knoten verschoben und der Geschwisterknoten gelöscht. Die Wurzel ist der einzige Knoten, der weniger Elemente enthalten darf (in diesem Fall besteht der Baum allerdings auch nur aus der Wurzel).

Falls ein Knoten hingegen seine maximale Elementanzahl erreicht hat, wird er in zwei Knoten unterteilt. Dazu wird das Element in der Mitte des zu unterteilenden Knotens gewählt und alle Elemente, die größer sind, in einen neuen Knoten verschoben. Anschliessend wird das gewählte Element in den Vaterknoten verschoben und der neu erstellte Knoten als Kind darunter gehängt. Sämtliche Balancierungs-Operationen können den Vaterknoten beeinflussen, weswegen dieser anschließend überprüft werden muss.

### 5.9.3 Die Klasse Map

Die Implementierung der `Map` beruht auf dem vorgestellten Baum in Verbindung mit der Klasse `KeyValuePair`. Diese hält einen Schlüssel zusammen mit einem ihm zugeordneten Wert. Sämtliche Vergleiche betrachten allerdings nur die Schlüssel, was

für die Eindeutigkeit der Schlüssel im Baum und die Suche wichtig ist. Die Klasse `KeyValuePair` soll nicht explizit durch den Benutzer verwendet werden, wodurch sich die Schnittstelle der `Map` aus der des Baumes wie im Folgenden aufgezeigt ergibt. Die Parameter `Key` und `Value` sind Templates, die den Klassen der zu haltenden Werte entsprechen.

- `void insert(const Key& key, const Value& value)`
  Fügt einen Schlüssel mit zugehörigem Wert ein.
- `void remove(const Key& key)`
  Löscht einen Schlüssel (samt Wert).
- `bool contains(const Key& key)`
  Prüft, ob der gegebene Schlüssel existiert.
- `Value get(const Key& key)`
  Liefert den Wert, der zum Schlüssel gehört, zurück.
- `bool empty()`
  Prüft, ob Elemente enthalten sind.
- `string printMap()`
  Erstellt einen String, der den Inhalt enthält.

# Kapitel 6

# Suffix-Arrays

## 6.1 Zielsetzung

Ziel war es, einen funktionsfähigen Algorithmus zur Erstellung von Suffix-Arrays zu implementieren, der im Externspeichermodell effizient ist. Dieser Algorithmus sollte mit einem entsprechenden internen Algorithmus verglichen werden. Dafür musste ein interner Algorithmus auf XAVER portiert werden.

## 6.2 Interner Algorithmus

Als internen Algorithmus zum Vergleichen haben wir den DC3-Algorithmus von Karkäinnen und Sanders [14] benutzt. Dieser Algorithmus dient gleichzeitig als Grundlage für den weiter unten vorgestellten externen Algorithmus.

Der DC3-Algorithmus benutzt einen rekursiven Ansatz, um Suffix-Arrays zu erstellen. Er teilt die Suffixe in 2 Gruppen auf: Diejenigen Suffixe, die an einer Position modulo 3 kongruent zu 0 starten ($A_0$) und die übrigen Suffixe ($A_{12}$).

Für $A_{12}$ wird rekursiv ein Suffix-Array erstellt. Dafür werden jeweils 3 Buchstaben logisch zu einem Tripel zusammengefasst. Dies geschieht implizit und ohne Rechenaufwand. Die Positionen aus $A_{12}$ werden mit einem 3-Stufen RADIXSORT lexikographisch sortiert, wobei jeweils das erste Tripel eines Suffixes als Vergleichswert benutzt wird.

Danach wird ein neues Alphabet generiert, das einen Buchstaben für jedes in $A_{12}$ vorkommende Tripel hat. Hierfür werden die Tripel in sortierter Reihenfolge durchlaufen.

Mit Hilfe dieses Alphabets wird ein neuer String ($S_{12}$) generiert, der erst alle Tripel enthält, die an einer Position modulo 3 kongruent zu 1 sind und danach alle Tripel, die modulo 3 kongruent zu 2 sind. Statt der Tripel wird der dem Tripel zugeordnete Buchstabe abgespeichert.

Da die Tripel in sortierter Reihenfolge durchlaufen werden, ändert sich die Ordnung der Suffixe nicht. $S_{12}$ hat eine Länge von $\frac{2}{3}n$, wenn $n$ die Länge des Strings ist. Das Alphabet kann sich allerdings deutlich vergrößern. Die Länge von $S_{12}$ ist eine obere Schranke für die Alphabetgröße.

Für $S_{12}$ und damit gleichzeitig für $A_{12}$ wird rekursiv ein Suffix-Array ($SA_{12}$) erstellt.

Als nächstes wird für $A_0$ ein Suffix-Array ($SA_0$) erstellt. Hierbei wird ausgenutzt, dass die relative Ordnung der Suffixe modulo 3 kongruent zu 1 bekannt ist. Ein Suffix aus $A_0$ kann dargestellt werden als sein erstes Zeichen gefolgt von dem nachfolgenden Suffix, von dem die Sortierung bereits bekannt ist.

Um $SA_0$ zu erhalten, wird $A_0$ zuerst nach den Suffixen aus $SA_{12}$ sortiert und anschließend mit einem RadixSort nach den Anfangsbuchstaben der Suffixe. Der RadixSort ist stabil.

Zuletzt müssen $SA_0$ und $SA_{12}$ zusammengefasst werden. Beim Vergleich eines Suffixes aus $SA_0$ mit einem Suffix aus $SA_{12}$ wird ausgenutzt, dass die Suffixe zerlegt werden können in ihr erstes Zeichen und den nachfolgenden Suffix, falls der Suffix aus $SA_{12}$ modulo 3 kongruent zu 1 ist, bzw. in ihre ersten beiden Zeichen und den darauffolgenden Suffix sonst.

Die Ordnung der bei dieser Zerlegung benötigten Suffixe ist jeweils aus $SA_{12}$ bekannt.

Die Rekusion kann abgebrochen werden, wenn die Alphabetgröße der Stringlänge entspricht, da dann jeder Buchstabe eindeutig ist und das Suffix-Array durch Sortieren der Buchstaben erzeugt werden kann.

## 6.3 Externer Algorithmus

Der externe Algorithmus ist eine angepasste Version des oben vorgestellten DC3-Algorithmus. Zuerst wird die generelle Umsetzung des Algorithmus beschrieben. Danach werden Verbesserungen vorgestellt, die in einem Engineering-Schritt nachträglich eingebaut worden sind, um die Laufzeit zu verbessern.

Da der generelle Algorithmus bereits bekannt ist, werden wir hier die Umsetzungen der einzelnen Schritte nur punktuell erläutern.

### 6.3.1 Erzeugen von Suffix-Arrays

An vielen Stellen erhalten wir Suffix-Arrays, benötigen aber eigentlich das Reverse-Array. Der interne Algorithmus zur Erzeugung des Suffix-Arrays ist trivial.

Extern geschieht dies, indem an das Suffix-Array die Index-Werte annotiert werden, also Werte, die von 0 bis n-1 aufsteigend geordnet sind. Wenn man jetzt das Suffix-Array zusammen mit den Indexwerten nach den Positionen der Suffixe sortiert, kann man an den Index-Werten das Reverse-Array ablesen. Zum Sortieren kann der NaturalNumberSort aus der Xaver Template Library verwendet werden.

### 6.3.2 Generieren des Strings für den rekursiven Aufruf

#### Tripelgenerierung

Die Tripelgenerierung, die im internen Algorithmus nur implizit erfolgte, muss hier explizit durchgeführt werden. Beim Sortieren müssen zum Vergleich alle 3 Buchstaben des Tripels zur Verfügung stehen. Dies wird dadurch gewährleistet, dass die 3 Buchstaben zu einer Struktur zusammengefasst werden. In dieser Struktur wird der Index explizit mit abgespeichert, da er später benötigt wird.

Nebenher wird auch das Array $A_0$ erstellt. Dies ist eigentlich nicht notwendig, erspart aber später den kompletten String noch einmal zu lesen.

#### Realphabetisieren

Zum Erstellen des neuen Alphabets müssen die Tripel erst lexikalisch sortiert werden. Hier kann ein RadixSort eingesetzt werden. Dieser steht aber in der Xaver Template Library nicht zur Verfügung, daher wird der dort verfügbare MergeSort benutzt.

Danach kann durch einen Scan das neue Alphabet bestimmt werden. Es werden die neuen Buchstaben zusammen mit den Index-Werten in ein neues Array geschrieben. Dieses wird anschließend nach den Indexwerten sortiert, um die Buchstaben in die richtige Reihenfolge für den rekursiven Aufruf zu bringen.

Abschließend wird durch einen Scan der String für den rekursiven Aufruf erstellt, die Index-Werte werden nicht mehr benötigt. Danach folgt der rekursive Aufruf.

### 6.3.3 Erzeugen von $SA_0$

Nachdem im rekursiven Aufruf das Reverse-Array für die Positionen modulo 3 kongruent zu 1 berechnet wurden[1], kann das Suffix-Array für die Positionen modulo 3 kongruent zu 0 berechnet werden.

Hierzu werden an das Array $A_0$ noch die Positionen der Suffixe, die an Positionen modulo 3 kongruent zu 1 starten und Index-Werte annotiert. Danach kann das entstandene Array nach Buchstaben und reversen Positionen sortiert werden.

An den Index-Werten kann danach das Suffix-Array $SA_0$ abgelesen werden.

### 6.3.4 Zusammenfassen von $SA_{12}$ und $SA_0$

Um die beiden Suffix-Arrays zusammenzufassen, muss entschieden werden können, welcher Suffix lexikalisch kleiner ist. Hierbei wird immer ein Suffix aus $SA_{12}$ und ein Suffix aus $SA_0$ verglichen. Um den Vergleich analog zu dem internen DC3-Algorithmus ausführen zu können, müssen alle nötigen Informationen an das Suffix-Array annotiert werden.

Da die benötigten Informationen (u.a. die Anfangsbuchstaben der Suffixe und verschiedene Informationen aus dem Reverse-Array) nicht in derselben Sortierung wie das Suffix-Array vorliegen, werden die Informationen zusammen mit Index-Werten an die Suffix-Arrays annotiert, welche danach nach den reversen Positionen sortiert werden, also in Suffix-Arrays überführt werden.

Danach kann das Zusammenfassen durch einen parallelen Scan durch die beiden Suffix-Arrays geschehen.

### 6.3.5 Rekursionsabbruch

Die Rekursion kann wie im internen Algorithmus abgebrochen werden, wenn das Alphabet genauso groß ist, wie der String lang. Es kann ein Suffix-Array erzeugt werden, indem die Buchstaben, an denen Index-Werte annotiert sind, nach Größe geordnet werden. An den Index-Werten kann anschließend das Suffix-Array abgelesen werden.

### 6.3.6 Semi-Interne Berechnung der Tripel

Bei kleinen Alphabeten kann die Tripelberechnung zum Großteil intern erfolgen. Dafür wird eine Lookup-Table angelegt, in dem für jedes mögliche Tripel der neue Buchstabe gespeichert wird. Dies geschieht in 2 Schritten.

Im ersten Schritt wird der String einmal gescannt. Dabei werden die vorkommenden Tripel in der Lookup-Table durch eine 1 markiert. Die anderen Positionen bleiben 0.

Im zweiten Schritt wird in die Zellen der vorkommenden Tripel die Präfixsumme als neuer Buchstabe geschrieben.

Danach wird der String erneut gescannt und mit Hilfe der Lookup-Table der String für den rekursive Aufruf geschrieben.

[1] Das Reverse-Array für die Positionen modulo 3 kongruent zu 2 wurde auch berechnet, wird hier aber nicht benötigt.

Insgesamt kommt man also mit 2 Scans und komplett ohne Sortieraufrufe aus, um den neuen String zu berechnen. Dies funktioniert natürlich nur, wenn die Lookup-Table in den internen Speicher passt.

Im wesentlichen entspricht das vorgehen einem 1-stufigen Radix-Sort. Ein 3-stufiger Radix-Sort wäre zwar näherliegend, würde aber entsprechend mehr IO-Zugriffe benötigen.

### 6.3.7 Verkürzung des Strings

#### Idee

Erste Tests des Algorithmus haben ergeben, dass in vielen Fällen in der Rekursion Strings auftreten, deren Alphabetgröße in der Nähe der Stringgröße liegen. Dies liegt daran, dass es wenige relativ lange Sequenzen gibt, die sich wiederholen.

Die wesentliche Idee ist, dass in diesen Fällen die Position der meisten Suffixe bereits durch den Anfangsbuchstaben des jeweiligen Suffixes eindeutig bestimmbar ist. Die Positionen dieser Suffixe durch rekursive Aufrufe zu berechnen, ist unnötig.

#### Umsetzung

Ähnlich dem Rekursionsabbruch wird das Suffix-Array durch Sortieren der Buchstaben erstellt. Buchstaben, die mehrfach vorkommen, sind natürlich noch nicht in der richtigen Reihenfolge. Daher haben wir dieses vorläufige Suffix-Array „pseudo Suffix-Array“ genannt.

Um mehrfach vorkommende Buchstaben zu sortieren, wird ein String generiert, der jede Kopie eines mehrfach vorkommenden Buchstaben enthält und jeden Buchstaben, der hinter einem mehrfach vorkommenden Buchstaben steht. Für diesen String wird rekursiv ein Suffix-Array erzeugt.

Dieses Suffix-Array wird dann mit dem pseudo Suffix-Array zusammengefasst. Dazu müssen zuerst die Anfangsbuchstaben der Suffixe an das Suffix-Array annotiert werden. Danach reicht ein gleichzeitiger Scan durch das Suffix-Array und das pseudo Suffix-Array, um die beiden zusammenzufassen.

Solange die Buchstaben im Pseudo Suffix-Array eindeutig sind, können die Positionen von dort kopiert werden. Sind die Buchstaben nicht eindeutig, so müssen die Positionen aus dem rekursiv erzeugten Suffix-Array kopiert werden. Die annotierten Buchstaben sind nötig, um die Suffixe zu überspringen, die schon eindeutig waren und nur dazu benötigt wurden, die korrekte Reihenfolge sicherzustellen.

## 6.4 Implementierung

Als Datenstruktur, um die Suffix-Arrays zu speichern, wurde die Klasse `SuffixArray` eingeführt. Diese speichert den String, das Suffix-Array, das Reverse-Array und soweit vorhanden die LCP-Tabelle und die LLCP- und RLCP-Werte, die zum Suchen benötigt werden. Sie enthält die Funktionen zum Erstellen der LCP-Tabellen und zum Suchen von Teilstrings.

Die Erstellung des Suffix-Arrays wurde bewusst aus dieser Klasse ausgelagert, um mehrere Algorithmen implementieren zu können. Die Algorithmen werden als statische Methoden implementiert, die als Rückgabewert das fertige Suffix-Array liefern.

### 6.4.1 Beschreibung der Schnittstelle der Klasse `SuffixArray`

- `SuffixArray(xaver::FarPointer<CharType> string, xaver::FarPointer<StringLen> suffixArray, xaver::FarPointer<StringLen> reverseArray, StringLen stringLen)`
  Konstruktor der Klasse. Benötigt als Parameter den Eingabestring, dessen Länge, das Suffix-Array und das dazugehörige Reverse-Array.

- `void createLCPArray()`
  Diese Methode erstellt die LCP Tabelle. Wir haben hier einen in [10] beschriebenen Algorithmus verwendet.

- `void createLLCPandRLCP()`
  Diese Methode erstellt die LLCP- und die RLCP-Tabelle. Auch hier wurde ein in [10] beschriebener Algorithmus verwendet. Vorher muss `createLCPArray` aufgerufen werden.

- `void search(XRan searchString)`
  Sucht den String `searchString` im zum Suffix-Array gehörenden String. Die Fundstellen werden ausgegeben. Vorher müssen `createLCPArray` und `createLLCPandRLCP` aufgerufen werden.

- `void destroy()`
  Gibt den Speicher für das Suffix-Array, das Reverse-Array und soweit vorhanden für die LCP-Tabellen wieder frei. Der Speicher für den String wird nicht freigegeben, da er noch benötigt werden könnte.

### 6.4.2 Algorithmen zur Erstellung von Suffix-Arrays

Zum Erstellen eines Suffix-Arrays gibt es zwei verschiedene in Xaver implementierte Algorithmen: Den von uns implementierten Algorithmus und einen Vergleichsalgorithmus.

### 6.4.3 SuffixArrayGenerator

Diese Klasse beinhaltet den von uns implementierten Algorithmus.

#### Beschreibung der Schnittstelle

- `SuffixArray<StringLen, CharType> generateArray(ExternalString<CharType, StringLen> string)`
  Erstellt ein Suffix-Array. Als Parameter benötigt es den Eingabestring `string`. Er darf das Zeichen 0 nicht enthalten. Das Array für den String muss um 2 Zeichen länger sein, als der eigentliche String. Diese Positionen müssen mit 0 gefüllt sein.

#### Implementierungsdetails

Der generelle Algorithmus wird weiter vorne im Abschnitt „Externer Algorithmus“ beschrieben. Ein paar Anmerkungen zur konkreten Implementierung sind hier aufgeführt.

**Templates:** Alle verwendeten Methoden sind mit Template-Parametern versehen. Dies ermöglicht, immer den kleinsten passenden Datentypen zu wählen. Der Ursprungsstring wird z.B. mit `char` als Zeichentyp auskommen, während die rekursiv erzeugten Strings zum Teil deutlich größere Alphabete haben.

Da sich der Datentyp im Laufe des Algorithmus ändert, ist die Hauptmethode `generateArray` in mehrere Methoden aufgeteilt worden – jeweils mit einer Fallunterscheidung bezüglich der Template-Parameter.

**Rückgabetypen:** Dort, wo die Methoden mehr als einen Wert zurückgeben müssen, werden Strukturen zurückgegeben, die alle nötigen Werte enthalten.

**Arrays:** An vielen Stellen werden Arrays über mehrere Werte benötigt. Dafür sind jeweils eigene Strukturen definiert worden.

**Comparatoren:** Die Comparatoren entsprechen dem STL-Standard. Da an verschiedenen Stellen das selbe Attribut in unterschiedlichen Strukturen verglichen werden muss, erhalten viele Comparatoren die Struktur als Template-Parameter.

Die Comparatoren implementieren alle den Vergleichsoperator der STL. Dort, wo es sinnvoll ist, ist auch der Operator implementiert, der zu einem Objekt die endgültige Position im Array zurückgibt, wie es der NaturalNumberSort aus der XTL erfordert.

**Sortierverfahren:** Als Sortierverfahren kommen der MergeSort und - wo möglich - der NaturalNumberSort aus der XTL zum Einsatz.

### 6.4.4 SuffixArrayGeneratorNaiv

#### Beschreibung der Schnittstelle

- `SuffixArray<StringLen, CharType>`
  `generateArray(ExternalString<CharType, StringLen> string)`
  Erstellt ein Suffix-Array, hat das selbe Interface wie der um Als Parameter benötigt es den Eingabestring `string`. Er darf das Zeichen 0 nicht enthalten. Das Array für den String muss um 2 Zeichen länger sein als der eigentliche String. Diese Positionen müssen mit 0 gefüllt sein. Der Zeichentyp muss gross genug gewählt werden, um auch die in den Rekursionsschritten erstellten String mit größerem Alphabet aufnehmen zu können, da der Algorithmus den Speicherplatz für die Rekursionen wiederverwendet. Daher sollte sicherheitshalber Chartype genausogroß sein wie StringLen.

#### Die Funktionsweise

Um einen Vergleichsalgorithmus zu haben, haben wir die Beispielimplementierung, die in [14] vorgestellt wurde, in das von `SuffixArrayGenerator` vorgegebene Klassengerüst eingefügt und hat deshalb auch auch dieselben Template-Parameter obwohl der Zeichentyp fest ist.

Der Algorithmus wurde an die Bedürfnisse von Xaver angepasst. Hierfür war es lediglich nötig, die C-Pointer durch Farpointer zu ersetzen und die Speicherallozierung mit `new` durch Allozierungen mit `newExternalArray` aus Xaver zu ersetzen.

## 6.5 Mögliche Verbesserungen

### 6.5.1 Pipelining

Wir haben uns bemüht, den Code möglichst modular zu gestalten und Teilaufgaben in einzelne Methoden zu verschieben. Dies hat zum Teil Auswirkungen auf die Laufzeit.

Es passiert häufig, dass eine Methode eine Sequenz in den Externspeicher schreibt, die sofort danach wieder von der nächsten Methode gelesen wird. Durch eine engere Verzahnung der Methoden könnten diese IO-Zugriffe vermieden werden.

### 6.5.2 Effizientere Sortierverfahren

An vielen Stellen setzen wir noch den externen MergeSort aus der XTL ein, obwohl ein RadixSort auch funktionieren würde, wie z.B. beim Sortieren der Tripel. Ein Austausch des Sortierverfahrens würde zumindest die interne Laufzeit verringern, wie dies schon durch Einsatz des NaturalNumberSort aus der XTL geschehen ist.

# Kapitel 7

# Vergleichsstudien

Um den Erfolg der PG zu messen, haben wir unsere auf den externen Speicher optimierten Algorithmen und Datenstrukturen gegen Standard-Algorithmen und -Container antreten lassen. Die Tests umfassten zum einen die XTL-Klassen, die sich gegenüber der STL behaupten mussten, zum anderen selbstverständlich unsere Implementierung der Suffix-Arrays, die gegenüber einer auf `FarPointer` umgestellten Version aus [14] getestet wurde. Die Ergebnisse zeigen eigentlich durchweg, dass es sich lohnt, intelligent mit den reservierten Blöcken umzugehen.

Wir haben unsere Studien jeweils auf den Rechnern im PG-Pool durchgeführt, um eine optimale Vergleichbarkeit der Ergebnisse zu gewährleisten. Es handelt sich um Intel Pentium IV Maschinen mit 2,4 GHz, 512 KB Cache, 1GB RAM und Debian 4.0 als Betriebssystem. Unsere Sourcen haben wir mit dem gcc 4.1.2 compiliert. Bei Tests, in denen jeweils nur die Anzahl der Speicherzugriffe gezählt werden musste, liefert Xaver sowohl im Real- als auch im Simulationsmodus identische Ergebnisse. Somit haben wir solche Untersuchungen im schnelleren Simulationsmodus durchgeführt. Lag der Kern der Betrachtung auf den Ausführungsdauern, so wurde der Realmodus gewählt.

## 7.1 Xaver-Allocator

Um die Vergleichsstudien durchführen zu können, mussten wir eine Schnittstelle finden, an der sich die Standard-Datenstrukturen an Xaver andocken konnten.

Zum Einen ist dies mit Hilfe der `FarPointer` möglich: Jede Template-Datenstruktur kann selbstverständlich als Objekttyp der Elemente des Containers einen FarPointer beliebigen Typs auswählen. Somit bekommt Xaver jede Dereferenzierung der Daten mit und hat somit eine Vergleichsmöglichkeit. Diese Variante wurde bei der Analyse des Sortieralgorithmus (Kapitel 7.4) und der Caching-Strategien (Kapitel 7.6) verwendet.

Der große Nachteil hierbei ist allerdings, dass sich der Algorithmus oder Container problemlos „an Xaver vorbei“ noch zusätzlichen Speicher reservieren kann, ohne dass dieser mit in die Analyse eingehen würde. Dies bringt insbesondere bei komplizierteren Datenstrukturen einen unberechtigten Wettbewerbsvorteil. Aus diesem Grunde haben wir uns entschieden, einen `Allocator` für Xaver zu entwickeln.

Ein Allokator ist generell durch ein Interface definiert und kann prinzipiell allen STL-Datenstrukturen als Template-Parameter übergeben werden. Wird kein Allokator explizit angegeben, wird ein Standard-Allokator verwendet, der über `new` und `delete` Speicher anfordert und freigibt.

Mit Hilfe des Allokator-Konzepts war es uns möglich, die Standard-C++ Befehle für das Allozieren und Freigeben von Speicherplatz zu überladen. Dazu wurde

der `XaverAllocator` entwickelt, der die entsprechenden Aufrufe für das Allozieren bzw. Freigeben von Speicher an den `ObjectController` weiter deligiert, der dann Speicher bei XAVER anfordert und verwaltet.

Bei der Umsetzung standen uns allerdings einige Probleme ins Haus. So mussten wir erkennen, dass die GNU-STL bezüglich der Allokatoren sehr fehlerhaft ist. So wurde zwar die Typ-Definition für einen Zeiger-Typ vom Allokator übernommen, dann aber innerhalb des Codes nicht verwendet. Somit war es nicht möglich den normalen C-Pointer durch FarPointer zu ersetzen.

Es war uns aber möglich, eine STL-Implementierung zu finden, die das Allokatoren-Prinzip korrekter umsetzt. Die STDCXX wurde – nachdem sie über zehn Jahre komerziell durch Rogue Wave Software vertrieben wurde – 2005 an die Apache Software Foundation gestiftet, um dort als Open Source Projekt weiter entwickelt zu werden. Mit ihr war es möglich, die Analyse des Trees auf den Allokatoren laufen zu lassen (Kapitel 7.3). Im Bereich von Listen und deren Iteratoren war die Umsetzung jedoch auch problembehaftet, so dass uns für den Fall der Liste nichts anderes übrig blieb, als eine naive Liste, die auf XAVER aufbaut, neu zu implementieren (Kapitel 7.2).

## 7.2 List

### 7.2.1 Test-Szenario

Wegen der in 7.1 beschriebenen Gründe war es notwendig, zum Vergeleich mit der Liste der XTL eine simple doppeltverkette Liste zu implementieren. `NaiveList` benutzt als Zeiger `FarPointer` und allokiert Speicherplatz über XAVER, ist aber ansonsten überhaupt nicht auf Externspeicher optimiert. Ein Knoten der Liste nimmt ein Element auf und hat je einen `FarPointer` auf Vorgänger und Nachfolger. Zum Sortieren wird ein einfacher MERGESORT verwendet, der in situ arbeitet und standardmäßig ab 40 oder weniger Elementen auf Insertionsort umstellt.

Die beiden Listenimplementierungen wurden bezüglich folgender Funktionalitäten verglichen:

- Einfügen
- Sortiern
- Durchlauf durch die Liste

Beim Einfügen läuft ein Iterator durch die Liste und fügt an der aktuellen Position mit Wahrscheinlichkeit 1/3 eine zufällig gleichverteilt gewählte Zahl ein. Ist der Iterator am Ende der Liste angekommen, wird am Beginn der Liste fortgefahren. Sobald die vorgebene Anzahl an Elementen eingefügt wurde, wird das Einfügen beendet.

Diese Liste wird dann mit dem von der Listenimplementierung bereitgestellten Sortierverfahren sortiert.

Die sortierte Liste wird dann mit einem Iterator durchlaufen. Dabei werden alle Elemente der Liste aufaddiert, wobei Überläufe ignoriert werden, da es lediglich darum geht, zu verhindern, dass der Durchlauf den Optimierungen des Compilers zum Opfer fällt. Da der Durchlauf auf der sortierten Liste stattfindet, werden die Elemente nicht mehr in der Reihenfolge durchlaufen, in der sie eingefügt wurden.

Die drei Funktionalitäten wurden dabei unter verschiedenen Variablen untersucht:

- variable Blockgröße
- variable Datenmenge

- variable Größe des internen Speichers

### 7.2.2 Ergebnisse

Bei Variierung der Blockgröße profitiert die XTL-Liste von größeren Blöcken, während die naive Implementierung sogar ausgebremst werden kann, wie in Abbildungen 7.1, 7.2 und 7.3 zu sehen ist. Während die XTL-Liste bei geringen Blockgrößen durch den entstehenden Overhead der naiven Liste unterlegen ist, nivelliert sich dieser Abstand und kehrt sich mit weiter steigender Blockgröße sogar um.

Dieses Verhalten der naiven Liste wird vermutlich durch die geringe Größe des internen Speichers verstärkt, ist aber asymptotisch auch bei größerem Speicher zu erwarten. Denn bei zufälliger Verteilung der Elemente über die Blöcke bleibt pro Zugriff die Wahrscheinlichkeit für einen Cache-Miss über alle Blockgrößen gleich. Allerdings müssen bei einem Miss bei größeren Blöcken mehr Daten geladen werden.

Das starke Abfallen der naiven Implementierung bei Blockgrößen jenseits von 2 KB in Abbildung 7.1 lässt sich durch das Testszenario erklären. Die Elemente werden sukzessive vom ObjectController angefordert, dieser legt sie immer in einem Block an, bis er voll ist. Wird nun beim Iterieren dieser Block aus dem Speicher geworfen, muss für jede neue Anforderung ein IO-Zugriff erfolgen.

Beim Durchlaufen der sortierten Listen in Abbildung 7.3 zeigt sich, dass sich die Optimierung der XTL-Liste auf genau diesen Fall sowohl bei IO-Zugriffen als auch bei der reinen Rechenzeit auszahlt. Der Abstand zwischen beiden Implementierungen wächst sogar mit zunehmender Blockgröße.

Bei der Variierung der Datenmenge und Wahl einer Blockgröße von 512 Bytes, die sich für die XTL-Liste als nicht optimal erwiesen hat, zeigt sich in den Abbildungen 7.5 und 7.6, dass sie der naiven Implementierung dennoch in den Fällen überlegen ist, auf die sie optmiert ist: Sortieren und vor allem Durchlaufen.

In Abbildung 7.4 fällt auf, dass ab einer Datenmenge von 2 MB die benötigte Zeit stärker steigt, während die Steigung der IO-Zugriffe nicht ändert. Diese Unterschiede zwischen reinen IO-Zugriffen und Laufzeit ergeben sich vermutlich dadurch, dass ab dieser Datengröße eine weitere Speicherhierarchiestufe des verwendeten Computers verwendet wurde, da sie sowohl die naive als auch die optimierte Implementierung betreffen.

Bei Veränderung des zur Verfügung stehenden internen Speichers zeigt sich, dass sich die XTL-Liste bis aufs Sortieren invariant gegenüber seiner Größe verhält. So profitiert in Abbildung 7.7 nur die naive Implementierung von der Vergrößerung des internen Speichers wesentlich.

Beim Sortieren in Abbildung 7.8 profitiert die XTL-Liste sprunghaft ab gewissen Vielfachen der Speichergröße, da sich genau dann vermutlich die benötigte Rekursionstiefe des verwendeten MergeSort Algorithmus um eins reduziert. Die naive Implementierung profitiert ebenfalls, aber nicht sprunghaft, sondern kontinuierlich durch eine Verringerung der Cache-Misses. Insgesamt zeigt sich hier, dass die XTL-Liste um einen Faktor 10 schneller ist als die naive Implementierung.

Dieser Faktor zeigt sich auch beim Iterieren, bleibt dort aber überraschenderweise über alle Speichergrößen konstant, wie in Abbildung 7.9 gezeigt. Die Erwartung, dass sich durch Vergrößerung des internen Speichers die Anzahl der Cache-Misses beim Iterieren über die naive Liste stetig reduziert, erfüllt sich nicht. Trotz des höheren Aufwandes für die Verwaltung reduziert sich ebenfalls die nötige Rechenzeit, um die XTL-Liste zu durchlaufen. An diesem Diagramm kann ebenfalls schön gesehen werden, dass fast alle Operationen im internen Speicher ausgeführt werden, da die benötigte Zeit für die IO-Operationen gegen null geht.

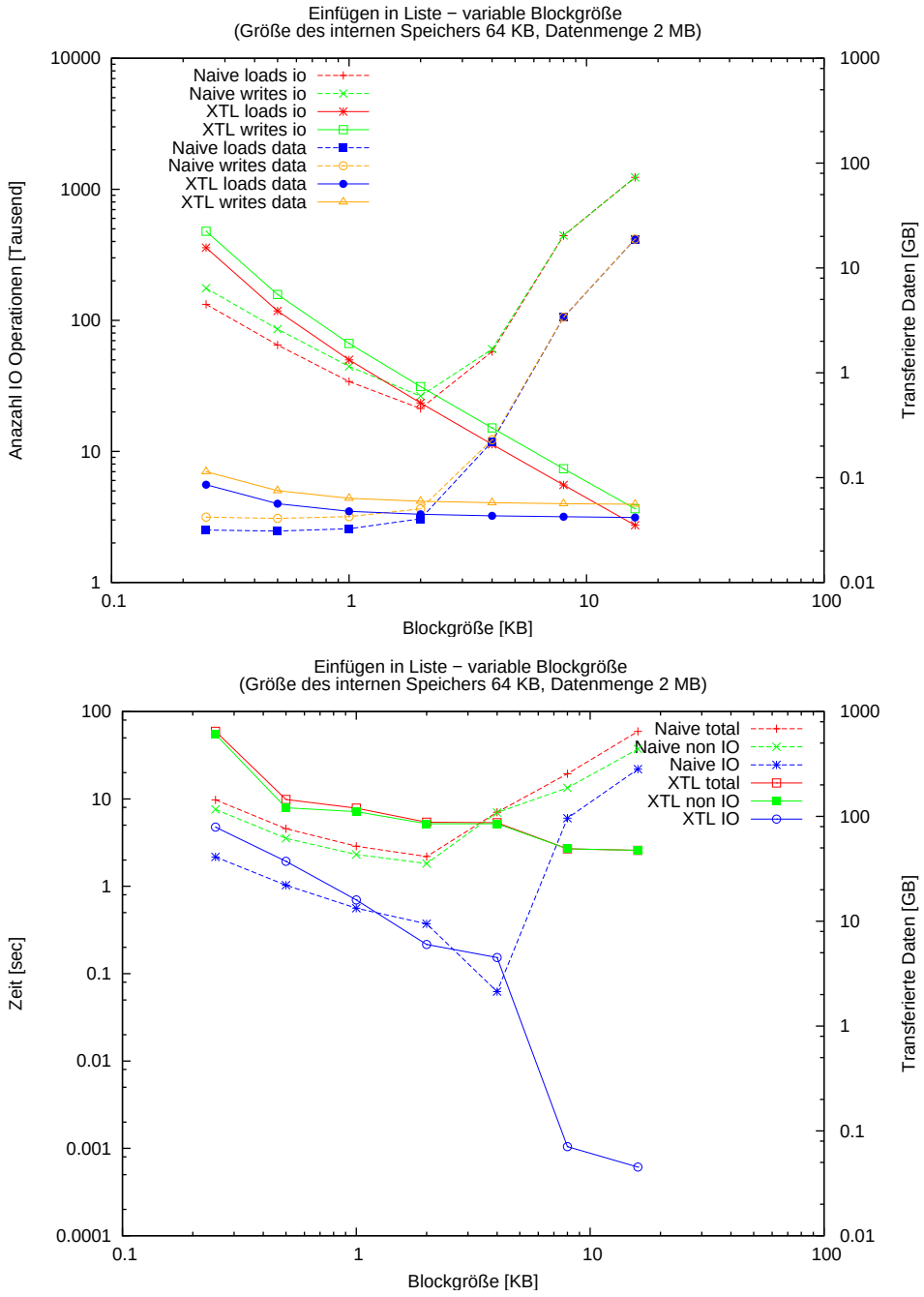


Abbildung 7.1: Liste – Einfügen bei variabler Blockgröße

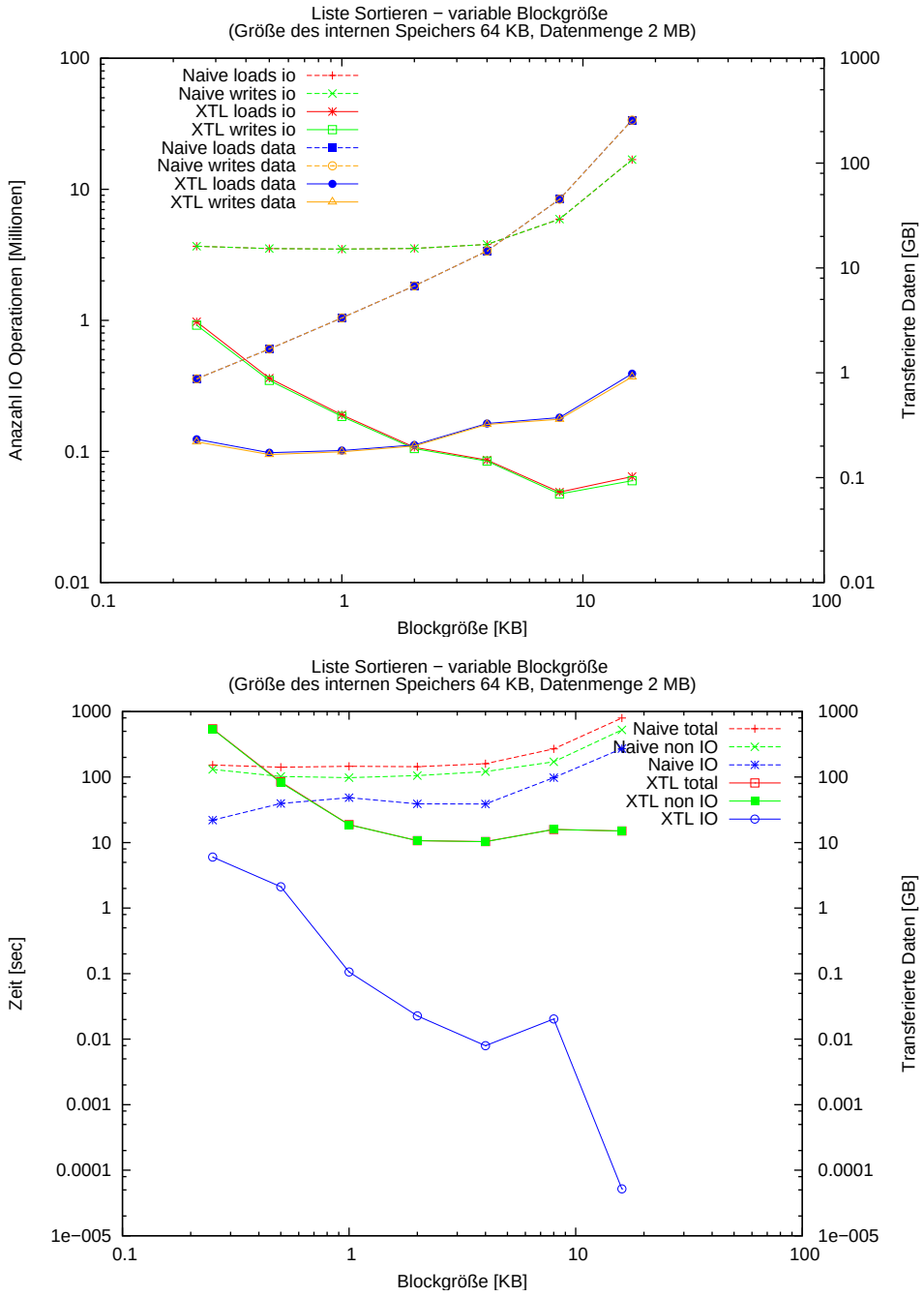


Abbildung 7.2: Liste – Sortieren bei variabler Blockgröße

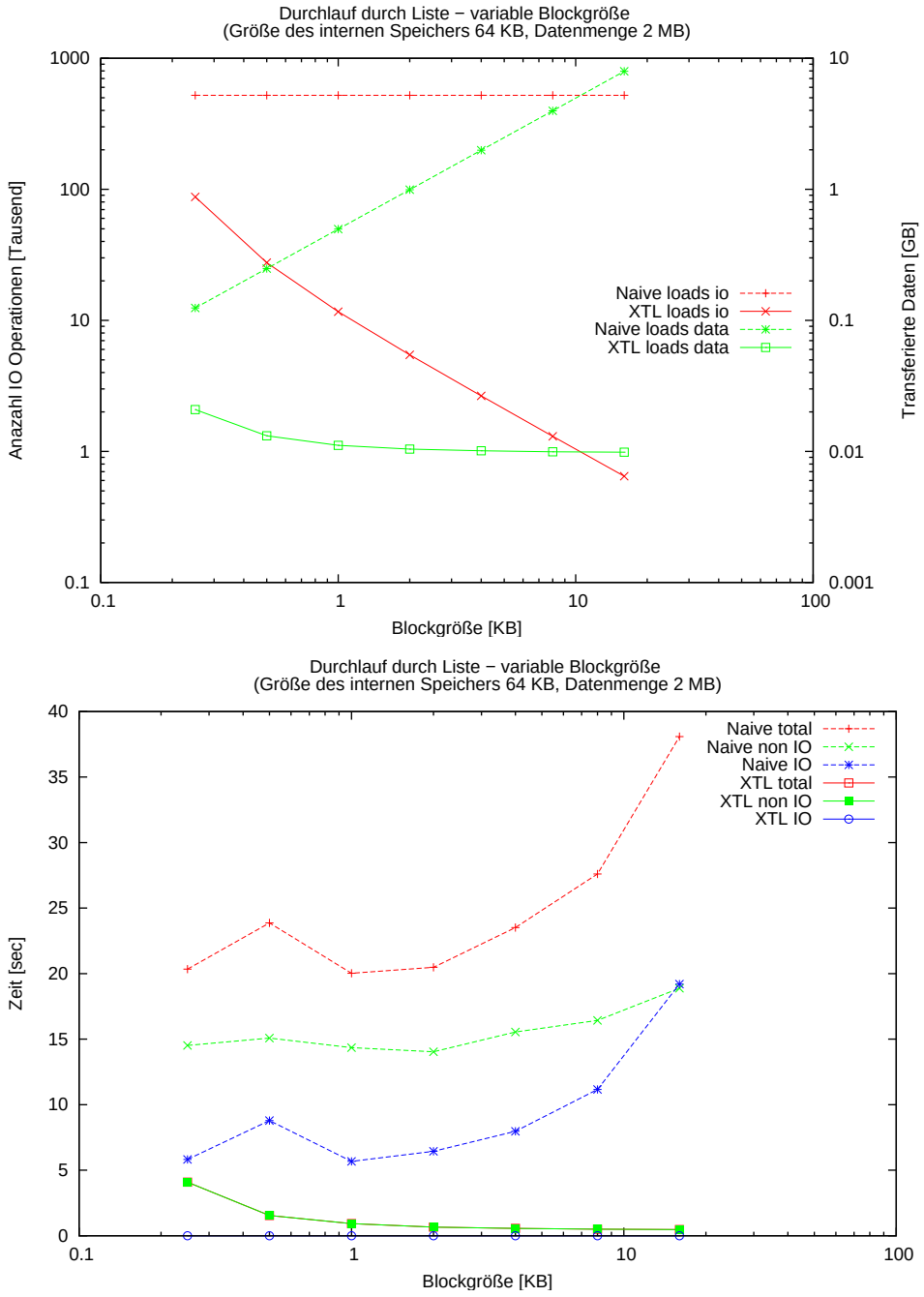


Abbildung 7.3: Liste – Durchlauf bei variabler Blockgröße

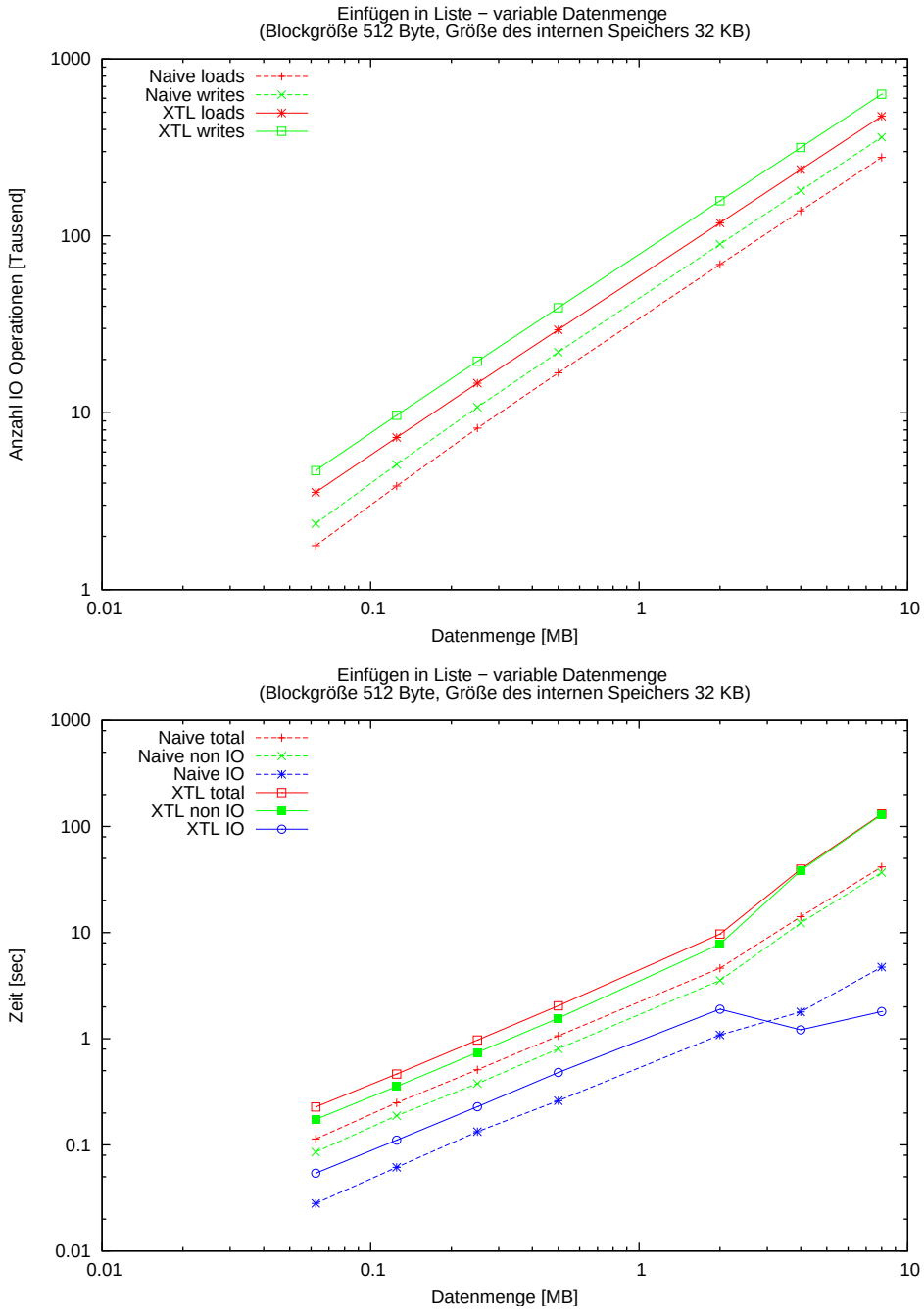


Abbildung 7.4: Liste – Einfügen bei variabler Datenmenge

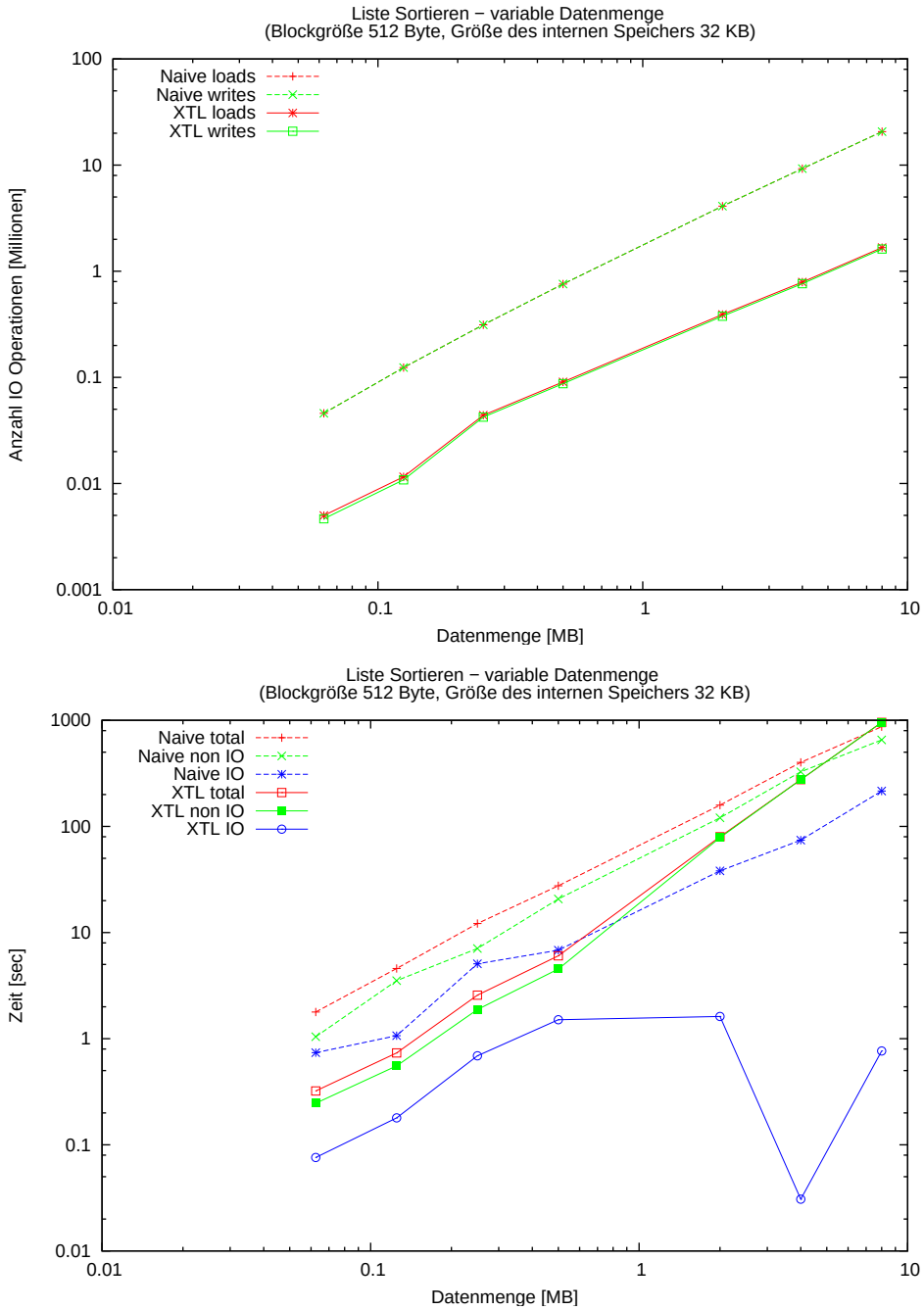


Abbildung 7.5: Liste – Sortieren bei variabler Datenmenge

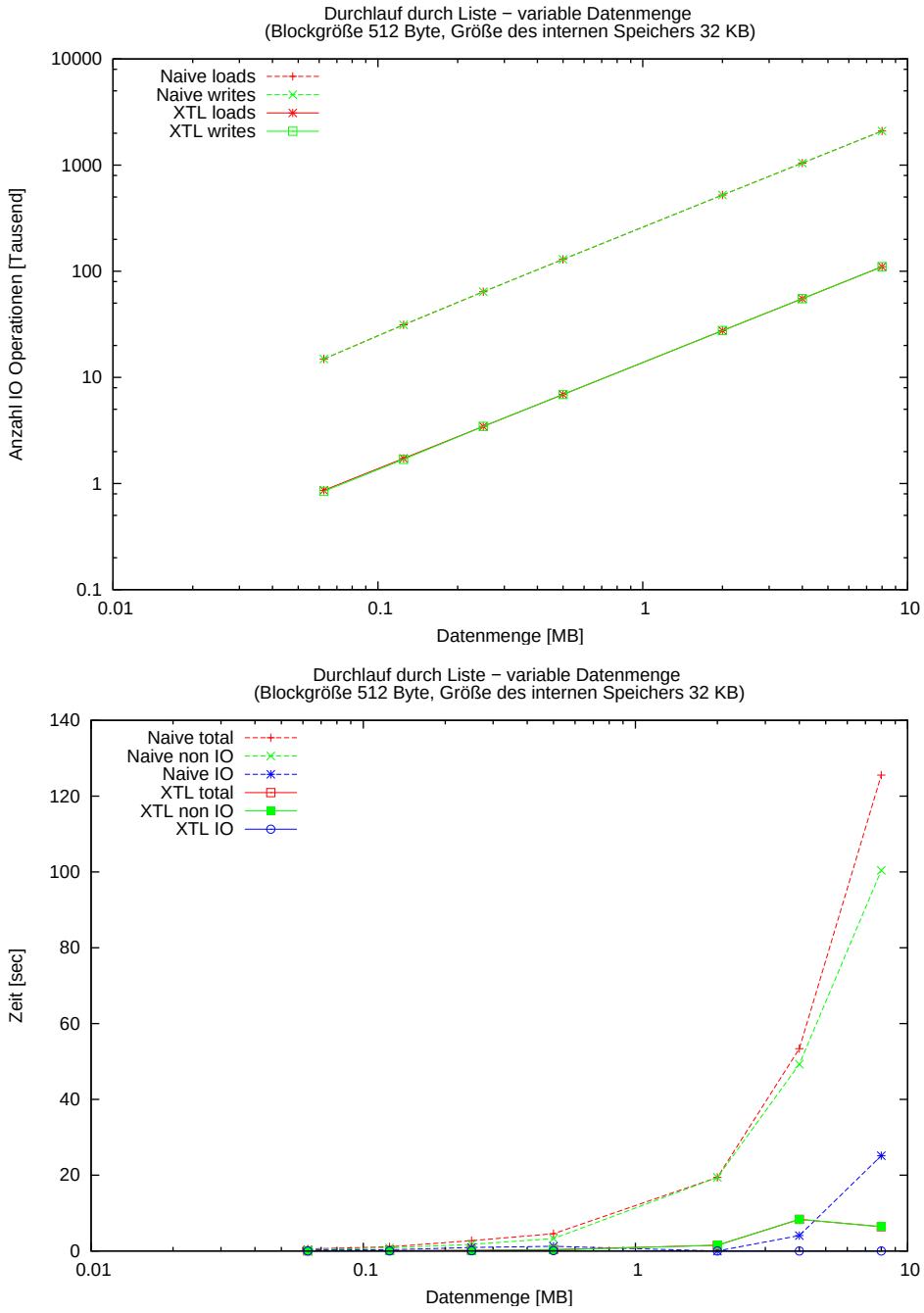


Abbildung 7.6: Liste – Durchlauf bei variabler Datenmenge

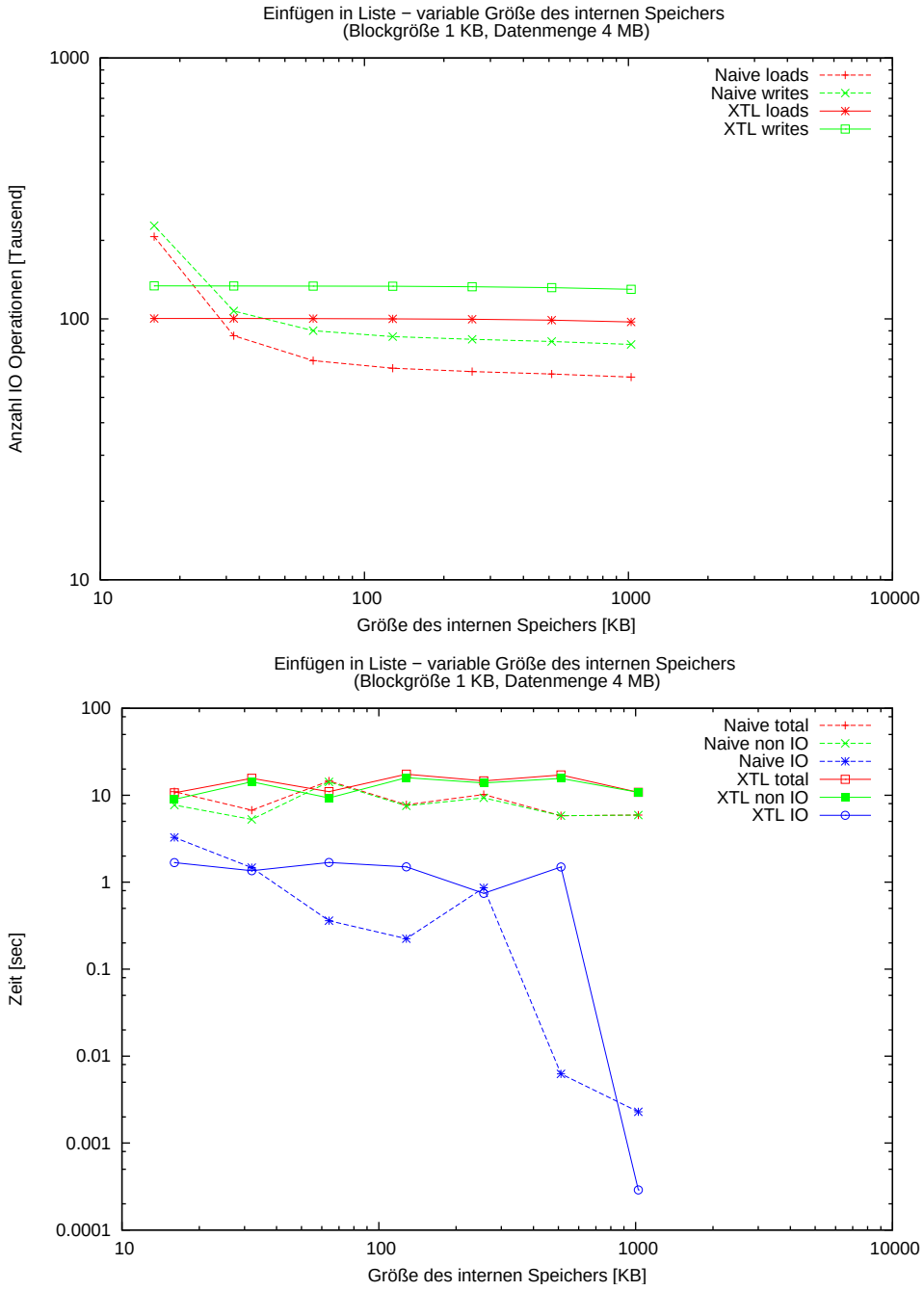


Abbildung 7.7: Liste – Einfügen bei variabler Größe des internen Speichers

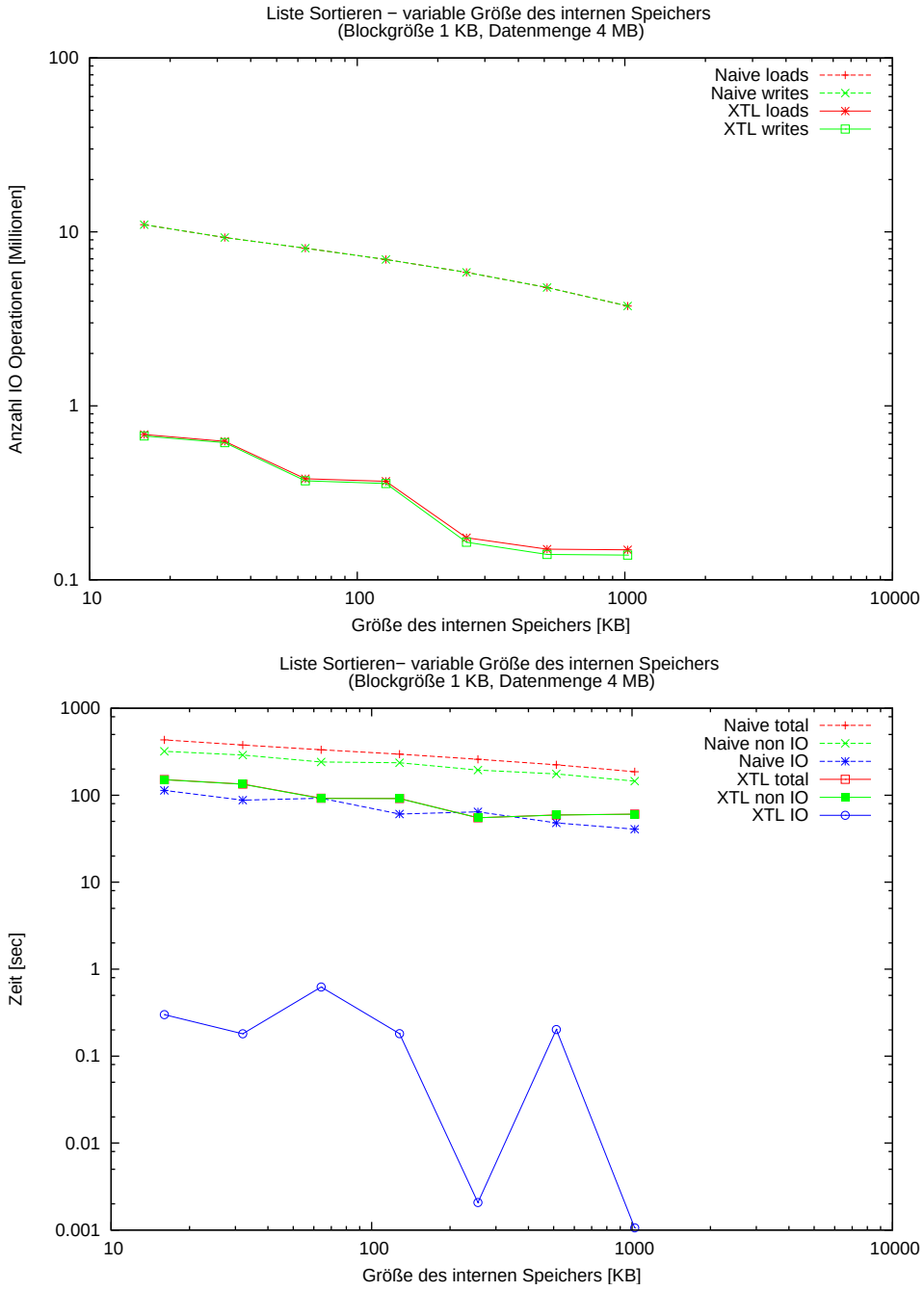


Abbildung 7.8: Liste – Sortieren bei variabler Größe des internen Speichers

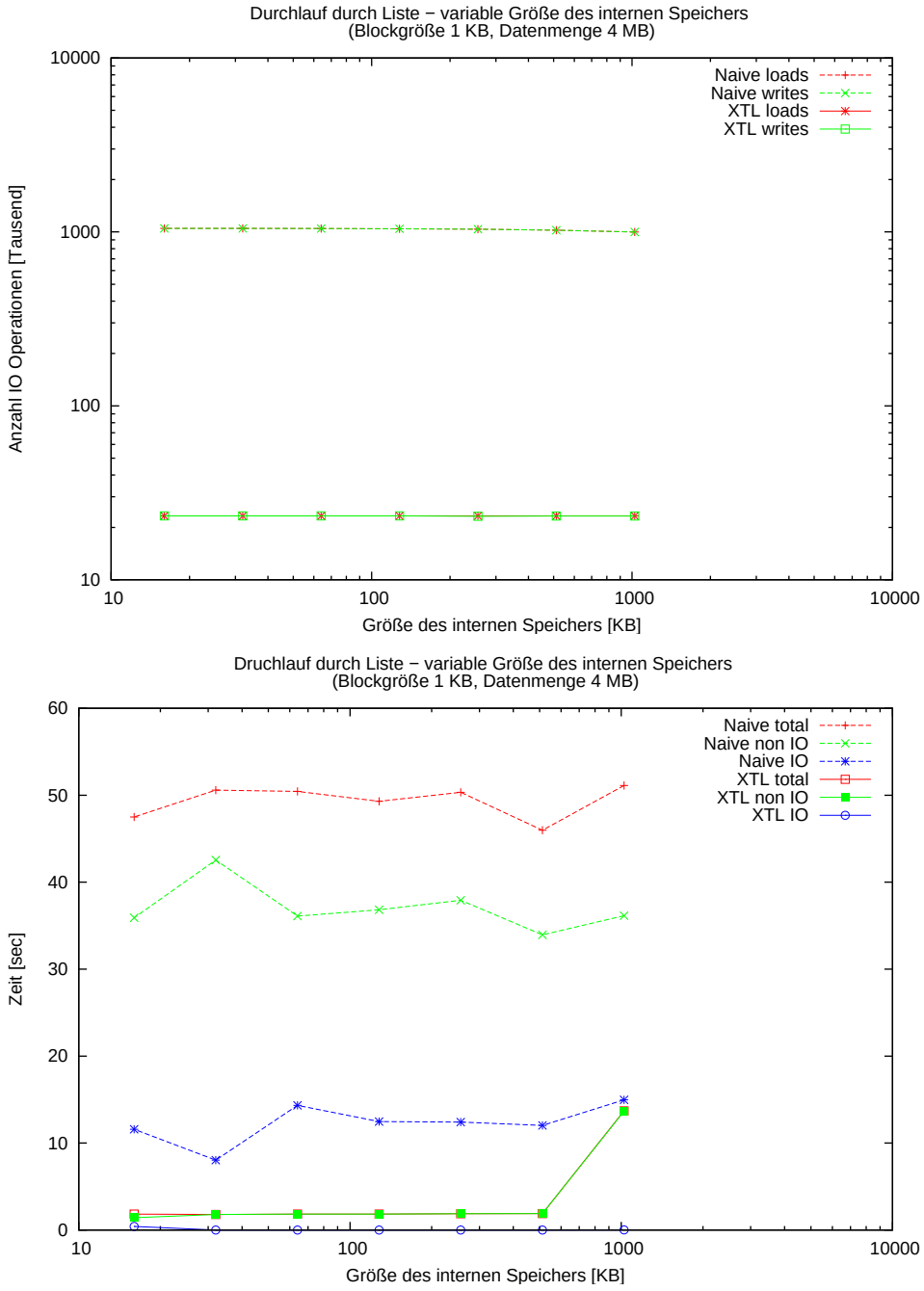


Abbildung 7.9: Liste – Durchlauf bei variabler Größe des internen Speichers

## 7.3 Tree

### 7.3.1 Test-Szenario

Die Datenstruktur des Baumes zeichnet sich insbesondere dadurch aus, dass ein geschicktes Einfügen der Daten in die innere Struktur des Containers eine schnelle Suche der einzelnen Elemente ermöglicht. Aus diesem Grunde ergibt es im Prinzip keinen Sinn, nur das Einfügen von Daten in den Baum zu testen, da sich der hieraus möglicherweise resultierende Overhead positiv auf die Such-Laufzeit auswirkt.

Aus diesem Grund haben wir uns dafür entschieden, ein Szenario aus gemischten Einfüge- und Such-Operationen zu erstellen:

1. Füge ein Element in den Baum ein
2. Merke dieses Element mit Wahrscheinlichkeit $\frac{1}{4}$ für eine spätere Suche
3. Führe nun mit Wahscheinlichkeit $\frac{1}{4}$ eine Suche durch:
   - mit Wahrscheinlichkeit $\frac{1}{2}$ suche ein vorher germerktes Element (erfolgreiche Suche)
   - mit Wahrscheinlichkeit $\frac{1}{2}$ führe eine höchst wahrscheinlich erfolglose Suche nach einem zufälligen Element durch

### 7.3.2 Durchführung und Ergebnisse

Wir haben bei der Vergleichsstudie die STL-`map` gegen die Tree-Datenstruktur der XTL antreten lassen. Der STL-Variante wurde hierbei zur Speicherverwaltung der Xaver-Allokator übergeben.

Bei allen durchgeführten Tests wird deutlich, dass die XTL-Implemenentierung der nicht auf den externen Speicher optimierten Variante überlegen ist. Dies ist insbesondere darauf zurückzuführen, dass diese direkt geschickt innerhalb der von Xaver bereitgestellten Blöcken arbeiten kann, wohingegen die STL-Variante die Möglichkeiten von Xaver nicht ausschöpfen kann.

Als erstes wurde untersucht, wie sich die Performance verhält, wenn Speichergröße, Blockgröße und zu verarbeitende Datenmenge parallel im gleichen Maße erhöht werden. Abbildung 7.10 zeigt, dass die Anzahl der Ladezugriffe linear in den anderen Größen wächst. Dies liegt insbesondere daran, dass die Anzahl der Suchoperationen sich durch Vergrößerung der Datenmenge ebenfalls erhöht hat.

Alle weiteren Tests bestätigen den Vorteil der XTL-Variante. Es ist bei der Betrachtung der Diagramme zu beachten, dass es sich bei den Achsen der Datenmengen immer um logarithmische Skalierungen handelt. Hier bedeutet ein paralleler Anstieg zweier Kurven im Prinzip ein exponentielles Auseinanderwandern der normal skalierten Werte. Abbildung 7.11 zeigt hier, dass sich das Verhältnis von XTL und STL bei steigender Datenmenge immer deutlicher zu Gunsten der externen Variante entwickelt.

Das analoge Verhalten bei veränderter Speicher- und Block-Größe ist in den Abbildungen 7.12 und 7.13 zu erkennen.

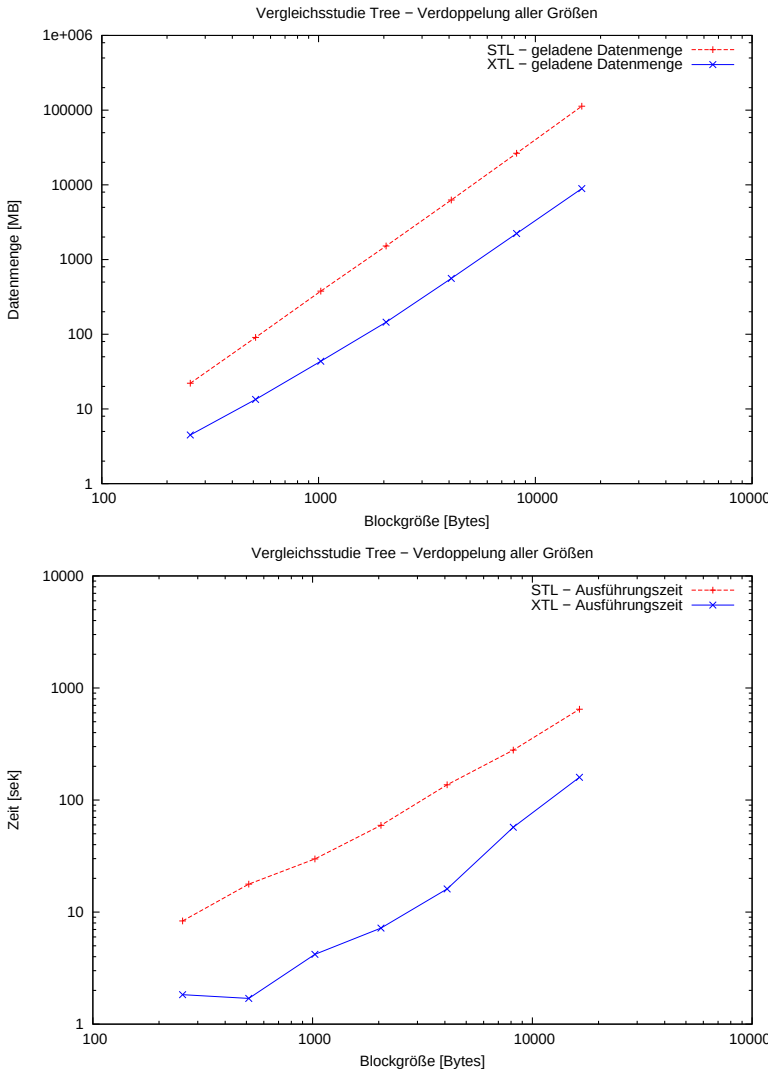


Abbildung 7.10: Tree – Verhalten bei Veränderung aller Hardware-Größen

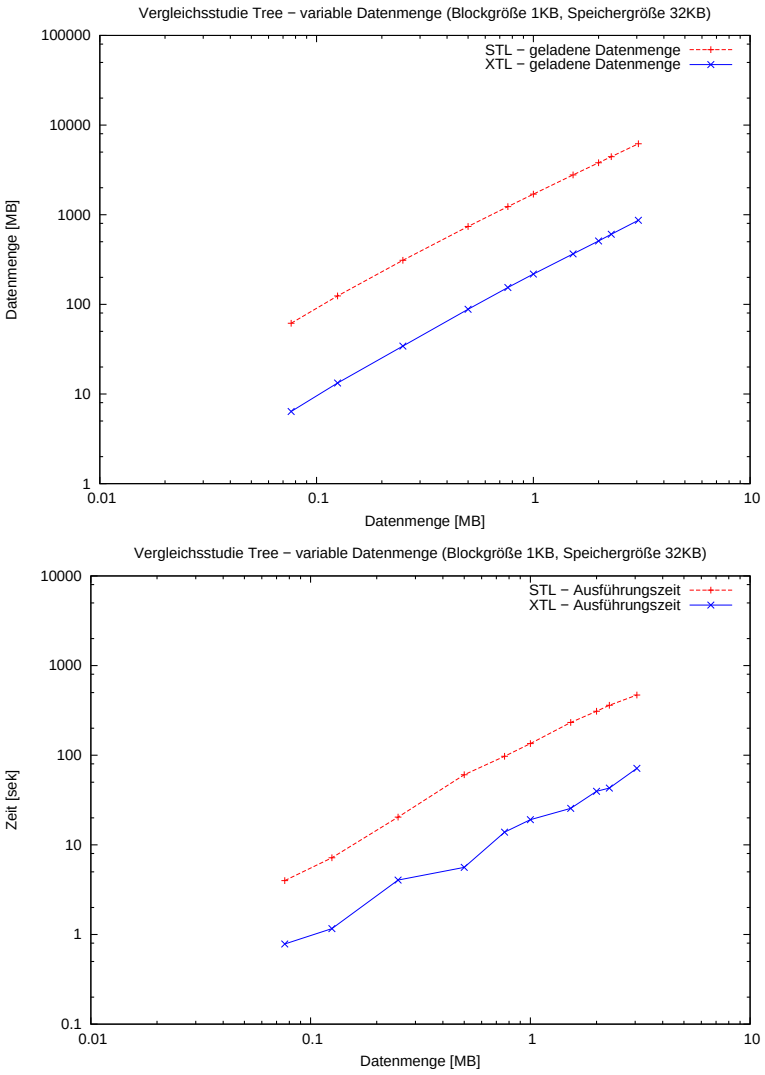


Abbildung 7.11: Tree – Verhalten bei variabler Datenmenge

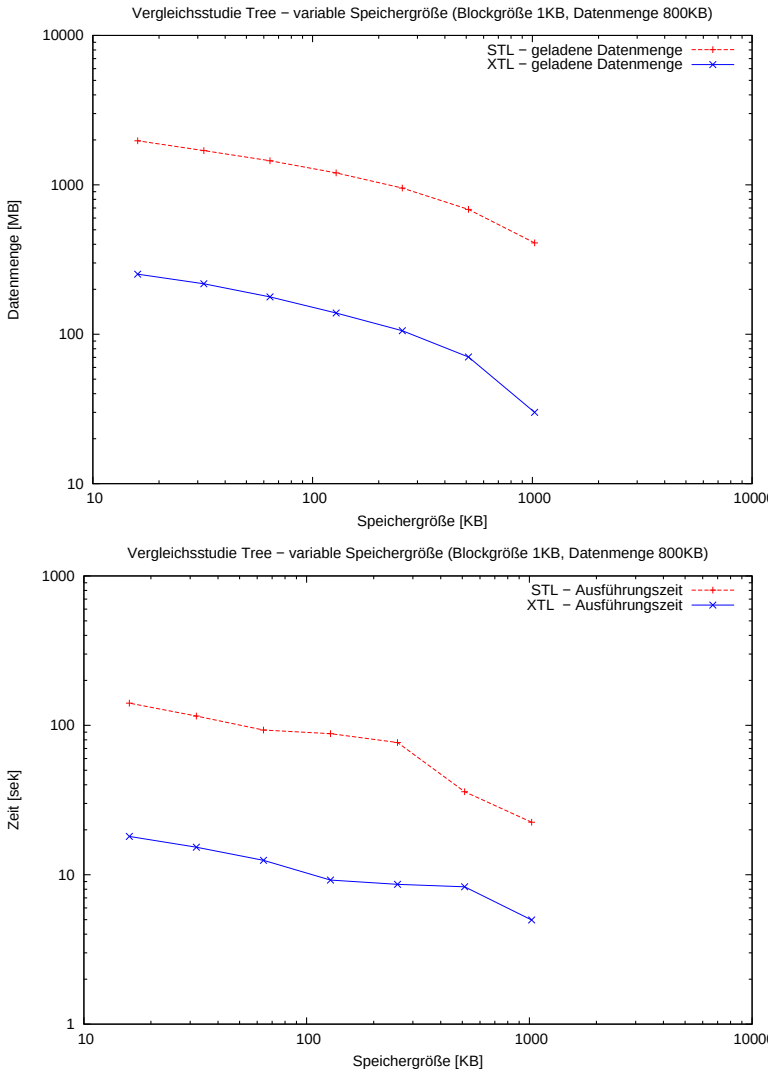


Abbildung 7.12: Tree – Verhalten bei variabler Speichergröße

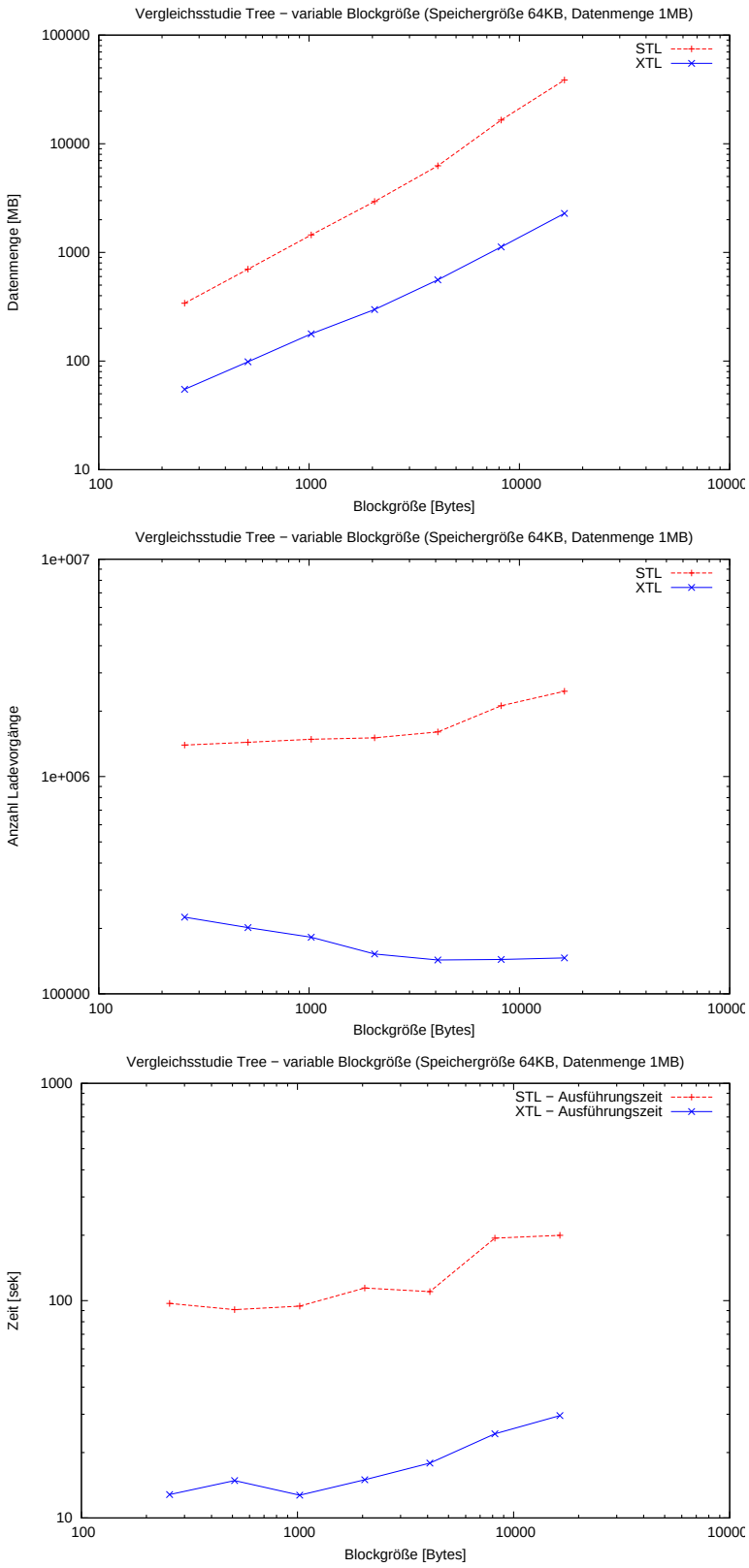


Abbildung 7.13: Tree – Verhalten bei variabler Blockgröße

## 7.4 Sortieren

### 7.4.1 Test-Szenario

Es wird eine variable Anzahl von im Bereich $[-2147483648; 2147483647]$ zufällig gleichverteilten Intergerwerten in ein von XAVER angefordertes externes Array eingefügt. Dieses Array wird dann mit der `sort` Methode der GNU-STL bzw. der XTL sortiert. Dazu wird jeweils ein `FarPointer` auf Anfang und Ende des Arrays, sowie *kleiner* als Vergleichsfunktion, die eine strenge schwache Ordnung realisiert, als Paramter übergeben.

### 7.4.2 Ergebnisse

Betrachtet man die Ergebnisse, so stellt man fest, dass der externspeicheroptimierte Sortieralgorithmus in allen getesteten Fällen den Sortieralgorithmus der STL schlägt. Dennoch gibt es an einigen Stellen Besonderheiten, auf die im Folgenden eingegangen werden soll.

Zunächst wird dabei der Testfall mit variabler Blockgröße betrachtet. Die Ergebnisse sind im Diagramm in Abbildung 7.14 dargestellt. Es ist klar, dass sowohl der Sortieralgorithmus der STL als auch der der XTL von größeren Blöcken profitieren und dann weniger IO-Zugriffe brauchen, da mit einem IO-Zugriff mehr Daten bewegt werden können. Interessanter ist hier deshalb die Betrachtung der Datenmenge, die bewegt wird: Während die Kurve bei der STL nur ganz minimal steigt, steigt die Kurve der XTL immer in sehr deutlichen Sprüngen. Dies ist dadurch zu erklären, dass je größer die Blockgröße ist, um so weniger Blöcke in den internen Speicher passen und der XTL-Sort deswegen mehr Iterationen ausführen muss, bis die Folge sortiert ist. Damit führt hier eine größere Blockgröße zu mehr Durchläufen und damit auch zu mehr bewegten Daten. Gerade bei sehr großen Blockgrößen (und sehr wenigen Blöcken im internen Speicher) ist dieser Effekt sogar so extrem, dass die Laufzeit des XTL-Sort dann sogar schlechter ist.

Betrachten wir nun das Verhalten bei variabler Datenmenge, welches in Abbildung 7.16 dargestellt ist. Wie zu erwarten, steigt die Anzahl der Blockzugriffe hier beim XTL-Sort deutlich langsamer; bei genauer Betrachtung sieht es sogar nach einem asymptotisch langsameren Wachstum aus. Interessanter ist hier das Zeitverhalten: Bei sehr kleinen Datenmengen ist der STL-Sort sogar deutlich schneller als der XTL-Sort, was daran liegen dürfte, dass der XTL-Sort seine Geschwindigkeit im Wesentlichen daraus bezieht, dass er IO-Zugriffe einspart. Gerade bei sehr kleinen Datenmengen gibt es aber kaum IO-Zugriffe, so dass hier kaum Einsparpotential liegt.

Betrachten wir nun abschließend das Verhalten bei variabler Größe des internen Speichers. Auch hier liegt der XTL-Sort deutlich unter dem STL-Sort. Bei der Anzahl der IO-Zugriffe sieht man sehr deutlich die Sprünge, wenn der interne Speicher groß genug ist, so dass der XTL-Sort eine Iteration weniger benötigt. Bezüglich der IO-Zugriffe profitiert der XTL-Sort also sehr deutlich von einem großen internen Speicher mit vielen Blöcken. Dennoch kommt es zu dem verblüffenden Effekt, dass der XTL-Sort auf sehr großem internen Speicher plötzlich wieder langsamer wird. Um diesen Effekt zu erklären, sollte man die konsekutiven Blockzugriffe betrachten und bedenken, dass eine reale Maschine eben nicht nur das von XAVER simulierte zweistufige Speichermodell hat, sondern mehr Schichten besitzt. Der XTL-Sort arbeitet zwar sehr lokal, was das Verhalten innerhalb der Blöcke angeht, der Zugriff zwischen verschiedenen Blöcken ist jedoch relativ wahlfrei, so dass hier kaum konsekutive Zugriffe stattfinden. Der STL-Sort weiß nichts von Blockgrößen und die einzige Möglichkeit, sich hier einigermaßen abzusichern, ist möglichst nur konsekutive Zugriffe vorzunehmen, was er unseren Experimenten zufolge auch macht.

An dieser Stelle kommt nun die Speicherhierachie der realen Maschine ins Spiel (respektive vermutlich der Level-2-Cache dieser): Gibt es im Level-2-Cache nicht genug Blöcke, so dass vom XTL-Sort aus jedem Run ein Block im Level-2-Cache liegt, kommt es dort zu sehr vielen Cache-Misses. Der STL-Sort hingegen arbeitet immer lokal, so dass die Anzahl der Cache-Misses nicht sprunghaft ansteigt. Wird der interne Speicher nun zu groß, verbraucht der XTL-Sort vermutlich viel zu viel Zeit damit, auf das Nachladen der Daten im Level-2-Cache zu warten, so dass bei der Wahl der Größe des internen Speichers immer auch noch die Begebenheiten der Cache-Hierachien der realen Maschine zu beachten sind.

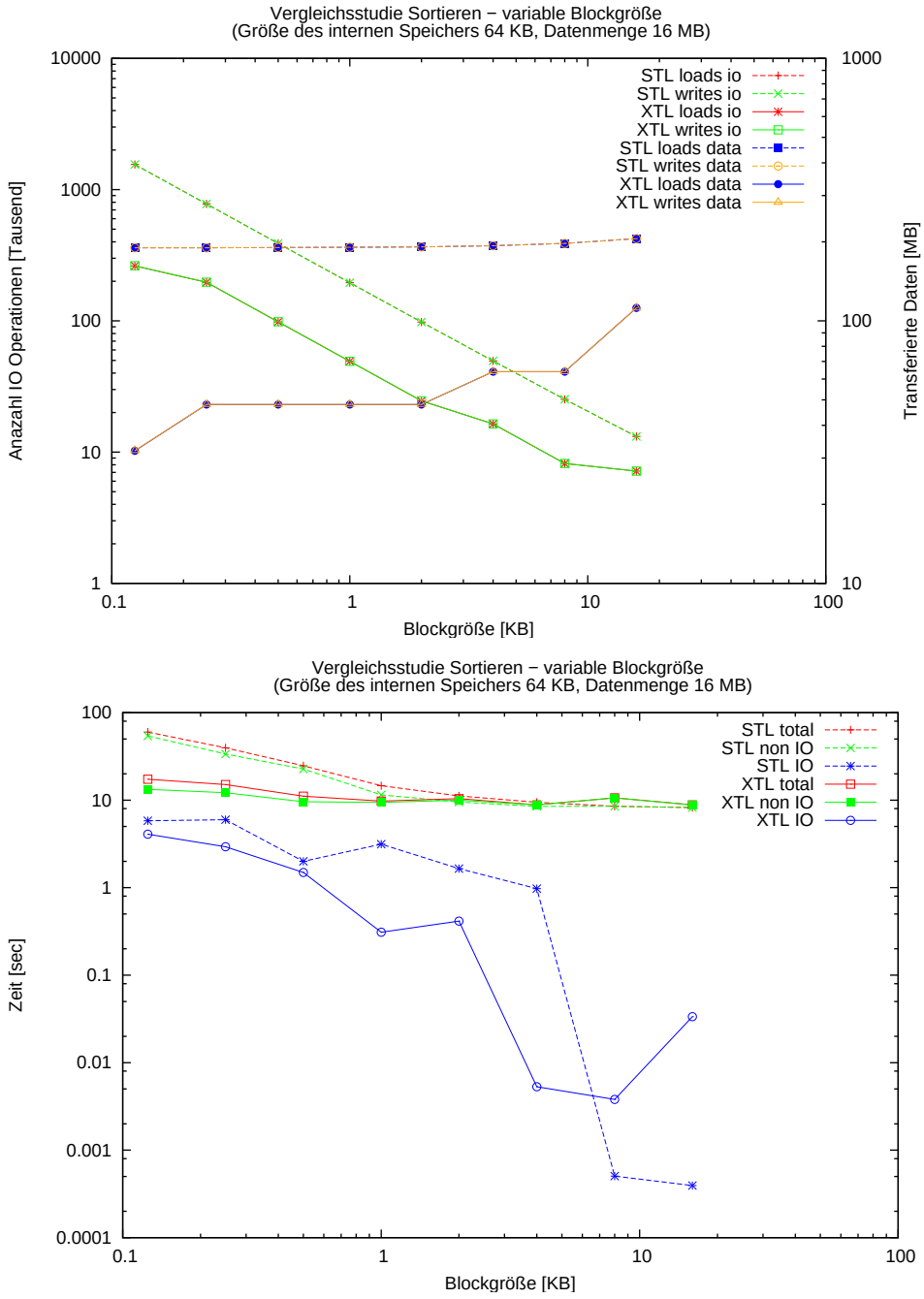


Abbildung 7.14: Sortieren – Verhalten bei variabler Blockgröße

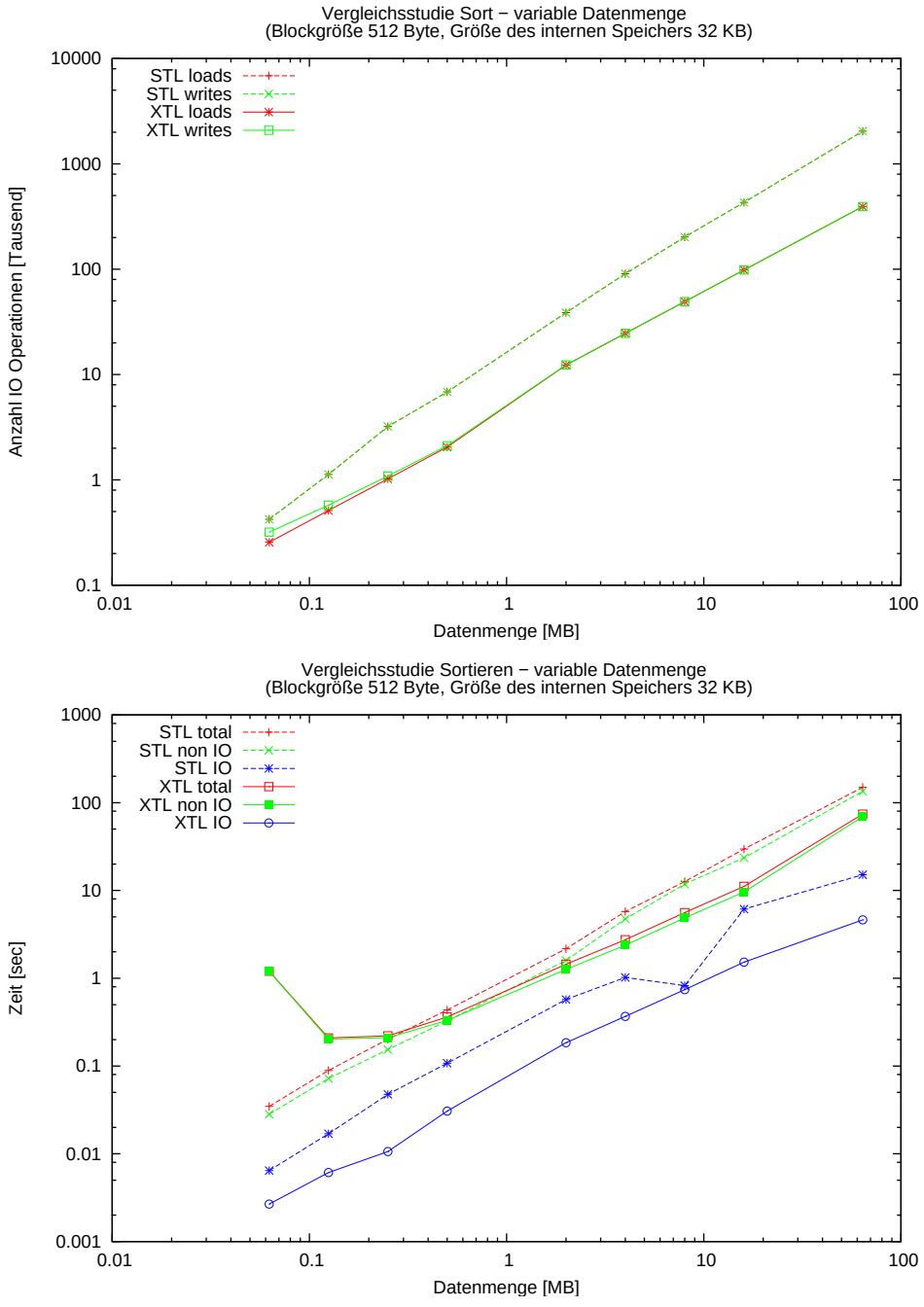


Abbildung 7.15: Sortieren – Verhalten bei variabler Datengröße

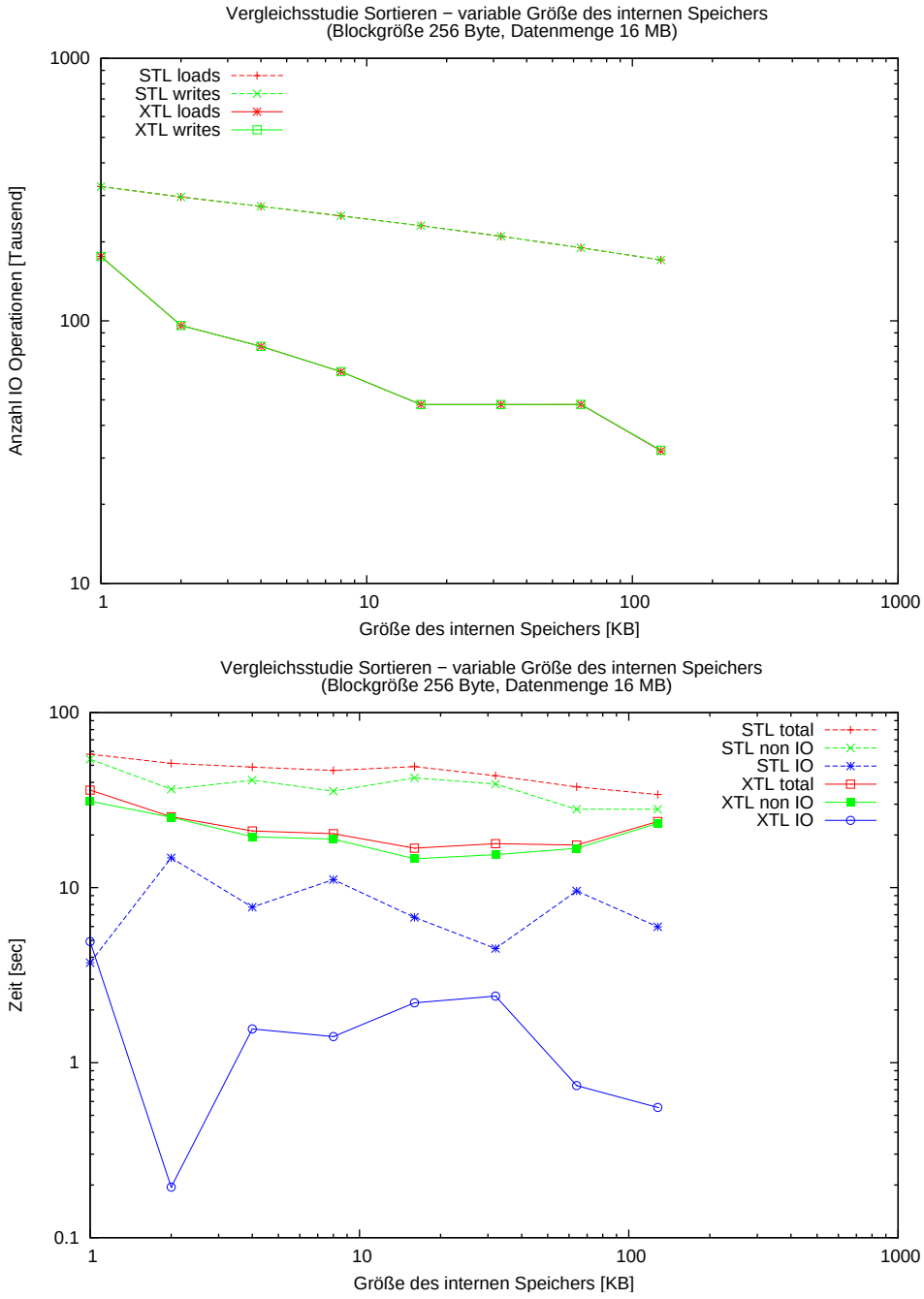


Abbildung 7.16: Sortieren – Verhalten bei variabler Größe des internen Speichers

## 7.5 Suffix-Arrays

### 7.5.1 Test-Szenario

Im Wesentlichen wurde die Erstellung der Suffix-Arrays mit zwei verschiedenen Strings getestet. Zugrunde gelegt wurde einmal das menschliche Genom. Der andere Teststring war die englische Bibel, um eventuelle Unterschiede zwischen Gendaten und Texten herauszufinden. In den Tests wurden jeweils die Anfänge der Strings bis zur angegebenen Länge verwendet. Bei den Gendaten wurden Zeilen ignoriert, die keine Gen-Informationen beinhalten. Es wurden jeweils der interne Algorithmus (Intern), der externe Algorithmus ohne String-Verkürzung (Extern) und der externe Algorithmus mit String-Verkürzung (Extern2) getestet. Auf einen Vergleich der Laufzeiten wurde verzichtet, da der interne Algorithmus derart häufig den Array-Operator benutzt, dass ein Großteil der Laufzeit durch Xaver verursacht wird. Ein fairer Vergleich wäre innerhalb von Xaver nicht möglich.

### 7.5.2 Ergebnisse

Die Tests haben ergeben, dass der externe Algorithmus deutlich weniger IOs benötigt als der interne, sobald die Stringlänge in die Größenordnung des internen Speichers kommt. Wie man in Abbildung 7.17 sieht, bringt der Einsatz der String-Verkürzung in der Praxis zumindest bei den Gendaten einen deutlichen Gewinn. Der Gewinn ist umso größer, je länger der String ist.

Der in dieser Abbildung zu sehende sublineare Anstieg der IO-Operationen liegt vermutlich daran, dass die Rekursionstiefe für den Beispielstring sublogarithmisch ansteigt. Die Rekursionstiefe wächst im wesentlichen logarithmisch mit der Länge der längsten sich wiederholenden Sequenz.

In der Abbildung 7.18 sieht man, dass der interne Algorithmus bei größeren Blöcken immer ineffizienter wird. Die Zahl der IO-Operationen ist fast unabhängig von der Blockgröße, da der interne Algorithmus hochgradig nicht-lokal arbeitet. Der externe Algorithmus ist wesentlich weniger von der Blockgröße abhängig. Der Anstieg der übertragenden Datenmenge erklärt sich durch die hinzukommenden Merging-Phasen der Sortierschritte bei extrem wenigen Blöcken im internen Speicher. Dieser Effekt tritt viel schwächer auf, wenn der interne Speicher nicht auf 1 MB beschränkt wird.

Der interne Algorithmus kann von einer Vergrößerung des Internspeichers kaum profitieren, wie man in Abbildung 7.19 sehen kann. Erst wenn der interne Speicher in der Größenordnung des erstellten Suffix-Arrays liegt, gehen die IO-Operationen deutlich zurück. In diesem Fall kann der interne Algorithmus sogar den externen schlagen, da dieser größere Datenstrukturen verwendet, die noch nicht in den internen Speicher passen. In dieser Abbildung kann man auch sehen, dass die Algorithmen bei Texten (Bibel) weniger IOs brauchen, als bei Gendaten. Dies liegt daran, dass es bei den Gendaten längere Sequenzen gibt, die sich wiederholen und daher mehr Rekursionschritte benötigt werden.

Dieser Effekt ist nur beim internen Algorithmus deutlich sichtbar. Beim externen Algorithmus ist er überhaupt nur dann sichtbar, wenn keine String-Verkürzung verwendet wird. Die String-Verkürzung reduziert die Größe dieser zusätzlichen Rekursionsschritte derart, dass kein Unterschied zwischen den Eingabetypen mehr zu erkennen ist.

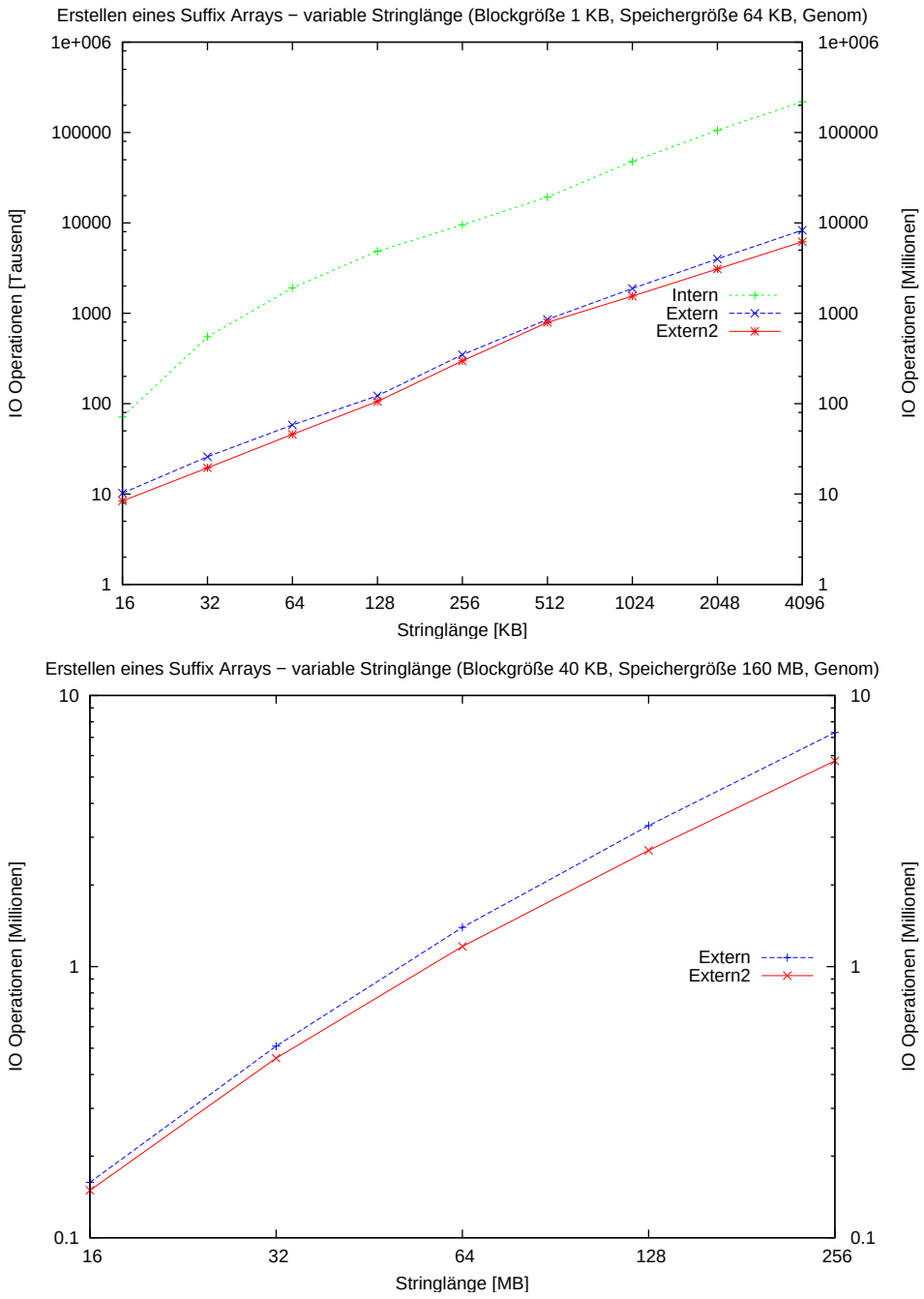


Abbildung 7.17: Suffix-Arrays – Verhalten bei Vergrößerung des Strings

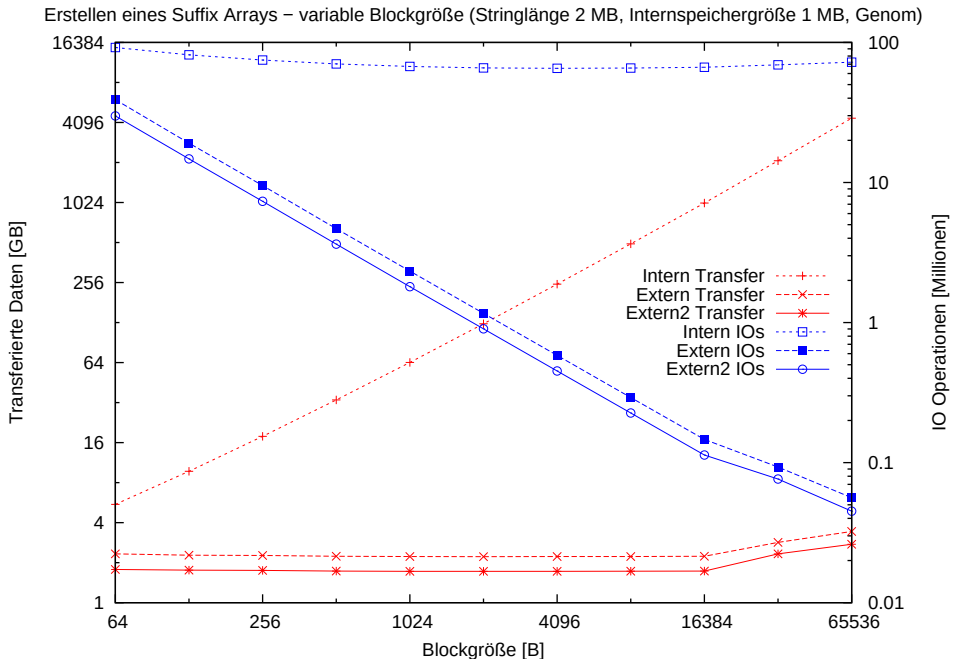


Abbildung 7.18: Suffix-Arrays – Verhalten bei unterschiedlichen Blockgrößen

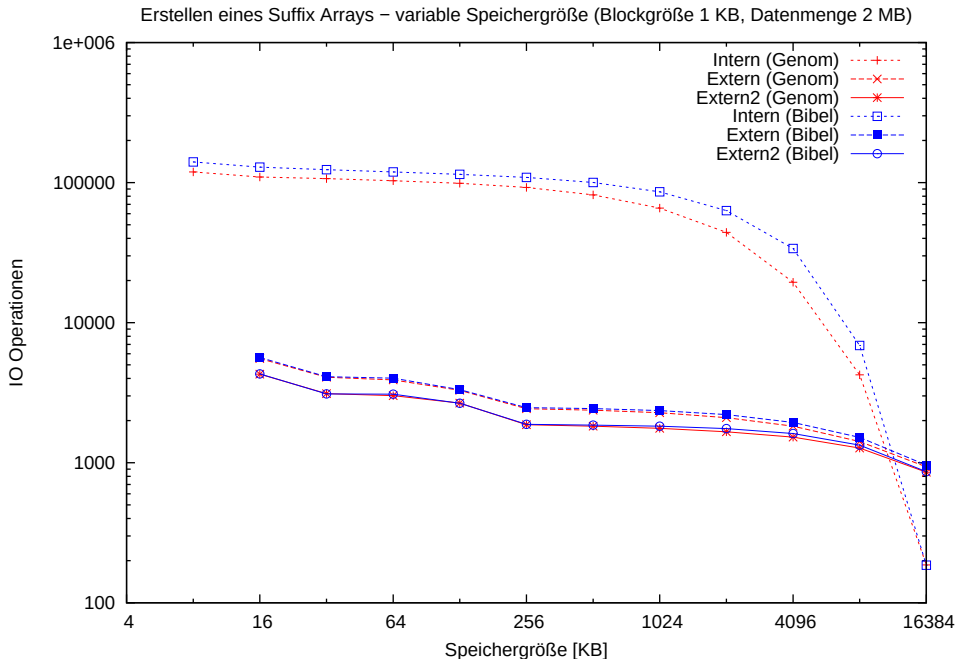


Abbildung 7.19: Suffix-Arrays – Verhalten bei wachsender Internspeichergröße

## 7.6 Caching-Strategien

### 7.6.1 Test-Szenario

Der Test der verschiedenen Caching-Strategien sollte möglichst auf solchen Datenstrukturen und Algorithmen ausgeführt werden, die selbst noch keine Externspeicher-Optimierung enthalten. Somit fallen insbesondere die Datenstrukturen der XTL weg, da diese von den Sperrmechanismen Gebrauch machen und so wichtige Böcke im Speicher halten können und dadurch die Caching-Strategien umgehen.

Wir haben uns deshalb für zwei Testfälle entschieden:

- Sortieren eines Arrays
- Mehrfache binäre Suche nach Elementen innerhalb des sortierten Arrays

Um die Lade-Vorgänge messen zu können, wurde der Speicher über ein FarPointer-Array mit XAVER allokiert. Auf diesem arbeiteten nun die Algorithmen, so dass jede Dereferenzierung von XAVER durchgeführt wurde.

Das konkrete Test-Szenario:

1. Reserviere ein Integer-Array der Größe 32 KB
2. Fülle es mit zufälligen Daten, merke dabei jeden Wert mit Wahrscheinlichkeit $\frac{1}{2}$
3. Sortiere das Array über STL-Sort
4. Suche nacheinander mit binärer Suche alle im zweiten Schritt gespeicherten Werte

Ziel dieser Untersuchung war es nicht, ein umfassenden Ergebnis über die unterschiedlichen Verhaltensweisen der Caching-Strategien zu erhalten. Wir wollten vielmehr ein erstes Gefühl dafür bekommen, wie sich XAVER bei unterschiedlichen Caching-Strategien auf einem alltäglichen algorithmischen Problem ohne Externspeicheroptimierung verhält.

### 7.6.2 Durchführung und Ergebnisse

Wir haben die Caching-Strategien FIFO, LRU, LFU und FIFO-2nd-Chance mit wachsender Speichergröße am obigen Testszenario getestet. Die Analyse unterteilte sich hierbei wie beschrieben in zwei Phasen.

#### Sortieren des Arrays

Die Sortierphase brachte keine großen Aufschlüsse über die Güte der Caching-Strategien. Abbildung 7.20 zeigt, dass fast alle Kurven einen ähnlichen Verlauf beschreiben. Lediglich LFU weicht von der Anzahl der Ladevorgänge ein wenig nach oben ab. Hier bringt also keine der Wahlen einen Vorteil.

#### Binäre Suche im Array

Interessanter wird das Verhalten bei der binären Suche. In Diagramm 7.21 ist auf der $x$-Achse absichtlich die Größe des internen Speichers in Blöcken angegeben. Hier wird deutlich, dass ab einer Speichergröße von $\log(\text{Datenmenge}) = \log(256) = 8$ zwar alle Caching-Strategien deutlich an Performance gewinnen, LRU und FIFO-2nd-Chance allerdings in ganz besonderem Maße. Dies ist darauf zurückzuführen, dass diese den Block des Datum in der Mitte des sortierten Arrays, auf den bei

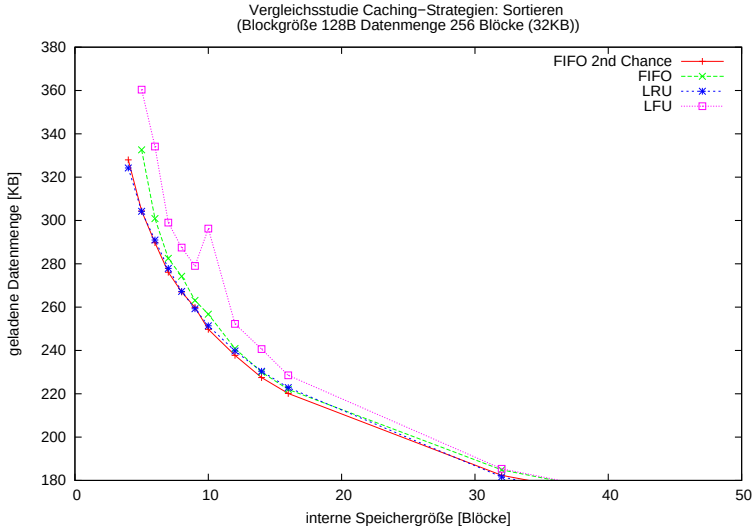


Abbildung 7.20: Caching-Strategien – Verhalten beim Sortieren auf verschiedenen Speichergrößen

jedem neuen Suchdurchlauf auf jeden Fall zugegriffen wird, möglichst im Speicher halten, wohingegen beispielsweise bei FIFO dieser immer wieder verdrängt wird.

Es bleibt also festzuhalten, dass die Güte der Caching-Strategie von der Problemstellung selbst abhängig ist. Allerdings dürften im Normalfall LRU und FIFO-2nd-Chance die besten Ergebnisse liefern.

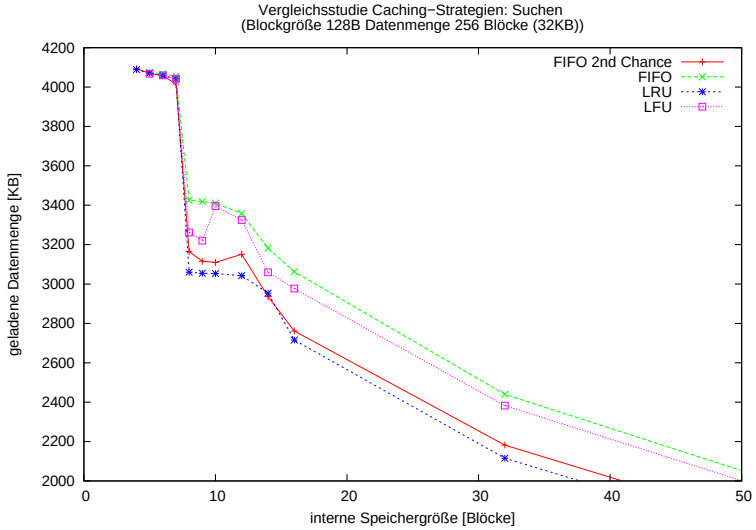


Abbildung 7.21: Caching-Strategien – Verhalten bei binärer Suche auf verschiedenen Speichergrößen

# Kapitel 8

# Fazit und Ausblick

Nachdem ein Mitglied die Gruppe verlassen hatte, wurden die Ursprungsziele der geringen Teilnehmerzahl angepaßt und kürzeste Weg Algorithmen fielen weg. Die verbliebenen Ziele wurden erreicht:

- Die Xaver-Library läuft stabil, verfügt über gute Loggingmethoden zum Bewerten von eingesetzten Algorithmen und stellt alle anvisierten Optionen zur Verfügung.
- Die Xaver Template Library ist umfangreich getestet und stellt Datenstrukturen bereit, die für einige Fälle deutlich schneller als herkömmliche Implementierungen in vergleichbarer Umgebung sind.
- Die Anwendung Suffix-Array-Generator berechnet das Suffix-Array deutlich schneller als herkömmliche Implementierungen, sobald die Datenmengen nicht mehr in den Hauptspeicher passen.
- Es existieren für alle wesentlichen Klassen aus dem Core und der Xaver Template Library sowie für das Suffix-Array Klassentests, um die Funktionalität der Klassen nach Änderungen und Erweiterungen sofort überprüfen zu können und gegebenenfalls Korrekturen vorzunehmen.
- Mittels Doxygen wurde eine Interface Beschreibung generiert, die sich leicht nach Änderungen aktualisieren lässt.
- Es existieren umfangreiche Analysen der Algorithmen hinsichtlich ihrer Laufzeit und ihres IO Verhaltens.

Weitere Verbesserungen sind vor allem bei der IDE-Integration nötig um, Xaver komfortabel verwenden zu können. Eine automatische Dereferenzierung von `FarPointern` und `NearPointer` im Debugger würde das Finden von Fehlern in anzupassenden Algorithmen enorm vereinfachen.

Für weitergehende Verbesserungen auf der Grundlage der Auswertungen der letzten Analysen fand sich leider nicht mehr genügend Zeit, so dass hier sicherlich noch weiteres Optimierungspotential verbleibt.

# Literaturverzeichnis


[1] Mohamed I. Abouelhoda, Stefan Kurtz, and Enno Ohlebusch. Replacing suffix trees with enhanced suffix arrays. *Journal of Discrete Algorithms*, 2:53–86, 2004.

[2] Lars Arge, Gerth S. Brodal, and Laura Toma. On external MST, SSSP and multi-way planar graph separation. In *Proc. 9th Scand. Workshop on Algorithmic Theory, in LNCS 1851*, pages 443–447. Springer, 2000.

[3] Adam Buchsbaum, Michael Goldwasser, Suresh Venkatasubramanian, and Jeffery Westbrook. On external memory graph traversal. In *Proc. 11th Ann. Symposium on Discrete Algorithms*, pages 859–860. ACM-SIAM, 2000.

[4] Andreas Crauser and Paolo Ferragina. A theoretical and experimental study on the construction of suffix arrays in external memory. *Algorithmica*, 32:1–35, 2002.

[5] Roman Dementiev, Juha Kärkäinen, Jens Mehnert, and Peter Sanders. Better external memory suffix array construction. In *Proceedings of ALENEX 05*, pages 171–192, 2005.

[6] Edsger W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[7] Leonor Frias, Jordi Petit, and Salvador Roura. Lists revisited: Cache conscious STL lists. In Carme Alvarez and Maria Serna, editors, *Experimental Algorithms, in LNCS 4007*, pages 121–133. Springer Verlag Berlin Heidelberg, 2006.

[8] Andrew V. Goldberg and Renato F. Werneck. Computing point-to-point shortest paths from external memory. In *Proceedings of ALENEX 05*, pages 26–40, 2005.

[9] The Open Group. The single unix specification, version 2 – threads. http://www.opengroup.org/onlinepubs/007908799/xsh/threads.html, 1997. [Online; Stand 15. Juli 2007].

[10] Volker Heun. Skriptum zur Vorlesung Algorithmen auf Sequenzen. Institut für Informatik; LMU München, 2005.

[11] Martin Holzer, Frank Schulz, and Dorothea Wagner. Engineering multi-level overlay graphs for shortest-path queries. In *Proceedings of ALENEX 06*, pages 156–170, 2006.

[12] Irit Katriel and Ulrich Meyer. Elementary graph algorithms in external memory. In Ulrich Meyer, Peter Sanders, and Jop F. Sibeyn, editors, *Algorithms for Memory Hierachies, in LNCS 2625*, pages 62–84. Springer Verlag Berlin Heidelberg, 2003.

[13] Vijay Kumar and Eric J. Schwabe. Improved algorithms and data structures for solving graph problems in external memory. In *Proc. 8th Symp. on Parallel and Distrib. Processing*, pages 169–177. IEEE, 1996.

[14] Juha Kärkkäinen and Peter Sanders. Simple linear work suffix array construction. In *Proceedings of 30th International Conference on Automata, Languages and Programming, in LNCS 2719*, pages 943–955, 2003.

[15] Anil Maheshwari and Norbert Zeh. A survey of techniques for designing i/o-efficient algorithms. In Ulrich Meyer, Peter Sanders, and Jop F. Sibeyn, editors, *Algorithms for Memory Hierachies, in LNCS 2625*, pages 36–61. Springer Verlag Berlin Heidelberg, 2003.

[16] Kurt Mehlhorn and Ulrich Meyer. External-memory breadth-first search with sublinear I/O. In *Proc. 10th Ann. European Symposion on Algorithms (ESA), in LNCS 2461*, pages 723–735. Springer, 2002.

[17] Kamesh Munagala and Abhiram Ranade. I/O Complexity of Graph Algorithms. In *Proc. 10th Ann. Symposium on Discrete Algorithms*, pages 687–694. ACM-SIAM, 1999.

[18] Robert C. Prim. Shortest connection networks and some generalizations. *Bell Systems Technical Journal*, pages 1389–1401, Nov. 1957.

[19] Robert E. Tarjan and Uzi Vishkin. An efficient parallel biconnectivity algorithm. *SIAM Journal of Computing*, 4(14):862–874, 1985.

# Abbildungsverzeichnis


3.1 LCP-Intervall-Baum für Mississippi . . . . . 15

4.1 Die Kommunikation mit Xaver . . . . . 19
4.2 Innenleben von Xaver . . . . . 20
4.3 Prinzip der Caching-Strategien . . . . . 31
4.4 Die externe Memory-Management-Unit im Überblick . . . . . 32
4.5 Aufteilung eines Blockes im `SimpleObject-Controller` . . . . . 41
4.6 Aufteilung eines Blockes im `QuickObjectController` . . . . . 42

5.1 Schematischer Aufbau einer Liste . . . . . 52
5.2 Löschen eines Elementes . . . . . 59
5.3 Einfügen bei vollem Block . . . . . 59
5.4 Struktur der Queue . . . . . 60
5.5 Aufbau eines Baumknotens . . . . . 63

7.1 Liste – Einfügen bei variabler Blockgröße . . . . . 75
7.2 Liste – Sortieren bei variabler Blockgröße . . . . . 76
7.3 Liste – Durchlauf bei variabler Blockgröße . . . . . 77
7.4 Liste – Einfügen bei variabler Datenmenge . . . . . 78
7.5 Liste – Sortieren bei variabler Datenmenge . . . . . 79
7.6 Liste – Durchlauf bei variabler Datenmenge . . . . . 80
7.7 Liste – Einfügen bei variabler Größe des internen Speichers . . . . . 81
7.8 Liste – Sortieren bei variabler Größe des internen Speichers . . . . . 82
7.9 Liste – Durchlauf bei variabler Größe des internen Speichers . . . . . 83
7.10 Tree – Verhalten bei Veränderung aller Hardware-Größen . . . . . 85
7.11 Tree – Verhalten bei variabler Datenmenge . . . . . 86
7.12 Tree – Verhalten bei variabler Speichergröße . . . . . 87
7.13 Tree – Verhalten bei variabler Blockgröße . . . . . 88
7.14 Sortieren – Verhalten bei variabler Blockgröße . . . . . 91
7.15 Sortieren – Verhalten bei variabler Datengröße . . . . . 92
7.16 Sortieren – Verhalten bei variabler Größe des internen Speichers . . 93
7.17 Suffix-Arrays – Verhalten bei Vergrößerung des Strings . . . . . 95
7.18 Suffix-Arrays – Verhalten bei unterschiedlichen Blockgrößen . . . . . 96
7.19 Suffix-Arrays – Verhalten bei wachsender Internspeichergröße . . . . 96
7.20 Caching-Strategien – Verhalten beim Sortieren auf verschiedenen Speichergrößen . . . . . 98
7.21 Caching-Strategien – Verhalten bei binärer Suche auf verschiedenen Speichergrößen . . . . . 99