



Entwicklung und Benchmarking eines succincten Dateisystems mithilfe von **FLOUDS**

Bachelorarbeit

vorgelegt von:

Sebastian Brunnert

Matrikelnummer: 539546

Studiengang: Informatik

Arbeit betreut durch:

Dr. Dietmar Lammers

Zweitgutachter:

Jun.-Prof. Dr. Florian Kurpicz

Münster, 7. April 2026

Zusammenfassung

Inode-basierte Dateisysteme verwalten Strukturinformationen verteilt auf dem Datenträger mit erheblichem Speicheroverhead. Die FLOUDS-Struktur löst dieses Problem durch den Einsatz paralleler succincter Datenstrukturen, die Ordnerhierarchien darstellen, beschränkte sich bislang jedoch auf lesenden Zugriff und ließ offen, wie Daten konkret auf einem Speichermedium abgelegt werden. Diese Arbeit ergänzt FLOUDS um Änderungsoperationen und implementiert auf dieser Basis ein dynamisches Dateisystem. Als Implementierungsrahmen wird FUSE eingesetzt, das die Entwicklung und Einbindung eines Dateisystems im Linux-Userspace zulässt. Instabile Knotenindizes sind ein zentrales Problem bei der Dynamisierung von FLOUDS, denn das Einfügen oder Löschen eines Knotens verschiebt alle nachfolgenden FLOUDS-Nummern. Um dieses Problem zu lösen, wird die Delta-Stabilisierung entwickelt, ein Mechanismus, der eine Historie von Änderungsoperationen führt und instabile FLOUDS-Nummern bijektiv auf stabile FUSE-Inode-Nummern, die durch das Virtual Filesystem gecacht werden, abbildet. Die Implementierung folgt einer komponentenbasierten Architektur. Als Basis werden zuvor theoretisch eingeführte succincte Datenstrukturen (Bitvektor, Wavelet Tree und verschiedene Namenssequenzen) hinter einer abstrakten Schnittstelle gekapselt. Verschiedene Allokationsstrategien und Metadatenverwaltungsansätze, die für das Benchmarking austauschbar sind, werden implementiert. Das Benchmarking zeigt, dass FLOUDS bei reinen Ordnerstrukturen durch minimale fixe Initialisierungskosten bis zu 97% Speicherplatz gegenüber ext4 einspart. Auch bei realen Datensätzen bleibt der Speichervorteil deutlich messbar. Bei lese-dominanten Workloads kann FLOUDS das ext4-Dateisystem deutlich übertreffen. Die Schreibperformance hingegen stellt die zentrale Schwäche des Dateisystems dar.

Inhaltsverzeichnis

1	Einleitung	1
1.1	Ziel der Arbeit	1
1.2	Aufbau der Arbeit	1
2	Succincte Datenstrukturen	3
2.1	Bitvektor	3
2.1.1	Bitoperationen auf Maschinenwörtern	4
2.2	Dynamischer Bitvektor	4
2.2.1	Ansatz mit balanciertem Suchbaum	5
2.2.2	Ansatz mit verallgemeinertem Teilsummen-Problem . . .	7
2.2.3	Adaptive Dynamic Bit Vector	8
2.3	Wavelet Tree	9
2.3.1	Konstruktion	10
2.3.2	Operationen	10
2.3.3	Praktische Erwägungen	11
2.4	Trie	12
2.4.1	Dynamischer Trie	13
2.5	LOUDS	14
2.5.1	Kodierung	15
2.5.2	Operationen	16
3	Dateisysteme	17
3.1	Physisches Layout	18
3.2	Allokationsstrategien	19
3.2.1	Zusammenhängende Allokation	20
3.2.2	Block-Mapping	20
3.2.3	Extents	21
3.2.4	Spekulative Allokation	21
3.2.5	Delayed allocation	22
3.3	Hierarchie-Ebene	22
3.3.1	Directory Entries	22
3.3.2	Hash Tree Directories	22
3.3.3	Auflösen von Pfaden	23
3.4	Ergänzende Konzepte	23
3.4.1	Journaling	23
3.4.2	Log-structured File System	23

3.5	Nutzer-Schicht	24
3.5.1	FUSE	24
4	FLOUDS	26
4.1	Konstruktion	26
4.2	Navigation	27
4.3	Pfadauflösung und Namenssequenz	28
4.3.1	Lineare Sequenz	28
4.3.2	Hierarchische Organisation	29
4.3.3	Fazit und Handles	31
4.4	Dynamisierung	31
4.4.1	Änderungsoperationen	32
4.4.2	Instabile FLOUDS-Nummern	32
4.4.3	Integration mit File System Manager	35
4.4.4	Multi-FLOUDS	35
4.4.5	Bewegen von Teilbäumen	36
5	Implementierung	37
5.1	Entwicklungsumgebung und Softwarestack	37
5.1.1	Projektstruktur	37
5.1.2	Bibliotheken	38
5.2	Systemarchitektur	38
5.2.1	Succincte Datenstrukturen	38
5.2.2	Computing Component	40
5.2.3	Abstraktionsschicht Block Device	41
5.2.4	Abstraktionsschicht Allocation Manager	41
5.2.5	File System Manager	41
5.2.6	FUSE-Integration	44
5.2.7	Zusammenspiel der Komponenten	45
5.3	Umsetzung der Kernfunktionen	45
5.3.1	Dateioperationen	45
5.3.2	FLOUDS-Nummern	47
5.3.3	Metadaten	47
5.3.4	Serialisierung und Speicherlayout	48
5.3.5	Allokationsstrategien	49
6	Benchmarking	51
6.1	Methodik	51
6.1.1	Vergleichssysteme	51
6.1.2	Workloads	51
6.1.3	Versuchsaufbau und Durchführung	53
6.2	Ergebnisse	53
6.3	Auswertung	54
6.3.1	Speicherbedarf	54

6.3.2	Baseline und FUSE-Overhead	54
6.3.3	Allokationsstrategien	56
6.3.4	Bitvektorimplementierung	56
6.3.5	Namens- und Metadatenverwaltung	57
6.3.6	Delta-Stabilisierung	58
6.3.7	Optimierte Gesamtkonfiguration	58
7	Fazit	60
7.1	Einsatzbereich	60
7.2	Limitationen	61
7.3	Zukünftige Arbeiten	61
	Literaturverzeichnis	62
	Abbildungsverzeichnis	65
	Tabellenverzeichnis	66
	Anhang	67

1 Einleitung

Gängige Dateisysteme wie ext4 oder XFS belegen durch Verwaltungs- und Indexinformationen oft mehr Speicherplatz, als tatsächlich an gespeicherten Informationen bereitgestellt wird [25, Abs. IV]. Gerade für ressourcenbeschränkte oder eingebettete Systeme sind daher besonders schlanke Dateisysteme wichtig. Für Szenarien, in denen nur lesender Zugriff auf statische Daten erforderlich ist, gibt es bereits bewährte Lösungen wie Squashfs [26]. Da die Daten in solchen Systemen komprimiert abgelegt werden, eignen sie sich jedoch nicht für Anwendungen, in denen sich die Dateistrukturen ändern müssen.

Der FLOUDS-Ansatz nutzt sogenannte succincte Datenstrukturen, um die Hierarchien einer Ordnerstruktur effizient abzubilden [20]. Allerdings handelt es sich hierbei ebenfalls um ein reines Read-Only-Dateisystem, und es wird nicht beschrieben, wie Daten konkret auf dem Speichermedium abgelegt werden.

1.1 Ziel der Arbeit

Ziel dieser Arbeit ist es, FLOUDS um Änderungsoperationen zu erweitern und darauf aufbauend ein vollwertiges Dateisystem zu implementieren. Es soll untersucht werden, welche Vor- und Nachteile ein succinctes Dateisystem im Vergleich zu herkömmlichen Dateisystemen bietet. Durch umfassende Benchmarks sollen konkrete Anwendungsfälle identifiziert werden, in denen ein solches Dateisystem besonders geeignet ist. Darüber hinaus sollen Strategien entwickelt werden, um das ursprüngliche System gezielt zu verbessern.

Zur vereinfachten Entwicklung des Dateisystems soll FUSE eingesetzt werden. FUSE ist ein Framework des Linux-Kernels, der es ermöglicht, Dateisysteme im Userspace zu implementieren und diese in das bestehende Kernel-Dateisystem einzubinden, sodass sie wie native Dateisysteme genutzt werden können [10].

1.2 Aufbau der Arbeit

Zunächst sollen im zweiten Kapitel die verwendeten succincten Datenstrukturen beschrieben werden. Dabei werden verschiedene Ansätze zur Implementierung und Konstruktion erklärt und beispielhaft demonstriert. Insbesonde-

re werden der (dynamische) Bitvektor, der Wavelet Tree, der Trie und die LOUDS-Struktur beschrieben.

Im dritten Kapitel wird beschrieben, was Dateisysteme grundsätzlich sind und welche Rolle sie innerhalb eines Betriebssystems einnehmen. Anhand des ext4-Dateisystems sollen gängige Konzepte und Strategien bei der Entwicklung eines Dateisystems beschrieben werden. Ebenfalls soll an dieser Stelle der Aufbau des FUSE-Frameworks erklärt und in diesen Kontext eingeordnet werden.

Das vierte Kapitel widmet sich schließlich der FLOUDS-Struktur. Zunächst wird deren grundlegende Funktionsweise erläutert. Anschließend werden die erforderlichen Algorithmen beschrieben, die notwendig sind, um Änderungen der Ordnerstrukturen zu ermöglichen und die Einbindung in ein Dateisystem vorzubereiten.

Im fünften Kapitel wird die Implementierung des succinthen Dateisystems mithilfe von FUSE beschrieben. Dabei werden zentrale Konzepte der Umsetzung erläutert, insbesondere das Zusammenspiel der FLOUDS-Struktur mit dem Dateisystem-Manager. Zudem wird der grundlegende Projektrahmen vorgestellt und dargelegt, wie Metadaten innerhalb des Dateisystems gespeichert und verwaltet werden. Bei der Implementierung wird insbesondere berücksichtigt, dass einzelne Funktionen gezielt deaktiviert oder alternative Strategien flexibel ausgetauscht werden können.

Das entwickelte Dateisystem wird im sechsten Kapitel einem ausführlichen Benchmarking unterzogen. Zunächst werden dazu umfangreiche Metriken und Workloads definiert. Neben der Bewertung des vollständig integrierten Dateisystems werden auch einzelne Komponenten separat analysiert, um die Ergebnisse fundiert einordnen und erklären zu können.

Abschließend sollen die Ergebnisse zusammengefasst werden und der praktische Anwendungsfall für das implementierte Dateisystem herausgearbeitet werden. Ebenso soll ein Ausblick auf zukünftige Herausforderungen gegeben werden.

2 Succincte Datenstrukturen

Schon in den 1940er-Jahren wurden erste Überlegungen angestellt, wie Daten - und insbesondere Texte - komprimiert gespeichert werden können. In seiner Dissertation „Succinct static data structures“ [13] prägte Guy Jacobson den Begriff der succincten Datenstruktur. Seitdem werden diese platzeffizienten Datenstrukturen kontinuierlich erforscht und es erscheinen regelmäßig neuartige Ansätze und Algorithmen. Das Ziel succincter Datenstrukturen ist es, die Informationen eines strukturierten Datenobjekts (etwa einer Baumstruktur) mit möglichst wenigen Bits und nahe am informationstheoretischen Minimum darzustellen. Gleichzeitig sollen effiziente Operationen zum Zugriff oder zur Navigation über die Datenstruktur bereitgestellt werden [17, S. 2].

Zur theoretischen Analyse dieser Datenstrukturen und Operationen wird oft das Word-RAM-Modell angenommen. Dabei können Wörter der Größe ω in konstanter Zeit geladen und bearbeitet werden, einschließlich arithmetischer oder bitweiser logischer Operationen. Bei einer Problemgröße von n wird üblicherweise eine Wortgröße von $\omega = \Theta(\log n)$ gewählt [17, S. 6].

2.1 Bitvektor

Ein Bitvektor der Länge n ist eine Datenstruktur $B[1..n]$, die n Bits aus $b \in \{0, 1\}$ speichert. Es werden folgende Operationen unterstützt:

- **Access:** Für einen Index $i \in [1..n]$ liefert

$$B.\text{access}(i) = B[i],$$

den Wert des i -ten Bits.

- **Rank:** Für ein Bit $b \in \{0, 1\}$ und einen Index $i \in [1..n]$ liefert

$$B.\text{rank}_b(i) = |\{j \mid 1 \leq j \leq i, B[j] = b\}|,$$

die Anzahl der Vorkommen von b unter den ersten i Bits.

- **Select:** Für ein Bit $b \in \{0, 1\}$ und ein $j \in [1, B.\text{rank}_b(n)]$ liefert

$$B.\text{select}_b(j) = \min\{i \mid 1 \leq i \leq n, B.\text{rank}_b(i) = j\},$$

den Index, an dem b zum j -ten Mal erscheint.

Bitvektoren bilden die Grundlage vieler succincter Datenstrukturen. Je nach konkreter Definition und Implementierung können Details variieren, etwa ob die Indizes bei Rank und Select inklusiv oder exklusiv gezählt werden oder ob die Zählung bei 0 oder 1 beginnt [17, S. 65f.] [14, Abs. 1] [19, Abs. 1] [32, Abs. 2].

Wir betrachten beispielhaft einen Bitvektor der Länge $n = 10$:

$$B = 1011001010$$

Beispiele für Operationen:

- $B.\text{access}(4) = 1$ (das 4. Bit ist 1)
- $B.\text{rank}_0(7) = 3$ (unter den ersten 7 Bits kommen drei 0-Bits vor)
- $B.\text{select}_1(5) = 9$ (der 5. 1-Bit erscheint an Index 9)

2.1.1 Bitoperationen auf Maschinenwörtern

In der Praxis werden Bitvektoren üblicherweise wortweise gespeichert, also als Folge von Maschinenwörtern fester Breite ω (z.B. 64 Bits). Das Bit mit dem Index i befindet sich dabei im Wort $\lfloor i/\omega \rfloor$ an Position $i \bmod \omega$. Diese Darstellung ermöglicht es, SIMD-Operationen für die Beantwortung der Anfragen auszunutzen, indem mehrere Bits eines Wortes gleichzeitig verarbeitet werden. Für die Beantwortung von **rank**-Anfragen wird typischerweise die CPU-Operation der *sideways addition* genutzt, mit der sich die Anzahl gesetzter Bits innerhalb eines Wortes bestimmen lässt. Durch geeignete Bitmasken kann dabei die Zählung auf die relevanten Bits zur angefragten Position eingeschränkt werden. Für **select**-Anfragen wird analog vorgegangen, indem mithilfe paralleler Bitoperationen und kumulativer Zählung zunächst das Maschinenwort identifiziert wird, das das gesuchte j -te Bit enthält, und anschließend dessen exakte Position innerhalb des Wortes bestimmt wird [32, Abs. 3]. Auf manchen Prozessorarchitekturen lassen sich diese Operationen stark optimieren, sodass sie trotz einer theoretischen Laufzeit von $\mathcal{O}(n)$ in der Praxis eine sehr hohe Effizienz erreichen [19, Abs. 5].

Durch geschickt definierte Blockstrukturen lassen sich die Anfragen an den Bitvektor im Rahmen des Word-RAM-Modells auch in konstanter $\mathcal{O}(1)$ -Zeit beantworten. Allerdings entsteht dabei ein - wenn auch minimaler - zusätzlicher Speicheraufwand [14, Abs. 4].

2.2 Dynamischer Bitvektor

Die meisten Blockstrukturen für Bitvektoren sind statisch ausgelegt. Für ein dynamisches succinctes Dateisystem wird allerdings die Unterstützung von Einfüge- und Löschungsoperationen auf dem Bitvektor benötigt:

- **Insert:** Für ein Bit $b \in \{0, 1\}$ und einen Index $i \in [1..n + 1]$ fügt

$B.\text{insert}_b(i)$

b zwischen $B[i - 1]$ und $B[i]$ ein

- **Delete:** Für einen Index $i \in [1..n]$ löscht

$B.\text{delete}(i)$

den Eintrag bei $B[i]$

- **Update:** Für ein Bit $b \in \{0, 1\}$ und einen Index $i \in [1..n]$ setzt

$B.\text{update}_b(i)$

den Bit an Stelle $B[i]$ zu b

Betrachtet man den Bitvektor als Folge von Maschinenwörtern, ist das Einfügen (und analog das Löschen) nur in linearer $\mathcal{O}(n)$ -Zeit möglich. Jedes Wort muss um einen Bit nach rechts verschoben werden und das höchstwertige Bit muss als Überlauf in das nächste Wort übernommen werden.

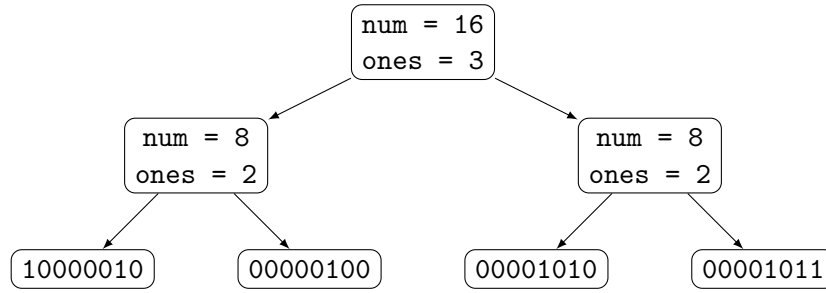
Quelltext 2.1: Einfügen in einem 64-Bit-Wort-basierten Bitvektor (MSB) mit Overflow Behandlung

$W[i] = (W[i] \gg 1) \mid ((W[i-1] \gg 63) \ll 63);$

Es gibt zwei zentrale Ansätze, um einen Bitvektor dynamisch zu gestalten und gleichzeitig eine effiziente Bearbeitung der Operationen zu ermöglichen. Beide Methoden basieren grundsätzlich auf der Idee eines Baumes, unterscheiden sich jedoch in ihrer Umsetzung und Zielsetzung: Der eine Ansatz ist eher praktisch und intuitiv, während der andere das Prinzip auf einer theoretisch verallgemeinerten Grundlage aufbaut.

2.2.1 Ansatz mit balanciertem Suchbaum

Ein dynamischer Bitvektor kann mithilfe eines balancierten, zeigerbasierten Suchbaums (z.B. einem AVL-Baum) realisiert werden, dessen Blätter jeweils Ausschnitte des Bitvektors enthalten. Jedes Blatt enthält dabei $\Theta(\omega^2)$ Bits, während innere Knoten aggregierte Informationen über ihren linken Teilbaum sichern, nämlich die Anzahl der dort gespeicherten Bits (**num**) sowie die Anzahl der gesetzten 1-Bits (**ones**). Diese Zusatzinformationen erlauben es, **access**, **rank** und **select** effizient entlang des Baum-Pfades auszuführen. Für Änderungen werden die Zähler entlang dieses Pfades sowie der im Blatt gespeicherte Bitvektor angepasst. Da jedes Blatt $\Theta(\omega^2)$ Bits speichert, gibt es nur $\mathcal{O}(n/\omega^2)$ Blätter und entsprechend viele innere Knoten. Insgesamt benötigt diese Struktur $n + \mathcal{O}(n/\omega)$ Bits. Anfragen und Updates lassen sich in $\mathcal{O}(\omega)$ -Zeit ausführen,



$B = 10000010 \ 00000100 \ 00001010 \ 00001011$

Abbildung 2.1: Darstellung eines dynamischen Bitvektors als balancierter Suchbaum. Innere Knoten speichern aggregierte Metadaten, während die Blätter die eigentliche Bitsequenz B enthalten.

denn die Navigation im balancierten Baum benötigt $\mathcal{O}(\log n)$ Schritte. Die Operationen auf den Blatt-Bitvektoren können mithilfe von Maschinenwortoperationen ebenfalls in $\mathcal{O}(\omega)$ -Zeit durchgeführt werden [17, S. 450–455].

Beispiele für Operationen:

- $B.\text{access}(10) = 0$

Wurzel $\rightarrow$ linker Teilbaum hat 16 Bits, gehe links $\rightarrow$ Position $10 - 8 = 2$ im rechten Blatt $\rightarrow 00000100.\text{access}(2) = 0$.

- $B.\text{rank}_1(21) = 4$

Linker Teilbaum enthält 3 Einsen $\rightarrow$ im rechten Teilbaum erste $21 - 16 = 5$ Bits $\rightarrow 00001010.\text{rank}_1(5) = 1 \Rightarrow$ insgesamt $3 + 1 = 4$.

- $B.\text{select}_1(4) = 21$

Linker Teilbaum hat 3 Einsen, gehe rechts $\rightarrow$ suche $(4 - 3)$ -te Eins im linken Blatt $\rightarrow 00001010.\text{select}_1(1) = 5 \Rightarrow 16 + 5 = 21$.

- $B.\text{insert}_1(7)$

Füge eine 1 an Position 9 ein. Navigiere zum entsprechenden Blatt (10000010), füge das Bit ein (100000**1**10), aktualisiere **num** und **ones** entlang des Pfades zur Wurzel. Bei Überlauf ($> \Theta(\omega^2)$ Bits) wird das Blatt gespalten und der Baum rebalanciert.

- $B.\text{delete}(5)$

Lösche Bit an Position 5. Navigiere zum Blatt (10000010), entferne das Bit (1000010), aktualisiere **num** und **ones**. Bei Unterlauf ($< \Theta(\omega^2)/2$ Bits) werden benachbarte Blätter verschmolzen und der Baum rebalanciert.

2.2.2 Ansatz mit verallgemeinertem Teilsummen-Problem

Das Problem der *searchable partial sums* bildet einen Ansatz zur Realisierung dynamischer Bitvektoren. Sei $S[1..n]$ eine Folge von Ganzzahlen, wobei jedes Element in k Bits dargestellt werden kann, also $0 \leq S[i] \leq 2^k - 1$ für alle $i \in [1..n]$. Es sollen folgende Operationen effizient unterstützt werden:

- **Sum:** Für einen Index $i \in [1..n]$ liefert

$$S.\text{sum}(i) = \sum_{j=1}^i A[j]$$

die Summe aller Werte bis i .

- **Update:** Für einen Index $i \in [1..n]$ und ein δ so, dass $0 \leq A[i] + \delta \leq 2^k - 1$, setzt

$$S.\text{update}(i, \delta)$$

den Wert an Stelle i zu $A[i] + \delta$.

- **Search:** Für ein $j \in [0, S.\text{sum}(n)]$ liefert

$$S.\text{search}(j) = \min\{i \mid 1 \leq i \leq n, S.\text{sum}(i) \geq j\},$$

den Index, an dem die Summe mindestens j beträgt.

Diese Datenstruktur wird mithilfe einer zweistufigen Baumhierarchie mit sorgfältig gewählten Parametern umgesetzt werden. Zunächst wird ein m -ärer Baum über S konstruiert, wobei $m = (\log n)^\epsilon$ für ein $0 < \epsilon < 1$ ist. Die Blätter dieses Baums entsprechen Blöcken von S . An jedem inneren Knoten werden die Summen der m Kinder gespeichert. Operationen in diesem Baum erfordern das Traversieren von der Wurzel zu einem Blatt, was $\mathcal{O}(\log_m n) = \mathcal{O}(\log n / \log \log n)$ Ebenen entspricht. In jeder Ebene muss unter den m Kindern gesucht oder aktualisiert werden, was durch die Verwendung von Q-Heaps in konstanter Zeit möglich ist.

Die Eingabe wird in Blöcke mit je $\Theta((\log n)^2)$ Elementen partitioniert und in den Blättern der ersten Baumstruktur gespeichert. Jeder dieser Blöcke wird durch einen b -ären Baum mit $b = (\log n)^{\frac{1}{2}}$ dargestellt, was Operationen innerhalb des Blocks in $\mathcal{O}(\log \log n)$ -Zeit ermöglicht - vernachlässigbar gegenüber der Hauptlaufzeit. Durch diese Parameterwahl erreichen alle drei Operationen eine Laufzeit von $\mathcal{O}(\log n / \log \log n)$ bei einem Speicherverbrauch von $kn + o(kn)$ Bits [24, Abs. 2].

Ein dynamischer Bitvektor ist ein Spezialfall der Searchable Partial Sums-Struktur, bei dem alle gespeicherten Ganzzahlen auf die Werte 0 oder 1 beschränkt sind ($k = 1$). Die **rank**-Operation auf dem Bitvektor entspricht direkt

der **sum**-Operation auf der Partial-Sum-Struktur, während **select**-Operationen durch die **search**-Operation realisiert werden können. Änderungsoperationen auf dem dynamischen Bitvektor entsprechen analog Änderungen der grundlegenden Strukturen [22, Abs. 2.2.2].

Die C++-Bibliothek DYNAMIC [22] implementiert diese Searchable Partial Sums-Struktur als B-Baum. Es wird auf eine zweite Baumstruktur verzichtet. Dadurch benötigt der dynamische Bitvektor $n + o(n)$ Bits und komplexe Operationen lassen sich in einer Laufzeit von $\mathcal{O}(\log n)$ ausführen. Es existieren zudem andere, stark optimierte Implementierungen, die speziell für die effiziente Darstellung von Bitvektoren entwickelt wurden. Diese erreichen in der Praxis eine deutlich bessere Performance, unter anderem durch das Puffern von Änderungen, was insbesondere bei Anwendungen mit häufigen Updates einen großen Vorteil bietet [5].

2.2.3 Adaptive Dynamic Bit Vector

Der von Navarro beschriebene *Adaptive Dynamic Bitvector* [16] erweitert klassische dynamische Bitvektorstrukturen um einen Mechanismus, der den Bitvektor an das tatsächliche Anfrageverhalten anpasst. Ziel ist es, häufig abgefragte Bereiche automatisch in statische, besonders schnelle Strukturen umzuwandeln, während selten genutzte Bereiche dynamisch bleiben. Die Datenstruktur basiert auch auf einem balancierten binären Baum mit der Höhe $h = \mathcal{O}(\log n)$. Die inneren Knoten speichern neben der Anzahl der enthaltenen Bits (**nums**) und der Anzahl gesetzter 1-Bits (**ones**) auch die Anzahl der seit dem letzten Update durchgelaufenen Anfragen (**queries**).

Die Blätter des Baumes sind entweder statisch oder dynamisch. Dynamische Blätter speichern Bitvektoren der Größe $b = \Theta(\omega \log n)$, wodurch Zugriffe in $\mathcal{O}(1)$ -Zeit und Rank-/Select-Operationen mittels Maschinenwortoperationen in $\mathcal{O}(\log n)$ -Zeit möglich sind. Statische Blätter enthalten hingegen größere Ausschnitte des Bitvektors, die mit vollständigen Rank- und Select-Strukturen ausgestattet sind [17, S. 74–76], sodass alle Operationen in konstanter Zeit $\mathcal{O}(1)$ unterstützt werden.

Gilt innerhalb eines inneren Knotens v $v.\text{queries} \geq \Theta \cdot v.\text{size}$ für eine gewählte Konstante Θ , so wird der gesamte Teilbaum von v durch ein einzelnes statisches Blatt ersetzt. Änderungsoperationen erfolgen analog zu klassischen dynamischen Bitvektorstrukturen in $\mathcal{O}(\log n)$ -Zeit. Soll ein Update auf einem statischen Blatt ausgeführt werden, so wird dieses rekursiv aufgespalten, bis das betroffene Teilsegment in einem dynamischen Blatt liegt, das klein genug für die lokale Änderung ist.

Ist q die Anzahl der Anfragen pro Änderung, werden alle Operationen in $\mathcal{O}(\log(n/q))$ bei einem Speicherbedarf von $n + o(n)$ unterstützt. In der Praxis können so insbesondere auf Bitvektoren mit statischen Bereichen sehr gute Ergebnisse erzielt werden. Da vor allem höhere Schichten einer Ordner-Hierarchie

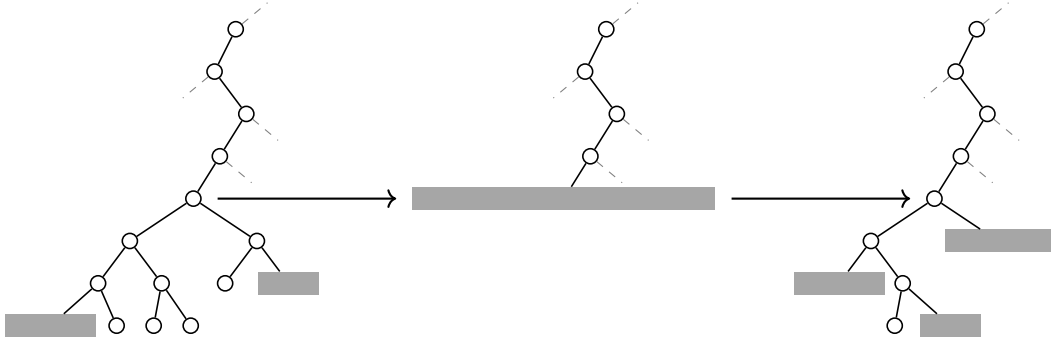


Abbildung 2.2: Restrukturierung im *Adaptive Dynamic Bitvector* nach Navarro [16]: Ein häufig angefragter Teilbaum wird zunächst zu einem statischen Blatt umgewandelt. Tritt anschließend eine Schreiboperation auf, erfolgt eine rekursive Aufspaltung des statischen Blattes.

wenigen Änderungen unterliegen, kann eine solche Implementierung auch für unser succinctes Dateisystem hilfreich sein.

2.3 Wavelet Tree

Bisher wurden Bitvektoren betrachtet, die ausschließlich binäre Werte speichern. Ein Wavelet Tree [1, Abs. 1-2] [4, Abs. 2] [17, S. 128–134] [18] verallgemeinert die Operationen eines Bitvektors auf Sequenzen über ein beliebiges endliches Alphabet. Dazu wird jedes Zeichen des Alphabets auf eine Ganzzahl abgebildet. Konkret speichert die Datenstruktur eine Sequenz $T[1..n]$ des Integer-Alphabets $\Sigma = [0, \sigma)$. Ein Buchstabe kann folglich mit $\lceil \log_2 \sigma \rceil$ Bits kodiert werden. Der Wavelet Tree unterstützt dabei dieselben grundlegenden Anfragen wie ein Bitvektor, jedoch bezogen auf beliebige Symbole des Alphabets:

- **Access:** Für einen Index $i \in [1..n]$ liefert

$$T.\text{access}(i) = T[i],$$

das an Position i gespeicherte Symbol.

- **Rank:** Für ein Symbol $s \in \Sigma$ und einen Index $i \in [1..n]$ liefert

$$T.\text{rank}_s(i) = |\{j \mid 1 \leq j \leq i, T[j] = s\}|$$

die Anzahl der Vorkommen von s in dem Präfix $T[1..i]$.

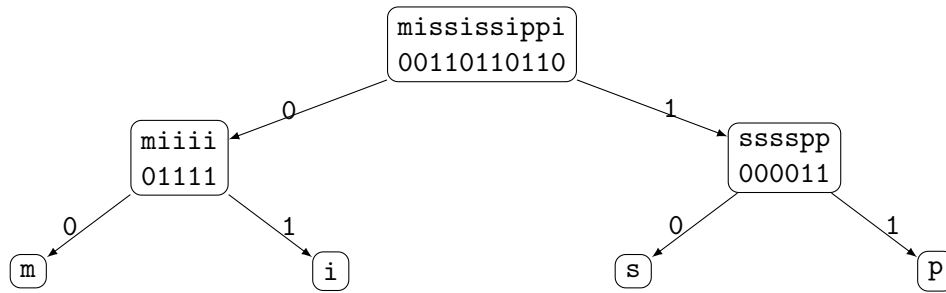
- **Select:** Für ein Symbol $s \in \Sigma$ und ein $j \in [1..T.\text{rank}_s(n)]$ liefert

$$T.\text{select}_s(j) = \min\{i \mid 1 \leq i \leq n, T.\text{rank}_s(i) = j\},$$

die Position des j -ten Vorkommens von s in T .

2.3.1 Konstruktion

Ein Wavelet Tree speichert die Sequenz T in einem Baum der Höhe $\lceil \log_2 \sigma \rceil$. Jeder Knoten des Baums speichert einen Bitvektor, der sich direkt aus der Binärdarstellung der Integer-Symbole ergibt: In der obersten Ebene wird das höchstwertige Bit jedes Symbols betrachtet, in den darunterliegenden Ebenen jeweils das nächsthöhere Bit. Symbole, deren Bit an der aktuellen Position 0 ist, werden dem linken Kindknoten zugeordnet, Symbole mit einem 1-Bit dem rechten Kindknoten. Dieser Vorgang wird rekursiv durchgeführt, bis jeder Blattknoten genau ein Symbol repräsentiert [17, S. 134–136].



$$T = 00\ 01\ 10\ 10\ 01\ 10\ 10\ 01\ 11\ 11\ 01 \hat{=} \text{mississippi}$$

Abbildung 2.3: Beispiel eines Wavelet Trees mit $|\Sigma| = 4$. Innere Knoten speichern Bitvektoren, die die Symbole auf linke und rechte Teilbäume verteilen.

2.3.2 Operationen

Wavelet Trees ermöglichen Anfragen auf der gespeicherten Sequenz, indem sie Operationen auf den Bitvektoren der Knoten nutzen. Diese Operationen lassen sich wie folgt auf der Datenstruktur umsetzen [1, Abs. 2.1] [18, Abs. 2.2]:

Access. Um das Symbol an Position i in T zu bestimmen, beginnt man an der Wurzel und verfolgt die Bitvektoren entlang des Pfades zu einem Blattknoten. In jedem Knoten mit Bitvektor B wird gezählt, wie viele Symbole mit demselben Bit vor Position i liegen. Entsprechend wird der Index für den Kindknoten-Bitvektor angepasst: $i \leftarrow B.\text{rank}_{B[i]}(i)$. Abhängig von $B[i]$ wird rekursiv beim linken oder rechten Kindknoten fortgesetzt.

Rank. Um die Vorkommen von $s \in \Sigma$ im Präfix $T[1..i]$ zu bestimmen, verfolgt man wie bei Access den Pfad vom Wurzel- zum Blattknoten, das s repräsentiert. In jedem Knoten wird das entsprechende Bit b aus s für diese

Ebene abgeleitet, und der Index angepasst: $i \leftarrow B.\text{rank}_b(i)$, wobei B der Bitvektor des entsprechenden Knotens der Ebene ist. Nach Erreichen des Blattes liefert der aktuelle Index i das Ergebnis.

Select. Um die Position des j -ten Vorkommens eines Symbols $s \in \Sigma$ in T zu bestimmen, beginnt man am Blattknoten, das s repräsentiert, und verfolgt den Pfad nach oben zur Wurzel. In jedem Knoten wird das entsprechende Bit b aus s für diese Ebene abgeleitet, und der Index im Bitvektor B angepasst: $i \leftarrow B.\text{select}_b(i)$. Nach Erreichen der Wurzel liefert der aktuelle Index i die gesuchte Position in T .

Vorausgesetzt, dass **rank** und **select** auf den verwendeten Bitvektoren in $\mathcal{O}(1)$ möglich sind, ergibt sich somit für alle drei Operationen eine Laufzeit von $\mathcal{O}(\log \sigma)$.

2.3.3 Praktische Erwägungen

Im Fall eines gleichverteilten Alphabets, d. h. wenn alle Symbole die gleiche Länge in der Binärdarstellung haben, kann pro Ebene ein konkatenierter Bitvektor gespeichert werden. Alternativ lassen sich die Bitvektoren aller Knoten zu einem einzigen, allgemeinen Bitvektor zusammenfassen: Die Länge des Bitvektors legt die Form des Baumes fest, sodass die Bereiche der einzelnen Knoten klar bestimmt sind [18, Abs. 2.2].

Dieses Konzept wird in der Wavelet-Matrix weitergeführt: Auf jeder Ebene wird zusätzlich zum Bitvektor auch die Anzahl der 0-Bits erfasst. Dadurch lässt sich jede Position effizient auf die nächste Ebene abbilden, was die Datenstruktur kompakt hält und cache-effizienter macht. Die Anzahl der benötigten **rank**-Operationen wird dadurch reduziert, was die praktische Performance verbessert [18, Abs. 2.3] [1, Abs. 5].

Wavelet Trees sind grundsätzlich als statische Datenstruktur konzipiert. Werden die zugrunde liegenden Bitvektoren jedoch dynamisch implementiert, lassen sich in trivialen Fällen (z.B. kleine, bekannte Alphabetgröße), einfach Änderungen am Wavelet Tree vornehmen. Ist h die Höhe des Wavelet Trees, so können Updates unter Verwendung der im vorherigen Kapitel beschriebenen dynamischen Bitvektoren in $\mathcal{O}(h \cdot \log n)$ ausgeführt werden. In bestimmten Szenarien kann jedoch eine vollständige Neukonstruktion des Baums effizienter sein, da moderne Prozessoren Operationen unterstützen, die sinnvoll während der Konstruktion genutzt werden können. So zeigen empirische Ergebnisse, dass diese Methoden eine bis zu 2,7-fach schnellere Erstellung ermöglicht als andere bekannte Algorithmen [4, Abs. 5].

2.4 Trie

Ein Trie [17, S. 252–256] [30, Abs. 1] [15, S. 28.1–28.3] ist eine Datenstruktur zur effizienten Speicherung und Suche nach Schlüsseln, typischerweise Zeichenketten.

Konkret verwaltet ein Trie T eine endliche Menge von Schlüsseln $\mathcal{K} \subset \Sigma^*$. Zur theoretischen Analyse sei $\Sigma = [0, \sigma)$ ein Integralalphabet. Oftmals wird ein Trie verwendet, um ein Dictionary zu implementieren. Ein eindeutiger Schlüssel K wird also mit einem eindeutigen Wert assoziiert. Konkret werden folgende Operationen ermöglicht:

- **Insert:** Für ein $S \in \Sigma^*$ und einen Wert V fügt

$$T.\text{insert}(S, V)$$

S zu $\mathcal{K}$ hinzu und assoziiert S mit V .

- **Lookup:** Für ein $S \in \Sigma^*$ liefert

$$T.\text{lookup}(S)$$

den mit S assoziierten Wert oder $\perp$ falls $S \notin \mathcal{K}$.

- **Delete:** Für einen Schlüssel $K \in \mathcal{K}$ entfernt

$$T.\text{delete}(K)$$

den Schlüssel aus $\mathcal{K}$.

- **Prefix:** Für ein $S \in \Sigma^*$ liefert

$$T.\text{prefix}(S) = \{K \in \mathcal{K} \mid S \text{ ist Präfix von } K\}$$

alle Schlüssel K aus $\mathcal{K}$ die den Präfix S haben.

Im Gegensatz zu einem Wavelet Tree, der für Abfragen auf Sequenzen optimiert ist, dient ein Trie der strukturierten Speicherung von Schlüsseln anhand gemeinsamer Präfixe. Ein Trie ist ein Baum, in dem jeder Pfad von der Wurzel aus Schritt für Schritt eine Zeichenkette $S \in \Sigma^*$ aufbaut. Jede Kante des Baums trägt genau ein Symbol aus dem Alphabet Σ . Ein Schlüssel ist genau dann im Trie gespeichert, wenn der Pfad, der ihn beschreibt, in einem Knoten endet, der mit einem assoziierten Wert markiert ist.

Komprimierte Tries (auch Patricia Tries) zeichnen sich dadurch aus, dass alle unären Pfade zusammengefasst sind, sodass jeder interne Knoten mindestens zwei Kinder hat [30, Abs. 1.2].

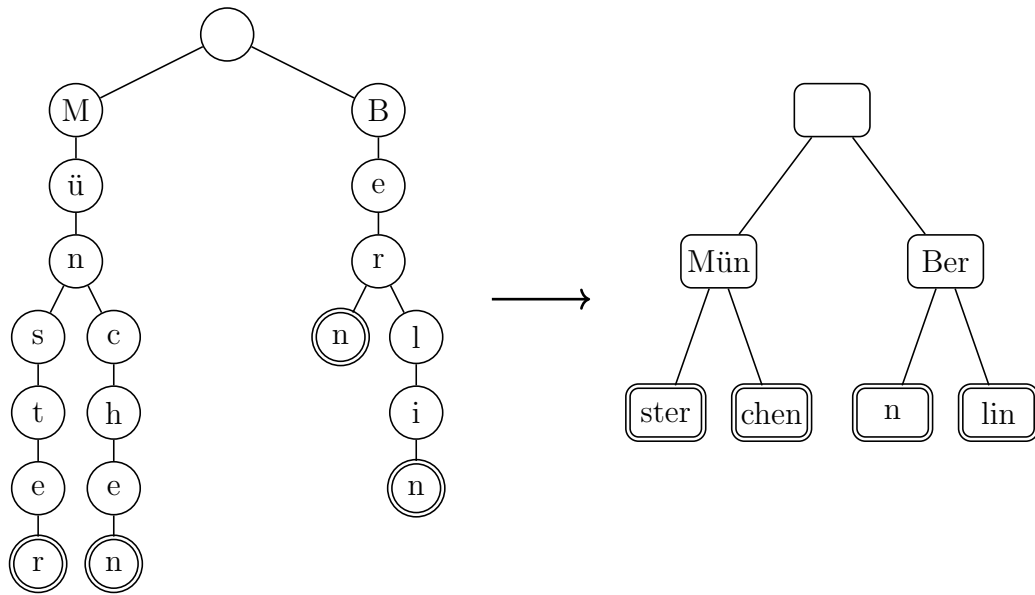


Abbildung 2.4: Beispiel eines Tries und seines komprimierten Patricia-Tries. Links pro Buchstabe ein Knoten, rechts werden Zeichenkette zusammengefasst. Doppelkreise markieren Endknoten von Wörtern. $\mathcal{K} = \{\text{Münster, München, Berlin, Bern}\}$

2.4.1 Dynamischer Trie

Aus algorithmischer Sicht sind Tries grundsätzlich gut für die Dynamisierung geeignet. Insbesondere zeigerbasierte Implementierungen erlauben das Einfügen und Löschen von Schlüsseln durch lokale Änderungen entlang eines Pfades. Allerdings geht diese Flexibilität häufig mit einem erhöhten Speicherbedarf einher. Klassische dynamische Tries speichern für jeden Knoten explizite Verweise auf seine Kinder [15, S. 28–5], was insbesondere bei großen Alphabeten oder stark verzweigten Strukturen zu einem erheblichen Overhead führen kann [17, S. 254].

c-trie++

Bei c-trie++ handelt es sich um eine hochoptimierte und gängige Implementierung eines Tries. Kazuya beschreibt und implementiert einen Trie, der die grundlegende Idee nutzt, den Trie in zwei Ebenen aufzuteilen: Eine grobe Ebene für die Navigation über große Distanzen und eine feine Ebene für die lokale Suche. Der Parameter $\alpha = \omega / \log_2 \sigma$ gibt an, wie viele Zeichen gleichzeitig in einem Maschinenwort verarbeitet werden können. Der Trie wird so unterteilt, dass nach jeweils genau α gelesenen Zeichen ein Knoten im sogenannten Makro-Trie erreicht wird. Jeder dieser Makro-Trie-Knoten verwaltet einen kleinen Mikro-Trie, der ausschließlich die nächsten α Zeichen abdeckt. Ist m die Länge des Suchwortes, so ergibt sich für die vollständige Traversierung eine Laufzeit von $\mathcal{O}(m/\alpha + \log \alpha)$, bestehend aus der Traversierung des Makro-

Tries und der anschließenden Suche im jeweiligen Mikro-Trie. Der Speicherbedarf liegt bei $n \lg \sigma + \Theta(k \lg n)$ Bits, wobei n die Gesamtlänge aller Keywords und k die Anzahl der Keywords ist [30, Abs. 2]. In der Praxis zeigt sich, dass `c-trie++` besonders bei Präfixsuchen zu den schnellsten getesteten Strukturen zählt. Dies kann auf die hierarchische Aufteilung und die Verarbeitung mehrerer Zeichen auf einmal zurückgeführt werden, wodurch die Anzahl notwendiger Vergleichsoperationen reduziert wird [30, Abs. 3].

Improved Practical Compact Dynamic Tries

Poyias und Raman präsentieren mit `m-Bonsai` eine weitere Datenstruktur für dynamische Tries, die auf dem `Bonsai`-Ansatz aufbaut. Die grundlegende Idee besteht darin, Knoten nicht durch Zeiger zu verbinden, sondern ihre Position durch Hashfunktionen zu berechnen. Alle Knoten werden in einem einzigen Array gespeichert, wobei für jeden Knoten und jedes mögliche Kindersymbol ein Schlüssel berechnet und gehasht wird.

Ein Vorteil, den `m-Bonsai` ausnutzt, ist die Beobachtung, dass die meisten Knoten nahe an ihrer idealen Position im Array liegen. Anstatt für jeden Knoten Kollisionsinformationen (etwa zum linearen Sondieren) zu speichern, genügt es, kleine Verschiebungswerte vorzuhalten. Diese lassen sich sehr kompakt kodieren, da sie typischerweise klein sind. Dadurch erreicht `m-Bonsai` einen asymptotisch optimalen Speicherbedarf von $(1 + \epsilon)n(\log \sigma + \mathcal{O}(1))$ Bits für n Knoten und eine Konstante $\epsilon > 0$. Anfrage- und Änderungsoperationen können in $\mathcal{O}(m)$ Zeit durchgeführt werden, da für jedes Zeichen lediglich eine Hash-Operation in konstanter Zeit durchgeführt wird. Eine Präfixsuche ist in dieser Implementierung nicht effizient realisierbar. In der Praxis zeigt sich eine hohe Speicherreduktion und Effizienz gegenüber klassischen Implementierungen [21].

2.5 LOUDS

Für die Analyse von Ordnerhierarchien ist die effiziente Darstellung von Baumstrukturen eine zentrale Grundlage. Zeigerbasierte Ansätze ermöglichen eine schnelle Navigation innerhalb der Baumstruktur sowie effiziente Operationen auf Teilbäumen, führen jedoch in der Praxis zu einem erheblichen Speicherverbrauch. Alternativ wurden succincte Repräsentationen wie *Balanced Parentheses* oder die auf Tiefensuche basierende *Depth-First Unary Degree Sequence* (DFUDS) entwickelt. Allerdings bringen auch diese Ansätze zusätzlichen Verwaltungsaufwand mit sich [20, Abs. 2A].

Daher betrachten wir im Rahmen dieser Arbeit ausschließlich die *Level-Order Unary Degree Sequence* (LOUDS) zur Baumdarstellung. Zwar bietet LOUDS weniger direkte Operationen auf der Baumstruktur als andere Ansätze, dafür ist sie extrem speichereffizient: Für einen Baum mit n Knoten werden

nur $2n + 1$ Bits benötigt, was dem theoretischen Minimum für die Darstellung allgemeiner Bäume entspricht [17, S. 212].

2.5.1 Kodierung

Die LOUDS-Kodierung eines Baumes basiert auf einer Breitensuche (Level-Order-Traversierung [15, S. 3–8]). Dabei wird jeder Knoten in eine Bitfolge übersetzt: Ein Knoten mit d Kindern wird durch d aneinanderfolgende 1-Bits, gefolgt von einem abschließenden 0-Bit dargestellt (1^d0). Die gesamte LOUDS-Bitfolge entsteht durch die Aneinanderreihung dieser Kodierungen in Ebenenreihenfolge. Damit die Wurzel korrekt dargestellt wird, wird der Baum gedanklich an eine imaginäre Wurzel angehängt. Für einen Baum (mit existierender Wurzel) ergibt sich dadurch der Präfix 10 in der LOUDS-Darstellung [17, S. 212–214] [3, Abs. 3].

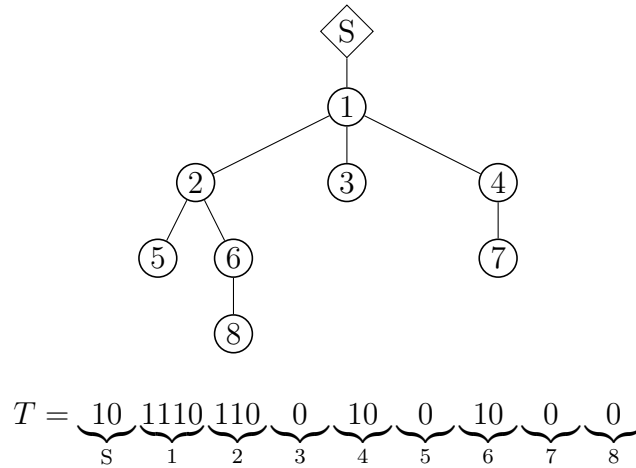


Abbildung 2.5: Beispiel einer LOUDS-Repräsentation mit *One-based Numbering*. Jeder Knoten wird in Level-order durch eine Bitfolge 1^d0 dargestellt.

Ein Baum T mit n Knoten entspricht als LOUDS also einem Bitvektor $B[1..2n + 1]$. Knoten werden nicht durch Speicherreferenzen ausgedrückt, sondern durch einen Index $i \in [1, 2n + 1]$. Jeder Knoten in der LOUDS-Kodierung wird zweifach repräsentiert: Entweder als 1-Bit innerhalb der Bitfolge seines Elternknotens oder als abschließendes 0-Bit seiner eigenen Bitfolge. Daher existieren zwei mögliche Indexierungsschemata [3, Abs. 3]. Wir verwenden im Folgenden ausschließlich das *One-based Numbering*, bei dem der i -te Knoten in Breitensuchreihenfolge durch die Position des i -ten 1-Bits identifiziert wird. Die Wurzel hat folglich den Index $T.\text{root} = 1$.

2.5.2 Operationen

Über einen Baum T mit n Knoten, der intern durch den Bitvektor $B[1..2n+1]$ repräsentiert wird, werden folgende Operationen [3, Abs. 3] definiert:

- **Parent:** Für einen Knoten an Position $i \in [3, 2n+1]$ ($B.\text{access}(i) = 1$) liefert

$$T.\text{parent}(i) = B.\text{select}_1(B.\text{rank}_0(i))$$

die Position des Elternknotens.

- **Children Count:** Für einen Knoten an Position $i \in [1, 2n+1]$ ($B.\text{access}(i) = 1$) liefert

$$T.\text{children_count}(i) = B.\text{select}_0(B.\text{rank}_1(i)+1) - B.\text{select}_0(B.\text{rank}_1(i)) - 1$$

die Anzahl der Kinder des Knotens.

- **Children:** Für einen Knoten an Position $i \in [1, 2n+1]$ ($B.\text{access}(i) = 1$) und ein $k \in [1, T.\text{children_count}(i)]$ liefert

$$T.\text{child}(i, k) = B.\text{select}_0(B.\text{rank}_1(i)) + k$$

die Position des k -ten Kindes.

Beispiele für Operationen basierend auf 2.5:

- $T.\text{children_count}(0) = B.\text{select}_0(1) - B.\text{select}_0(0) - 1 = 3$

Es wird das abschließende 0-Bit des Knotens sowie das abschließende 0-Bit des nachfolgenden Knotens bestimmt. Die Anzahl der Kinder ergibt sich aus der Anzahl der (1-)Bits zwischen diesen beiden Positionen.

- $T.\text{child}(0, 3) = B.\text{select}_0(1) + 3 = 4$

Es wird das abschließende 0-Bit des Knotens bestimmt. Das k -te darauf folgende (1-)Bit entspricht der Position des k -ten Kindes.

- $T.\text{parent}(4) = B.\text{select}_1(1) = 1$

Es wird die Anzahl der abschließenden 0-Bits gezählt. Das entsprechende 1-Bit markiert die Position des Elternknotens.

Das Hinzufügen oder Löschen eines Blatts kann umgesetzt werden, indem die beiden 1- und 0-Bits des Knotens an den entsprechenden Stellen eingefügt oder gelöscht werden [17, 470f.]. Auf dieser Grundlage baut die in Kapitel 4 beschriebene dynamische Version von FLOUDS auf.

3 Dateisysteme

Das Dateisystem ist ein zentraler Bestandteil eines Betriebssystems und ist für die persistente Speicherung und Organisation von Daten auf dem Massenspeicher verantwortlich. Es stellt dem Nutzer eine abstrahierte Sicht auf die Daten bereit in Form von Dateien, die die eigentlichen Daten enthalten, und Ordnern, die Dateien und andere Ordner hierarchisch gruppieren. Alle Dateien und Ordner teilen sich einen Namensraum, sodass jeder Name innerhalb eines Ordners eindeutig ist. Zusätzlich verwalten Dateisysteme wichtige Metadaten, wie Dateigrößen oder Besitzinformationen. Dateien und Ordner werden über Pfade adressiert, die ihre Position in der Verzeichnishierarchie beschreiben. Absolute Pfade beginnen an der Wurzel des Dateisystems und spezifizieren den vollständigen Weg zu einer Datei, während relative Pfade von einem aktuellen Arbeitsverzeichnis ausgehen [27, S. 470–490].

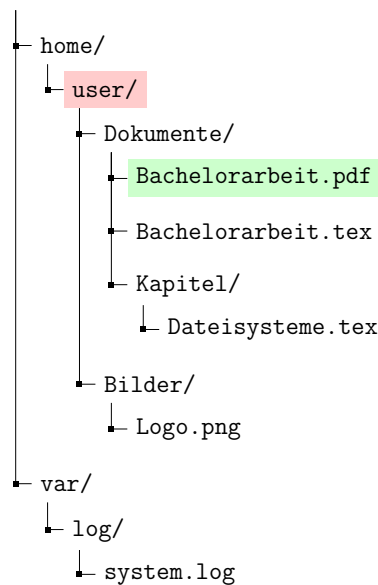


Abbildung 3.1: Beispielhafte Verzeichnisstruktur. Das aktuelle Arbeitsverzeichnis `user` ist rot gekennzeichnet. Der absolute Pfad zur grün gekennzeichneten Datei lautet `/home/user/Dokumente/Bachelorarbeit.pdf`, während der relative Pfad `Dokumente/Bachelorarbeit.pdf` ist.

Unterhalb dieser logischen Schicht existieren weitere Ebenen, die für die tatsächliche Organisation und Speicherung der Daten verantwortlich sind. Un-

terschiedliche Dateisysteme implementieren diese Schichten, je nach Anwendungsfall und Umgebung, auf unterschiedliche Weise. Dennoch teilen gängige Endnutzer-Betriebssysteme grundlegende Gemeinsamkeiten und Konzepte, die im Folgenden konzeptionell am Beispiel des ext4-Dateisystems dargestellt werden.

3.1 Physisches Layout

Massenspeicher wie HDDs oder SSDs können Daten nur in Blöcken fester Größe lesen oder schreiben, den sogenannten *disk blocks*. Das Dateisystem fasst ein Vielfaches solcher Disk Blocks zu einer größeren Einheit zusammen, den *file system blocks*, die als kleinste Verwaltungseinheit dienen (oft 4096 Bytes). Die Daten des Dateisystems werden innerhalb einer Partition organisiert, also einer zusammenhängenden Sammlung von Blöcken, die für das Dateisystem reserviert sind [11, S. 8]. Eine ext4-Partition ist wiederum in mehrere *block groups* unterteilt, die jeweils dasselbe interne Layout aufweisen. Die Verwendung von Block Groups reduziert die Fragmentierung und verbessert durch Ausnutzung der Lokalität die Performance des Dateisystems [6, Abs. 2: High Level Design].

Superblock und Group Descriptors. Der erste Block eines Dateisystems, der sogenannte Superblock, stellt die zentrale Datenstruktur des Dateisystems dar und enthält grundlegende Metadaten wie die definierte Blockgröße, Zeitstempel oder einen Magic-String, der das verwendete Dateisystem identifiziert. Da der Superblock für die Integrität des Dateisystems essenziell ist, wird er redundant in mehreren Block Groups gespeichert. Metadaten zu den einzelnen Block Groups, insbesondere die Positionen der weiteren Datenstrukturen, sind in den sogenannten Group Descriptors abgelegt [12, S. 43–51].

Block Bitmap und Inode Bitmap. Darauf folgt eine Bitfolge, in der jedes Bit einen Block der jeweiligen Block Group repräsentiert. Ein 1-Bit signalisiert, dass der entsprechende Block belegt ist, während ein 0-Bit einen freien Block kennzeichnet. Der Einsatz von Bitvektoren ermöglicht eine effiziente Suche nach freiem Speicherplatz bei der Allokation neuer Daten. Da die Bitfolge maximal die Größe eines Blocks besitzen darf, ergibt sich bei einer Blockgröße von 4096 Bytes eine maximale Anzahl von $4096 \cdot 8 = 32.768$ adressierbaren Datenblöcken pro Gruppe und entsprechend eine maximale Größe einer Block Group von $32.768 \cdot 4096 = 128\text{MiB}$. Analog dazu existiert eine Inode Bitmap, die mithilfe eines Bitvektors kennzeichnet, welche Inodes innerhalb der Block Group bereits belegt beziehungsweise noch frei sind [7, Abs. 2].

Inode Table. Die Inode Table ist ein zusammenhängender Bereich, der alle Inodes einer Block Group speichert. Ein *Inode* ist eine zentrale Datenstruktur, die die Metadaten (z.B. Dateigröße, Berechtigungen) einer Datei oder eines

Ordners enthält. Außerdem werden hier Verweise auf die Datenblöcke, in denen der eigentliche Inhalt gespeichert ist, verwaltet. Unter ext4 benötigt ein Inode standardmäßig 256 Bytes [12, S. 55].

Data Blocks. Der verbleibende und größte Teil einer Block Group besteht aus den *Data Blocks*, in denen tatsächliche Inhalte gespeichert werden [7, Abs. 2].

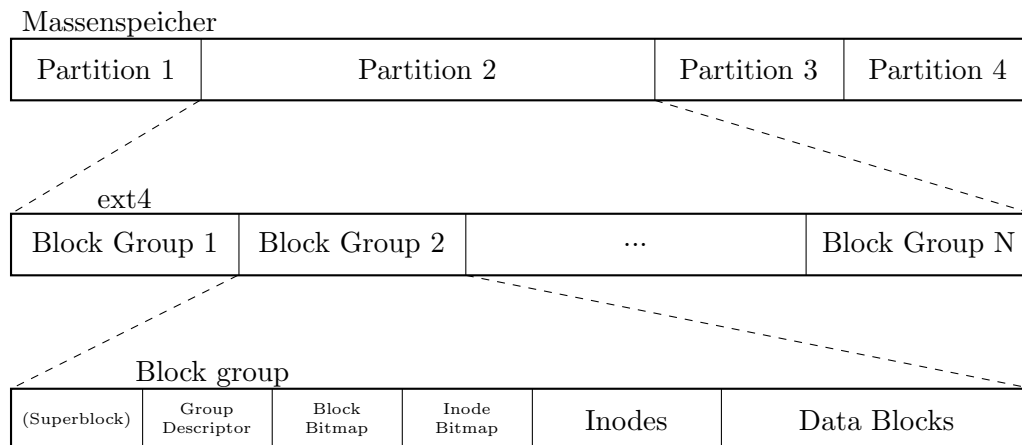


Abbildung 3.2: Vereinfachte Struktur eines ext4-Dateisystems auf einem partitionierten Massenspeicher.

3.2 Allokationsstrategien

Die Allokationsstrategien eines Dateisystems bestimmen maßgeblich dessen Performance. Wenn eine neue Datei erstellt oder eine bestehende Datei erweitert wird, muss entschieden werden, welche freien Blöcke auf dem Massenspeicher zur Speicherung der Daten verwendet werden. Diese Entscheidung wird durch die jeweilige Allokationsstrategie des Dateisystems bestimmt. Dabei gilt es, unterschiedliche und teils konkurrierende Ziele zu berücksichtigen [11, S. 99], beispielsweise:

- **Fragmentierung:** Die Fragmentierung, also die Aufsplittung von Dateien in viele kleine, über das Speichermedium verteilte Fragmente, soll minimiert werden. Zusammenhängende Speicherbereiche ermöglichen sequentielle Zugriffe, die insbesondere bei rotierenden Festplatten deutlich schneller möglich sind als ein zufälliger Zugriff [11, S. 101].
- **Lokalität:** Verwandte Daten, etwa Dateien desselben Ordners oder Fragmente einer großen Datei, sollten physisch nah beieinander platziert werden. [6, Abs. 2.4]

- **Speicherausnutzung:** Der verfügbare Speicherplatz sollte nicht durch übermäßige Lücken verschwendet werden.

3.2.1 Zusammenhängende Allokation

Die zusammenhängende Allokation ist eine einfache Heuristik für Allokationsstrategien. Dabei werden die Daten einer Datei stets in einer zusammenhängenden Liste von Blöcken auf dem Massenspeicher abgelegt. Diese Methode ist leicht zu implementieren und ermöglicht eine optimale Leseperformance. Allerdings kann der Speicher schnell fragmentiert werden. Wird eine Datei nachträglich vergrößert, kann es im schlimmsten Fall erforderlich sein, die gesamte Datei neu zu kopieren [27, 491f.].

First Fit. Bei der First-Fit-Strategie durchsucht das Dateisystem die Block Bitmap und wählt den ersten freien Bereich aus, der groß genug ist, um die gewünschte Datei zu schreiben.

Best Fit. Die First-Fit-Strategie tendiert dazu, am Anfang des Speicherbereichs viele kleine Lücken zu hinterlassen, während größere Bereiche nur teilweise genutzt werden. Die Best-Fit-Strategie durchsucht alle verfügbaren Bereiche und wählt den kleinsten Bereich, der aber noch groß genug für die Anforderungen ist [27, 253f.].

3.2.2 Block-Mapping

Ältere ext-Dateisysteme nutzen direktes und indirektes *Block-Mapping*, eine klassische Technik, die bereits in früheren Unix-Dateisystemen zum Einsatz kam [27, S. 545]. Jeder Inode enthält ein Array von 15 Block-Pointern. Die ersten 12 Pointer zeigen direkt auf Datenblöcke und werden daher *direkte Block-Pointer* genannt. So kann besonders schnell auf kleine Dateien zugegriffen werden. Der 13. Pointer im Inode ist ein *indirekter Pointer*. Er verweist auf einen Block, der wiederum weitere Block-Pointer enthält. Der 14. Pointer ist ein zweifach indirekter Pointer und zeigt auf einen Block, der Pointer auf weitere indirekte Blöcke enthält, die schließlich auf die eigentlichen Datenblöcke zeigen. Der 15. Pointer erweitert dieses Prinzip um eine dritte Ebene, sodass auch sehr große Dateien gespeichert werden können [12, 60f.].

Soll das Dateisystem auf einen Datenbereich einer Datei zugreifen, der nicht durch einen direkten Pointer abgedeckt ist, folgt das Dateisystem den indirekten Pointern. Je Pointer-Ebene bedeutet das also einen zusätzlichen Blockzugriff, bevor die Daten gelesen werden können. Selbst wenn die Datenblöcke einer Datei zusammenhängend auf der Festplatte liegen, erzeugen die indirekten Pointer zusätzlichen Speicher-Overhead, da für jede Pointer-Ebene eigene Blöcke nur zur Adressierung der Daten reserviert werden.

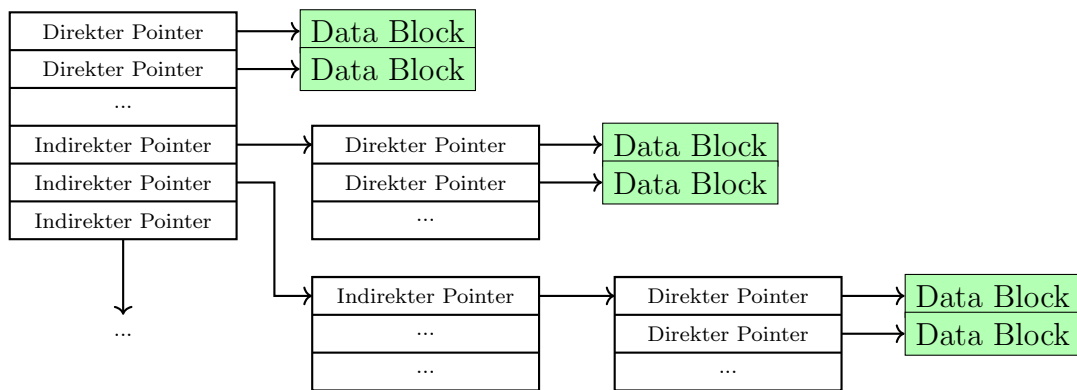


Abbildung 3.3: Darstellung der direkten, einfach indirekten und doppelt indirekten Block-Pointer eines Inodes.

3.2.3 Extents

Mittlerweile verwendet ext4 hauptsächlich das Konzept der *Extents*. Anstatt für jeden einzelnen Block einen Pointer zu speichern, fasst ein Extent die Lage-Informationen einer zusammenhängenden Liste von Blöcken zusammen [7, Abs. 4]. Ein Extent besteht aus drei Informationen: der logischen Position innerhalb der Datei, dem physischen Startblock und der Länge des zusammenhängenden Block-Bereichs. In ext4 kann ein einzelner Extent bis zu 32768 Blöcke abdecken, was bei 4096-Bytes-Blöcken 128 MiB entspricht.

Für kleine Dateien mit bis zu vier zusammenhängenden Bereichen können die Extents direkt im Inode gespeichert werden. Größere beziehungsweise stärker fragmentierte Dateien erfordern jedoch einen *Extent Tree*. Der Wurzelknoten dieses Baums wird direkt im Inode gespeichert. Weitere Knoten des Baums liegen in dafür reservierten Blöcken des Dateisystems. Die inneren Knoten des Baumes enthalten Index-Einträge, die anhand der logischen Position innerhalb der Datei auf tiefere Ebenen verweisen. In den Blattknoten werden die eigentlichen Extents mit den physischen Blockadressen gespeichert. Um die Daten an einer logischen Position innerhalb einer Datei zu finden, folgt das Dateisystem vom Inode ausgehend den inneren Knoten, bis es den relevanten Blattknoten gefunden hat [6, Abs. 4.2].

Zusammenfassend lässt sich festhalten, dass diese klassischen Strukturen zwar eine hohe Performance und Robustheit gewährleisten, jedoch nicht sehr platzsparend sind. Diese Überlegungen motivieren den Einsatz und die Erweiterung des FLOUDS-Ansatzes für die Implementierung eines succinthen Dateisystems.

3.2.4 Spekulative Allokation

Wenn eine Anwendung Daten schreibt, ist es sinnvoll, nicht nur den unmittelbar benötigten Speicherplatz, sondern einen größeren Block zu reservieren,

in der Annahme, dass die Datei zeitnah weiter wachsen wird. Wenn eine Datei schrittweise wächst und bei jeder Erweiterung nur der minimal benötigte Platz allokiert würde, entstünden schnell viele kleine, verteilte Fragmente [11, S. 107].

3.2.5 Delayed allocation

Eine weitere Optimierungstechnik in ext4 ist die *delayed allocation*. Hierbei wird die physische Zuweisung von Blöcken auf dem Massenspeicher so lange hinausgezögert wie möglich. Wenn eine Anwendung Daten schreibt, werden diese zunächst nur im Hauptspeicher gepuffert. Zum Zeitpunkt des konkreten Flushes kennt das System dann die tatsächliche Größe der zu schreibenden Daten und es kann eine fundiertere Allokationsentscheidung getroffen werden [6, Abs 2.4].

3.3 Hierarchie-Ebene

Während Inodes die Metadaten von Dateien speichern, muss das Dateisystem auch die hierarchische Struktur von Ordnern und die Zuordnung von Dateinamen zu Inodes verwalten. Diese Aufgabe übernehmen in ext4 die Directory Entries.

3.3.1 Directory Entries

Ein Ordner ist eine spezielle Datei, deren Inhalt aus einer Liste von Directory Entries besteht. Jeder Directory Entry enthält eine Inode-Nummer sowie einen zugehörigen Dateinamen. Der Name einer Datei wird also nicht mit den Metadaten einer Datei gespeichert, sondern im übergeordneten Verzeichnis. Diese Struktur erklärt auch, warum ext4 Hard Links unterstützt, denn auf diese Weise können mehrere Dateinamen in verschiedenen Verzeichnissen auf dieselben Daten verweisen [12, S. 65].

In kleinen Ordnern mit wenigen Einträgen werden die Directory Entries einfach linear in den Datenblöcken des Verzeichnisses abgelegt. Bei der Suche nach einem bestimmten Dateinamen muss das System die Liste sequentiell durchlaufen, was bei einer überschaubaren Anzahl von Einträgen ausreichend performant ist [6, Abs. 4.3.1].

3.3.2 Hash Tree Directories

Große Verzeichnisse würden bei einer linearen Suche jedoch zu erheblichen Performance-Problemen führen. Ext4 löst dieses Problem durch die Verwendung einer Hash-Tree-Struktur, die Einträge des Ordners indiziert. Der Name

des Eintrags wird mittels einer Hash-Funktion auf einen numerischen Wert abgebildet. Dieser Hash-Wert dient als Schlüssel in einer baumartig organisierten Indexstruktur. Die Wurzel des Baumes und die inneren Knoten enthalten durch Hash-Bereiche Verweise auf weitere Knoten. Die Blattknoten enthalten die eigentlichen Directory Entries [6, Abs. 4.3.2].

3.3.3 Auflösen von Pfaden

Der Prozess der Pfadauflösung, also das Übersetzen eines Dateipfads in den entsprechenden Inode, nutzt die beschriebenen Mechanismen. Bei einem absoluten Pfad wie `/home/user/Dokumente/Bachelorarbeit.pdf` beginnt das Dateisystem bei einem festgelegten Wurzel-Inode. Das System liest den Inode der Wurzel, erkennt, dass es sich um ein Verzeichnis handelt, und sucht in dessen Directory Entries nach dem Eintrag `home`. Dieser Eintrag enthält die Inode-Nummer des Verzeichnisses `/home`. Das System liest diesen Inode und durchsucht dessen Inhalt nach `user`, erhält wieder eine Inode-Nummer und setzt diesen Prozess fort, bis es den letzten Pfadbestandteil erreicht. Bei jedem Schritt führt das Dateisystem entweder eine lineare Suche durch oder nutzt die Hash-Tree-Struktur. Es bietet sich an diesen Prozess durch zusätzliches Caching oft aufgerufener Teil-Pfade zu optimieren [25, Abs. II].

3.4 Ergänzende Konzepte

3.4.1 Journaling

Viele Dateisystemoperationen bestehen aus mehreren Schritten, die nacheinander ausgeführt werden. Beispielsweise erfordert das Löschen einer Datei die Freigabe der Datenblöcke in dem Block- und Inode Bitmap sowie das Entfernen des Directory Entries. Wenn das System zwischen diesen Schritten abstürzt, kann das Dateisystem in einem inkonsistenten Zustand zurückbleiben. Eine Transaktion (eine Liste verschiedener Änderungsoperationen auf dem Massenspeicher) ist atomar, soll also entweder vollständig oder gar nicht durchgeführt werden [11, 111f.].

Ext4 implementiert Journaling als optionalen Mechanismus zur Sicherstellung der Konsistenz [6, S. 6.2]. Das Journal ist ein speziell reservierter Bereich auf dem Speichermedium, in dem geplante Änderungen protokolliert werden, bevor sie tatsächlich durchgeführt werden können. Nach einem Systemabsturz analysiert das System beim nächsten Startvorgang das Journal und führt unvollständige Transaktionen zu Ende oder macht sie rückgängig [11, 114f.].

3.4.2 Log-structured File System

Da die Speicherkapazitäten des Arbeitsspeichers und Massenspeichers in den 1990er-Jahren stetig zunahmen, der Massenspeicher jedoch nicht proportional

schneller wurde, entstand das Konzept der *Log-structured File systems* (LFS). Dieses setzte sich allerdings bei Endnutzer-Dateisystemen weitgehend nicht durch. Bei LFS werden zunächst alle Schreibvorgänge im Arbeitsspeicher gepuffert und anschließend periodisch als große, zusammenhängende Segmente am Ende eines kontinuierlichen Logs auf den Massenspeicher geschrieben. Ein Segment kann Inodes, Directory Entries oder Data Blocks enthalten, jeweils mit einem Segment-Header, der den Inhalt beschreibt. Strukturinformationen des Dateisystems werden im Arbeitsspeicher zwischengespeichert. Da der Speicherplatz auf der Festplatte begrenzt ist, sorgt ein Hintergrund-Thread dafür, dass alte, nicht mehr benötigte Daten freigegeben werden. LFS verbessert die Performance bei kleinen Schreibvorgängen und ermöglicht eine sehr gute Ausnutzung der gesamten Festplatte, bringt jedoch eine äußerst komplexe Verwaltungslogik mit sich [27, S. 524–526].

3.5 Nutzer-Schicht

Unter Linux stellt das *Virtual Filesystem* (VFS) eine einheitliche Abstraktionsschicht dar, die es ermöglicht, verschiedene Dateisysteme wie ext4, XFS oder Squashfs über eine gemeinsame Schnittstelle anzusprechen. Das VFS definiert abstrakte Operationen für typische Dateisystemfunktionen wie `open()`, `read()`, `write()`, `mkdir()` oder `unlink()`. Jedes konkrete Dateisystem implementiert diese Operationen gemäß seiner internen Struktur. Anwendungen und Nutzer können über das VFS unabhängig vom darunterliegenden Dateisystem mit Dateien und Ordnern arbeiten. Des Weiteren übernimmt das VFS einige Caching-Tätigkeiten. Ebenfalls wird durch Locks sichergestellt, dass sich gegenseitig ausschließende Operationen (z.B. Schreiben und Löschen) nicht parallel ausgeführt werden können [9].

3.5.1 FUSE

FUSE (Filesystem in Userspace) ist ein weit verbreitetes Linux-Framework, um Dateisysteme außerhalb des Betriebssystem-Kernels zu implementieren. FUSE-Dateisysteme sind nicht fest im Kernel integriert, sondern laufen im User-Space, was die Programmierung erheblich erleichtert. Im Kernel registriert sich FUSE als Dateisystem-Treiber beim VFS und fungiert dort als Vermittler: Anstatt die Operationen selbst auszuführen, leitet der Treiber die Anfrage über ein Device-File an den entsprechenden Userspace-Daemon weiter. Es existieren zwei unterschiedliche FUSE-APIs für den Userspace: Eine Low-level API, die nah am Kernel und direkt mit Inodes arbeitet, und eine High-level API, die direkt mit abstrahierten Dateipfaden arbeitet [31].

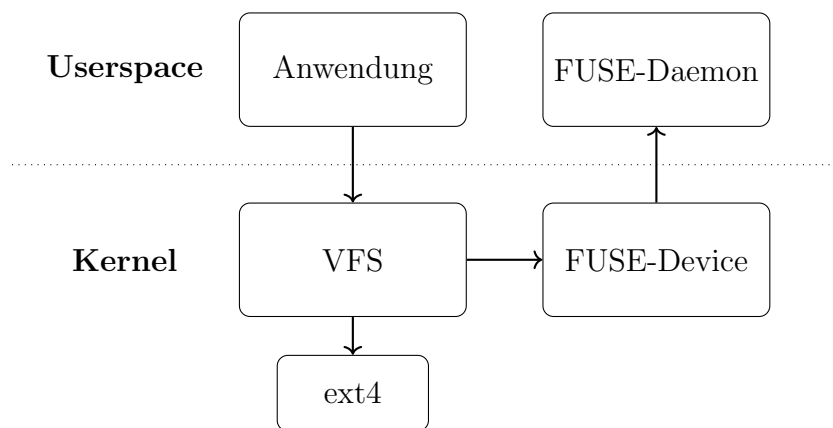


Abbildung 3.4: Zusammenspiel der Dateisystem-Komponenten unter Linux: Anfragen aus dem Userspace werden an das VFS weitergeleitet und von da aus entweder an das implementierte Dateisystem oder über das FUSE-Device wieder zurück an den Userspace.

4 FLOUDS

FLOUDS [20] ist eine alternative Darstellungsform für Dateisysteme beziehungsweise Ordnerhierarchien, die sich deutlich von Inode-basierten Ansätzen unterscheidet. Ursprünglich wurde FLOUDS entwickelt, um die Integrität eines Dateisystems mithilfe von Hash-Werten überprüfen zu können. Anstatt die Form einer Ordnerhierarchie indirekt über Inodes und Directory Entries zu speichern, wird diese mit einer Variation der LOUDS-Struktur gespeichert. Ein Dateisystem kennt typischerweise t verschiedene Knotentypen: reguläre Dateien, Ordner, leere Ordner, Links und weiteren Typen. Die Unterscheidung zwischen Ordnern und leeren Ordnern ist dabei strukturell relevant und wird daher explizit kodiert.

4.1 Konstruktion

FLOUDS besteht aus mehreren parallelen Datenstrukturen, die während einer Breitensuche über den Dateisystembaum aufgebaut werden:

Struktursequenz. Die Baumtopologie wird analog zu LOUDS in einem Bitvektor $B[1..n]$ kodiert. Für einen Knoten wird eine 1 geschrieben, wenn er das erste Kind seines Elternknotens ist, andernfalls eine 0. Diese Struktur unterscheidet sich insofern von LOUDS, als dass ein Knoten nur eine einzige Repräsentation in B besitzt. Den Index dieses Auftretens verwenden wir im Folgenden als Referenzindex für den entsprechenden Knoten an verschiedenen Stellen des Dateisystems.

Dateitypsequenz. Für jeden Knoten wird ein kleiner vordefinierter Integer-Wert gespeichert, der den Dateitypen repräsentiert. Zur Vereinfachung verwenden wir zunächst nur: 0 für reguläre Dateien, 1 für Ordner (mit Inhalt) und 2 für leere Ordner. Weitere Dateitypen lassen sich analog ergänzen. Da die Sequenz $T[1..n]$ Werte aus dem Alphabet $\Sigma = [0, t)$ mit $t = 3$ Dateitypen enthält, wird sie als Wavelet Tree gespeichert.

Namens-Sequenz. $N[1..n]$ ordnet jedem Index, also jeder Datei und jedem Ordner, einen Namen zu. Die konkrete Umsetzung dieser Struktur stellt eine zentrale Designentscheidung dar und wird im Folgenden näher erläutert.

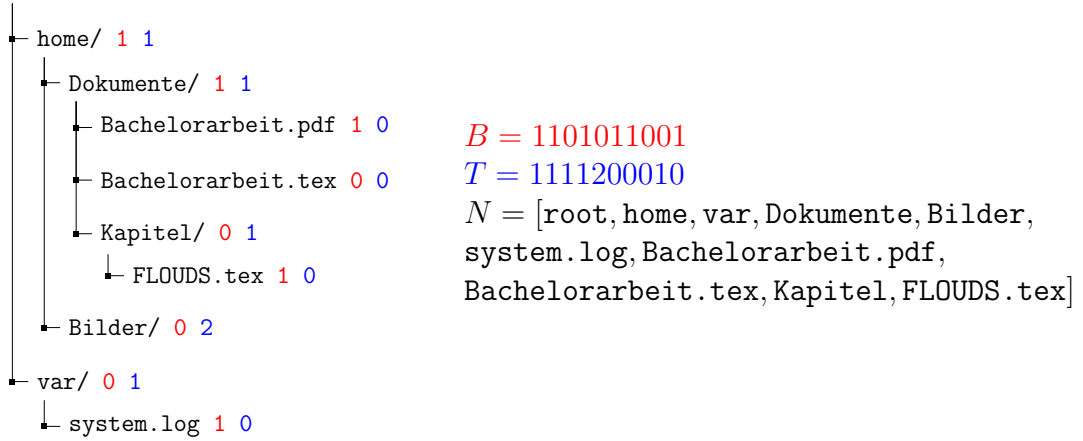


Abbildung 4.1: Beispielhafte Verzeichnisstruktur kodiert mit FLOUDS. Jeder Knoten hat einen Eintrag in der Struktursequenz B (rot) und in der Dateitypsequenz T (blau). Die imaginäre Wurzel root wird ergänzt.

4.2 Navigation

Die Navigation über FLOUDS funktioniert ganz analog zu den bereits beschriebenen Operationen über LOUDS. Ist $F = (B, T, N)$ definieren wir folgende Operationen:

- **Parent:** Für eine Datei/einen Ordner an Position $i \in [2, n]$ liefert

$$F.\text{parent}(i) = T.\text{select}_1(B.\text{rank}_1(i) - 1)$$

den Index des übergeordneten Ordners.

- **Children Count:** Für einen nicht-leeren Ordner an Position $i \in [2, n]$ ($T.\text{access}(i) = 1$) liefert

$$F.\text{children_count}(i) = B.\text{select}_1(T.\text{rank}_1(i) + 1) - B.\text{select}_1(T.\text{rank}_1(i))$$

die Anzahl der enthaltenen Dateien und Unterordner.

- **Child:** Für einen nicht-leeren Ordner an Position $i \in [1, n]$ ($B.\text{access}(i) = 1$) und ein $k \in [0, F.\text{children_count}(i)]$ liefert

$$F.\text{child}(i, k) = B.\text{select}_1(T.\text{rank}_1(i) + 1) + k - 1$$

die Position der k -ten enthaltenen Datei beziehungsweise des k -ten Unterordners. Für $k = 0$ wird die Position unmittelbar vor dem ersten Kind zurückgegeben, die später für die Einfügeoperationen benötigt wird.

4.3 Pfadauflösung und Namenssequenz

Die Frage, wie Pfade im entwickelten Dateisystem aufgelöst werden, hängt eng damit zusammen, wie die Namen der Dateien und Ordner gespeichert werden. Für die Umsetzung existieren unterschiedliche Ansätze, die im Folgenden analysiert werden sollen. Dabei ist besonders zu berücksichtigen, dass die verwendeten Strukturen sowohl platzsparend als auch dynamisch sein müssen, da das Ziel darin besteht, ein vollwertiges dynamisches Dateisystem zu implementieren.

4.3.1 Lineare Sequenz

Ein einfacher Ansatz zur Speicherung der Knotennamen besteht darin, diese sortiert nach ihren FLOUDS-Nummern abzulegen. Im Paper wird vorgeschlagen, alle Namen konkateniert in einer Zeichenkette zu speichern und zusätzlich einen Bitvektor B_N zu verwenden, der den Beginn jedes Namens markiert. Über $B_N.\text{select}_1(i)$ kann dann die Startposition des i -ten Namens bestimmt werden. Wird die konkatenierte Zeichenkette wie vorgeschlagen als Wavelet Tree über einem Alphabet der Größe σ kodiert, ergibt sich für das Auslesen eines Namens der Länge k eine Laufzeit von $\mathcal{O}(k \log \sigma)$. Diese Variante erreicht zwar einen praktisch minimalen Speicherbedarf, jedoch sind Änderungen an der Zeichenkette aufwendig und der Zugriff auf einzelne Namen ineffizient. Bei jeder Modifikation muss die gesamte Struktur des Wavelet Trees angepasst werden, was für ein dynamisches Dateisystem unpraktikabel ist.

Alternativ kann die konkatenierte Zeichenkette auch klassisch als Folge von Alphabetsymbolen gespeichert werden, beispielsweise als String. Dies erhöht den Speicherbedarf zwar, ermöglicht jedoch Random Access auf die Namen in $\mathcal{O}(k)$ -Zeit. Das Entfernen oder Einfügen von Elementen ist im schlechtesten Fall in $\mathcal{O}(n)$ möglich, da alle nachfolgenden Einträge verschoben werden müssen. Ohne zusätzlichen Bitvektor B_N können die Namen auch als pointerbasierte Liste oder als Array von Strings gespeichert werden, wobei letzteres durch bessere Cache-Lokalität in der Praxis zu bevorzugen ist.

Tabelle 4.1: Vergleich der verschiedenen Varianten, um Knotennamen zu speichern. Betrachtet wird ein Alphabet der Größe σ , t Namen mit einer Gesamtlänge von n und ein beispielhaftes Zugriffswort der Länge k .

Variante	Speicher [15]	Implementierung	Zugriff	Einfügen / Entfernen
Wavelet Tree + B_N	$n \log \sigma + o(n \log \sigma)$	Hoher Aufwand	$\mathcal{O}(k \log \sigma)$	Teuer, praktisch Rebuild
String + B_N	$n \log \sigma + \Theta(n)$	Mittlerer Aufwand	$\mathcal{O}(k)$	$\mathcal{O}(n)$
Array	$n \log \sigma + \Theta(t)$	Geringer Aufwand	$\mathcal{O}(k)$	$\mathcal{O}(n)$
Pointerbasierte Liste	$n \log \sigma + \Theta(t)$	Geringer Aufwand	$\mathcal{O}(n)$	$\mathcal{O}(1)$

Um einen Pfad wie `/home/Dokumente/Bachelorarbeit.pdf` in die entsprechende FLOUDS-Nummer aufzulösen, wird der Pfad zunächst in seine Komponenten zerlegt. Ausgehend von der Wurzel wird für jede Komponente linear

durch die Kinder des aktuellen Knotens iteriert und deren Namen verglichen, bis eine Übereinstimmung gefunden wird.

Algorithmus 1 Auflösen eines Pfades durch lineare Suche in den Kindern jedes Knotens. Integritätsprüfungen (z. B. Existenz des Pfades oder Knotentyps) werden zur Abstraktion ausgelassen.

```

current = root
for all component  $\in$  path do
  first_child  $\leftarrow F.\text{child}(\text{current}, 1)$ 
  last_child  $\leftarrow \text{first\_child} + F.\text{children\_count}(\text{current}) - 1$ 
  for  $i \leftarrow \text{first\_child} \rightarrow$  do
    if  $N[i] = \text{component}$  then
      current  $\leftarrow i$ 
      break
    end if
  end for
end for

```

Durch geschickt gewählte Index-Strukturen über die Namenssequenz lassen sich die Einfüge- und Löschooperationen schneller umsetzen. Mit einem B-Baum [15, 15-3ff.], also einem selbstbalancierten Suchbaum, der in einem inneren Knoten mehrere Verzweigungen speichert und dadurch die Höhe des Baumes reduziert, lässt sich der Zugriff auf Namen immer noch effizient realisieren. Änderungsoperationen sind mit einer Laufzeit von $\mathcal{O}(\log t + n/t)$ deutlich effizienter umsetzbar, da $\log t$ die Höhe des Baumes beschreibt und damit die Anzahl der Knoten, die im Worst-Case angepasst werden müssen, während n/t den Aufwand für das lokale Verschieben der Schlüssel innerhalb eines Knotens darstellt. Die Laufzeit ist amortisiert, weil aufwendige Anpassungen wie Splits oder Merges nur gelegentlich auftreten, während die meisten Änderungsoperationen schnell lokal durchgeführt werden können.

4.3.2 Hierarchische Organisation

Bei zusammenhängender Speicherung aller Namen entsteht ein fundamentales Problem: Die gesamte Namenssequenz mit einer Gesamtlänge von potenziell mehreren Megabytes muss vollständig im Speicher vorgehalten werden, selbst wenn nur kleine Ausschnitte von Interesse sind. Dies ist besonders bei ressourcenbeschränkten Systemen problematisch.

Im Hinblick auf die Cachelokalität ist es daher sinnvoll, einen zusammenhängenden Bereich vorzuhalten, der die aktuell relevanten Namen enthält. Zwar wurde der zuvor erwähnte B-Baum für ähnliche Zwecke entwickelt und findet in vielen Datei- und Datenbanksystemen breite Anwendung, jedoch erfüllt dieser unsere Anforderungen an den Speicherbedarf durch viele Pointer nur bedingt. Darüber hinaus ist es sinnvoll, ausschließlich logisch zusam-

menhängende Namensbereiche im Speicher vorzuhalten, was ein B-Baum durch Selbstbalancierung nicht sicherstellt. Beispielsweise lädt ext4 über die Directory Entries stets alle Einträge eines Ordners als logischen Namensbereich. Es wäre darüber hinaus auch denkbar, größere Namensbereiche (etwa mehrere Ordnersebenen) zusammenzufassen und auf einmal zu speichern und laden.

Sei $F = (B, T)$ eine FLOUDS-Struktur mit n Knoten und $D = \{i \in [1, n] \mid T.\text{access}(i) = 1\}$ die Menge aller Ordner. Für jeden Ordner $i \in D$ mit d_i Kindern definieren wir eine lokale Namensstruktur $N_i[1..d_i]$. Ein Array von Zeigern $NP[1..|D|]$ mit $NP[T.\text{rank}_1(i)] = N_i$ ermöglicht den direkten Zugriff auf die Namensstruktur eines Ordners. $N_i[k]$ beschreibt also den Namen des k -ten Kindes von i . Die Zuordnung einer FLOUDS-Nummer f zu ihrem Namen erfolgt über:

$$N[f] = N_p[f - F.\text{child}(p, 1) + 1] \quad \text{wobei} \quad p = F.\text{parent}(f)$$

Der Term $f - F.\text{child}(p, 1) + 1$ berechnet dabei den relativen Index von f unter den Kindern seines Elternordners p . Für die konkrete Umsetzung von N_i existieren verschiedene Ansätze, die sich hinsichtlich Speicherbedarf und Zugriffszeiten unterscheiden. Es ist auch denkbar, unterschiedliche Datenstrukturen zu verwenden, abhängig davon wie viele Namen verwaltet werden müssen.

Lineare Sequenz. Für Ordner mit wenigen Kindern bietet sich die Speicherung in einer linearen Sequenz an. Da hierbei lediglich die Einträge eines einzelnen Ordners verwaltet werden, können diese lexikographisch sortiert werden, um klassische Kompressionsverfahren wie mithilfe von Suffix-Arrays [17, 395f.] einzusetzen.

Trie. Die Verwendung eines klassischen komprimierten Tries zur Verwaltung der lokalen Namensstrukturen N_i stellt eine besonders intuitive Methode zur Pfadauflösung dar. Die Suche nach einer Pfadkomponente der Länge k innerhalb eines Ordners lässt sich effizient über die Operation `lookup` mit einem Aufwand von $\mathcal{O}(k)$ realisieren. Ein wesentlicher Nachteil dieses Ansatzes im Kontext von FLOUDS ist jedoch die Beantwortung inverser Anfragen: Während die Abbildung von einem Namen auf einen Index effizient ist, lässt sich der Name zu einer gegebenen FLOUDS-Nummer f nur schwierig extrahieren. Diese Rückwärts-Suche erfordert entweder eine vollständige Traversierung des Tries oder den Einsatz zusätzlicher Metadaten, was den succinthen Charakter der Gesamtdatenstruktur beeinträchtigen würde.

Hashing. Eine andere Möglichkeit zur Realisierung dieser bidirektionalen Abbildung bietet ein Hash-basierter Ansatz, wie in m-Bonsai beschrieben. Im Gegensatz zur linearen Sequenz, bei der Löschoperationen ein zeitaufwendiges Verschieben aller nachfolgenden Elemente in $O(n)$ -Zeit erfordern, nutzt

m-Bonsai Hashfunktionen, um Namen direkt auf die Position in einem kompakten Array abzubilden. Die Pfadauflösung (Name $\rightarrow$ FLOUDS-Nummer) erfolgt dabei in $\mathcal{O}(k)$ -Zeit für einen Namen der Länge k . Für die inverse Richtung (FLOUDS-Nummer $\rightarrow$ Name) lässt sich der Name durch die Umkehrung der Hash-Funktion direkt aus den Daten an der Position im kompakten Array ableiten, da der relative Index im Ordner unmittelbar als Lookup-Key fungiert [21, Abs. 3.5]. Ein zentraler Vorteil dieses Ansatzes ist es, dass im Gegensatz zum klassischen Array bei Löscho- oder Einfügeoperationen keine physischen Verschiebungen der Datensätze vorgenommen werden. Stattdessen werden freie Positionen lediglich markiert [21, Abs. 2.3].

Die Pfadauflösung innerhalb einer hierarchischen Organisation folgt dem Prinzip des Algorithmus der linearen Sequenz, nutzt jedoch die strukturellen Vorteile lokaler Namensbereiche. In jedem Schritt der Auflösung müssen die entsprechenden lokalen Datenstrukturen geladen werden, falls der benötigte Ordner gerade nicht im Cache vorgehalten wird. Zudem muss nicht mehr zwangsläufig linear über alle Kinder eines Knotens iteriert werden. Stattdessen kann die jeweilige Funktionsweise der gewählten Datenstruktur, etwa die Lookup-Operation eines Tries, ausgenutzt werden.

4.3.3 Fazit und Handles

Keiner der beschriebenen Ansätze ist für sich genommen optimal: Entweder verursachen die verwendeten Datenstrukturen, wie beim Hashing [21, Abs. 5], einen hohen Overhead, Anfragen, insbesondere Änderungsoperationen an Namen, sind nicht ausreichend schnell oder der Gesamteinsatz basiert auf der Nutzung von Pointern, was bei einem succincten Dateisystem vermieden werden sollte.

Bei der Implementierung des Dateisystems ist es daher notwendig, im Rahmen einer benchmarking-getriebenen Entwicklung die verschiedenen Ansätze zu einer möglichst sinnvollen Gesamtlösung zu kombinieren. Dabei sollten die einzelnen Ansätze gezielt entsprechend ihrer jeweiligen Stärken in den unterschiedlichen Anwendungsfällen des Dateisystems eingesetzt werden. Um insbesondere die aufwendige Pfadauflösung möglichst selten durchführen zu müssen, sollte die Implementierung nicht direkt mit Pfaden arbeiten, sondern die von der Low-Level-API bereitgestellten File-Handles verwenden [31, Abs. 2.2]. Dadurch kann direkt mit FLOUDS-Nummern gearbeitet werden.

4.4 Dynamisierung

Die Autoren von FLOUDS erwähnen, dass sich FLOUDS grundsätzlich für ein vollwertiges dynamisches Dateisystem eignet, wenn die zugrunde liegenden succincten Datenstrukturen dynamisch effizient gestaltet werden [20, Abs. V].

Für ein vollwertiges Dateisystem müssen einerseits Operationen zum Erstellen, Umbenennen und Löschen von Dateien und Ordnern innerhalb der FLOUDS-Struktur definiert werden. Andererseits stellt die Dynamisierung mehrere fundamentale Herausforderungen dar, die im Folgenden analysiert und gelöst werden sollen.

4.4.1 Änderungsoperationen

Die Dynamik von FLOUDS basiert auf der Dynamik der genutzten Datenstrukturen (B, T, N) . Wie in Kapitel 2.2 beschrieben, unterstützt der dynamische Bitvektor B Modifikationen in $\mathcal{O}(\log n)$ -Zeit. Da in unserem Fall T nicht viele Ebenen besitzt, lässt sich dieser Wavelet Tree auch gut dynamisieren. Im vorangegangenen Abschnitt wurde beschrieben, wie N dynamisch verwaltet werden kann.

Einfügen eines Blattknotens. Um eine neue Datei oder einen neuen Ordner als Kind eines Ordners p einzufügen, müssen B und T (und N) entsprechend angepasst werden. Beim Einfügen in einen leeren Ordner ($T.\text{access}(p) = 2$) muss zunächst der Typ des Elternordners zu einem nicht-leeren Ordner ($T.\text{access}(p) = 1$) geändert werden. Da der neue Knoten das erste Kind von p ist, wird in B an Position $F.\text{child}(p, 1)$ ein 1-Bit eingefügt. Beim Einfügen in einen nicht-leeren Ordner wird das neue Kind nach dem letzten existierenden Kind von p als 0-Bit eingefügt. Ebenso muss der Dateityp des neuen Knotens (entweder leerer Ordner oder Datei) an der berechneten Position in T eingefügt werden. Die Laufzeit beider Einfügeoperationen beträgt $\mathcal{O}(\log n)$, insgesamt also auch $\mathcal{O}(\log n)$.

Löschen eines Blattknotens. Um eine Datei oder einen leeren Ordner mit der FLOUDS-Nummer f aus der FLOUDS-Struktur zu entfernen, muss jeweils das f -te Element aus den Sequenzen B , T und N entfernt werden. Falls f das einzige Kind seines ursprünglichen Elternknotens p war, so muss dieser anschließend als leerer Ordner markiert werden. War hingegen f das erste Kind von p , muss das zweite Kind, das jetzt die FLOUDS-Nummer f hat, als neues erstes Kind gekennzeichnet werden.

Bei einem mit FUSE implementierten Dateisystem müssen nur die Operationen für das Löschen von leeren Verzeichnissen (`rmdir`) und Dateien (`unlink`) implementiert werden [10]. Das Löschen nicht-leerer Verzeichnisse erfordert das rekursive Löschen aller enthaltenen Dateien und Ordner. Diese Aufgabe wird unter Linux durch den Userspace übernommen.

4.4.2 Instabile FLOUDS-Nummern

Ein fundamentales Problem bei der Dynamisierung von FLOUDS besteht darin, dass die FLOUDS-Nummern bei Änderungen nicht stabil bleiben. Da die

Algorithmus 2 Einfügen eines Blattknotens mit dem Typ $t \in \{0, 2\}$ und dem Namen n als Kind des Ordners p .

```

empty  $\leftarrow (T.\text{access}(p) = 2)$ 
if empty then
   $T[p] \leftarrow 1$ 
  children_count  $\leftarrow 0$ 
else
  children_count  $\leftarrow F.\text{children\_count}(p)$ 
end if
insert_pos  $\leftarrow F.\text{child}(p, \text{children\_count}) + 1$ 
if empty then
   $B.\text{insert}_1(\text{insert\_pos})$ 
else
   $B.\text{insert}_0(\text{insert\_pos})$ 
end if
 $T[\text{insert\_pos}] \leftarrow t$ 
 $N[\text{insert\_pos}] \leftarrow n$ 

```

Nummern in Breitensuch-Reihenfolge vergeben werden, führt das Einfügen oder Löschen eines Knotens dazu, dass alle nachfolgenden Knoten um eins verschoben werden. Dies ist problematisch, da externe Referenzen auf Knoten, etwa in der Namensstruktur oder in Metadaten zum Verweis auf Datenblöcke, ungültig werden.

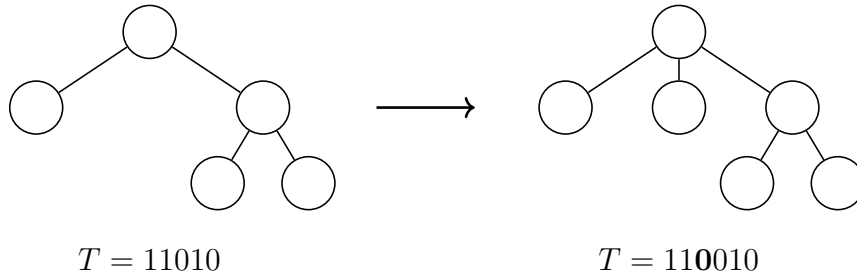


Abbildung 4.2: Einfügen eines neuen Knotens in zweiter Ebene verschiebt alle nachfolgenden Bits in Struktursequenz und damit die FLOUDS-Nummern um eine Stelle nach rechts.

Delta-basierte Stabilisierung. Es lässt sich beobachten, dass das Einfügen oder Löschen eines Knotens f ausschließlich die nachfolgenden Knoten in Breitensuchreihenfolge beeinflusst. Wird der Knoten f neu eingefügt, so erhöht sich die FLOUDS-Nummer aller Knoten mit einer ursprünglichen FLOUDS-Nummer $\geq f$ um 1. Beim Entfernen des Knotens f verringert sich entsprechend die FLOUDS-Nummer aller Knoten mit ursprünglicher Nummer $> f$ um 1.

Algorithmus 3 Entfernen eines Blattknotens f .

```
parent_index =  $F.parent(f)$ 
parent_children_count =  $F.children\_count(parent\_index)$ 
first_child =  $B.access(f)$ 
 $B.delete(f)$ 
 $T.delete(f)$ 
 $N.delete(f)$ 
if parent_children_count then
     $T[parent\_index] = 1$ 
else if first_child and parent_children_count > 1 then
     $B.update_1(f)$ 
end if
```

Wir speichern die letzten m Änderungsoperationen als Historie $(op_1, op_2, \dots, op_m)$, wobei jede Operation op_i definiert ist durch:

$$op_i = \begin{cases} (\text{insert}, f_i) & \text{Einfügen an Position } f_i \\ (\text{delete}, f_i) & \text{Löschen an Position } f_i \end{cases}$$

Die in op_i gespeicherte Position f_i bezieht sich auf die durch frühere Operationen $op_1, \dots, op_{i-1}$ bereits veränderte Nummerierung. Für einen Knoten mit ursprünglicher FLOUDS-Nummer f_0 ergibt sich die aktuelle FLOUDS-Nummer f_{current} also durch rekursive Anwendung der Historie:

$$\begin{aligned} f^0 &= f_0 \\ f^{(i)} &= f^{(i-1)} + \delta_i(f^{(i-1)}) \quad \text{für } i = 1, \dots, m \\ f_{\text{current}} &= f^m \end{aligned}$$

wobei

$$\delta_i(f) = \begin{cases} +1 & \text{falls } op_i = (\text{insert}, f_i) \text{ und } f \geq f_i \\ -1 & \text{falls } op_i = (\text{delete}, f_i) \text{ und } f > f_i \\ 0 & \text{sonst} \end{cases}$$

Damit die Historie nicht zu groß wird und die Anfragen mit einer Verzögerung durch die $\mathcal{O}(m)$ -Laufzeit der Neuberechnung effizient bleiben, ist es sinnvoll, periodisch die Änderungen aus der Historie analog zum LFS in die permanenten Strukturen zu propagieren und diese bei Bedarf neu zu konstruieren. Grundsätzlich müssen alle übrigen Dateisystemkomponenten so aufgebaut sein, dass sie die Delta-basierte Stabilisierung unterstützen.

Darüber hinaus lässt sich die Effizienz steigern, indem die Historie kontinuierlich bereinigt wird: Operationen, die sich gegenseitig aufheben (beispielsweise ein **insert** gefolgt von einem **delete** an derselben Position), werden aus der Liste entfernt.

Mapping. Ein naheliegender Ansatz zur Stabilisierung besteht darin, jedem Knoten bei seiner Erzeugung eine eindeutige stabile ID zuzuweisen und ein Mapping zwischen den jeweils aktuellen FLOUDS-Nummern und den stabilen IDs zu pflegen.

$$M : [1..n] \rightarrow \mathbb{N}$$

Eine zentrale Herausforderung dieses Ansatzes liegt jedoch in der effizienten Verwaltung des Mappings. Das Problem ist analog zur Speicherung der Namenssequenz, die bereits zuvor betrachtet wurde. Folglich bietet sich auch hier eine hierarchische Organisation an: Die stabile ID eines Knotens wird im jeweiligen Elternknoten gespeichert.

4.4.3 Integration mit File System Manager

Die FLOUDS-Autoren definieren für ein vollwertiges Dateisystem, das auf FLOUDS basiert, zwei notwendige Komponenten [20, Abs. V]: Die *Computing Component*, die Anfragen über die im Arbeitsspeicher abgelegte FLOUDS-Struktur beantwortet, sowie den *File System Manager*, der als Vermittler zwischen der Computing Component und den Datenblöcken fungiert.

Neben der reinen Struktur einer Ordnerhierarchie müssen zahlreiche Metadaten gespeichert werden: Berechtigungen, Zeitstempel, Dateigrößen sowie Verweise auf Datenblöcke, wie in Kapitel 3 beschrieben. Für die Speicherung dieser Metadaten existieren mehrere Ansätze, die eng mit der Wahl des Stabilisierungsansatzes für FLOUDS-Nummern verknüpft sind.

Bei Verwendung stabiler IDs können Metadaten direkt im jeweiligen Elternknoten mit oder anstatt der stabilen ID gespeichert werden. Alternativ lässt sich ein zentraler Inode-Bereich nach dem Vorbild von ext4 festlegen, wobei für jeden Knoten ein fester Inode reserviert wird. Dieser Ansatz definiert allerdings die maximale Anzahl von Knoten im Voraus und reserviert dafür Speicherplatz. Eine dritte Möglichkeit besteht darin, Metadaten parallel zu den FLOUDS-Sequenzen B , T und N in separaten Sequenzen zu organisieren, wobei der Zugriff über die FLOUDS-Nummer als Index erfolgt. Dies nutzt jedoch die Cache-Lokalität nicht optimal aus und größere Strukturen wie Extents können nicht effizient abgelegt werden.

Die Wahl des optimalen Ansatzes gilt es daher praxisorientiert zu ermitteln.

4.4.4 Multi-FLOUDS

FLOUDS-Operationen über kleine Bäume sind naturgemäß schneller als über große Bäume eines komplexen Dateisystems. Dies legt nahe, das Dateisystem nicht als einen einzelnen großen FLOUDS-Baum zu organisieren, sondern als mehrere unabhängige Teilbäume, die miteinander verknüpft werden. Dieses *Mounten* von Teilbäumen kann durch einen neuen internen Knotentyp reali-

siert werden, ergänzt durch eine zusätzliche Sequenz M , die Verweise auf die eingehängten FLOUDS-Strukturen hält.

Für die Aufteilung des Dateisystems in mehrere FLOUDS-Strukturen bieten sich verschiedene Strategien an: Eine Möglichkeit besteht darin, nach einer festen Anzahl von Ebenen automatisch eine neue FLOUDS-Struktur anzuhängen. Alternativ könnte die Aufteilung im Sinne der Data Blocks erfolgen, sobald eine FLOUDS-Struktur eine bestimmte Größe überschreitet.

Obwohl eine Performance-Steigerung durch kleinere, unabhängige Strukturen denkbar ist, wächst der Verwaltungsaufwand erheblich. Die Verwaltungsdaten für die verschiedenen FLOUDS-Strukturen müssen selbst gespeichert werden, und die Navigation zwischen verschiedenen Teilbäumen erfordert zusätzlichen Aufwand. Multi-FLOUDS ist mit Low-Level-FUSE nicht ohne weiteres umsetzbar, da wir nicht auf Pfaden arbeiten, sodass während der Traversierung des Baumes die einzelnen Teilbäume geladen werden können, sondern mit Inodes.

4.4.5 Bewegen von Teilbäumen

Das Verschieben von Teilbäumen innerhalb der Verzeichnishierarchie stellt bei FLOUDS eine fundamentale Herausforderung dar. Im Gegensatz zu klassischen Dateisystemen wie ext4, bei denen das Verschieben eines Teilbaums durch Umsetzen von Zeigern realisiert werden kann, ist dies strukturbedingt bei FLOUDS nicht möglich. Die Knoten eines Teilbaums sind nicht zusammenhängend im Speicher angeordnet, sondern über verschiedene Bereiche der FLOUDS-Sequenzen verteilt. Dieses Problem scheint zunächst nicht effizient lösbar. Um einen Teilbaum zu verschieben, muss dieser vollständig traversiert werden und die zusammenhängenden Inhalte eines Ordners können verschoben werden.

5 Implementierung

Nach den theoretischen Betrachtungen zu succincten Datenstrukturen, Dateisystemen und FLOUDS soll nun darauf aufbauend ein funktionsfähiges Dateisystem implementiert werden. Das Dateisystem wird als Low-Level-FUSE-Dateisystem für das Linux-VFS realisiert. Neben einem qualitativ hochwertigen Quellcode und einer sauberen Modularisierung wird besonders Wert auf eine Implementierung gelegt, die verschiedene Strategien benchmarkt, um für unterschiedliche Anwendungsfälle jeweils die geeignetste einsetzen zu können. Der Quellcode einschließlich einer Installationsanleitung ist unter <https://github.com/sebastianbrunnert/succinct-fs> einsehbar.

5.1 Entwicklungsumgebung und Softwarestack

5.1.1 Projektstruktur

Ein FUSE-Dateisystem ist letztendlich ein normales Programm, das im Userspace ausgeführt wird und mit dem Kernel kommuniziert. Um die offizielle (Low-Level-)Schnittstelle *libfuse* [8] zu nutzen, wird unser Projekt in C++ (Version 20) umgesetzt.

Als Build-System wird CMake eingesetzt, welches die plattformübergreifende Konfiguration, Kompilation und Verwaltung von Bibliotheken übernimmt. Mit der *Makefile* kann das Projekt kompiliert und getestet werden. Da FUSE ausschließlich unter Linux-Systemen verfügbar ist, verhindert die Konfiguration, dass das Projekt auf anderen Plattformen gebaut wird. Die *Makefile* enthält eine Regel, um ein beispielhaftes Dateisystem auf den Ordner `tmp` zu montieren, dessen Daten in der Datei `tmp.img` gesichert werden.

Der Einstiegspunkt des Dateisystem-Daemons befindet sich in der Datei `src/fuse.cpp`. In der `main`-Funktion wird eine FUSE-Session initialisiert und auf das VFS-Dateisystem montiert. Das Programm erwartet zwei Pflichtparameter: den Pfad zu einer Raw-Device-Datei sowie den Mountpoint, an dem das Dateisystem eingehängt werden soll. Da ein FUSE-Dateisystem nicht unmittelbar auf einer Partition arbeiten kann, wird diese über eine riesige Datei simuliert.

Zum Unit-Testing einzelner Komponenten wird GTest eingesetzt. Die Tests befinden sich im Ordner `tests/`.

5.1.2 Bibliotheken

Es werden verschiedene externe Bibliotheken eingesetzt:

- **libfuse:** Die offizielle Bibliothek zur Implementierung von FUSE-Daemons. Wir verwenden ausschließlich die Low-Level-API, die direkt mit Inode-Nummern arbeitet, um mehr Kontrolle über die Abbildung von FUSE-Operationen auf FLOUDS-Nummern zu haben. Die Bibliothek ist standardmäßig auf Linux-Systemen mit FUSE-Installationen verfügbar.
- **saskeli/bit_vector:** Eine optimierte Implementierung eines dynamischen Bitvektors auf Basis des Teilsummen-Ansatzes. Die Bibliothek baut konzeptionell auf dem DYNAMIC-Framework [22] auf, verwendet aber verschiedene Ansätze zur Verbesserung der praktischen Performance [5]. Die Implementierung wurde als Git-Modul eingebunden. Da diese für x86-Prozessoren optimiert wurde, muss auf diese Bitvektor-Strategie in anderen Umgebungen verzichtet werden.
- **Adaptive Dynamic Bitvector:** Implementierung des in Abschnitt 2.2.3 beschriebenen Adaptive Dynamic Bitvectors nach Navarro [16]. Auch diese Bibliothek wurde als Git-Modul eingebunden. Da es sich um C-Implementierung handelt, muss der Compiler dies unterstützen.
- **immer:** Effiziente Implementierung von verschiedenen Datenstrukturen, unter anderem Suchbäumen [23].

5.2 Systemarchitektur

Das entwickelte Dateisystem folgt einer komponentenbasierten Architektur, bei der klar abgegrenzte Module über strikt definierte Schnittstellen miteinander interagieren. Diese *Separation of Concerns* ermöglicht es, einzelne Komponenten unabhängig voneinander zu entwickeln, zu testen und auszutauschen. Das ist besonders für das angestrebte Benchmarking relevant, da so einzelne Komponenten isoliert verglichen werden können. Das Gesamtsystem lässt sich in Hauptkomponenten gliedern, die in den nachfolgenden Abschnitten detailliert beschrieben werden.

5.2.1 Succincte Datenstrukturen

Alle eingesetzten succincten Datenstrukturen sind hinter abstrakten Schnittstellen gekapselt, um verschiedene Strategien zu realisieren. Konkrete Implementierungen werden nicht direkt instanziiert, sondern über eine typisierte Factory-Function erzeugt.

Quelltext 5.1: Entkopplung der Bitvektor-Implementierung durch Template-basierte Strategiewahl.

```

template <typename Strategy> BitVector*
    create_bitvector(size_t n);

create_bitvector<ArrayBitVectorStrategy>(10);
create_bitvector<WordBitVectorStrategy>(10);

```

Bitvector. Die abstrakte Klasse `BitVector` definiert die in Abschnitt 2.1 beschriebenen Operationen eines dynamischen Bitvektors als rein virtuelle Methoden. Eine Test-Suite führt für jede Implementierungs-Strategie dieselben Tests über einen Bitvektor durch. Dadurch wird sichergestellt, dass alle Bitvektoren dasselbe Verhalten aufweisen. Der implementierte Bitvektor nutzt 0-basierte Indizes; `rank0`- oder `rank1`-Operationen erfolgen inklusive der angegebenen Position und um noch effizienter zu werden, muss der Aufrufer die Integrität eines Aufrufs sicherstellen, falls etwa ein nichtexistentes Auftreten eines Bits gesucht wird. Als Referenz wurde eine Strategie implementiert, die einen Vektor von Booleans als Bitvektor interpretiert. Der klassische Maschinenwort-basierte Ansatz wird durch die `WordBitVectorStrategy` implementiert. Außerdem existieren zwei Strategien, welche die Bitvektoren der beiden eingebundenen Bibliotheken an unsere Schnittstelle anpassen.

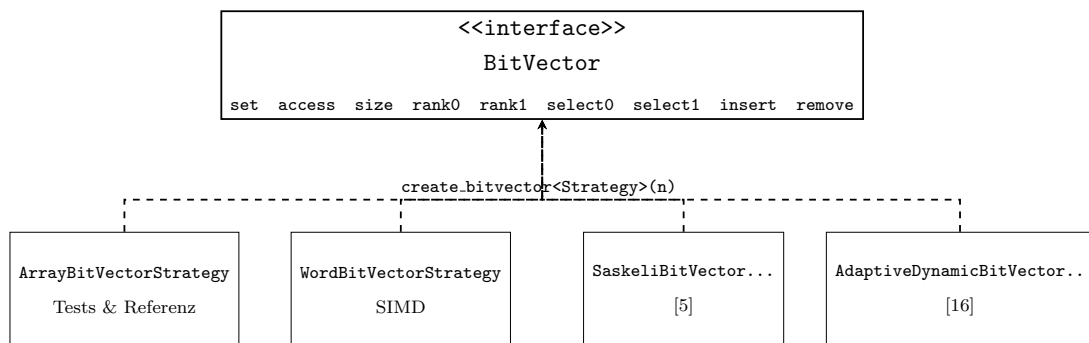


Abbildung 5.1: Abstrakte Schnittstelle `BitVector` mit den vier implementierten Strategien. Konkrete Instanzen können mit der initialen Größe erzeugt werden.

Wavelet Tree. Da der Wavelet Tree ausschließlich für die Typsequenz eingesetzt wird, genügt eine Implementierung, die eine Alphabetgröße von $\sigma = 4$ verwalten kann. Aufgrund dieses kleinen Alphabets können auch einfache, naive Ansätze zur Dynamisierung des Wavelet Trees verwendet werden. Bei der Instanziierung eines `TwoBitWaveletTree` kann ausgewählt werden, welche Bitvektor-Strategie intern genutzt werden soll. Der Wavelet Tree verwaltet dabei drei logisch zusammenhängende Bitvektoren: `root_bv` für die erste Ebene sowie `left_bv` und `right_bv` für das linke beziehungsweise rechte Kind. Das Interface stellt die Operationen `set`, `access`, `size`, `rank`, `select`, `insert` und

`remove`, jeweils mit ähnlichen Verträgen wie der `BitVector`, bereit. Auch hier muss der Aufrufer sicherstellen, dass zum Beispiel nur Symbole aus dem definierten Alphabet angefragt werden. Bei der Implementierung dynamischer Operationen ist es besonders entscheidend, die zusammenhängenden Bitvektoren konsistent zu halten.

Name Sequence. Zur Verwaltung von Namenssequenzen wird ein Interface `NameSequence` bereitgestellt, das die Operationen `set`, `access`, `insert` und `remove` definiert. Es existieren mehrere Implementierungen dieses Interfaces: Eine naive Variante verwendet intern einen String-Vektor. Eine weitere Strategie setzt den Ansatz des FLOUDS-Papers um, bei dem Namen in einem einzigen konkatenierten String gespeichert und über einen Bitvektor Startpunkte der Namen markiert werden.

Zwei zusätzliche Implementierungen basieren auf succincten Datenstruktur: Die erste nutzt den `flex_vector` der Bibliothek *immer* [23], einen RRB-Baum. Die zweite Implementierung verwendet Hashing-Verfahren durch die Nutzung einer ungeordneten Map.

5.2.2 Computing Component

Die *Computing Component* bildet die algorithmische Grundlage des Dateisystems und kapselt sämtliche Navigations- und Änderungsoperationen hinter der einheitlichen Schnittstelle der Klasse `Flouds`. Diese verwaltet drei parallele Datenstrukturen: den dynamischen Bitvektor `structure`, den Wavelet Tree `types` sowie die Namenssequenz. Die Navigation im Verzeichnisbaum erfolgt anhand der in Kapitel 4.2 beschriebenen Berechnungen. Die Methode `parent` bestimmt den Elternknoten über eine `rank1`-Operation auf dem Strukturvektor, gefolgt von einer `select`-Operation auf dem Typvektor. Die Kindknoten beziehungsweise deren Anzahl können über die Methoden `child` und `children_count` ermittelt werden. Beim Einfügen eines neuen Knotens wird gegebenenfalls zunächst der Typ des Elternknotens von einem leeren Ordner zu einem regulären Ordner geändert, bevor an der berechneten Position in allen drei Datenstrukturen die entsprechenden Werte des neuen Knotens eingetragen werden. Das Entfernen eines Knotens erfolgt analog. Zusätzlich bietet die Methode `path` die Möglichkeit, aus einem Pfad die zugehörige Knotennummer zu bestimmen, indem komponentenweise über alle infrage kommenden Knoten iteriert wird, bis der gesuchte Knoten gefunden ist. Die Pfadauflösung als Teil der Computing Component zu implementieren, ermöglicht es, dass komplexere Namensauflösungen wie unser Trie-Ansatz nah an der Struktur arbeiten können.

Die Verträge der Computing Component definieren, dass der Aufrufer die Integrität der Operationen sicherstellt, also etwa dass beim Einfügen eines neuen Knotens tatsächlich ein Ordner als Elternknoten übergeben wird. Dadurch

wird zusätzlicher Berechnungsaufwand vermieden, der ohnehin bereits durch das VFS übernommen wird.

5.2.3 Abstraktionsschicht Block Device

Ein *Block Device* ist ein Speichermedium, das Daten ausschließlich in Blöcken fester Größe lesen und schreiben kann [12, S. 56]. Da ein FUSE-Dateisystem nicht direkt auf einer Partition operieren kann, wird das Block Device durch eine Datei im regulären Dateisystem simuliert. Die Klasse `BlockDevice` kapselt den Zugriff auf diese Datei über die Methoden `read_block` und `write_block`, um den Zugriff auf bestimmte Datenblöcke zu simulieren. Beim Erstellen eines Block Devices müssen neben der Blockgröße auch der Pfad zur Datenträgerdatei angegeben werden.

5.2.4 Abstraktionsschicht Allocation Manager

Diese Schicht arbeitet auf dem Block Device und abstrahiert von physischen Speicherbereichen auf logisch zusammenhängende Speicherbereiche. Die zentrale Aufgabe dieser Komponente ist die Verwaltung der verwendeten Blöcke sowie die Zuteilung von Speicher.

Über `allocate` kann ein neuer logischer Speicherbereich einer bestimmten Größe angefordert werden. Als Ergebnis erhält der Aufrufer einen Handle, also den eindeutigen Bezeichner für diesen logischen Speicherbereich. Intern muss dieser Speicherbereich nicht zwingend physisch zusammenhängend auf dem Speichermedium liegen, kann also auch fragmentiert sein. Mithilfe dieses Handles kann über `read` und `write` auf den Speicherbereich zugegriffen werden. Beide Methoden akzeptieren neben dem Handle auch die logischen Beschreibungsgrößen `size` und `offset`, sodass der Aufrufer die Daten der entsprechenden Blöcke lesen und beschreiben kann, ohne die physische Lage der Daten zu kennen.

Die Größe eines bestehenden Speicherbereichs lässt sich nachträglich über `resize` anpassen. Je nach zugrundeliegender Allokationsstrategie kann erforderlich sein, dass die vorhandenen Daten physisch in einen neuen Bereich kopiert und die alten Blöcke freigegeben werden. Daher kann `resize` auch einen neuen Handle zurückgeben, der den Speicherbereich nun identifiziert.

Wird ein Speicherbereich nicht mehr benötigt, kann dieser mit `free` anhand des Handles freigegeben werden. Die zugehörigen Blöcke werden als verfügbar markiert und können neu allokiert werden.

5.2.5 File System Manager

Der *File System Manager* (FSM) ist die zentrale Komponente, die als Schnittstelle zwischen der FLOUDS-Struktur und den darunter liegenden Speicherschichten vermittelt. Sie koordiniert das Zusammenwirken zwischen FLOUDS

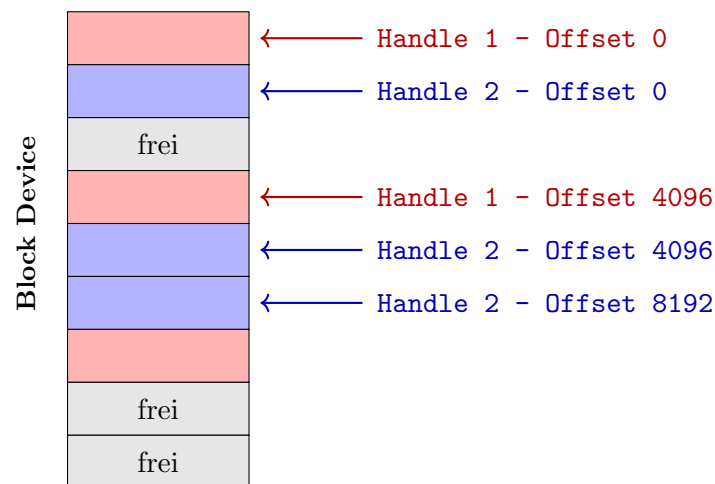


Abbildung 5.2: Fragmentierte Allokation zweier logischer Speicherbereiche auf einem Block Device.

zur Navigation, dem Inode Manager für die Metadatenverwaltung sowie dem Allocation Manager für den Zugriff auf die eigentlichen Dateiinhalte. Konkret muss die Konsistenz der Knotenindizes zur FLOUDS-Struktur sowie zur Metadaten-Struktur gewährleistet werden. Zudem steuert der FSM die Zuweisung von Speicherbereichen über den Allocation Manager und stellt die notwendigen Verwaltungsdaten des Dateisystems bereit.

Inode Manager. Die Verwaltung der Metadaten erfolgt durch die Subkomponente *Inode Manager*, die jedem FLOUDS-Knoten eine Inode-Struktur zuordnet. Dabei dient analog zur Namensstruktur der FLOUDS-Index als eindeutiger Schlüssel, um die Metadaten eines Knotens aufzurufen. Der Inode Manager stellt mit `get_inode`, `insert_inode`, `remove_inode` Methoden zum Abrufen, Einfügen und Löschen von Inodes bereit. Der FSM aktualisiert die Inode-Sequenz dazu über die definierte Schnittstelle.

Tabelle 5.1: Übersicht der Metadaten, die innerhalb einer Inode-Struktur gespeichert werden.

Bezeichnung	Verwendung	Ordner / Datei
<code>allocation_handle</code>	Logische Speicherbereich des Dateiinhalts	Nur Datei
<code>size</code>	Größe des Dateiinhalts	Nur Datei
<code>access_time</code>	Zeitpunkt des letzten Schreibzugriffs	Beide
<code>modification_time</code>	Zeitpunkt des letzten Lesezugriffs	Beide
<code>creation_time</code>	Zeitpunkt der Erstellung	Beide
<code>mode</code>	Berechtigungen innerhalb dieser Datei	Beide

Formatieren. Beim Einhängen des Dateisystems muss bei Aufruf der Methode `mount` ein zugehöriges Block Device beziehungsweise ein Pfad zur entsprechenden Datei angegeben werden. Der FSM prüft dann, ob das Block Device ein kompatibles Format aufweist. Dazu wird der erste Block ausgelesen und als `FloudsHeader` interpretiert. Beginnt dieser Block mit dem Magic-String “FLOUDS”, ist das Block Device korrekt formatiert und kann geladen werden. Der `FloudsHeader` enthält außerdem die Handles und die Größen der Speicherbereiche, wo die Informationen des Allocation Managers, die FLOUDS-Struktur sowie die Inodes gespeichert sind. Zunächst initialisiert sich der Allocation Manager anhand seines Handles selbst, da alle weiteren Ladeoperationen dessen Funktionalität voraussetzen. Danach können die FLOUDS-Struktur sowie der Inode Manager in den Arbeitsspeicher geladen werden.

Ist der Magic-String hingegen nicht vorhanden, legt der FSM einen neuen `FloudsHeader` an, initialisiert die drei leeren Komponenten und speichert die Verwaltungsinformationen auf den Datenträger.

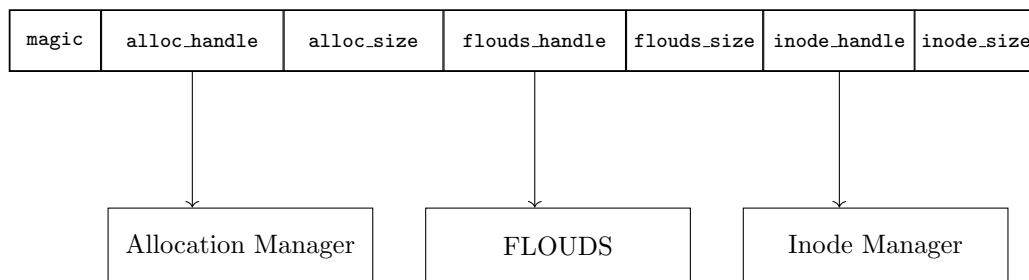


Abbildung 5.3: Aufbau des `FloudsHeader`: Handles verweisen auf die drei zentralen Dateisystemkomponenten.

Speichern. Wenn das Dateisystem erstmals initialisiert wurde oder Änderungen an der Dateienstruktur vorgenommen wurden, müssen über den Aufruf von `save` die drei Komponenten auf dem Datenträger aktualisiert werden. Zunächst wird für eine Komponente der Speicherbedarf bestimmt. Anschließend wird über den Allocation Manager ein passender logischer Bereich allokiert beziehungsweise der existierende Bereich gegebenenfalls vergrößert, falls bereits ein Handle vorliegt. Da der Allocation Manager seinen eigenen Speicherbereich selbst verwaltet, kann die Zuweisung von Speicher für sich selbst den eigenen Speicherbedarf erneut erhöhen. Daher wird die Größe nach jeder Allokation erneut geprüft und der Prozess so lange wiederholt, bis sich die benötigte Größe stabilisiert hat. Die serialisierten Daten der Komponenten werden dann in den entsprechenden logischen Bereich geschrieben. Nachdem alle Komponenten geschrieben wurden, wird abschließend der `FloudsHeader` mit aktualisierten Handles und Größenangaben in den ersten Block des Block Devices geschrieben.

5.2.6 FUSE-Integration

Der Einstiegspunkt des Dateisystem-Daemons initialisiert eine FUSE-Session über `fuse_session_new` und registriert die implementierten Operationen. Beim Einhängen des Dateisystems wird gleichzeitig der FSM initialisiert, auf den alle Operationen zugreifen. Nach jedem schreibenden Zugriff wird `save` aufgerufen, was die Konsistenz nach einem unerwarteten Absturz garantiert, jedoch zu Lasten der Schreibperformance geht.

Die Operationen greifen über `get_floods` auf die FLOODS-Struktur zu, um Anfragen an die Ordnerhierarchie zu beantworten. Um keine Fehler in den unteren Schichten der Dateisystem-Implementierung zu verursachen, übernimmt die FUSE-Schnittstelle die Integritätsprüfung einer Anfrage.

Tabelle 5.2: Übersicht der implementierten FUSE-Operationen.

Operation	Beschreibung
<code>init</code>	Die FUSE-Session wurde gestartet. Daher wird unser FSM initialisiert und der angegebene Pfad wird als Block Device montiert.
<code>destroy</code>	Die FUSE-Session wurde beendet. Unser FSM wird über <code>unmount</code> verworfen und das Dateisystem wird noch einmal abschließend auf dem Block Device gespeichert.
<code>lookup</code>	Sucht einen Eintrag innerhalb eines Verzeichnisses durch Iteration über alle Kinderknoten in der FLOODS-Struktur und Vergleich der Namen.
<code>getattr</code>	Liest die Attribute eines Knotens aus dem Inode Manager und konvertiert diese in die von FUSE erwartete <code>stat</code> -Struktur.
<code>setattr</code>	Aktualisiert die Metadaten eines Knotens über den Inode Manager, wobei Berechtigungen, Größe und Zeitstempel angepasst werden können.
<code>open</code>	Prüft über die FLOODS-Struktur, ob der angegebene Knoten eine Datei ist und erlaubt den Zugriff.
<code>read</code>	Liest Dateiinhalte über den FSM.
<code>write</code>	Schreibt Dateiinhalte über den FSM.
<code>readdir</code>	Iteriert über alle Kindknoten eines Ordners in der FLOODS-Struktur und konvertiert diese in FUSE-Directory-Entries inklusive der Spezialeinträge <code>.</code> und <code>...</code>
<code>mkdir</code>	Erstellt einen neuen Ordnerknoten über den FSM.
<code>create</code>	Erstellt einen neuen Dateiknoten über den FSM.
<code>unlink</code>	Löscht eine Datei innerhalb eines Verzeichnisses durch Iteration über alle Kinderknoten in der FLOODS-Struktur und Vergleich der Namen.
<code>rmdir</code>	Löscht einen leeren Ordner analog zu <code>unlink</code> .

Tabelle 5.2 – Fortsetzung

Operation	Beschreibung
<code>statfs</code>	Fragt Dateisystem-Statistiken über den FSM für Benchmarking ab. Anfragen werden an das Block Device weitergeleitet.

Caching. Die Low-Level-API arbeitet mit stabilen Inode-Nummern, die FUSE als unveränderlich voraussetzt. Da FLOUDS-Nummern bei Änderungsoperationen jedoch verschoben werden, können diese nicht direkt als FUSE-Inodes verwendet werden. Ein naiver Ansatz wäre, alle Attribut-Timeouts auf null Sekunden zu setzen, sodass FUSE keine Inodes cacht. Dies würde aber zu unnötig häufigen Anfragen des VFS führen. Die Komponente `Delta-Stabilization` löst dieses Problem durch die delta-basierte Stabilisierung. Vor jeder Operation wird die stabile FUSE-Inode über `stable_inode_to_floods_inode` in die aktuelle FLOUDS-Nummer übersetzt. Bei der Antwort an das VFS erfolgt eine Umkehrung über `floods_inode_to_stable_inode`. Einfügungen und Löschungen werden dazu über `record.insert` und `record.remove` in der Historie festgehalten.

5.2.7 Zusammenspiel der Komponenten

Die Architektur folgt einem mehrschichtigen Aufbau, bei dem jede Schicht ausschließlich mit der direkt darunterliegenden über klar definierte Schnittstellen kommuniziert. Eingehende VFS-Anfragen werden durch die FUSE-Schicht entgegengenommen und aufbereitet an den FSM weitergeleitet. Dieser koordiniert die FLOUDS-Struktur für die Navigation sowie den Inode Manager für den Zugriff auf Metadaten. Beide verwenden zur Speicherung den Allocation Manager, der die logischen Speicherbereiche auf dem Block Device verwaltet. Insbesondere auf den unteren Schichten, etwa der Computing Component und dem Allocation Manager zur Verwaltung freier Blöcke, können succincte Datenstrukturen zum Einsatz kommen.

5.3 Umsetzung der Kernfunktionen

5.3.1 Dateioperationen

Der File System Manager stellt die Methode `add_node` zur Verfügung, die nach vorangegangener Integritätsprüfung durch die FUSE-Integration aufgerufen wird. Die Computing Component fügt einen Knoten in die FLOUDS-Struktur ein. Der resultierende FLOUDS-Index dient als neuer Schlüssel für die Inode-Struktur, wodurch sich alle nachfolgenden Inode-Schlüssel verschieben.

Beim Beschreiben einer Datei erlaubt die FUSE-Schnittstelle neben den zu schreibenden Daten und deren Größe auch die Angabe eines Offsets, der

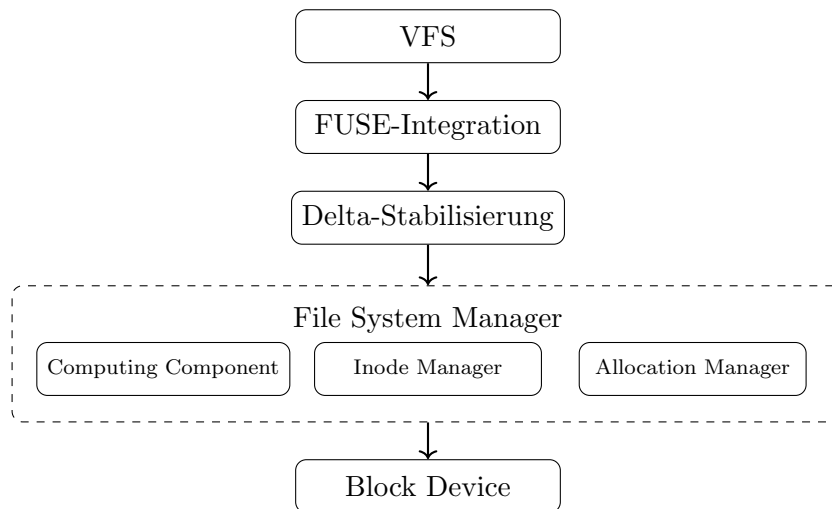


Abbildung 5.4: Schichtenarchitektur unseres Dateisystems. Schichten kommunizieren über klar definierte Schnittstellen und nutzen succincte Datenstrukturen.

bestimmt, wo die Daten logisch in der Datei beginnen sollen. Die Methode `write_file` prüft anhand der Metadaten, ob der verfügbare Platz ausreicht ($\text{offset} + \text{size} > \text{inode.size}$). Falls nicht, vergrößert der Allocation Manager den verfügbaren Platz oder allokiert einen neuen Bereich. Anschließend beschreibt der Allocation Manager den logischen Speicherbereich. Die Methode `read_file` funktioniert analog und leitet entsprechende Leseanfragen an den Allocation Manager weiter.

Um eine bessere Allokationsentscheidung zu treffen und diese möglichst weit hinauszuzögern wird optional im Sinne der Delayed Allocation ein temporärer Schreibpuffer mit einer Kapazität von 16 Blöcken definiert. Bei `write_file` werden Daten zunächst in diesen Puffer übernommen, statt sofort eine physische Allokation durchzuführen. Die Methode `read_file` berücksichtigt den Puffer ebenfalls, sodass noch nicht persistente Änderungen gelesen werden können. Ein Flush erfolgt erst dann, wenn der Puffer voll ist, wenn auf eine andere Datei geschrieben werden soll oder Änderungen an der Ordnerstruktur durchgeführt werden sollen.

Zum Entfernen eines Knotens ruft die FUSE-Integration nach Ermittlung des entsprechenden Knotens die Methode `remove_node` auf. Der Allocation Manager gibt zunächst den logischen Speicherbereich frei. Anschließend entfernt die Computing Component den Knoten aus der FLOUDS-Struktur. Der zugehörige Schlüssel wird ebenfalls aus der Inode-Struktur entfernt, sodass alle nachfolgenden Inode-Schlüssel verschoben werden.

5.3.2 FLOUDS-Nummern

Wird die Delta-Stabilisierung eingesetzt, übernimmt die Komponente **Delta-Stabilization** die bijektive Übersetzung zwischen stabilen Inode-Nummern und aktuellen FLOUDS-Nummern. Da für die spätere Rückübersetzung auch die Position im Log relevant ist, an der eine FLOUDS-Nummer einem stabilen Inode zugeordnet wurde, wird in den höchstwertigen Bits des stabilen Inodes die aktuelle Position innerhalb des Logs kodiert. Die verbleibenden Bits enthalten die FLOUDS-Nummer.

Quelltext 5.2: Übersetzung einer instabilen FLOUDS-Nummer in einen stabilen Inode durch Zusammensetzung der aktuellen Log-Länge und der FLOUDS-Nummer.

```
uint64_t flouds_inode_to_stable_inode(uint64_t
    flouds_inode) {
    uint64_t inode_number = flouds_inode + 1;
    return (next_operation_index << 48) | inode_number;
}
```

Diesen zusammengesetzten 64-Bit-Integer wird FUSE nun (möglichst lange) cachen und für weitere Operationen verwenden. Bei der Rückübersetzung wird die zugehörige Position im Log bestimmt und es werden alle seither erfolgten Operationen rückgängig gemacht.

Damit der Log nicht zu lang wird, wodurch die Übersetzung ineffizient werden würde, und da sonst die Log-Position nicht in 16 Bits passen würde, begrenzen wir die Länge des Logs auf die letzten 256 Einträge. Versucht FUSE einen Inode mit nicht mehr existenten Daten anzufragen, wird **ESTALE** zurückgegeben, wodurch das VFS den Inode erneut ermittelt. Die Log-Position wird bewusst größer kodiert als der Log maximal lang sein kann, damit zwischen abgelaufenen und aktiven Inodes differenziert werden kann.

5.3.3 Metadaten

Neben einer naiven Strategie zur sequentiellen Speicherung von Inodes implementieren wir eine zweite Strategie mit hierarchischer Speicherung der Inodes. Da FLOUDS-Nummern in Breitensuchreihenfolge entlang der Verzeichnishierarchie vergeben werden, liegen die Nummern von Knoten desselben Ordners nah beieinander. Wir können im Sinne der Lokalität diese Eigenschaft nutzen, indem die logische sequentielle Inode-Liste in lokale Abschnitte unterteilt wird.

Konkret teilen wir die Inode-Liste in viele Elemente der Struktur **InodeChunk** auf. Da unsere Inode-Struktur 48 Bytes benötigt, sollte bei einer Blockgröße von 4096 Bytes ein Chunk maximal $\lfloor 4096/48 \rfloor = 85$ Inodes beinhalten. Zu jedem Chunk speichern wir neben dem Handle des logischen Speicherbereichs auch den Index des ersten Inodes und die Anzahl der Inodes in diesem Chunk. Über die Hilfsmethode **find_chunk** kann der Chunk eines Inodes gefunden werden und dann durch **load_chunk** in den Cache geladen werden. Beim Einfügen

eines Inodes in einen Chunk wird geprüft, ob der Chunk noch ausreichend Platz zur Verfügung hat. Falls nicht, wird ein zweiter Chunk erstellt und aufgeteilt.

Verändert sich der lokal gecachte Chunk, wird dieser über seinen Handle mithilfe des Allocation Managers auf das Block Device zurückgeschrieben. Während die naive sequentielle Strategie sämtliche Inodes des Dateisystems direkt liest oder auf das Block Device schreiben muss, organisiert die hierarchische Strategie nur einige Verwaltungsdaten und muss diese zusätzlich zu kleinen Teilbereichen lesen und auf das Block Device schreiben.

5.3.4 Serialisierung und Speicherlayout

Um sauber auf den Massenspeicher zu schreiben und von diesem zu lesen, wird die Schnittstelle `Serializable` definiert. Sie wird von allen succinthen Datenstrukturen (Bitvektor, Wavelet Tree, Namenssequenzen) sowie von den übergeordneten Schichten implementiert, die auf das Block Device geschrieben werden müssen (FLOUDS, Allocation Manager, Inode Manager).

Die Methode `get_serialized_size` gibt zurück, wie viele Bytes das serialisierte Datenobjekt benötigt, damit vorab ein entsprechend großer Bereich reserviert werden kann. Der wortbasierte Bitvektor etwa wird serialisiert, indem zunächst die Anzahl der Bits und anschließend jedes einzelne Wort geschrieben wird. Bei n Bits und einer Wortgröße von 64 Bits ergibt sich ein Bedarf von $8 + 8 \cdot \lceil n/64 \rceil$ Bytes.

Die Methode `serialize` schreibt das Datenobjekt ab einem gegebenen Offset in einen Byte-Buffer ausreichender Größe und aktualisiert den Offset anschließend auf die neue Position. Bei der Serialisierung der FLOUDS-Struktur beispielsweise werden alle drei enthaltenen Datenstrukturen nacheinander in denselben Buffer geschrieben, wobei sich der Offset gemäß des Vertrags nach jeder Teilserialisierung entsprechend verschiebt.

Quelltext 5.3: Serialisierung des wortbasierten Bitvektors und der FLOUDS-Struktur. Beim Bitvektor wird zunächst die Anzahl der Bits, dann jedes Wort in den Buffer geschrieben, wobei der Offset jeweils aktualisiert wird. Bei FLOUDS werden die enthaltenen Strukturen nacheinander serialisiert.

```
void WordBitVectorStrategy::serialize(char* buffer ,
    size_t* offset) override {
    size_t size = this->size();
    memcpy(buffer + *offset , &size , sizeof(size_t));
    *offset += sizeof(size_t);
    for (size_t i = 0; i < words.size(); i++) {
        memcpy(buffer + *offset , &words[i] ,
            sizeof(size_t));
        *offset += sizeof(size_t);
    }
}
```

```

void Flouds::serialize(char* buffer , size_t* offset) {
    structure->serialize(buffer , offset);
    types->serialize(buffer , offset);
    names->serialize(buffer , offset);
}

```

Wurden die Bytes eines Datenobjekts vom Massenspeicher geladen, wird zur Deserialisierung **deserialize** auf einer leeren Instanz des entsprechenden Objekts aufgerufen. Übergeben werden der Byte-Buffer sowie der Offset, ab dem das Datenobjekt beginnt. Auch hier soll der Offset nach abgeschlossener Deserialisierung aktualisiert werden.

Beim erstmaligen Formatieren des Dateisystems, werden alle drei leeren Strukturen serialisiert und in einen neu allokierten Block abgelegt. Zusätzlich wird der erste Block für den Header reserviert. Darum benötigt ein leeres FLOUDS-Dateisystem bei einer Blockgröße von 4096 genau 16 KiB. Das ist deutlich weniger Overhead als bei ext4 durch Group Descriptors, Inode-Tabellen sowie Block- und Inode-Bitmaps.

5.3.5 Allokationsstrategien

Wir implementieren zwei Allocation Manager: einen für zusammenhängende Best-Fit-Allocation sowie einen für Extent-basierte Allokation. Beide Strategien verwalten einen dynamischen Bitvektor, der belegte Blöcke mit einem 1-Bit markiert. Da der Bitvektor beliebig vergrößert werden kann, wenn neue Blöcke benötigt werden, abstrahieren wir theoretisch unbegrenzten Speicherplatz. Die Methoden **get_total_blocks** und **get_used_blocks** liefern Informationen zum verwendeten Speicherplatz, die später über den FUSE-Endpunkt **statfs** beim Benchmarking relevant sein werden.

Best-Fit-Allocation. Bei der Allokation durchsucht diese Strategie den Bitvektor linear und identifiziert den kleinsten zusammenhängenden Bereich, der gerade noch groß genug ist, um die Anforderungen zu erfüllen. Die entsprechenden Blöcke werden als belegt markiert, und der erste Block dieses Bereichs wird als Handle zurückgegeben. Falls kein ausreichend großer freier Bereich existiert, werden neue Blöcke am Ende des Bitvektors eingefügt.

Das Lesen und Schreiben innerhalb dieses logischen Bereichs ist einfach, da lediglich der physische Block über **block_index = handle + [(offset + bytes_processed/block_size)]** bestimmt werden muss. Da Zugriffe nicht zwingend an Blockgrenzen ausgerichtet sind, müssen bei Schreibvorgängen Blöcke zunächst gelesen, modifiziert und zurückgeschrieben werden.

Beim Verkleinern des Bereichs werden die letzten nicht mehr benötigten Blöcke im Bitvektor als frei markiert, während das Handle unverändert bleibt.

Beim Vergrößern prüft die Strategie zunächst, ob die nachfolgenden Blöcke noch frei sind. Falls nicht, werden ein neuer Bereich allokiert, die Daten kopiert, der alte Bereich freigegeben und der neue Handle zurückgegeben.

Extent-basierte Allokation. Die Struktur `Extent` besteht aus dem Index des Startblocks (`start_block`) sowie der Anzahl der zusammenhängenden Blöcke (`num_blocks`). Um $\mathcal{O}(\log n)$ -Zugriff auf die Extents eines Handles zu ermöglichen, werden diese in einem Dictionary verwaltet. Als Schlüssel dient der Startblock des ersten Extents eines Handles, der als Wert eine Liste aller zugehörigen Extents assoziiert.

Die Hilfsmethode `allocate_extents` reserviert für eine gegebene Anzahl benötigter Blöcke eine Liste von Extents. Dazu durchsucht sie den Bitvektor nach möglichst großen freien Bereichen, die mindestens ein Drittel der Anforderung erfüllen, um Fragmentierung zu vermeiden. Verbleibender Speicherbedarf wird am Ende des Block Devices durch Erweiterung des Bitvektors reserviert. Bei Allokation oder Vergrößerung werden durch diese Methode neue Extents reserviert und zur Extent-Liste des Handles hinzugefügt.

Lesen und Schreiben von Speicher funktioniert analog zur Best-Fit-Allocation, wobei zunächst über die logische Position der korrekte Extent identifiziert werden muss. Hierbei wird iterativ über alle Extents eines Handles summiert, bis der Zielbereich des Offsets erreicht ist.

6 Benchmarking

Ziel des Benchmarkings ist es, das implementierte Dateisystem und dessen Variationen anhand der Laufzeit, gemessen als Durchsatz in Operationen pro Sekunde, sowie der Speichereffizienz, gemessen an tatsächlich belegten Speicherplatz, zu bewerten. Diese Dimensionen erlauben eine fundierte Einschätzung, für welche Anwendungsfälle ein FLOUDS-Dateisystem geeignet ist.

6.1 Methodik

Das Benchmarking orientiert sich methodisch einerseits an einer Performance-Charakterisierung des FUSE-Frameworks [31], die zeigt, dass sowohl der Workload als auch die konkrete FUSE-Konfiguration den Overhead durch FUSE erheblich beeinflussen. Andererseits wird berücksichtigt, dass verschiedene Parameter wie der Cache-Zustand oder die beobachtete Größe das Ergebnis eines Dateisystems-Benchmarkings stark variieren können [29].

6.1.1 Vergleichssysteme

Es werden drei Dateisysteme verglichen:

- **ext4 (nativ)**: Das etablierte ext4-Dateisystem mit einer Blockgröße von 4096 Bytes wird als 100GB-System über eine Image-Datei formatiert und direkt über das VFS eingebunden.
- **ext4 (Passthrough)**: Um den FUSE-spezifischen Overhead vom Overhead des Dateisystems zu trennen, wird ext4 auch über einen FUSE-Passthrough (bindfs) eingebunden [31].
- **FLOUDS**: Unser FLOUDS-Dateisystem über Low-Level-FUSE.

6.1.2 Workloads

Die durchsatzspezifisch Workloads werden mit Filebench [28] durchgeführt. Filebench erlaubt es eine steuerbare und reproduzierbare Last auf ein Dateisystem zu erzeugen. Die verwendeten Workloads decken vier Kategorien ab:

- **Struktur**: Erstellen, Löschen und Traversieren von Dateien und Ordnern. Hier wird insbesondere die Hierarchieverwaltung berücksichtigt.

- **Lesen / Schreiben:** Sequentielle und zufällige Lese- oder Schreiboperationen auf verschiedenen Datenmengen.
- **Makro:** Synthetische Workloads mit gemischten Lese- und Schreiboperationen, die reale Anwendungsfälle nachbilden.
- **Speicher:** Definierte Datenmengen, die auf dem Dateisystem abgelegt werden. Relevant ist hier ausschließlich der belegte Speicher.

Die Kategorisierung der Workloads folgt der Klassifikation von Tarasov [29], der zwischen metadaten- und datenintensiven Workloads unterscheidet. Die Metadaten-Workloads sind für FLOUDS besonders aussagekräftig und relevant, da das Dateisystem durch den succinten Ansatz insbesondere für Strukturoperationen optimiert ist. Metadaten-intensive Workloads erzeugen einen großen FUSE-Overhead [31], was ein Vergleichs-Benchmarking mit dem Pass-through umso wichtiger macht. Konkret verwenden wir folgende Workloads:

Tabelle 6.1: Übersicht der verwendeten Workloads.

Workload	Beschreibung	Kategorie
append_small_1000	Append-Operationen auf 1.000 kleinen Dateien in verschachtelter Ordnerstruktur	Lesen / Schreiben
create_dirs_deep_250000	Erstellen einer tiefen Ordnerstruktur mit 250.000 Knoten	Struktur
create_dirs_flat_250000	Erstellen einer flachen Ordnerstruktur mit 250.000 Knoten	Struktur
dirs_250000	Tiefe Ordnerstruktur mit 250.000 Knoten	Speicher
create_small_5000	Erstellen einer Ordnerstruktur mit 5.000 kleinen Dateien	Struktur
delete_small_5000	Löschen von 5.000 kleinen Dateien aus einer Ordnerstruktur	Struktur
dirops_deep_5000	Verschiedene Strukturoperationen in einer tiefen Ordnerstruktur mit 5.000 Knoten	Struktur
dirops_flat_5000	Verschiedene Strukturoperationen in einer flachen Ordnerstruktur mit 5.000 Knoten	Struktur
fileserver_read_500	Simulierter lesender Zugriff auf einen Dateiserver mit 500 Knoten	Makro
fileserver_read_5000	Simulierter lesender Zugriff auf einen Dateiserver mit 5.000 Knoten	Makro
fileserver_rw_500	Simulierter lesender und schreibender Zugriff auf einen Dateiserver mit 500 Knoten	Makro
fileserver_rw_5000	Simulierter lesender und schreibender Zugriff auf einen Dateiserver mit 5.000 Knoten	Makro
flouds_src	Quellcode des FLOUDS-Dateisystems	Speicher
linux_src	Quellcode des Linux-Kernels v6.19	Speicher
open_close_5000	Öffnen- und Schließen auf Ordnerstruktur mit 5.000 kleinen Dateien	Struktur
openwrt_rootfs	Rootfs eines eingebetteten Linux-Systems (OpenWrt 23.05.3)	Speicher
remove_dirs_deep_20000	Löschen einer tiefen Ordnerstruktur mit 20.000 Knoten	Struktur
remove_dirs_flat_20000	Löschen einer flachen Ordnerstruktur mit 20.000 Knoten	Struktur
rread_1g	Zufälliges Lesen von 4KB-Blöcken aus einer 1GB-Datei	Lesen / Schreiben
rwrite_1g	Zufälliges Schreiben von 4KB-Blöcken in eine 1GB-Datei	Lesen / Schreiben
seqread_1g	Sequentielles Lesen von 1MB-Blöcken aus einer 1GB-Datei	Lesen / Schreiben
seqwrite_1g	Sequentielles Schreiben von 1MB-Blöcken in eine 1GB-Datei	Lesen / Schreiben

Die Unterscheidung zwischen flachen und tiefen Ordnerhierarchien ist auf die Beschaffenheit von FLOUDS zurückzuführen. Da die LOUDS-Struktur in Breitensuchreihenfolge kodiert, kann die Hierarchietiefe die Performance beeinflussen und ein Vergleich mit dem pointerbasierten ext4-Ansatz erscheint hier sinnvoll.

6.1.3 Versuchsaufbau und Durchführung

Alle Workloads werden auf einer NVMe M.2 SSD (Ediloca EN605, 256 GB) sowie einer SATA-HDD (Toshiba, 5.400 rpm, 960 GB) durchgeführt. Die Benchmarks werden auf einem System mit 16 GB Arbeitsspeicher und einer AMD Ryzen 5 7600 CPU durchgeführt. Verwendet wird Debian 13, das jeweils auf dem zu testenden Datenträger installiert ist.

Jeder durchsatzspezifische Workload wird pro Konfiguration fünfmal mit einer Maximaldauer von 10 Sekunden durchgeführt. Durch mehrfache Wiederholung können Mittelwert und Standardabweichung berechnet werden. Benchmarks mit einer Standardabweichung von mehr als 10% werden wiederholt, da vermutlich äußere Einflüsse die Ergebnisse verändert haben. Vor jedem Lauf wird das Dateisystem neu initialisiert und eingehängt.

Die Benchmarking-Suite ist als Python-Skript implementiert und steuert Initialisierung, Ausführung und Auswertung vollständig automatisiert. Die Ergebnisse werden als CSV-Daten geschrieben, auf dessen Basis die Auswertung erfolgt.

6.2 Ergebnisse

Die Benchmarking-Ergebnisse der Baseline- sowie der verschiedenen FLOUDS-Konfigurationen sind nach Kategorien gruppiert im Anhang aufgeführt. Ausgewiesen ist jeweils der Mittelwert der fünf Durchläufe in Operationen pro Sekunde. Bei den Speicher-Workloads wird hingegen der über `statfs` belegte Speicherplatz ausgewiesen, der unabhängig vom verwendeten Speichermedium ist.

Um den Einfluss einzelner Komponenten isoliert bewerten zu können, wurde jeweils ausgehend von einer Standardkonfiguration genau eine Komponente variiert. Die Standardkonfiguration verwendet Best-Fit-Allokation, sequentielle Inode-Verwaltung, einen Maschinenwort-basierten Bitvektor sowie eine RRB-Namenssequenz (*immer*) und verzichtet auf Delayed Allocation und Delta-Stabilisierung. Konkret wurden folgende Variationen untersucht:

Tabelle 6.2: Übersicht der getesteten FLOUDS-Konfigurationen.

Konfiguration	Variation	Hypothese
<i>Allokation</i>		
Standard	Referenz	-
Delayed Allocation	Blockzuweisung bei Erstellung	Verbesserte Schreibperformance
Extent	Extent-basierte Allokation	Reduzierte Fragmentierung, verbesserte Schreibperformance
<i>Bitvektoriimplementierung</i>		
Adaptive BV	Adaptiver dynamischer Bitvektor [16]	Schnellere Anfragen bei statischen Daten
Optimized BV	Hochoptimierter Bitvektor [5]	Schnellere Strukturoperationen- und Anfragen
<i>Namens- und Metadatenverwaltung</i>		
Concatenated NS	Konkatenierte Namenssequenz	Schlechtere Änderungsperformance

Tabelle 6.2 – Fortsetzung

Konfiguration	Variation		Hypothese
Hash NS	Hash-basierte Namenssequenz		Schnellere Pfadauflösung durch $\mathcal{O}(1)$ -Lookup, dadurch bessere Gesamtpformance
Hierarchy	Hierarchische Verwaltung	Inode-	Bessere Cache-Lokalität, höherer Speicheraufwand
<i>Integration</i> Delta	Delta-basierte Nummern	FLOUDS-	Effizienteres FUSE-Caching durch stabile Inode-Nummern
<i>Kombination</i> Optimiert	Hash NS + Delta + Hierarchy		Kombination der vielversprechendsten Einzelvarianten

6.3 Auswertung

Im Folgenden werden die Ergebnisse des Benchmarkings in den Gesamtzusammenhang succincter Datenstrukturen im Kontext von Dateisystemen eingeordnet.

6.3.1 Speicherbedarf

Eines der Kernziele von FLOUDS ist die Reduzierung des praktischen Speicherbedarfs. Bei stark verschachtelten, synthetischen Ordnerstrukturen (dirs_250-000) gelingt es durch den succincten Ansatz, bis zu 97 % gegenüber ext4 einzusparen. Auch die realen Datensätze werden deutlich platzsparender gespeichert. Der relative Vorteil fällt dabei umso größer aus, je kleiner der Datensatz ist: Da FLOUDS keinen fixen initialen Speicherbedarf für leere Datenstrukturen wie eine Inode-Tabelle hat, sondern die Strukturen dynamisch wachsen, entfällt der bei ext4 stets anfallende Verwaltungs-Overhead. Bei großen Datensätzen wie dem Linux-Kernel-Quellcode (linux_src) nähern sich die Werte erwartungsgemäß an (1682,02 MiB zu 1714,24 MiB, -2%), da der Overhead relativ zur Nutzdatenmenge sinkt. Es ist aber anzumerken, dass die vorliegende Implementierung weniger Funktionen als ext4 bietet, da beispielsweise das Journaling sowie Hard- und Softlinks fehlen, was den Vergleich zugunsten von FLOUDS verzerrt. Dennoch lässt sich festhalten, dass succincte Dateisysteme insbesondere für speicherplatzbeschränkte und leichtgewichtige Systeme eine attraktive Option darstellen. Zwischen den FLOUDS-Konfigurationen zeigen sich kaum Unterschiede, wobei die hierarchische Inode-Verwaltung jedoch erwartungsgemäß minimal mehr Speicher belegt.

6.3.2 Baseline und FUSE-Overhead

Die beobachteten Ergebnisse der Baseline-Benchmarks von ext4 mit und ohne FUSE-Passthrough decken sich mit den Erkenntnissen von Vangoor et al. [31], die zeigen, dass der FUSE-Overhead abhängig vom verwendeten Workload

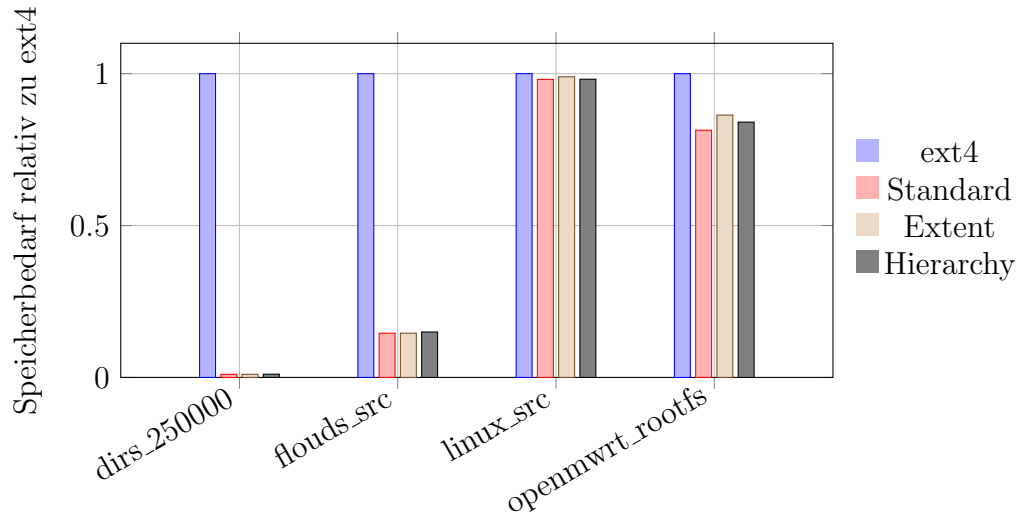


Abbildung 6.1: Speicherverbrauch ausgewählter Konfigurationen im Vergleich zu ext4.

stark variieren kann. So lässt sich beim Workload `append_small_1000` ein Performanceverlust von 75% beobachten, während beim Workload `delete_small_5000` keine signifikanten Veränderungen festzustellen sind. Dies rechtfertigt einen Vergleich des FLOUDS-Dateisystems ausschließlich mit dem ext4-Pass-through, da nur so der Overhead des Dateisystems selbst isoliert betrachtet werden kann. Ein direkter Vergleich mit ext4 ist also nicht sinnvoll.

Auffällig ist, dass bereits bei den Baseline-Benchmarks die HDD-Ergebnisse im Durchschnitt lediglich 1,3% unter denen der SSD liegen (ohne `seqwrite_1g`). Das lässt sich darauf zurückführen, dass die betrachteten Workloads primär metadatenintensiv sind und durch das Page-Caching [27, S. 517] des Betriebssystems die Laufzeit durch den Overhead des Dateisystems, aber nicht durch die Geschwindigkeit des Speichermediums, bestimmt wird. Lediglich bei schreibintensiven Workloads mit großen Datenmengen wie `seqwrite_1g` zeigen sich erwartungsgemäß Unterschiede zwischen HDD und SSD. Dass der Datenträger und speicherintensive Prozesse keinen Einfluss auf die Ergebnisse haben und diese *memory-bound* sind, hat den Vorteil, dass die Benchmarking-Ergebnisse die konkreten Eigenschaften des Dateisystems isoliert widerspiegeln und nicht durch externe Effekte überlagert werden [29]. Der Flaschenhals muss also in den höheren Schichten des VFS liegen, was sich insbesondere durch die Workloads `create_dirs_deep_20000` und `create_dirs_flat_20000` zeigt, bei denen alle Konfigurationen nahezu identische Ergebnisse erzielt haben. Dadurch liegen die Ergebnisse nahe an einer statischen Grenze.

6.3.3 Allokationsstrategien

Die Verwendung der Delayed Allocation hat einen positiven Einfluss auf die Performance der Workloads, bei denen kleine Dateien geschrieben werden (`append_small_1000`: +78,3%, `fileserver_rw_500`: +18,9%). Es ist kein signifikanter positiver oder negativer Effekt beim Schreiben großer Dateien oder lesender Zugriffe festzustellen. Die Nutzung der Delayed Allocation kann in der Praxis sinnvoll sein, damit nah beieinanderliegende Dateien auch in lokaler Nähe auf dem Datenträger gesichert werden. Dieser potenzielle Vorteil wird allerdings durch die definierten Benchmarks nicht direkt gemessen. Die Nutzung der Delayed Allocation belegt konstant 16 Blöcke Arbeitsspeicher, also insgesamt 64 KiB. Deswegen ist sie trotz eines insgesamt leicht signifikanten Einflusses (+6,1%) im Rahmen eines succinthen Dateisystems nicht sinnvoll.

Die Extent Allocation bringt einen hohen Verwaltungsoverhead mit sich, der insbesondere bei kleinen Daten sichtbar wird (`create_small_5000`, durchschnittlich -19,6%). Eine Ausnahme bildet der Workload `seqwrite_1g` auf der HDD: Hier erzielt die Extent-Konfiguration mit 310 gegenüber 191 Operationen pro Sekunde einen deutlichen Vorteil von +62,3%. Dieser Unterschied lässt sich auf die physischen Eigenschaften rotierender Festplatten zurückführen. Muss ein logischer Speicherbereich beim Vergrößern in einen neuen, zusammenhängenden Bereich umkopiert werden, sind die dabei entstehenden Schreibzugriffe auf einer HDD verhältnismäßig teuer. Extent-basierte Allokation verzichtet auf diese Kopiervorgänge. Auf einer SSD ist dieser Vorteil nicht beobachtbar, da zufällige und sequentielle Zugriffe annähernd gleich schnell sind. Für den Zielanwendungsfall von FLOUDS, also in ressourcenbeschränkten Systemen mit kleinen Hierarchien, überwiegt der Verwaltungsoverhead dem potentiellen Nutzen deutlich. Der beschriebene Vorteil der Extent-Allokation würde sich erst bei intensiver Nutzung und damit eingehender Fragmentierung des Datenträgers bemerkbar machen. Die Best-Fit-Allokation stellt daher für unseren Anwendungsfall die geeignetere Strategie dar.

6.3.4 Bitvektorimplementierung

Die hochoptimierte Bitvektorimplementierung zeigt entgegen der Erwartungen bei strukturintensiven Workloads einen systematischen Nachteil (-19,1%) gegenüber der einfachen Maschinenwort-Implementierung. Bei Speichermediums- oder VFS-limitierten Workloads hingegen ist kein Unterschied messbar. Dieses Muster lässt sich auf eine zentrale Designentscheidung der Bibliothek zurückführen: Die Implementierung puffert Änderungsoperationen an den Blättern des B-Baumes, um Einfüge- und Löschoperationen auf Kosten der Rank- und Select-Anfragen zu beschleunigen [5]. Da die FLOUDS-Navigation (`parent`, `child`, `children_count`) direkt auf Rank- und Select-Operationen abgebildet wird, erhöht sich dieser Overhead bei navigationsintensiven Workloads erheblich. Da Anfragen und Änderungen in einem Dateisystem verschränkt statt in

getrennten Batches auftreten, ist diese Implementierung für den Anwendungsfall eines Dateisystems ungeeignet.

Der adaptive Bitvektor zeigt bei leseintensiven Workloads keinen signifikanten Vor- oder Nachteil. Bei schreibintensiven Workloads hingegen ist ein erheblicher Leistungseinbruch festzustellen: Der Workload `rwrite_1g` fällt von 19.895 auf nur 92 Operationen pro Sekunde. Dies lässt sich durch das beschriebene Verhalten dieser Strategie erklären: Trifft ein Update auf ein statisches Blatt, wird dieses kostspielig rekursiv aufgespalten. Bei zufälligen Schreibzugriffen trifft dieser Fall bei nahezu jeder Operation auf.

Insgesamt zeigt sich, dass die Bitvektorimplementierung nicht der Flaschenhals des Dateisystems ist und durch alternative Implementierungen kaum Gewinne erzielt werden können. Die naive Implementierung ist bei ausgeglichener Nutzung von Lese- und Schreiboperationen die geeignetste Wahl.

6.3.5 Namens- und Metadatenverwaltung

Die Wahl der Namens- und Metadatenverwaltung erweist sich als Konfigurationsvariable mit einem breiten, workloadübergreifenden Effekt, der sich deutlich von den isolierten Einflüssen der Bitvektor- oder Allokationsvarianten unterscheidet.

Die konkatenierte Namenssequenz erzielt teils deutliche Vorteile gegenüber der Standardkonfiguration bei Workloads, die Änderungsoperationen auf kleinen Strukturen ausführen (`append_small_1000`: +78,8%, `fileserver_rw_5000`: +25%). Workloads mit vielen aufeinanderfolgenden Such-Operationen hingegen brechen massiv ein. Die konkatenierte Darstellung ermöglicht schnellen Zugriff, erfordert bei Lookup-Operationen jedoch das sequentielle Durchlaufen der gesamten Struktur. Diese Implementierung ist also für Anwendungsfälle geeignet, bei denen Daten überwiegend geschrieben und selten gesucht werden, also etwa für Archivierungssysteme.

Die Hash-basierte Namenssequenz zeigt ein konsistent positives Verhalten (insgesamt +32,4%). Strukturintensive Workloads, insbesondere mit vielen Einfüge- und Löschoperationen, erzielen vergleichsweise die stärksten Verbesserungen: `create_small_5000` und `remove_dirs_deep_20000` verdoppeln sich jeweils. Die Implementierung profitiert davon, dass Änderungsoperationen ohne physisches Verschieben von Einträgen realisierbar sind. Sie stellt damit für den typischen Dateisystem-Anwendungsfall die geeignetste Namenssequenz dar.

Die hierarchische Inode-Verwaltung zeigt bei leseintensiven Workloads einen klaren Vorteil, der auf bessere Cache-Lokalität zurückzuführen ist. Schreibintensive Workloads weisen hingegen einen Nachteil auf (`fileserver_rw_500`, -22,1%), der auf den Overhead beim Aufteilen der Chunks zurückzuführen ist. Dieser Overhead amortisiert sich jedoch mit wachsender Strukturgröße, da der relative Anteil der Chunk-Verwaltung an der Gesamtlaufzeit sinkt. Für klas-

sische Anwendungsfälle ist die hierarchische Inode-Verwaltung daher klar zu bevorzugen.

Es lässt sich bei der Effizienz von FLOUDS kein signifikanter Unterschied zwischen flachen und tiefen Ordnerstrukturen feststellen.

6.3.6 Delta-Stabilisierung

Auf Workloads, bei denen FUSE Inode-Nummern über mehrere Operationen hinweg wiederholt nutzt, also solche, bei denen das VFS von einem funktionierenden Inode-Caching profitiert, hat die Delta-Stabilisierung einen signifikant positiven Effekt (insgesamt +26,9%). Bei strukturintensiven Workloads sind die Verbesserungen erheblich (+30,8%). Der Overhead der Delta-Berechnung scheint vernachlässigbar gering zu sein.

Besonders aufschlussreich ist, dass der Effekt den Größenunterschied zwischen kleinen und großen Strukturen relativiert. Ohne Delta-Stabilisierung ist der `fileserv_read`-Workload mit 500 Knoten um den Faktor 1,24 schneller als mit 5.000 Knoten. Das ist erwartbar, da bei größeren Strukturen mehr Inode-Lookups anfallen. Mit der Delta-Stabilisierung reduziert sich dieses Verhältnis auf 1,02. Beide Strukturgrößen sind nahezu gleich performant, weil der Cache bei stabilen Inode-Nummern unabhängig von der Strukturgröße effektiv wirkt.

Bei leseintensiven Workloads übertrifft die Delta-Konfiguration den `ext4_Passthrough` als Referenz deutlich: `open_close_5000` liegt um +322,8% über dem `Passthrough`-Wert, `fileserv_read_500` um +100,4% und `fileserv_read_5000` um +142%. Hier zeigt sich der systematische Vorteil des FLOUDS-Ansatzes, der Inodes nicht erst laden muss.

Bei strukturveränderenden Workloads hingegen ist kein signifikanter Effekt messbar (`append_small_1000`: +2,9%, `dirops_flat_5000`: +3,6%, `remove_dirs_flat_20000`: +1,6%). Dies ist konsistent mit der Erwartung, denn Schreiboperationen invalidieren zwangsläufig Teile des Inode-Caches, sodass der Caching-Vorteil nicht mehr zum Tragen kommt.

6.3.7 Optimierte Gesamtkonfiguration

Die optimierte Konfiguration kombiniert die drei Varianten mit den größten Einzeleffekten: Hash-Namenssequenz, Delta-Stabilisierung und die hierarchische Inode-Verwaltung. Es lässt sich beobachten, dass diese Effekte sich dabei nicht nur summieren, sondern sich gegenseitig sogar verstärken. Beim Workload `fileserv_read_500` wäre beispielsweise ein durchschnittliches Ergebnis von 35.424 Operationen pro Sekunde zu erwarten. Tatsächlich erreicht die Kombination 61.336 Operationen pro Sekunde. Durch die Gesamtintegration kommt jeder Effekt für sich selber besser zum Tragen. Der Speicherbedarf bleibt gegenüber der Standardkonfiguration nahezu unverändert. Im Vergleich

zum ext4-Passthrough schneidet die optimierte Konfiguration bei einigen lesenden Workloads deutlich besser ab:

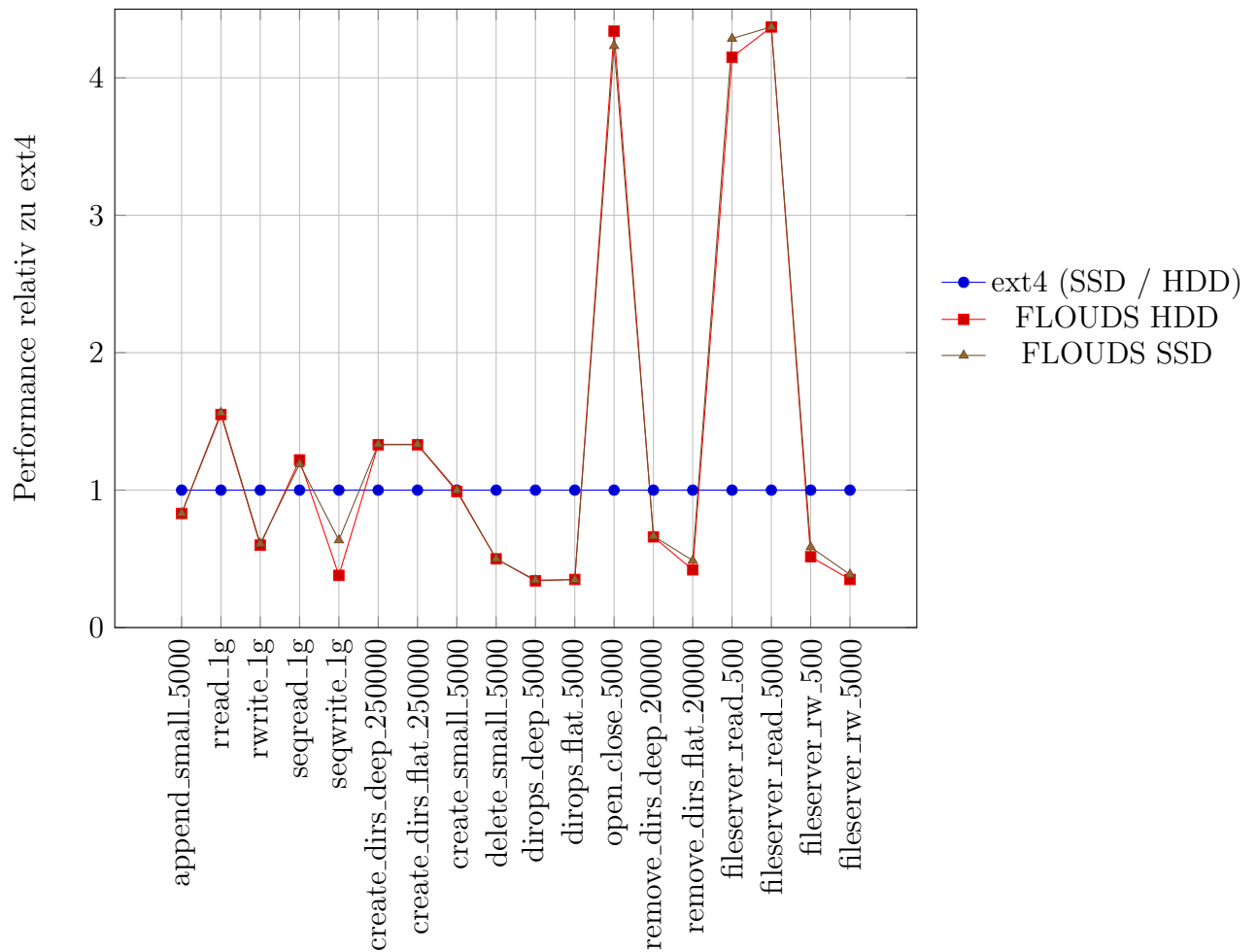


Abbildung 6.2: Relative Performance der optimierten Gesamtkonfiguration zum ext4-Passthrough.

7 Fazit

Ziel dieser Arbeit war es, die FLOUDS-Struktur um Änderungsoperationen zu erweitern und auf dieser Basis ein funktionsfähiges Dateisystem zu implementieren. Dieses sollte anschließend mit herkömmlichen Ansätzen verglichen werden, um konkrete Anwendungsfelder und Optimierungspotenziale zu ermitteln.

Die theoretische Auseinandersetzung mit succinthen Datenstrukturen und der FLOUDS-Struktur hat gezeigt, dass die Dynamisierung von FLOUDS grundsätzlich möglich ist, jedoch mehrere nicht-triviale Herausforderungen mit sich bringt. Insbesondere das Problem instabiler FLOUDS-Nummern erfordert einen Stabilisierungsmechanismus und das Verschieben von Teilbäumen ist eine zentrale Hürde.

Das Benchmarking hat gezeigt, dass FLOUDS bei lesedominanten Workloads mit ext4 mithalten und dieses in einigen Szenarien deutlich übertreffen kann. Die Delta-Stabilisierung erlaubt dabei effektives Caching. Insbesondere konnte aber auch gezeigt werden, dass FLOUDS als integriertes Dateisystem durch fehlende Fixkosten, aber auch durch kompakte Speicherung, verfügbaren Speicherplatz effizienter nutzt als herkömmliche Dateisysteme. Die Schreibperformance stellt hingegen die zentrale Schwäche des implementierten Systems dar.

7.1 Einsatzbereich

Mit dieser Arbeit konnte gezeigt werden, dass succinthe Datenstrukturen im Dateisystem-Kontext einen messbaren Vorteil liefern können. Der zentrale Mehrwert von FLOUDS liegt in der Kombination aus minimalem Speicherbedarf und trotzdem schnellen Lesezugriffen. FLOUDS bietet sich daher nicht nur für statische Dateistrukturen an, sondern auch für selten veränderliche Strukturen wie Konfigurationsdateien oder lesedominanten Datei- und Webservern. In Szenarien, in denen Delta-Stabilisierung und Inode-Caching besonders gut greifen, ist FLOUDS damit sinnvoll einsetzbar.

Aufgrund minimaler Fixkosten ist das Dateisystem außerdem besonders attraktiv für eingebettete Systeme. Beispielhaft genannt werden kann der ESP32-Mikrocontroller, der bislang auf das Dateisystem SPIFFS [2] angewiesen ist, das allerdings keine echte Ordnerhierarchie implementiert. Der Mikrocontroller bietet Schnittstellen, um Blöcke auf dem Flash-Speicher lesen und schreiben zu können. FLOUDS könnte hier eine echte hierarchische Struktur mit geringem Overhead liefern.

7.2 Limitationen

Das vorliegende System implementiert im Vergleich zu ext4 einen reduzierten Funktionsumfang. Fehlendes Journaling bedeutet, dass ein Absturz zwischen zwei Schreiboperationen zu einem inkonsistenten Dateisystemzustand führen kann. Für den identifizierten Hauptanwendungsfall ist diese Einschränkung weniger kritisch als in allgemeinen Desktopumgebungen, sollte jedoch vor einem produktiven Einsatz behoben werden. Ebenso fehlen Hard Links, Sym-links sowie die Rename-Operation, was den Vergleich mit ext4 zugunsten von FLOUDS leicht verzerrt.

Es ist darüber hinaus wichtig zu betonen, dass der FUSE/VFS-Rahmen strukturell auf Inode-basierte Dateisysteme ausgelegt ist. Ein nativer Einsatz auf einem Mikrocontroller ohne VFS könnte den FLOUDS-Ansatz weiter stärken, da das Inode-Caching-Problem sich dort gar nicht erst stellt und der FUSE-Overhead entfällt.

Schließlich ist anzumerken, dass die Benchmarking-Ergebnisse durch das Page-Caching des Betriebssystems geprägt sind: Die meisten Workloads sind memory-bound, sodass Unterschiede zwischen SSD und HDD kaum messbar sind. Das bedeutet einerseits, dass die Ergebnisse die Eigenschaften des Dateisystems isoliert widerspiegeln. Andererseits heißt das aber auch, dass das Verhalten bei ausgelastetem Speicher nicht vollständig abgebildet wird.

7.3 Zukünftige Arbeiten

Die zentrale nächste Herausforderung ist die vollständige Portierung des Dateisystems auf einen Mikrocontroller oder direkt in einen Betriebssystemkern. Erst dann lässt sich der eigentliche Zielanwendungsfall vollständig und ohne systematische Verzerrungen durch FUSE und VFS evaluieren.

Darüber hinaus sollte die Delta-Stabilisierung konsequent auf weitere relevante Komponenten wie die Namenssequenz und die Inode-Verwaltung ausgeweitet werden. Grundsätzlich bietet es sich an, succincte Datenstrukturen tiefer in das Dateisystem zu integrieren. Der Einsatz von Tries oder succincten Hash-Strukturen kann etwa auch bei der Extent- und Inode-Verwaltung sinnvoll sein, um den succincten Ansatz konsequenter anzuwenden. Schließlich wäre die Implementierung von Journaling oder alternativer Allokationsstrategien ein wichtiger Schritt in Richtung eines produktionsreifen Dateisystems.

Literaturverzeichnis

- [1] F. Claude, Gonzalo Navarro und A. Ordóñez. “The wavelet matrix: An efficient wavelet tree for large alphabets”. In: *Information Systems* 47 (2015), S. 15–32. DOI: 10.1016/j.is.2014.06.002. URL: <https://dl.acm.org/doi/10.1016/j.is.2014.06.002>.
- [2] Espressif Systems (Shanghai) Co. *SPIFFS Filesystem*. <https://docs.espressif.com/projects/esp-idf/en/stable/esp32/api-reference/storage/spiffs.html>. zuletzt geprüft am 29.03.2026. 2026.
- [3] O’Neil Delpratt, Naila Rahman und Rajeev Raman. “Engineering the LOUDS Succinct Tree Representation”. In: *Experimental Algorithms*. Hrsg. von Carme Àlvarez und María Serna. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, S. 134–145. ISBN: 978-3-540-34598-5.
- [4] Patrick Dinklage u. a. “Bit-Parallel (Compressed) Wavelet Tree Construction”. In: *Proceedings of the Data Compression Conference (DCC)*. IEEE, 2023, S. 81–90. DOI: 10.1109/DCC55655.2023.00016.
- [5] Saska Dönges, Simon J. Puglisi und Rajeev Raman. “On Dynamic Bit-vector Implementations”. In: *2022 Data Compression Conference (DCC)*. 2022, S. 252–261. DOI: 10.1109/DCC52660.2022.00033.
- [6] *ext4 Data Structures and Algorithms*. <https://docs.kernel.org/filesystems/ext4/>. The Linux Kernel documentation; zuletzt geprüft am 29.03.2026. The Linux Kernel Organization, Inc., 2025.
- [7] Kevin D. Fairbanks. “An analysis of Ext4 for digital forensics”. In: *Digital Investigation* 9 (2012). The Proceedings of the Twelfth Annual DFRWS Conference, S118–S130. ISSN: 1742-2876. DOI: 10.1016/j.diin.2012.05.010. URL: <https://www.sciencedirect.com/science/article/pii/S1742287612000357>.
- [8] Philippe Ferrucci. *libfuse API documentation*. <https://libfuse.github.io/doxygen/index.html>. zuletzt geprüft am 29.03.2026. 2025.
- [9] *Filesystems in the Linux kernel*. <https://docs.kernel.org/filesystems/index.html>. The Linux Kernel documentation; zuletzt geprüft am 29.03.2026. The Linux Kernel Organization, Inc., 2025.
- [10] *FUSE*. <https://www.kernel.org/doc/html/next/filesystems/fuse.html>. The Linux Kernel documentation; zuletzt geprüft am 29.03.2026. The Linux Kernel Organization, Inc., 2025.

- [11] Dominic Giampaolo. *Practical File System Design with the Be File System*. 1. Ausgabe. 1999.
- [12] Christian Hummert. *Mobile Forensics – The File Format Handbook*. 1. Ausgabe. 2022.
- [13] Guy J. Jacobson. “Succinct static data structures”. In: 1988. URL: <https://gwern.net/doc/cs/algorithm/information/compression/1988-jacobson.pdf>.
- [14] Florian Kurpicz. “Engineering Compact Data Structures for Rank and Select Queries on Bit Vectors”. In: *SPIRE*. Bd. 13617. Lecture Notes in Computer Science. Springer, 2022, S. 257–272. DOI: 10.1007/978-3-031-20643-6_19.
- [15] D.P. Mehta und S. Sahni. *Handbook of Data Structures and Applications*. Chapman & Hall/CRC Computer and Information Science Series. Taylor & Francis, 2004. ISBN: 9781584884354. URL: <https://books.google.de/books?id=kMSomAEACAAJ>.
- [16] Gonzalo Navarro. *(Worst-Case) Optimal Adaptive Dynamic Bitvectors*. 2025. arXiv: 2405.15088 [cs.DS]. URL: <https://arxiv.org/abs/2405.15088>.
- [17] Gonzalo Navarro. *Compact Data Structures: A Practical Approach*. Cambridge University Press, 2016.
- [18] Gonzalo Navarro. “Wavelet trees for all”. In: *Journal of Discrete Algorithms* 25 (2014). 23rd Annual Symposium on Combinatorial Pattern Matching, S. 2–20. ISSN: 1570-8667. DOI: 10.1016/j.jda.2013.07.004. URL: <https://www.sciencedirect.com/science/article/pii/S1570866713000610>.
- [19] Prashant Pandey, Michael A. Bender und Rob Johnson. “A Fast x86 Implementation of Select”. In: *CoRR* abs/1706.00990 (2017). arXiv: 1706.00990. URL: <http://arxiv.org/abs/1706.00990>.
- [20] Daniel Peters u. a. “FLOUDS: A Succinct File System Structure”. In: (Sep. 2017), S. 51–57. DOI: 10.15439/2017F535.
- [21] Andreas Poyias und Rajeev Raman. “Improved Practical Compact Dynamic Tries”. In: *String Processing and Information Retrieval*. Hrsg. von Costas Iliopoulos, Simon Puglisi und Emine Yilmaz. Cham: Springer International Publishing, 2015, S. 324–336. ISBN: 978-3-319-23826-5.
- [22] Nicola Prezza. “A Framework of Dynamic Data Structures for String Processing”. In: *International Symposium on Experimental Algorithms*. Leibniz International Proceedings in Informatics (LIPIcs). 2017.
- [23] Juan Pedro Bolivar Puente. *immer documentation*. <https://sinusoid.es/immer/>. zuletzt geprüft am 29.03.2026. 2026.

- [24] Rajeev Raman, Venkatesh Raman und S. Srinivasa Rao. “Succinct Dynamic Data Structures”. In: *Algorithms and Data Structures*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, S. 426–437. ISBN: 978-3-540-44634-7.
- [25] Sohail Shaikh. “Billion-files File Systems (BfFS): A Comparison”. In: *ArXiv* abs/2408.01805 (2024). URL: <https://arxiv.org/abs/2408.01805>.
- [26] *Squashfs 4.0 Filesystem*. <https://docs.kernel.org/6.2/filesystems/squashfs.html>. The Linux Kernel documentation; zuletzt geprüft am 29.03.2026. The Linux Kernel Organization, Inc., 2025.
- [27] Andrew S. Tannenbaum. *Modern Operating Systems*. 4. Ausgabe. 2014.
- [28] Vasily Tarasov, Erez Zadok und Spencer Shepler. “Filebench: A Flexible Framework for File System Benchmarking”. In: *login: The USENIX Magazine* 41.1 (Spring 2016), S. 6–14. URL: https://www.usenix.org/system/files/login/articles/login_spring16_02_tarasov.pdf.
- [29] Vasily Tarasov u. a. “Benchmarking File System Benchmarking: It *IS* Rocket Science”. In: *13th Workshop on Hot Topics in Operating Systems (HotOS XIII)*. Napa, CA: USENIX Association, Mai 2011. URL: <https://www.usenix.org/conference/hotosxiii/benchmarking-file-system-benchmarking-it-rocket-science>.
- [30] Kazuya Tsuruta u. a. “c-Trie++: A Dynamic Trie Tailored for Fast Prefix Searches”. In: *2020 Data Compression Conference (DCC)*. 2020, S. 243–252. DOI: 10.1109/DCC47342.2020.00032.
- [31] Bharath Kumar Reddy Vangoor, Vasily Tarasov und Erez Zadok. “To FUSE or Not to FUSE: Performance of User-Space File Systems”. In: *15th USENIX Conference on File and Storage Technologies (FAST 17)*. Santa Clara, CA: USENIX Association, Feb. 2017, S. 59–72. ISBN: 978-1-931971-36-2. URL: <https://www.usenix.org/conference/fast17/technical-sessions/presentation/vangoor>.
- [32] Sebastiano Vigna. “Broadword Implementation of Rank/Select Queries”. In: *Experimental Algorithms*. Hrsg. von Catherine C. McGeoch. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, S. 154–168. ISBN: 978-3-540-68552-4.

Abbildungsverzeichnis

2.1	Darstellung eines dynamischen Bitvektors als balancierter Suchbaum.	6
2.2	Restrukturierung im <i>Adaptive Dynamic Bitvector</i> nach Navarro.	9
2.3	Beispiel eines Wavelet Trees.	10
2.4	Beispiel eines Tries und seines komprimierten Patricia-Tries.	13
2.5	Beispiel einer LOUDS-Repräsentation.	15
3.1	Beispielhafte Verzeichnisstruktur.	17
3.2	Struktur eines ext4-Dateisystems.	19
3.3	Darstellung der Block-Pointer eines Inodes.	21
3.4	Zusammenspiel der Dateisystem-Komponenten unter Linux.	25
4.1	Beispielhafte Verzeichnisstruktur kodiert mit FLOUDS.	27
4.2	Einfügen eines neuen Knotens in FLOUDS.	33
5.1	Abstrakte Schnittstelle BitVector	39
5.2	Fragmentierte Allokation logischer Speicherbereiche.	42
5.3	Aufbau des FloudsHeader	43
5.4	Schichtenarchitektur unseres Dateisystems.	46
6.1	Speicherverbrauch der FLOUDS-Konfigurationen.	55
6.2	Performance der optimierten Gesamtkonfiguration.	59

Tabellenverzeichnis

4.1	Verschiedene Varianten zur Speicherung von Knotennamen. . . .	28
5.1	Metadaten innerhalb einer Inode-Struktur.	42
5.2	Implementierte FUSE-Operationen.	44
6.1	Übersicht der verwendeten Workloads.	52
6.2	Übersicht der getesteten FLOUDS-Konfigurationen.	53
1	Benchmarking-Ergebnisse: Baseline	67
2	Benchmarking-Ergebnisse: Lesen / Schreiben	68
3	Benchmarking-Ergebnisse: Struktur	69
4	Benchmarking-Ergebnisse: Makro	70
5	Benchmarking-Ergebnisse: Speicher	71

Anhang

Tabelle 1: Benchmarking-Ergebnisse: Baseline

Workload	HDD		SSD	
	ext4	Passthrough	ext4	Passthrough
<i>Lesen / Schreiben</i>				
append_small_1000	73.302	18.221	73.959	18.474
rread_1g	74.392	39.403	75.965	39.790
rwrite_1g	74.563	33.238	75.560	33.399
seqread_1g	11.765	10.413	12.968	10.953
seqwrite_1g	329	337	2.287	1.415
<i>Struktur</i>				
create_dirs_deep_20000	62.491	31.245	62.493	31.247
create_dirs_flat_20000	62.490	31.246	62.492	31.243
create_small_5000	14.998	7.499	14.996	7.498
delete_small_5000	2.500	2.500	2.500	2.500
dirops_deep_5000	42.925	5.670	43.182	5.742
dirops_flat_5000	37.552	6.199	37.876	6.253
open_close_5000	79.226	13.602	79.714	13.685
remove_dirs_deep_20000	19.994	4.999	19.994	4.999
remove_dirs_flat_20000	19.994	6.665	19.990	6.666
<i>Makro</i>				
fileserver_read_500	65.521	14.363	67.511	14.714
fileserver_read_5000	66.451	11.670	69.269	11.947
fileserver_rw_500	49.222	13.007	49.675	13.264
fileserver_rw_5000	47.174	10.272	46.728	10.544
<i>Speicher</i>				
dirs_250000	978,59 MiB			
flouids_src	2,54 MiB			
linux_src	1714,24 MiB			
openwrt_rootfs	14,67 MiB			

Tabelle 2: Benchmarking-Ergebnisse: Lesen / Schreiben

Konfiguration	append_small_1000		rread_lg		rwrite_lg		seqread_lg		seqwrite_lg	
	HDD	SSD	HDD	SSD	HDD	SSD	HDD	SSD	HDD	SSD
Standard	5.513	5.513	61.279	61.349	19.682	19.895	12.906	12.941	197	542
Delayed Allocation	9.832	9.617	61.253	61.465	20.222	20.255	12.825	12.653	209	542
Extent	3.841	3.921	56.742	56.310	16.576	16.729	12.790	12.665	310	416
Adaptive BV	3.598	3.563	61.259	61.624	92	93	13.063	12.307	91	92
Optimized BV	3.834	4.044	61.198	60.564	19.877	19.833	12.158	11.119	200	542
Concatenated NS	9.851	9.858	61.134	61.334	19.583	20.199	12.879	11.159	196	545
Hash NS	9.156	9.211	61.258	61.150	19.838	20.148	12.027	11.942	194	545
Delta	5.780	5.675	60.935	61.647	19.821	20.151	13.126	13.340	213	542
Hierarchy	8.328	8.555	61.325	61.996	19.865	19.982	12.763	13.344	226	541
Optimiert	15.153	15.421	61.618	61.867	20.298	20.071	12.394	13.392	214	544

Tabelle 3: Benchmarking-Ergebnisse: Struktur

Konfiguration	create_dirs_deep_20000		create_dirs_flat_20000		create_small_5000		delete_small_5000		dirops_deep_5000		dirops_flat_5000		open_close_5000		remove_dirs_deep_20000		remove_dirs_flat_20000	
	HDD	SSD	HDD	SSD	HDD	SSD	HDD	SSD	HDD	SSD	HDD	SSD	HDD	SSD	HDD	SSD	HDD	SSD
Standard	41.662	41.662	41.663	41.663	2.500	2.500	834	834	1.370	1.347	1.396	1.386	17.532	17.419	488	476	500	500
Delayed Allocation	41.664	41.663	41.662	41.662	2.500	2.500	834	834	1.379	1.377	1.380	1.434	17.422	17.596	465	476	500	488
Extent	41.661	41.662	41.662	41.659	2.143	1.875	834	834	1.355	1.363	1.453	1.371	17.332	17.487	476	488	500	488
Adaptive BV	41.663	41.663	41.662	41.663	2.143	2.143	834	834	1.177	1.234	1.196	1.239	15.406	15.579	377	377	392	385
Optimized BV	41.662	41.662	41.663	41.660	1.875	1.875	834	834	971	1.048	1.040	1.034	11.319	11.191	338	339	345	351
Concatenated NS	41.663	41.663	41.660	41.664	3.000	3.000	1.250	1.250	1.887	2.009	1.125	1.123	4.622	4.627	714	714	571	556
Hash NS	41.658	41.662	41.662	41.664	4.999	4.999	1.250	1.250	1.987	1.996	2.413	2.433	17.786	17.697	909	952	800	769
Delta	41.664	41.662	41.661	41.659	2.500	2.500	1.250	1.250	1.353	1.358	1.367	1.336	57.626	57.748	488	500	488	492
Hierarchy	41.663	41.662	41.662	41.664	3.000	3.000	1.250	1.250	1.491	1.487	1.463	1.552	16.914	17.083	606	769	610	800
Optimiert	41.661	41.662	41.663	41.664	7.498	7.489	1.250	1.250	1.957	2.007	2.160	2.204	57.570	59.326	3.333	3.333	3.250	2.857

Tabelle 4: Benchmarking-Ergebnisse: Makro

Konfiguration	filesver_read_500		filesver_read_5000		filesver_rw_500		filesver_rw_5000	
	HDD	SSD	HDD	SSD	HDD	SSD	HDD	SSD
Standard	17.848	17.429	13.945	14.043	7.979	8.329	1.956	1.945
Delayed Allocation	17.854	17.631	13.884	13.996	9.903	9.824	2.145	2.115
Extent	17.839	17.624	13.987	13.956	7.481	7.248	1.611	1.468
Adaptive BV	17.405	17.409	12.877	12.940	3.385	3.335	311	327
Optimized BV	17.366	16.692	10.246	10.173	7.336	7.208	1.558	1.566
Concatenated NS	15.895	15.522	6.118	6.069	10.258	10.111	3.352	3.329
Hash NS	17.890	17.698	14.146	13.957	10.171	10.027	3.414	3.421
Delta	28.950	29.332	28.466	28.696	9.540	9.540	2.083	2.185
Hierarchy	23.852	23.933	17.419	17.619	6.332	6.361	2.158	2.309
Optimiert	61.555	61.116	51.009	52.262	7.612	6.837	3.966	3.701

Tabelle 5: Benchmarking-Ergebnisse: Speicher

Konfiguration	dirs_250000		flouids_src		linux_src		openwrt_rootfs	
	HDD	SSD	HDD	SSD	HDD	SSD	HDD	SSD
Standard	10,06 MiB		0,37 MiB		1682,02 MiB		11,94 MiB	
Delayed Allocation	10,06 MiB		0,37 MiB		1682,02 MiB		11,94 MiB	
Extent	10,06 MiB		0,37 MiB		1696,47 MiB		12,67 MiB	
Adaptive BV	10,06 MiB		0,37 MiB		1682,02 MiB		11,94 MiB	
Optimized BV	10,06 MiB		0,37 MiB		1682,02 MiB		11,94 MiB	
Concatenated NS	10,06 MiB		0,37 MiB		1682,02 MiB		11,94 MiB	
Hash NS	10,06 MiB		0,37 MiB		1682,02 MiB		11,94 MiB	
Delta	10,06 MiB		0,37 MiB		1682,02 MiB		11,94 MiB	
Hierarchy	10,48 MiB		0,38 MiB		1682,92 MiB		12,33 MiB	
Optimiert	10,48 MiB		0,38 MiB		1682,92 MiB		12,33 MiB	

Eidesstattliche Erklärung

Hiermit versichere ich, dass die vorliegende Arbeit über „*Entwicklung und Benchmarking eines succinnten Dateisystems mithilfe von FLOUDS*“ selbstständig von mir und ohne fremde Hilfe verfasst worden ist, dass keine anderen Quellen und Hilfsmittel, als die angegebenen benutzt worden sind und dass die Stellen der Arbeit, die anderen Werken – auch elektronischen Medien und KI-Tools – dem Wortlaut oder Sinn nach entnommen wurden, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht worden sind. Mir ist bekannt, dass es sich bei einem Plagiat um eine Täuschung handelt, die gemäß der Prüfungsordnung sanktioniert werden kann.

Ich erkläre hiermit, dass ich Kenntnis von einer zum Zweck der Plagiatskontrolle vorzunehmenden Speicherung der Arbeit in einer Datenbank sowie von ihrem Abgleich mit anderen Texten zwecks Auffindung von Übereinstimmungen habe.

Ich versichere, dass ich die vorliegende Arbeit oder Teile daraus nicht anderweitig als Prüfungsarbeit eingereicht habe.

Ich erkläre hiermit, dass ich im Rahmen der Erstellung dieser Arbeit generative KI-Werkzeuge eingesetzt habe. Die Nutzung beschränkte sich auf redaktionelle Zwecke sowie das Strukturieren eigener Gedanken. Sämtliche inhaltliche Aussagen, Konzepte, Implementierungen und Ergebnisse stammen von mir.

Sebastian Brunnert, Münster, 7. April 2026