Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

Projekt-Bericht

ZRAM - RAM-Kompression

vorgelegt von

Benjamin Warnke

Fakultät für Mathematik, Informatik und Naturwissenschaften
Fachbereich Informatik
Arbeitsbereich Wissenschaftliches Rechnen

Studiengang:	Master Informatik
Matrikelnummer:	6676867

Hamburg, 2018-04-16

Abstract

ZRAM ist ein Kernel Modul welches virtuelle Blockdevices bereitstellen kann. Diese Bockdevices befinden sich komplett im Arbeitsspeicher. Um nicht zu viel Speicher zu belegen, wird der Inhalt der Blockdevices von ZRAM komprimiert. Aktuell benutzt ZRAM nur die Kompressionsalgorithmen, welche von der Crypto-API bereitgestellt werden. ZRAM komprimiert jede Seite einzeln, und beschränkt dadurch die Kompressionsalgorithmen auf einen sehr kleinen Kontext. Keiner der Kompressions-algorithmen in der Crypto-API ist darauf darauf ausgelegt, mit wenig Kontext gute Kompressions-Resultate zu erzielen.

Diese Ausarbeitung stellt einen neuen Algorithmus (zBeWalgo) vor, der genau auf die Anforderungen von ZRAM zugeschnitten ist. Insbesondere besteht das Ziel darin, die Kompressionsrate zu maximieren und gleichzeitig so wenig Zeit wie möglich zu verlieren. Durch die erhöhte Kompressionsrate soll es ermöglicht werden, Programme mit sehr hohem Arbeitsspeicherbedarf selbst dann noch auszuführen, wenn physikalisch nicht genügend Arbeitsspeicher vorhanden ist.

Beim Komprimieren muss man sich immer entscheiden, ob es wichtiger ist, eine hohe Kompressionsrate zu haben oder eine hohe Geschwindigkeit zu erzielen. Viele Algorithmen liegen auch irgendwo dazwischen, sodass die Wahl des benutzten Algorithmus nicht immer trivial ist.

Contents

1 Einleitung

Viele Kompressionsalgorithmen wurden mit verschiedenen Zielen erfunden. Manche Algorithmen sollen besonders schnell sein, während andere besonders hohe Kompressionsraten erzielen sollen. Wieder andere Algorithmen sind auf die speziellen Bitmuster Ihrer Anwendung optimiert. Bisher gibt es noch keinen Algorithmus, der in allen Disziplinen und Anwendungsgebieten hervorragende Ergebnisse erzielt.

Bei ZRAM geht es darum, Arbeitsspeicher zu komprimieren. Ein Ziel ist, dass das Anwenderprogramm mehr Arbeitsspeicher belegen kann als physikalisch verfügbar ist. Ein anderer Anwendungsfall besteht darin, mit Hilfe eines virtuellen Speichergerätes die Dateisystemgeschwindigkeit zu erhöhen, indem häufig benutzte Dateien direkt im RAM gespeichert werden.

In der aktuellen Version ist ZRAM in der Lage, Daten in ein backing_device zu schreiben, wenn diese nicht komprimierbar sind. Dadurch soll die Anzahl der Seiten, welche komprimiert im Speicher gehalten werden können, maximiert werden. Wenn kein backing_device verwendet wird, dann muss ZRAM auch die unkomprimierten Seiten im Arbeitsspeicher halten. Der häufigste Grund zur Benutzung von ZRAM ist der Geschwindigkeitsvorteil gegenüber einer einfachen SWAP-Partition auf einem anderen Speichermedium. Ein anderer Anwendungsfall wäre die Reduktion von Schreibzugriffen auf Speichermedien, welche nach einer begrenzten Anzahl von Schreibzyklen zerstört werden.

Eine mögliche Klasse von backing_devices sind HDDs. Diese haben allerdings den Nachteil, dass sie - verglichen mit dem Arbeitsspeicher - extrem langsam sind. Bei der Verwendung von SSDs ist zu beachten, dass diese insbesondere für die persistente Speicherung entwickelt wurden. Bei einem Versuch mit meiner NVME-SSDs überhitzte diese innerhalb weniger Minuten bei der Benutzung als SWAP-Speicher. Daraufhin blockierten Temperatur-Sensoren jeden weiteren Lese- und Schreibzugriff, wodurch die Festplatte mindestens temporär unbrauchbar war.

2 zBeWalgo

zBeWalgo ist ein neuer Kompressionsalgorithmus im Linux Kernel mit dem Ziel, ZRAM zu verbessern. zBeWalgo selbst ist ein Container-Algorithmus. Er kann verschiedene Kompressions- und Transformationsalgorithmen hintereinander ausführen. Im folgenden (und auch im Code) wird die Hintereinanderausführung von Algorithmen als Kombination bezeichnet. Zusätzlich zur Hintereinanderausführung von Algorithmen kann zBeWalgo auch verschiedene Kombinationen ausprobieren und die jeweils Beste für einen Datensatz verwenden. Durch dieses Ausprobieren von verschiedenen Kombinationen erzielt zBeWalgo sehr gute Kompressionsraten auf verschiedensten Datensätzen, welche andernfalls verschiedene Kompressionsalgorithmen oder Wörterbücher erfordern würden.

Es wäre extrem langsam, jede mögliche Kombination auf jedem Datensatz auszuprobieren. Aus diesem Grund startet zBeWalgo jeweils mit der Kombination, welche den letzten Datensatz am Besten komprimieren konnte. Wenn die Kompressionsrate für diese Daten ähnlich zu der Kompressionsrate von dem letzten Datensatz ist, dann wird diese Kombination direkt akzeptiert, ohne irgendwelche weiteren Kombinationen auszuprobieren. Selbst wenn es keine Garantie gibt, dass aufeinanderfolgende Daten zusammengehören, wird die Kompressionsgeschwindigkeit signifikant verbessert, ohne allzu viel Kompressionsrate zu verlieren.

Wenn eine Kombination nicht in der Lage ist, die Daten zu komprimieren, dann testet zBeWalgo so lange die nächste mögliche Kombination bis entweder eine passende Kombination gefunden wurde oder keine weitere Kombination mehr zur Verfügung steht. Wenn keine Kombination in der Lage ist, die Daten zu komprimieren, dann liefert zBeWalgo anstelle der Größe der komprimierten Daten einen negativen Wert, um auszudrücken, dass die Daten nicht komprimierbar sind.

ZRAM benutzt für die Allokation von Speicher die Funktion zsmalloc. Als Blockdevice bearbeitet ZRAM die Daten stets in Seiten zu je 4096 Bytes. Die größte *size-class* von *zsmalloc*, welche kleiner ist als eine ganze Seite, hat eine Kapazität von 3264 Bytes. Wenn die komprimierten Daten größer sind, dann verwirft ZRAM die komprimierten Daten und schreibt einfach die originalen Daten. Sobald klar wird, dass ZRAM das Ergebnis sowieso verwirft, bricht zBeWalgo die Komprimierung ab. Um zusätzliche Flexibilität zu gewährleisten, bietet zBeWalgo mithilfe von *sysfs* an, diesen Schwellwert zu konfigurieren. Dieser Schwellwert kann jeder Zeit geändert werden, sogar wenn gerade ein anderer Prozessorkern noch am Komprimieren ist.

Jede Kombination besteht aus bis zu 7 Kompressions- und Transformationsschritten. Transformationsschritte sollen die Daten so aufbereiten, dass nachfolgend eine effektivere Kompression stattfinden kann. Neue Kombinationen können jederzeit mit Hilfe der *sysfs*-Schnittstelle hinzugefügt und entfernt werden. Bereits komprimierte Daten können auch dann noch dekomprimiert werden, wenn die Kombination, die die Daten produziert hat,

gelöscht wurde. zBeWalgo kann theoretisch bis zu 256 benutzerdefinierte Kombinationen speichern. zBeWalgo würde alle diese Kombinationen ausprobieren, bevor ausgegeben wird, dass die Daten nicht komprimierbar sind. Dies wäre allerdings extrem zeitaufwändig und ist daher nicht zu empfehlen.

Um in der Lage zu sein, verschiedene Kompressions- und Transformations-Algorithmen aufzurufen, definiert zBeWalgo ein Interface, welches von allen in Frage kommenden Algorithmen implementiert werden muss. Nur dadurch ist zBeWalgo in der Lage, zur Laufzeit neue Kombinationen von dem Benutzer zu akzeptieren.

Innerhalb der Kombinationen können viele verschiedene Algorithmen verwendet werden. Einige dieser Algorithmen sind weit verbreitet, andere sind Modifikationen, um insbesondere mit wenig Kontext gute Ergebnisse zu erzielen.

2.1 Implementation

Während der Implementierung von zBeWalgo wurde darauf geachtet, so früh wie möglich ein komprimiertes Ergebnis zu liefern, ohne dadurch die Kompressionsrate allzu sehr zu verschlechtern. Figure 2.1 zeigt den Ablauf der Komprimierung mit zBeWalgo. Zu Beginn der Komprimierung (1) wird überprüft, dass die Eingabe nicht zu groß ist. Bei zu großen Eingaben würden einerseits Datentypen überlaufen und andererseits würde der vorher statisch allokierte Arbeitsspeicher nicht ausreichen. Wenn die Eingabe klein genug ist, dann iteriert zBeWalgo über alle verfügbaren Kombinationen (3). Beim Iterieren über die Kombinationen ist die Reihenfolge egal. Dies wird ausgenutzt, um mit den bisher erfolgversprechendsten Kombinationen anzufangen, wodurch die Kompressionsgeschwindigkeit häufig besser wird. Wenn keine Kombination eine besonders gute Kompressionsrate liefert, dann gibt zBeWalgo die am besten komprimierten Daten zurück (4). Wenn keine Kombination in der Lage ist, überhaupt zu komprimieren, dann liefert zBeWalgo einen negativen Rückgabe-Wert (4). Da jede Kombination mindestens einen Komprimierungsschritt beinhaltet, führt zBeWalgo anschließend den ersten Schritt der aktuellen Kombination aus (5). Anschließend iteriert zBeWalgo über die übrigen Schritte und führt die jeweilige Transformation oder Komprimierung durch (8, 9). Bei den Kompressionsschritten ist die Reihenfolge wichtig, da beim Dekomprimieren später genau die umgekehrte Folge genutzt werden muss, um die Daten korrekt wiederherzustellen. Jedes Mal, nachdem zBeWalgo einen Schritt durchgeführt hat, wird überprüft, ob die komprimierten Daten schon kleiner sind als der vorher definierte Schwellwert (6, 10). Wenn dies der Fall ist, dann kann zBeWalgo vorzeitig terminieren und spart somit Zeit.

2.2 Linux Mailinglisten

Nachdem der Algorithmus funktionsfähig war, wurde zBeWalgo als Patch zunächst an die linux-crypto Mailingliste gesendet. Einer der Entwickler dieser Liste half daraufhin, den Patch in eine korrekte Form zu bringen.

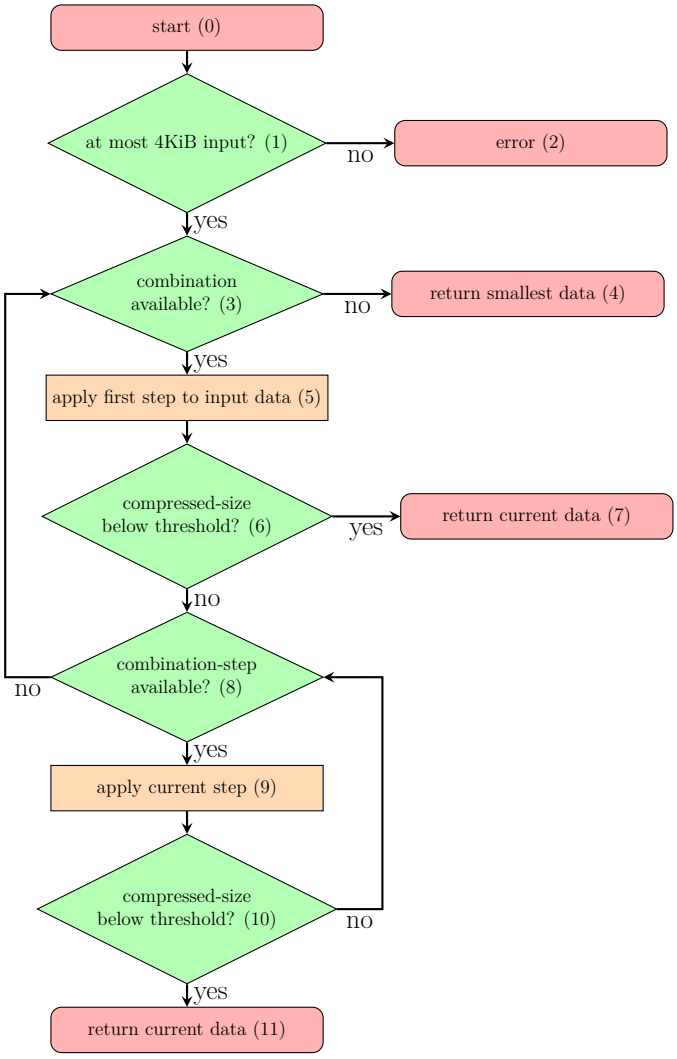

Figure 2.1: zBeWalgo Kompression Flussdiagramm

Auf ein weiteres Einsenden des Patches an die linux-crypto Mailingliste kam keine Antwort.

Daraufhin wurde der Patch zusätzlich an die linux-kernel Mailingliste gesendet. Dort fiel den Entwicklern auf, dass der Code beim Dekomprimieren nicht verhindert, dass ungültige Speicherbereiche beschrieben werden. Es wurde empfohlen, american fuzzy lop `http://lcamtuf.coredump.cx/afl/` zu verwenden, um die Fehlerfreiheit des Patches zu verbessern. AFL kann beliebigen Code und auch schon fertige Programme testen. Zum Testen benötigt AFL nur wenige gültige Eingabedaten. Diese Eingabedaten werden Stück für Stück von AFL manipuliert, indem zum Beispiel bits geflippt, Bytes inkrementiert und verschiedene Eingaben aneinander gehängt werden. Durch dieses Programm konnte die Testabdeckung deutlich über die Codeabdeckung hinaus gesteigert werden.

2.3 Enthaltene Algorithmen

Da ZRAM Daten stets in kleinen Blöcken von 4096 Bytes bearbeitet, wurden alle Algorithmen, die innerhalb von den Kombinationen benutzt werden können, entsprechend adaptiert. Um die Geschwindigkeit der (De-)Kompression zu erhöhen, wurde darauf geachtet, so wenig bedingte Sprünge wie möglich zu verwenden. Die implementierten Transformatoren und Kompressoren umfassen:

- BWT: Die Burrows-Wheeler-Transformation wurde von 'M. Burrows' und 'D. J. Wheeler' 1994 veröffentlicht. Die hier verwendete Implementation benutzt intern eine Variante von counting-sort, da hierdurch eine lineare Sortierzeit erzielt werden kann. Insbesondere ist es bei der Kompression nicht notwendig, die Daten fertig zu sortieren, da bereits ein Zwischenergebnis von Counting-sort ausreichend ist. Das originale Paper von 'M. Burrows' und 'D. J. Wheeler' ist verfügbar unter `http://www.hpl.hp.com/techreports/Compaq-DEC/SRC-RR-124.pdf`

- MTF: Der Move-To-Front-Algorithmus wurde ebenfalls von 'M. Burrows' und 'D. J. Wheeler' in genau derselben Veröffentlichung vorgestellt, wie ihre Transformation BWT.

- jbe: j-bit-encoding wurde von 'I Made Agus Dwi Suarjaya' 2012 vorgestellt. Sein Paper findet man online unter `https://arxiv.org/pdf/1209.1045.pdf`

- jbe2: Eine eigene Modifikation des jbe Algorithmus. Bevor jbe die Daten verarbeitet, werden Gruppen von jeweils 4 Bit umgestellt. Durch diese Umstellung kann die Kompressionsrate insbesondere dann verbessert werden, wenn bei vielen aufeinanderfolgenden Bytes die jeweils ersten 4 Bit 0 sind. Dies tritt insbesondere dann auf, wenn zuvor der Move-To-Front-Algorithmus angewendet wurde.

- rle: Eine detaillierte Beschreibung einer möglichen Run Length Encodierung wird im Internet unter `https://www.fileformat.info/mirror/egff/ch09_03.htm` beschrieben.

- huffman: Die Grundlage der verwendeten Huffman Kompression wurde in dem Paper 'Implementing the huffman Algorithm' von Gagarine Yaikhom `https://github.com/gyaikhom/huffman/blob/master/huffman.pdf` vorgestellt.

- BeWalgo: Diesen Algorithmus habe ich in meiner Bachelorarbeit `https://wr.informatik.uni-hamburg.de/_media/research:theses:benjamin_warnke_page_based_compression_in_the_linux_kernel.pdf` vorgestellt.

- BeWalgo2: Zu Beginn des Projekt-Moduls war dies der komplette Algorithmus. Dieser Algorithmus liest die Eingabe in Blöcken von 8 Bytes. Diese Blöcke werden anschließend in einen AVL-Baum einsortiert. Der AVL-Baum wird direkt in ein Array gespeichert, wodurch die einzelnen Knoten in dem Baum explizite Indices erhalten. Anschließend kann eine Variation von RLE verwendet werden, wobei anstelle der Daten nur die Indices in dem Baum abgespeichert werden. Durch

die Verwendung eines Baumes werden die komprimierten Daten noch kleiner im Vergleich zu nur RLE-Encoding.

3 Datensätze

Zum Vergleich verschiedener Algorithmen werden zunächst die verwendeten Datensätze aus verschiedenen Forschungsbereichen beschrieben.

3.1 Daten aus dem Forschungsbereich Geographie, Hydrologie und Klima

3.1.1 Isabella Hurrikane

Der Isabella Datensatz enthält Messergebnisse zu verschiedensten Variablen von dem Isabella Hurrikane. Jede Variable ist einer separaten Datei verfügbar, sodass diese unabhängig voneinander ausgewertet werden können. Zu jeder Variable ist zusätzlich ihr Wertebereich angegeben. Typ jeder Variablen ist float. Einige verschiedene Variablen haben sehr ähnliche Wertebereiche, sodass nur jeweils ein Datensatz als Stellvertreter gewählt wird, um eine möglichst große Varianz in den Datensätzen zu erzielen.

Eine Übersicht über alle Variablen ist unter `http://www.vets.ucar.edu/vg/isabeldata/readme.html` verfügbar.

Die hier verwendeten Variablen sind:

- P: Diese Variable hat einen sehr großen Wertebereich von -5471.85791 bis 3225.42578.
- TC: Diese Variable hat einen mittelgroßen Wertebereich von -83.00402 bis 31.51576.
- CLOUD: Diese Variable hat einen sehr kleinen Wertebereich von 0.00000 bis 0.00332.

Die Erwartung ist, dass kleinere Wertebereiche zu einer höheren Kompressionsrate führen, während die Variablen mit einer sehr großen Varianz schwieriger zu komprimieren sind.

`http://www.vets.ucar.edu/vg/isabeldata`

3.1.2 ECOHAM

Dieser Datensatz ist einer von mehreren für die Ozean-Simulationsanwendung ECOHAM. Wie in allen vorherigen Datensätzen auch werden hier Eigenschaften wie Wassertiefen und Chemikalienanteile in float-Arrays gespeichert. Messwerte an benachbarten Array Zellen sind ähnlich, da die Chemikalien sich im Wasser gut verteilen. Dies ermöglicht unabhängig vom Algorithmus extrem hohe Kompressionsraten.

3.1.3 Pflanzenwachstum

Dieser Datensatz enthält Aufzeichnungen zum Wachstum von Pflanzen über einen Verlauf von 12 Monaten. Alle Eigenschaften sind jeweils als 4-Byte große float-Werte abgespeichert. Die Daten sind unter `https://www.uni-frankfurt.de/45218031/data_download` verfügbar.

3.2 Daten aus dem Forschungsbereich Elektrotechnik

3.2.1 FGPA

Dieser Datensatz enthält Frequenzmessungen von vielen mit einem FGPA verbundenen Oszillatoren. Alle Frequenzmessungen sind im csv Format abgespeichert. Die originalen Daten sind verfügbar unter `http://rijndael.ece.vt.edu/puf/download.html`.

3.3 Daten aus dem Forschungsbereich Medizin

3.3.1 Lukas-Corpus

Der Lukas-Corpus enthält ein 16-bit zweidimensionales Bild von einem Radiographen. `http://www.data-compression.info/Corpora/LukasCorpus/index.html`

3.3.2 fMRI Daten

Dieser Datensatz beinhaltet Rohdaten aus Messungen von einem hochauflösenden 7-Tesla fMRI an 20 Probanden, die mit 25 Musik-Clips aus 5 Genres stimuliert wurden. `https://openfmri.org/dataset/ds000113b/`

3.3.3 Protein-Corpus

Dieser Datensatz enthält Daten über Proteinsequenzen verschiedener Bakterien, die sich aus der Genomanalyse ergaben. Diese Art von Datensatz ist schwer komprimierbar, da es kaum sich wiederholende Sequenzen gibt. `http://www.data-compression.info/Corpora/ProteinCorpus/index.html`

3.4 Daten aus dem Forschungsbereich Informatik

3.4.1 Linux Quelltext

Zum Testen wurden alle Quelltext-Dateien einer Version des Linux Kernels in einer tar-Datei zusammengefasst. Quelltext enthält nur kurze gleiche Textsequenzen wie zum Beispiel Variablennamen, da der Programmierer sonst Schleifen oder Funktionen verwenden würde. Daher ist die Suche nach für die Kompression verwertbaren Sequenzen

schwierig. Da es sich nur um Text-Daten handelt, ist der verwendete Zeichenvorrat begrenzt, wodurch Kompressions-Algorithmen die Möglichkeit haben, die verwendeten Zeichen effizienter zu kodieren.

3.5 Daten aus dem Forschungsbereich Astronomie

3.5.1 SAO Star Catalog

Dieser Datensatz enthält den gesamten Katalog des Smithsonian Astrophysical Observatories. Er enthält die Position, Größe und weitere Eigenschaften von 258.996 Sternen. `http://tdc-www.harvard.edu/software/catalogs/sao.html`

3.6 Daten aus anderen Bereichen

3.6.1 HPCG

Das HPCG Programm ist eigentlich ein Benchmark. Es ist dafür gedacht, Supercomputer zu bewerten und diese vergleichbar zu machen. Da nicht die Hardware, sondern Kompressionsalgorithmen bewertet werden sollen, wurde vom laufenden HPCG Programm ein Arbeitsspeicherabbild erstellt. In dem Moment, in dem das Speicherabbild erstellt wurde, führte HPCG gerade eine Multiplikation mit dünn besetzten Matrizen durch.

3.6.2 Partdiff

Partdiff löst partielle Differentialgleichungssysteme mit Hilfe einer Jakobi-Implementation. Im Arbeitsspeicher benutzt dieses Programm dafür ein großes Array von 8-Byte Fließkommazahlen. In diesem Programm beinhalten benachbarte Werte in dem Array ähnliche, aber dennoch verschiedene Werte.

4 Evaluation

Vor den Tests wurden zunächst doppelt vorhandene Seiten aus den Datensätzen entfernt, um die Vergleichbarkeit zu verbessern. Es ging darum, das bessere Abschneiden von Algorithmen zu verhindern, die nur wenige spezielle, häufig wiederkehrende Seiten gut komprimieren können. Dann wurden die Testdaten in ein user-space Programm geladen, um reproduzierbare Benchmarks zu erzielen.

Anschließend wurden die Daten direkt in ein ZRAM-device geschrieben, ohne vorher darauf ein Dateisystem zu initialisieren. Dies sollte zusätzlichen Overhead innerhalb des Dateisystems verhindern. Zwischen dem Lesen und Schreiben von Daten auf dem ZRAM-Device wurden jeweils 'sync' und 'echo 3 > /proc/sys/vm/drop_caches' ausgeführt. Alle Zeitmessungen sind 'wall-clock' Zeiten und verwenden jeweils nur 1 Thread zur Zeit. zBeWalgo wurde mit allen Algorithmen, die in der crypto-API vorhanden sind, verglichen.

Zusätzlich wurde eine effektive Geschwindigkeit berechnet und in den mit '⊘' markierten Spalten angegeben, um Algorithmen besser vergleichen zu können. Für diese Berechnung wurden einige Annahmen getroffen:

- Die Kapazität der ZRAM-Partition (/sys/block/zram0/disksize) ist groß genug, um die Test-Daten komplett aufzunehmen.
- Das Speicherlimit der ZRAM-Partition (/sys/block/zram0/mem_limit) wird so gewählt, dass nur der Algorithmus mit der höchsten Kompressionsrate gerade eben genügend Speicher hat.
- Alle unkomprimierbaren Seiten werden auf ein backing_device geschrieben.
- ZRAM bricht mit einem OutOfMemory Fehlercode ab, wenn das Speicherlimit voll ist, aber dennoch komprimierbare Seiten geschrieben werden. Für diese Berechnung wird jedoch angenommen, dass das Programm alle weiteren Daten, die ZRAM nicht speichern kann, auf das backing_device schreibt. Wenn ZRAM als SWAP verwendet wird, würde der Kernel ohnehin zum nächstbesten SWAP-device wechseln.
- Das backing_device ist groß genug, um die Test-Daten komplett aufzunehmen.

Für das backing_device wurde eine HDD angenommen, die Benchmarks hierfür werden in Table 4.1 dargestellt. Die effektive Geschwindigkeit berechnet sich nach den folgenden Formeln:

Definition der verwendeten Variablen:

- s_u : Summe aller Bytes auf unkomprimierbaren Seiten

- s_c : Größe der komprimierten Daten in Bytes
- s_o : Größe der originalen Daten in Bytes
- s_{cmin} : Anzahl Bytes, die der Algorithmus mit der höchsten Kompressionsrate benötigt, die dem Speicherlimit der ZRAM-Partition entsprechen
- r : Kompressionsrate
- t_c : Zeit für die Kompression

$$size_{backing\_device} = s_u + (s_c - s_{cmin}) * r$$
$$speed_{effective} = \frac{s_o}{t_c + \frac{size_{backing\_device}}{speed_{backing\_device}}}$$

Die folgenden Benchmarks zeigen, dass schnellere Algorithmen wie LZ4 häufig eine niedrigere Kompressionsrate erzielen als langsamere Algorithmen. In diesem Zusammenhang wird leider meistens übersehen, dass durch die niedrigere Kompressionsrate tendenziell häufiger auf das backing_device zurückgegriffen wird, welches extrem viel langsamer ist. Dies sollte bei der Betrachtung der Benchmarks berücksichtigt werden.

Einige der Datensätze enthalten zusätzlich Messungen mit dem Algorithmus zBeWalgo*. Bei diesem Algorithmus hat der Benutzer zur Laufzeit die verfügbaren Kombinationen so geändert, dass ausschließlich die für diese Datenart relevanten Kombinationen verfügbar sind.

4.1 /dev/zero

Dieser Datensatz wird verwendet, um die Geschwindigkeitsgrenzen von ZRAM zu messen. ZRAM filtert intern Seiten, welche komplett nur aus Nullen bestehen und ruft für diese Seiten keinen Kompressionsalgorithmus auf. Neben ZRAM wird dieser Datensatz ebenfalls benutzt, um die Lese/Schreibgeschwindigkeit von potentiellen backing_devices zu ermitteln. Table 4.1 zeigt alle Benchmarks mit diesem Datensatz. Wie zu erwarten, kann ZRAM sehr schnell erkennen, ob eine Seite komplett mit Nullen gefüllt ist, wodurch eine effiziente Speicherung möglich ist.

Algorithmus	Schreiben (MiB/s)	Lesen (MiB/s)
–ZRAM–	2571.47	2828.87
–HDD–	134.70	156.62

Table 4.1: Benchmarks mit dem /dev/zero Datensatz

4.2 Isabella Hurrikane CLOUD

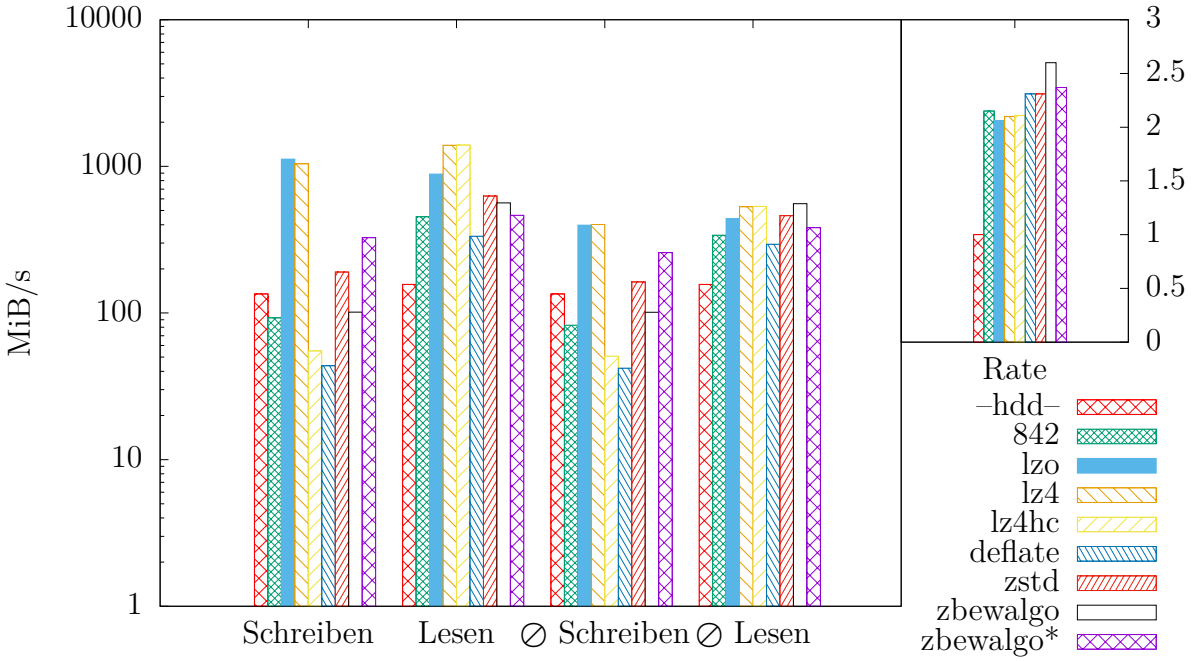

Figure 4.1: zBeWalgo Benchmarks mit dem Isabella CLOUD Datensatz

In der Figure 4.1 kann man sehen, dass alle Algorithmen sehr ähnliche Kompressionsraten erzielen. Wenn der Benutzer zBeWalgo unterstützt und die verfügbaren Kombinationen auf den Datensatz anpasst, kann die Kompressionsgeschwindigkeit verdreifacht werden, während die Kompressionsrate nur minimal schlechter wird. Beim Lesen und Schreiben ohne backing_dev erreichen insbesondere die LZ4 Varianten die höchsten Geschwindigkeiten. Sobald der verfügbare Speicher begrenzt ist und das backing_device verwendet werden muss, ist zBeWalgo zumindest im Dekomprimieren schneller als alle anderen Algorithmen, da weniger Zugriffe auf das langsame backing_device erfolgen.

4.3 Isabella Hurrikane PRESSURE

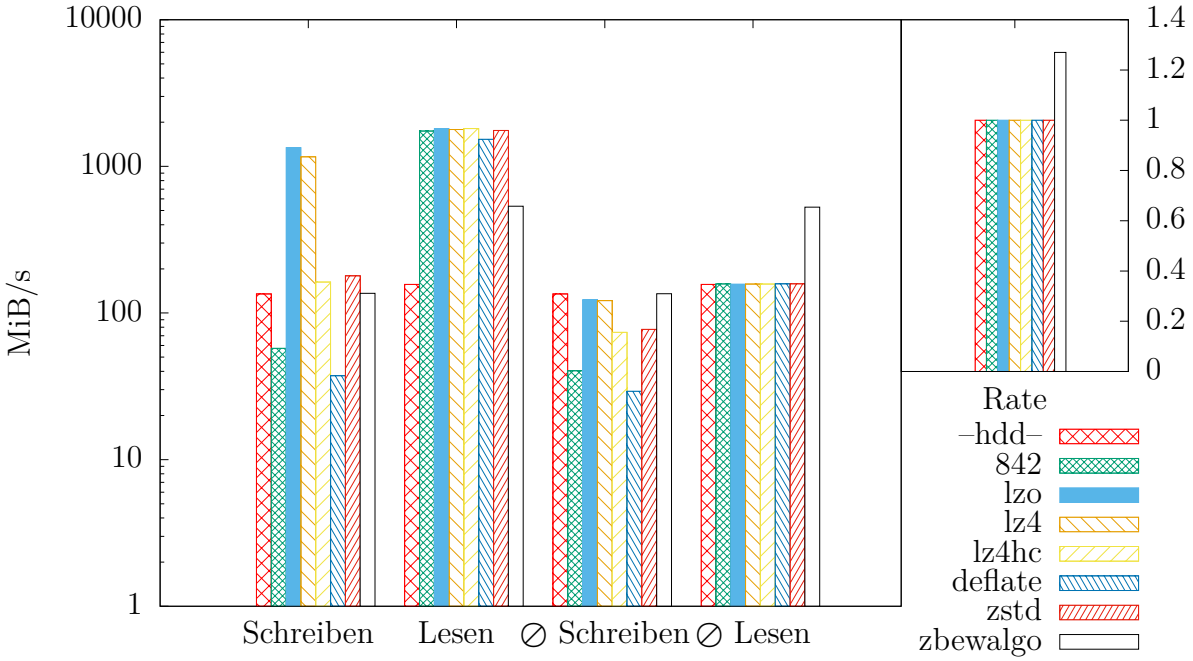

Figure 4.2: zBeWalgo Benchmarks mit dem Isabella PRESSURE Datensatz

Figure 4.2 zeigt, dass ausschließlich zBeWalgo in der Lage ist, die Daten zu komprimieren. Solange ausreichend Arbeitsspeicher verfügbar ist, benutzt ZRAM die unkomprimierten Seiten direkt aus dem Arbeitsspeicher, wodurch alle Algorithmen schneller erscheinen als zBeWalgo. Sobald der Arbeitsspeicher limitiert ist, müssen alle anderen Algorithmen einen Großteil ihrer Daten auf das backing_device schreiben. Im Endergebnis kann zBeWalgo eine deutlich höhere effektive Lesegeschwindigkeit liefern als alle anderen Algorithmen. Beim Schreiben ist zBeWalgo in diesem Benchmark genauso schnell wie eine HDD. Hierbei ist noch zu bemerken, dass die Geschwindigkeit einer HDD nicht mit der Anzahl der verfügbaren CPU-Kerne skaliert - zBeWalgo schon.

4.4 Isabella Hurrikane TEMPERATURE

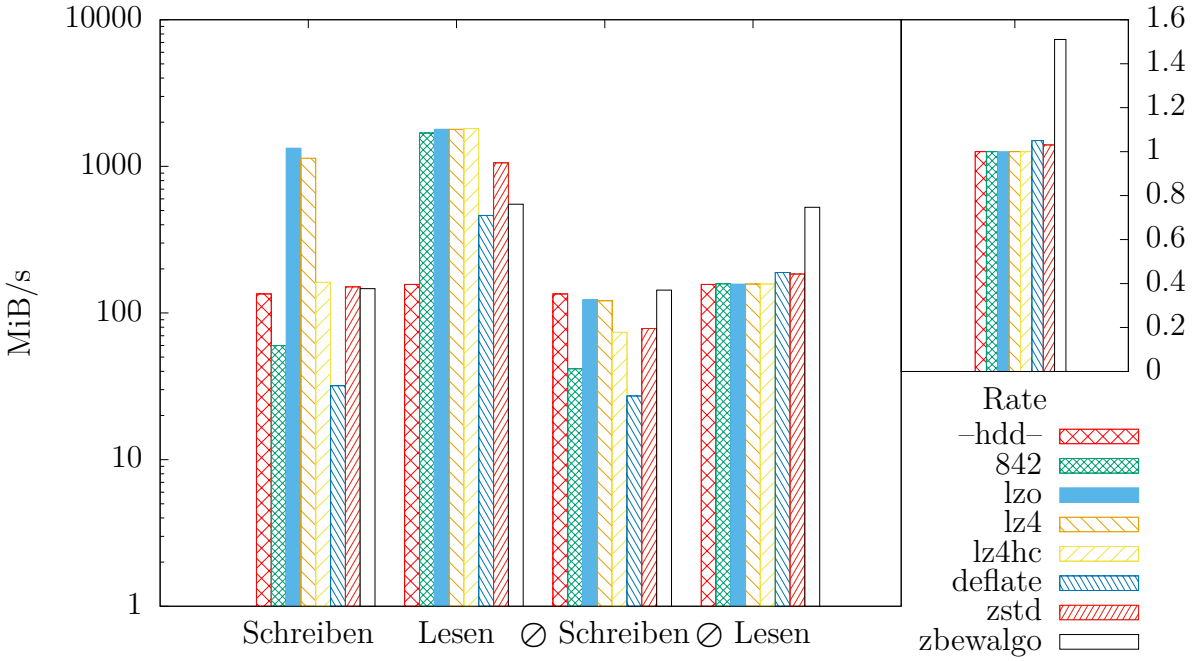

Figure 4.3: zBeWalgo Benchmarks mit dem Isabella TEMPERATURE Datensatz

Die Temperatur Daten in Figure 4.3 und die Druckmessungen der vorherigen Figure 4.2 zeigen, dass zBeWalgo besser als andere Algorithmen geeignet ist, wenn Arrays von ähnlichen float Werten komprimiert werden sollen. Aus diesen Ergebnissen kann man ableiten, dass zBeWalgo seine Kompressionsrate steigert, wenn die Wertebereiche kleiner werden. Erst wenn der Wertebereich sehr klein wird wie in dem Wolken Datensatz aus Figure 4.1, schaffen es auch andere Algorithmen, die Daten zu komprimieren.

4.5 Pflanzenwachstum

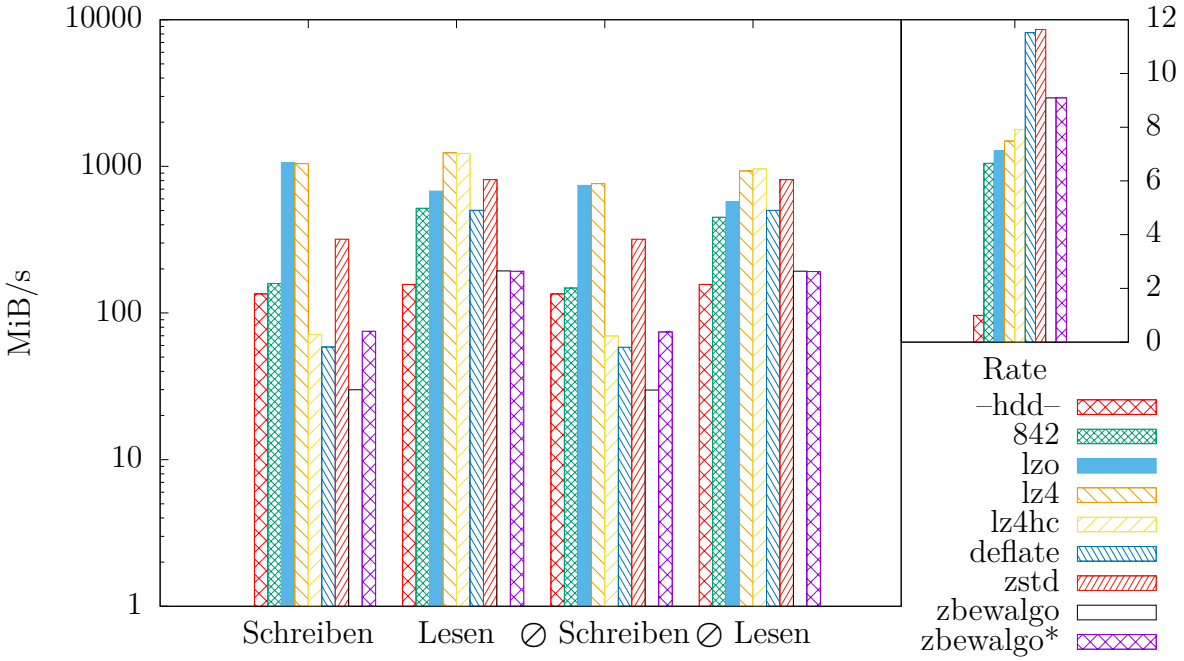

Figure 4.4: zBeWalgo Benchmarks mit dem Datensatz Pflanzenwachstum

Dieser Datensatz enthält viele Wiederholungen von 4-Byte-Blöcken in direkter Folge. In der aktuellen Implementation bearbeiten alle Kompressionsalgorithmen von zBeWalgo die Eingabedaten stets in Blöcken von 1 oder 8 Bytes. Dadurch kann zBeWalgo nicht ganz die Kompressionsrate von zstd und deflate erreichen. Obwohl zBeWalgo diesen Datensatz deutlich langsamer dekomprimiert als alle anderen Algorithmen, ist zBeWalgo beim Lesen trotzdem schneller als eine HDD.

4.6 ECOHAM

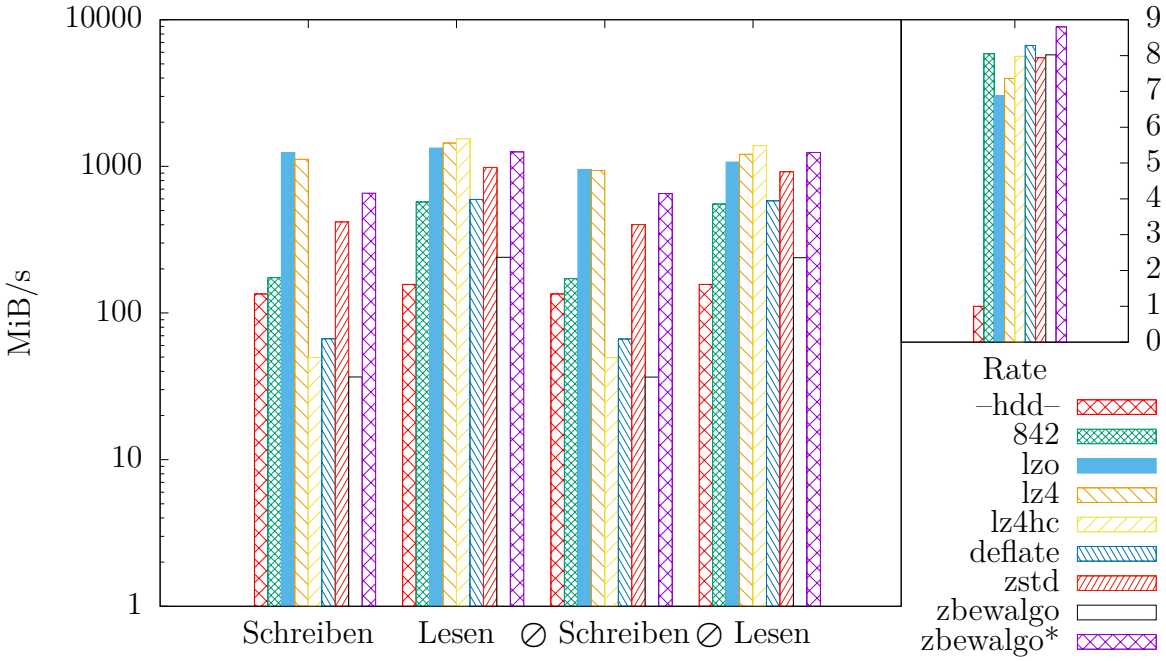

Figure 4.5: zBeWalgo Benchmarks mit dem ECOHAM Datensatz

Dieser Datensatz enthält sehr viele fast leere Seiten. Dies ermöglicht es jedem Algorithmus, der effizient lange Sequenzen von Nullen komprimieren kann, eine gute Performance zu zeigen. Wenn der Benutzer zBeWalgo unterstützt und nur die besten Kombinationen aktiviert, kann die Kompressions- und Dekompressionsgeschwindigkeit signifikant verbessert werden. Zudem wird auch noch die Kompressionsrate verbessert. Die Steigerung der Kompressionsrate durch eine Reduktion der verfügbaren Kombinationen kann dadurch erklärt werden, dass zBeWalgo abbricht, sobald eine gute Kombination gefunden wurde. Wenn mehrere ähnlich gute Kombinationen existieren, dann wählt zBeWalgo stets die erste, die gefunden wird, wodurch nicht immer die maximale Kompressionsrate gefunden wird.

4.7 FGPA

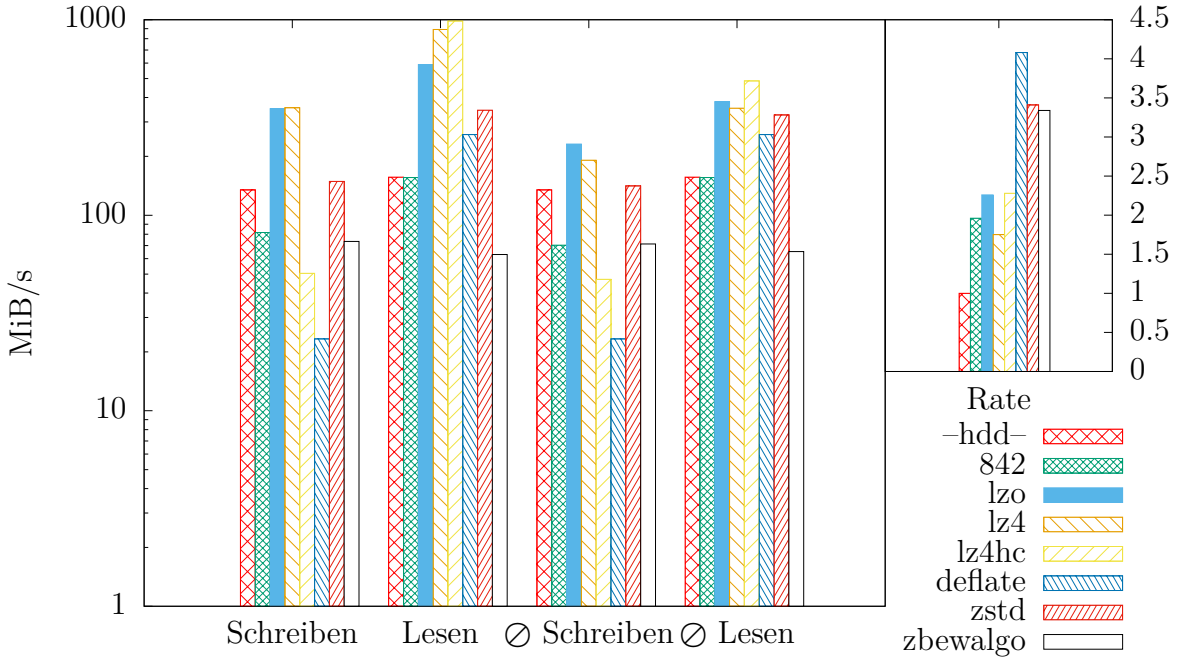

Figure 4.6: zBeWalgo Benchmarks mit dem FPGA Datensatz

Dieser als csv kodierte Datensatz enthält lange Ketten von kleinen und auch ähnlichen Fließkommazahlen. Fast alle der kodierten Werte haben jeweils 3 Stellen vor und nach dem Komma. Zusammen mit dem Dezimalpunkt sowie dem Trennzeichen Komma entsteht ein 8-Byte Muster. Dieses Muster kann dann unterschiedlich gut komprimiert werden. Da zBeWalgo unter anderem huffman-encoding verwendet, um diese Daten komprimieren zu können, ist die Geschwindigkeit sehr niedrig. Die Kompression von diesem Datensatz mit zBeWalgo lohnt sich somit nur, wenn zstd nicht verfügbar ist und das backing_device vermieden werden soll. Wenn eine sehr hohe Kompressionsrate wichtig ist, dann sollte dieser Datentyp mit deflate komprimiert werden.

4.8 Lukas-Corpus

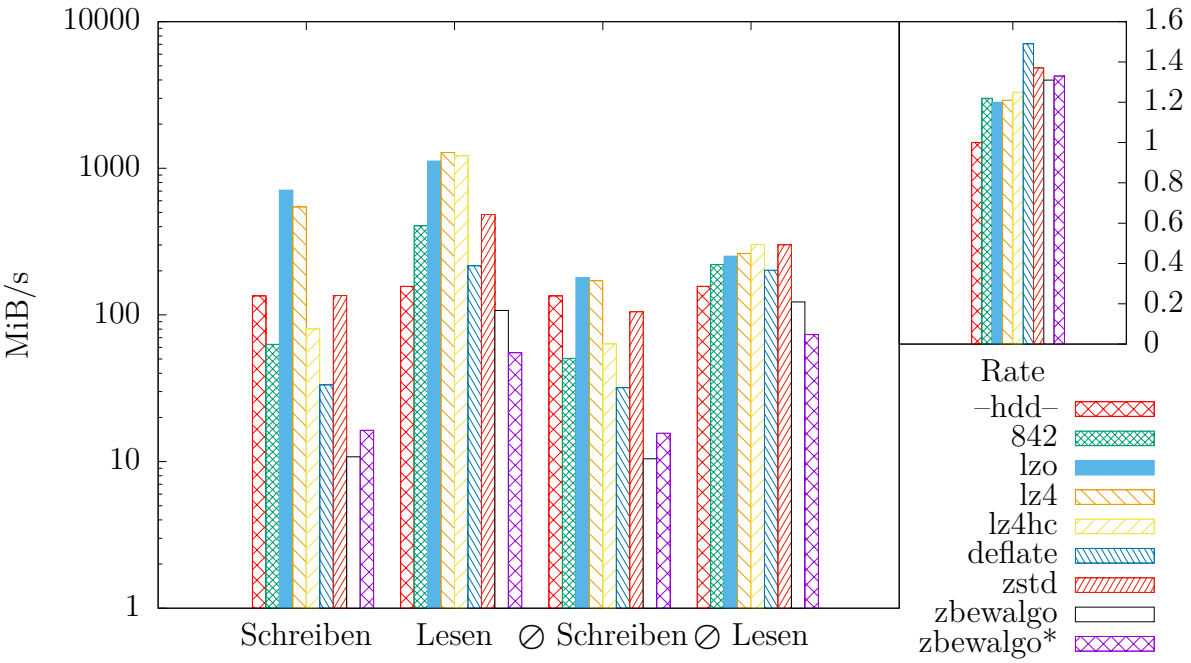

Figure 4.7: zBeWalgo Benchmarks mit dem Lukas-Corpus

Diesen Datensatz kann zBeWalgo nur bei sehr niedriger Geschwindigkeit etwas komprimieren. Die niedrige Geschwindigkeit lässt sich durch den hohen Anteil von nicht komprimierbaren Seiten erklären, bei denen zBeWalgo alle Kombinationen zunächst prüft und anschließend dennoch die unkomprimierten Seiten gespeichert werden.

4.9 fMRI Daten

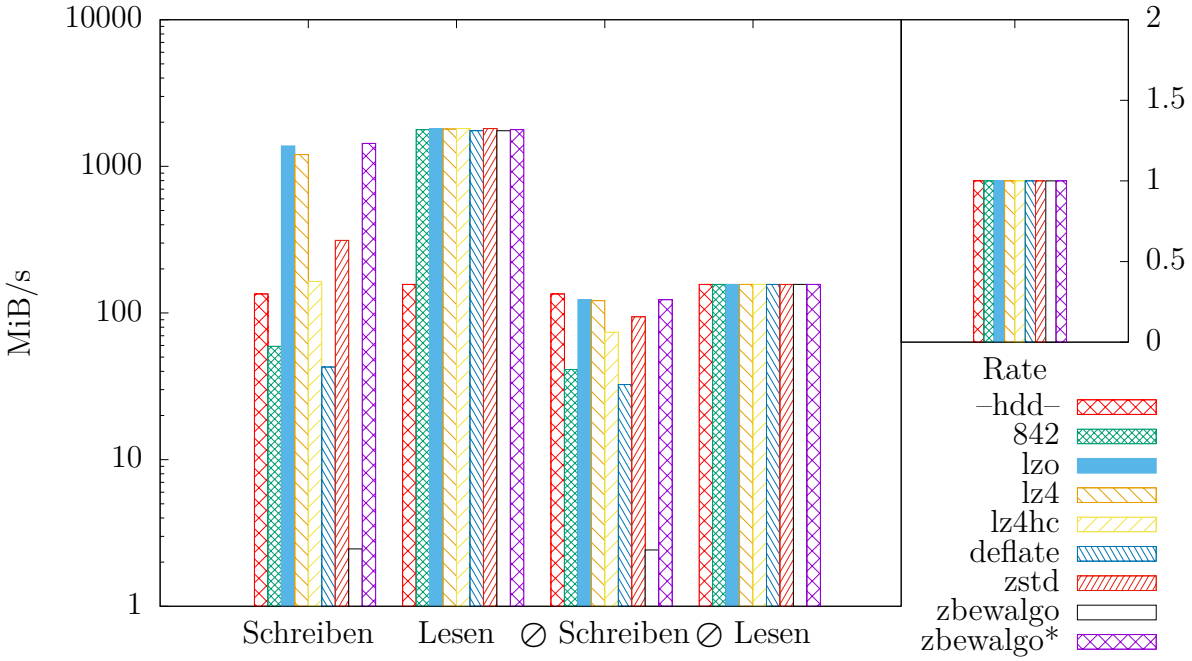

Figure 4.8: zBeWalgo Benchmarks mit dem fMRI Datensatz

Keiner der getesteten Algorithmen ist in der Lage gewesen, diesen Datensatz zu komprimieren. Dieser Datensatz stellt somit einen echten Anwendungsfall dar, bei dem Kompression zwar hilfreich wäre, aber bislang nicht möglich ist. Hier unterscheiden sich die Algorithmen nur darin, wie schnell sie den Datensatz als nicht komprimierbar erkennen.

4.10 Protein-Corpus

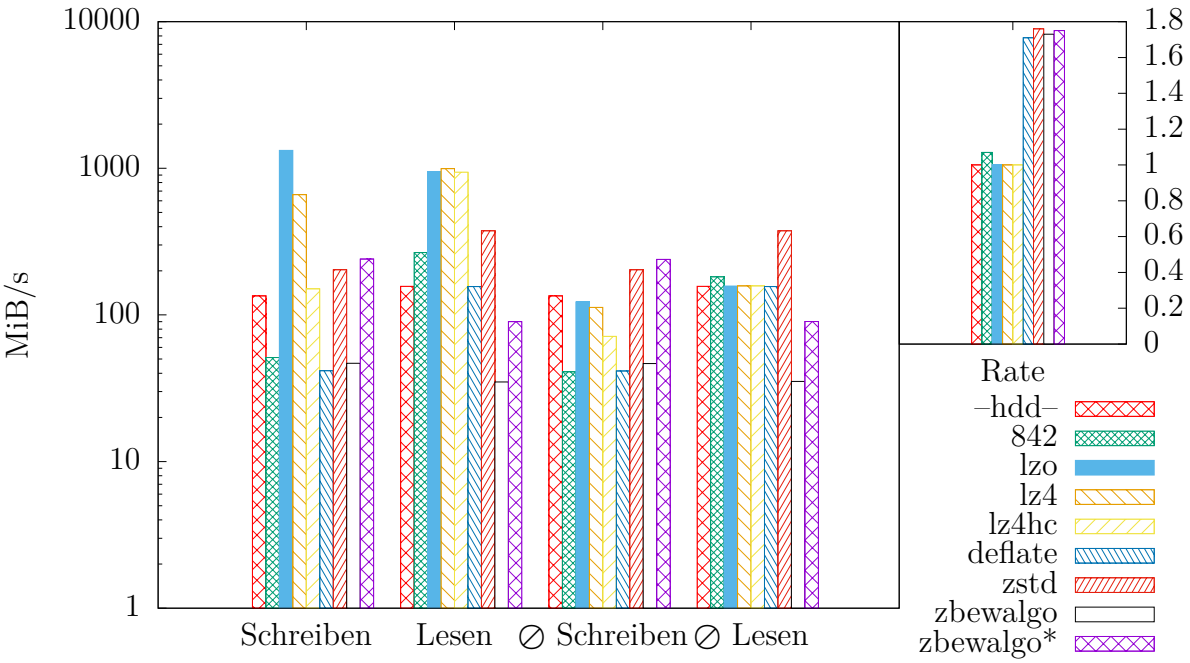

Figure 4.9: zBeWalgo Benchmarks mit dem Protein-Corpus

Schon in früheren wissenschaftlichen Arbeiten wurde dieser Datensatz verwendet, um einen Kompressionsalgorithmus speziell für diese Art von Daten zu entwerfen. Das originale Paper ist ist auf der Seite `http://www.data-compression.info/Corpora/ProteinCorpus/index.html` referenziert, aber aktuell nicht verfügbar. Bei meinen Tests waren zstd, deflate und zBeWalgo in der Lage, diese Daten zumindest etwas zu reduzieren. Mit einer sinnvollen Vorauswahl von Kombinationen, kann zBeWalgo auf diesem Datensatz die höchste Kompressionsgeschwindigkeit - im Vergleich mit den Algorithmen, die tatsächlich eine Kompression erzielen, - erwirken.

4.11 Linux

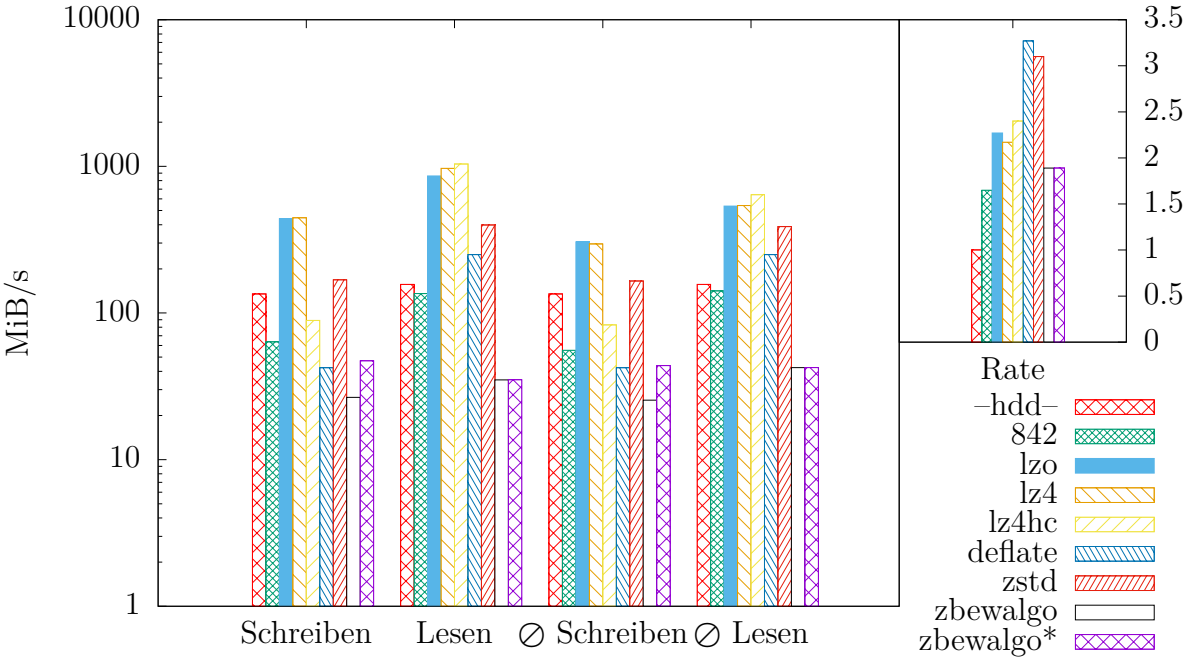

Figure 4.10: zBeWalgo Benchmarks mit dem Linux Datensatz

Dieser textbasierte Datensatz ist bisher nicht mit zBeWalgo komprimierbar, da noch kein Teilalgorithmus implementiert wurde, der diese Datenart gut komprimieren kann. Für die Kompression von Quelltexten sind bisher die meisten der anderen Algorithmen im Vorteil.

4.12 SAO Star Catalog

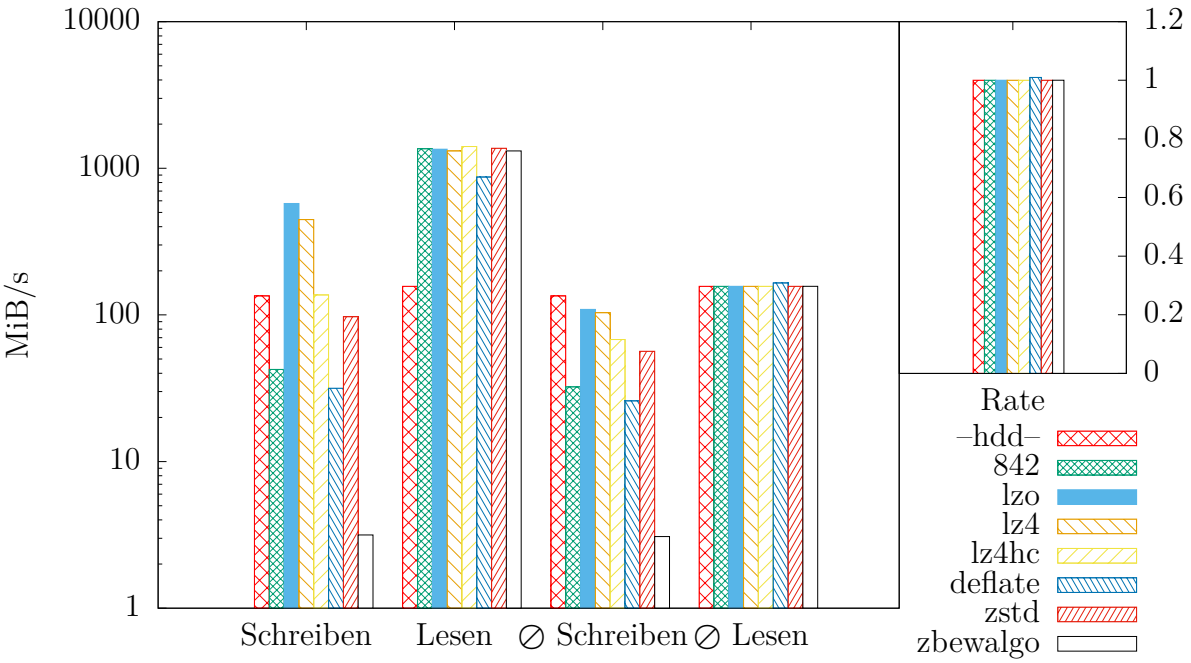

Figure 4.11: zBeWalgo Benchmarks mit dem SAO Star Katalog

Dieser Datensatz kann nur von deflate minimal komprimiert werden. Alle anderen Algorithmen unterscheiden sich hier nur darin, wie schnell sie erkennen, dass die Daten nicht komprimierbar sind. Bei diesem Datensatz braucht zBeWalgo extrem lange, bis der Datensatz als nicht komprimierbar abgebrochen wird.

4.13 HPCG

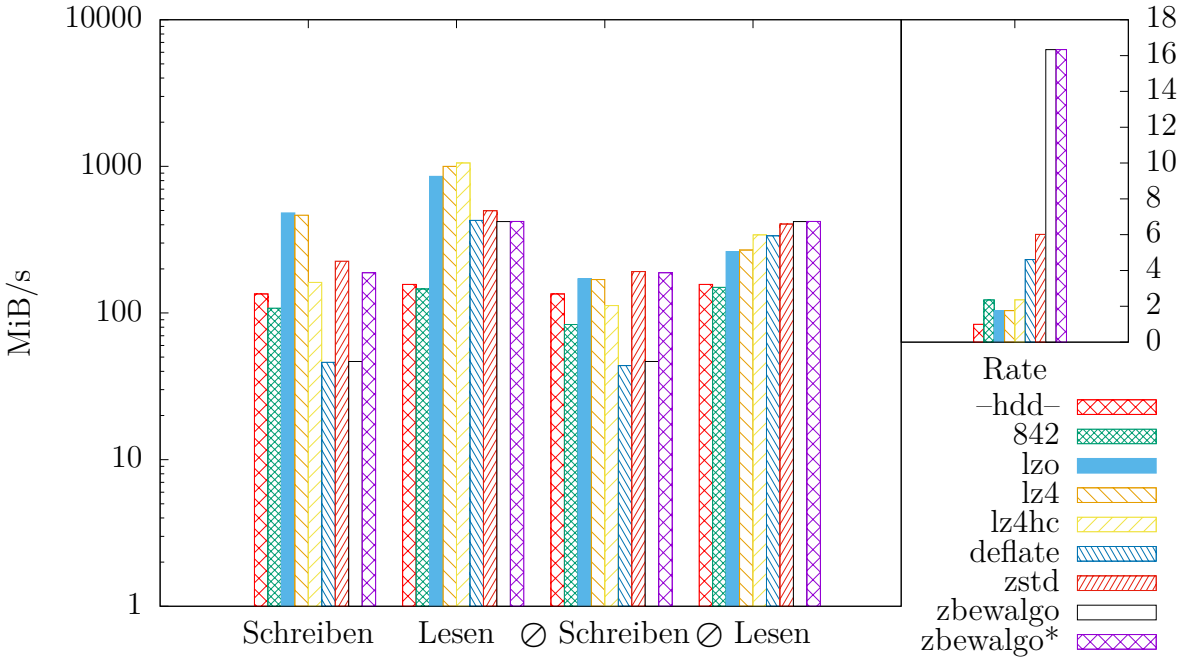

Figure 4.12: zBeWalgo Benchmarks mit dem HPCG Datensatz

Bei diesem Datensatz erreichte zBeWalgo mit großem Abstand die beste Kompressionsrate. Die Algorithmen der LZ Familie konnten keine so hohe Kompressionsraten erzielen. Solange der Arbeitsspeicher über genügend Kapazität verfügte, wurden die Seiten durch LZ Algorithmen mit insgesamt höherer Geschwindigkeit komprimiert als durch die übrigen Algorithmen. Bei sehr begrenztem Arbeitsspeicher ist zBeWalgo zusätzlich der Algorithmus, welcher am schnellsten komprimiert und dekomprimiert.

4.14 partdiff

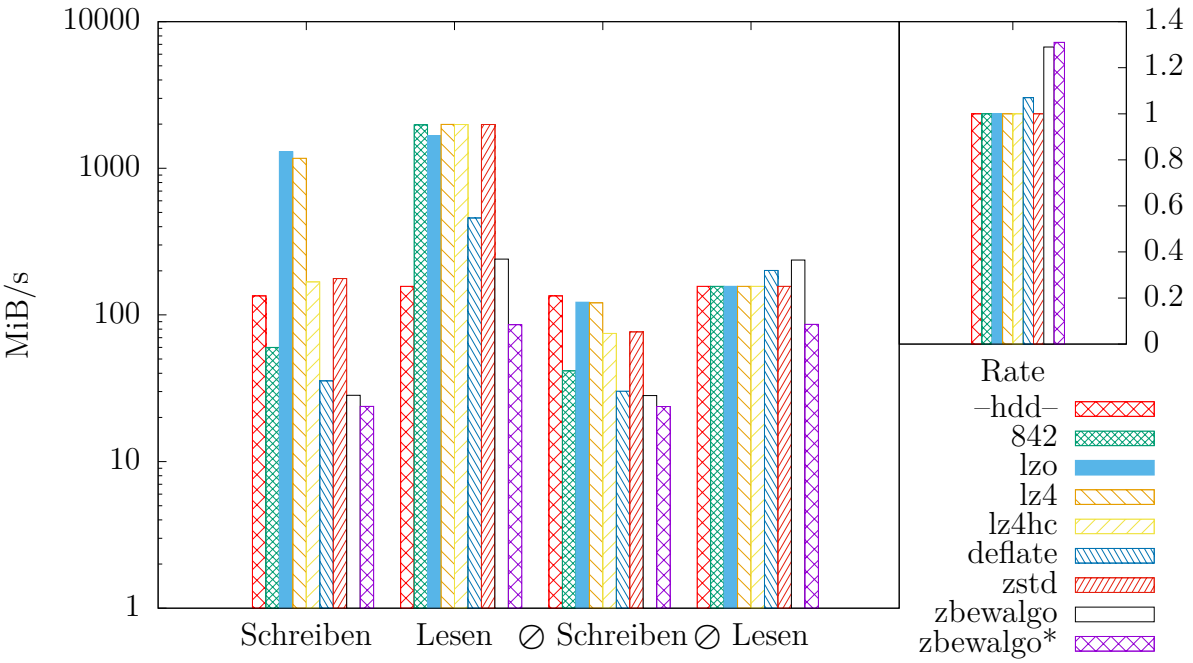

Figure 4.13: zBeWalgo Benchmarks mit dem Partdiff Datensatz

zBeWalgo ist für diese Art von Daten der am besten geeignete Algorithmus, da bisher keiner der anderen getesteten Algorithmen eine bemerkbare Kompressionsrate erzielen konnte. Die Anderen Algorithmen scheinen zwar schnell, komprimieren aber gar nicht. Die zBeWalgo Variante mit dem '*' erreicht eine höhere Kompressionsrate, indem nicht die Standardkombinationen sondern andere speziell ausgewählte Kombinationen verwendet werden. Ein Nebeneffekt besteht darin, dass die Kompressionsgeschwindigkeit niedriger wird. Die Schreibgeschwindigkeit bei der Benutzung von zBeWalgo ist hier zwar niedriger als die einer HDD, aber durch die Parallelisierbarkeit des Komprimierens zusammen mit dem Einsatzgebiet, in dem die Daten öfter gelesen als geschrieben werden, ist zBeWalgo in der Gesamt- Performance schneller als die HDD.

4.15 /dev/urandom

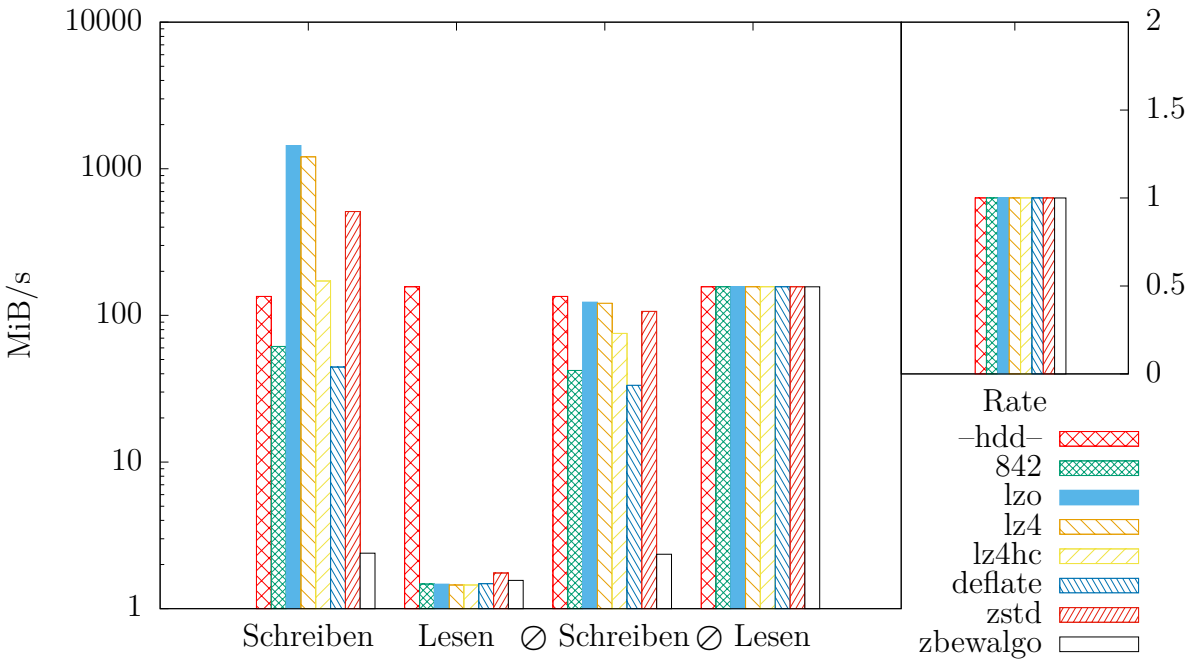

Figure 4.14: zBeWalgo Benchmarks mit dem Zufallszahlen Datensatz

Dieser Datensatz enthält Zufallsdaten und sollte somit durch keinen Kompressionsalgorithmus komprimiert werden können. Die LZ Varianten und zstd erkennen relativ schnell, dass die Daten nicht komprimierbar sind, während die anderen Algorithmen sehr viel Zeit verlieren. In Figure 4.14 kann man sehen, dass vollkommen unerwartet das Lesen der unkomprimierten Daten aus dem Arbeitsspeicher sehr lange dauert. Dies ist besonders ungewöhnlich, da im Fall von nicht komprimierbaren Daten die jeweiligen Dekompressionsalgorithmen niemals ausgeführt werden, und theoretisch nur ein einfaches memcpy verwendet wird.

5 Zusammenfassung und Ausblick

Im Verlauf des Projektes wurden verschiedenste Algorithmen auf verschiedenen Datensätzen ausprobiert. Es gibt bisher keinen Algorithmus, der für alle getesteten Datensätze optimale Ergebnisse liefert. Die meisten Algorithmen haben jeweils ihren eigenen Anwendungsfall, in dem sie entweder eine besonders hohe Kompressionsrate oder Geschwindigkeit erreichen. Je nach Anwendungsgebiet ist mal die Kompressions oder Dekompressions-Geschwindigkeit wichtig, wodurch wiederum verschiedene Algorithmen besser oder schlechter geeignet sind.

Der neu entwickelte Algorithmus zBeWalgo ist der einzige, der überhaupt in der Lage ist, einen Datensatz wie den von dem Partdiff Programm zu komprimieren. Bei Datensätzen wie z.B. HPCG ist zBeWalgo in allen Eigenschaften besser als die Vergleichsalgorithmen.

6 Anhang

6.1 Isabella Hurrikane CLOUD

Algorithmus	Kompressionsrate	Kompression (MiB/s)	Dekompression (MiB/s)	∅ Kompression (MiB/s)	∅ Dekompression (MiB/s)
–hdd–	1.00	134.70	156.62	134.70	156.62
842	2.15	92.68	453.38	82.59	339.27
lzo	2.06	1120.49	885.89	397.51	438.97
lz4	2.10	1040.36	1393.80	401.65	530.68
lz4hc	2.11	55.05	1397.29	50.80	533.45
deflate	2.31	43.67	334.07	42.05	294.45
zstd	2.31	190.22	626.60	162.70	460.91
zbewalgo	2.60	101.44	563.09	101.13	557.12

6.2 Isabella Hurrikane PREASSURE

Algorithmus	Kompressionsrate	Kompression (MiB/s)	Dekompression (MiB/s)	∅ Kompression (MiB/s)	∅ Dekompression (MiB/s)
–hdd–	1.00	134.70	156.62	134.70	156.62
842	1.00	57.42	1738.72	40.34	157.58
lzo	1.00	1342.14	1800.67	123.07	157.45
lz4	1.00	1159.76	1785.03	121.36	157.51
lz4hc	1.00	162.54	1810.00	73.91	157.53
deflate	1.00	37.30	1529.34	29.28	158.14
zstd	1.00	179.08	1756.10	77.34	158.13
zbewalgo	1.27	136.09	534.74	135.24	526.91

6.3 Isabella Hurrikane TEMPERATURE

Algorithmus	Kompressionsrate	Kompression (MiB/s)	Dekompression (MiB/s)	∅ Kompression (MiB/s)	∅ Dekompression (MiB/s)
–hdd–	1.00	134.70	156.62	134.70	156.62
842	1.00	60.00	1686.72	41.64	158.10
lzo	1.00	1331.26	1792.14	122.97	157.45
lz4	1.00	1137.80	1791.80	121.12	157.51
lz4hc	1.00	162.07	1810.07	73.82	157.53
deflate	1.05	31.90	463.15	27.13	188.92
zstd	1.03	150.66	1057.20	78.42	184.34
zbewalgo	1.51	146.45	552.43	143.29	525.52

6.4 Pflanzenwachstum

Algorithmus	Kompressionsrate	Kompression (MiB/s)	Dekompression (MiB/s)	∅ Kompression (MiB/s)	∅ Dekompression (MiB/s)
–hdd–	1.00	134.70	156.62	134.70	156.62
842	6.66	159.14	517.79	147.88	450.78
lzo	7.12	1059.62	677.37	741.38	573.34
lz4	7.48	1044.20	1236.66	761.39	929.52
lz4hc	7.92	71.12	1224.97	69.63	959.49
deflate	11.52	58.50	502.01	58.47	500.79
zstd	11.64	318.70	811.73	318.70	811.73
zbewalgo	9.09	29.98	193.76	29.81	192.64
zbewalgo*	9.09	75.17	192.71	74.16	191.63

6.5 ECOHAM

Algorithmus	Kompressionsrate	Kompression (MiB/s)	Dekompression (MiB/s)	∅ Kompression (MiB/s)	∅ Dekompression (MiB/s)
–hdd–	1.00	134.70	156.62	134.70	156.62
842	8.05	174.08	570.52	171.47	553.30
lzo	6.88	1242.85	1334.47	953.51	1069.85
lz4	7.36	1118.11	1440.75	936.70	1209.38
lz4hc	7.98	49.74	1544.46	49.50	1385.09
deflate	8.28	66.68	595.19	66.39	581.26
zstd	7.94	417.44	983.46	400.73	918.25
zbewalgo	8.02	36.65	239.72	36.52	238.13
zbewalgo*	8.80	657.17	1255.17	653.20	1244.29

6.6 FGPA

Algorithmus	Kompressionsrate	Kompression (MiB/s)	Dekompression (MiB/s)	∅ Kompression (MiB/s)	∅ Dekompression (MiB/s)
–hdd–	1.00	134.70	156.62	134.70	156.62
842	1.96	81.57	155.81	70.33	156.02
lzo	2.26	350.72	590.28	231.22	380.91
lz4	1.75	354.71	891.28	191.12	353.00
lz4hc	2.28	50.43	983.67	47.02	486.26
deflate	4.08	23.31	258.52	23.31	258.52
zstd	3.41	148.97	344.90	141.28	325.62
zbewalgo	3.34	73.49	63.02	71.32	65.20
zbewalgo*	3.40	77.22	65.78	75.06	67.76

6.7 Lukas-Corpus

Algorithmus	Kompressionsrate	Kompression (MiB/s)	Dekompression (MiB/s)	∅ Kompression (MiB/s)	∅ Dekompression (MiB/s)
–hdd–	1.00	134.70	156.62	134.70	156.62
842	1.22	63.04	405.55	50.56	220.60
lzo	1.20	709.85	1119.40	179.43	251.63
lz4	1.21	544.46	1281.23	171.30	263.09
lz4hc	1.25	80.34	1219.13	63.46	302.83
deflate	1.49	33.34	216.18	31.89	202.04
zstd	1.37	135.77	483.59	104.98	300.86
zbewalgo	1.31	10.77	107.24	10.44	122.54
zbewalgo*	1.33	16.34	55.22	15.62	73.42

6.8 fMRI Daten

Algorithmus	Kompressionsrate	Kompression (MiB/s)	Dekompression (MiB/s)	∅ Kompression (MiB/s)	∅ Dekompression (MiB/s)
–hdd–	1.00	134.70	156.62	134.70	156.62
842	1.00	59.24	1781.13	41.16	156.75
lzo	1.00	1377.43	1803.77	122.82	156.76
lz4	1.00	1203.84	1794.72	121.26	156.76
lz4hc	1.00	164.14	1808.47	74.03	156.77
deflate	1.00	42.78	1749.49	32.48	156.77
zstd	1.00	312.86	1810.90	94.23	156.77
zbewalgo	1.00	2.46	1750.45	2.42	156.59
zbewalgo*	1.00	1435.87	1781.55	123.25	156.74

6.9 Protein-Corpus

Algorithmus	Kompressionsrate	Kompression (MiB/s)	Dekompression (MiB/s)	∅ Kompression (MiB/s)	∅ Dekompression (MiB/s)
–hdd–	1.00	134.70	156.62	134.70	156.62
842	1.07	51.20	266.49	40.90	181.92
lzo	1.00	1325.32	948.49	122.78	157.22
lz4	1.00	660.71	993.75	112.58	157.60
lz4hc	1.00	150.76	942.34	71.44	157.67
deflate	1.71	41.66	155.94	41.44	155.95
zstd	1.76	203.22	374.83	203.22	374.83
zbewalgo	1.73	46.80	34.88	46.62	35.19
zbewalgo*	1.75	240.36	90.04	239.13	90.15

6.10 Linux

Algorithmus	Kompressionsrate	Kompression (MiB/s)	Dekompression (MiB/s)	∅ Kompression (MiB/s)	∅ Dekompression (MiB/s)
–hdd–	1.00	134.70	156.62	134.70	156.62
842	1.65	63.48	135.76	55.64	141.39
lzo	2.27	440.37	860.35	305.78	536.06
lz4	2.17	445.82	969.75	295.91	540.34
lz4hc	2.40	88.96	1036.66	82.93	640.53
deflate	3.27	42.36	250.24	42.35	250.24
zstd	3.10	168.45	397.68	165.07	387.87
zbewalgo	1.89	26.59	35.01	25.46	42.37
zbewalgo*	1.89	47.15	35.06	43.73	42.42

6.11 SAO Star Catalog

Algorithmus	Kompressionsrate	Kompression (MiB/s)	Dekompression (MiB/s)	∅ Kompression (MiB/s)	∅ Dekompression (MiB/s)
–hdd–	1.00	134.70	156.62	134.70	156.62
842	1.00	42.50	1357.04	32.31	156.65
lzo	1.00	575.61	1349.63	109.16	156.62
lz4	1.00	446.24	1313.04	103.47	156.62
lz4hc	1.00	136.84	1407.43	67.88	156.62
deflate	1.01	31.63	871.93	25.93	165.30
zstd	1.00	97.17	1370.72	56.45	156.62
zbewalgo	1.00	3.16	1314.78	3.08	156.62
zbewalgo*	1.00	3.15	1405.83	3.08	156.62

6.12 HPCG

Algorithmus	Kompressionsrate	Kompression (MiB/s)	Dekompression (MiB/s)	∅ Kompression (MiB/s)	∅ Dekompression (MiB/s)
–hdd–	1.00	134.70	156.62	134.70	156.62
842	2.35	107.91	145.75	83.50	149.54
lzo	1.76	481.22	854.58	171.17	262.18
lz4	1.76	464.56	1001.45	169.13	268.34
lz4hc	2.36	161.59	1054.60	112.40	341.13
deflate	4.61	46.13	428.44	43.75	335.92
zstd	6.02	225.69	498.49	191.47	404.35
zbewalgo	16.33	46.74	420.28	46.74	420.22
zbewalgo*	16.33	187.91	420.83	187.91	420.83

6.13 partdiff

Algorithmus	Kompressionsrate	Kompression (MiB/s)	Dekompression (MiB/s)	∅ Kompression (MiB/s)	∅ Dekompression (MiB/s)
–hdd–	1.00	134.70	156.62	134.70	156.62
842	1.00	60.06	1977.13	41.54	156.63
lzo	1.00	1298.41	1671.97	122.05	156.63
lz4	1.00	1169.54	1991.44	120.80	156.63
lz4hc	1.00	167.98	1989.73	74.76	156.63
deflate	1.07	35.53	458.06	30.23	201.06
zstd	1.00	177.12	1989.31	76.52	156.63
zbewalgo	1.29	28.33	240.06	28.19	236.99
zbewalgo*	1.31	23.80	85.65	23.73	86.23

6.14 /dev/urandom

Algorithmus	Kompressionsrate	Kompression (MiB/s)	Dekompression (MiB/s)	∅ Kompression (MiB/s)	∅ Dekompression (MiB/s)
–hdd–	1.00	134.70	156.62	134.70	156.62
842	1.00	61.40	1.47	42.17	156.62
lzo	1.00	1440.66	1.47	123.18	156.62
lz4	1.00	1204.79	1.45	121.16	156.62
lz4hc	1.00	171.46	1.45	75.43	156.62
deflate	1.00	44.40	1.48	33.39	156.62
zstd	1.00	510.31	1.75	106.57	156.62
zbewalgo	1.00	2.39	1.56	2.35	156.62