

Санкт-Петербургский государственный университет

Кафедра системного программирования

Группа 22.Б11-мм

Исследование проблем алгоритмов блочной компрессии в ядре Linux

Сичкар Георгий Константинович

Отчёт по производственной практике
в форме «Решение»

Научный руководитель:
Инженер-исследователь лаборатории технологий программирования
инфраструктурных решений СПбГУ А. И. Васенина

Санкт-Петербург
2025

Оглавление

Введение	3
1. Постановка задачи	5
1.1. Осенний семестр	5
1.2. Весенний семестр	5
1.3. Дальнейший план развития	5
2. Обзор	6
2.1. Алгоритмы компрессии	6
2.2. Bio struct	13
2.3. Обзор существующих решений	14
3. Архитектура решения	18
3.1. Требования	18
3.2. Пользовательский интерфейс	18
3.3. Архитектура	20
3.4. bcomp	22
3.5. bcomp_req	23
4. Реализация	24
4.1. Работа с памятью	24
4.2. Обработка IO-запросов	25
5. Тестирование	27
5.1. Проверка корректности	27
5.2. Эксперимент	27
Заключение	29
Список литературы	30

Введение

Данные являются неотъемлемой частью современного мира, большинство компаний сталкиваются с необходимостью ежедневно сохранять терабайты информации о пользователях, транзакциях или клиентах. Например, операторам связи приходится архивировать трафик, проходящий через них, не удаляя его на протяжении нескольких недель. Из-за этого эффективное хранение данных может быть критическим фактором для развития успешного бизнеса.

Возможным решением для оптимизации системы хранения данных является использование алгоритмов компрессии без потерь. Такой подход не является инновационным: например, файловая система ZFS [6] позволяет при конфигурации нового блочного устройства настроить сжатие хранимых данных. На данный момент существует множество библиотек с открытым исходным кодом, которые позволяют применять эти алгоритмы на практике. Одними из самых популярных решений являются: lz4 [4], основывающаяся на одноименном алгоритме, и zstd, которая использует Zstandard [9]. Немаловажным преимуществом этих библиотек является доступность API внутри Linux kernel [10]. Linux kernel — это ядро операционной системы с открытым исходным кодом, которое используется многими компаниями по всему миру в качестве основы для серверов. Из-за этого большинство системы хранения данных (СХД) также реализуются на базе Linux kernel. СХД внутри ядра обычно имеют представление блочного устройства. Это означает, что любые манипуляции с данными происходят блоками равного размера.

Однако реализация блочного устройства с компрессией не является тривиальной задачей. Так происходит по трем основным причинам. Во-первых, особенность работы алгоритмов сжатия: статически по входным данным невозможно определить во сколько раз они будут сжаты после применения алгоритма. Во-вторых, из-за высоких стандартов качества предъявляемых к СХД: пользовательские данные должны сохраняться без потери качества, а доступ к ним должен осуществляться достаточно быстро. Третья причина — это наличие инфраструктур-

ных ограничений внутри самого Linux kernel. Ярким примером такого ограничения является блочный интерфейс взаимодействия с данными. Все вышеперечисленное затрудняет разработку СХД с поддержкой компрессии. При этом на данный момент не существует универсального подхода для преодоления всех этих проблем в совокупности.

Для того чтобы иметь возможность на практике исследовать эффективность тех или иных инженерных решений, появилась необходимость в реализации некоторого отдельного блочного устройства для сжатия данных без потерь. Важным критерием которого является использование модульной архитектуры. Такой подход позволяет сравнивать разные реализации одних и тех же компонент.

1 Постановка задачи

Целью данной производственной практики является разработка прототипа блочного устройства с компрессией без потерь, и определение ключевых алгоритмических проблем появляющихся при реализации функциональности компрессии в коммерческом продукте.

Для достижения этой цели были поставлены следующие задачи.

1.1 Осенний семестр

1. Выполнить обзор алгоритмов компрессии без потерь, реализованных в ядре Linux, и существующих блочных устройств, использующих эти алгоритмы.
2. Разработать архитектуру блочного устройства, позволяющего подключать алгоритмы компрессии LZ4 и ZSTD.
3. Реализовать блочное устройство с компрессией на основе алгоритма LZ4 и линейной адресацией.
4. Протестировать целостность реализованного блочного устройства.

1.2 Весенний семестр

1. Реализовать нелинейную адресацию.
2. Добавить поддержку алгоритма ZSTD.

1.3 Дальнейший план развития

1. Произвести сравнение производительности с существующими решениями.

2 Обзор

2.1 Алгоритмы компрессии

Алгоритмы компрессии — это алгоритмы, позволяющие преобразовывать исходные данные в эквивалентные представления, использующие меньшее количество байт. Практическое применение этих алгоритмов подразумевает наличие двух основных компонент: во-первых, компрессор — устройство для кодирования последовательности данных; во-вторых, декомпрессор, используемый для получения из сжатого представления исходной последовательности. Существует два основных вида сжатия: без потери и с потерей данных. Различием является то, что для алгоритмов первого типа существуют декомпрессоры, способные восстановить оригинальную цепочку данных, в то время как для второго типа таких декомпрессоров нет, а данные возможно получить не полностью и с неточностями. Далее будут рассмотрены только алгоритмы, сжимающие данные без потерь.

При описании алгоритмов используется следующая терминология: строка — оригинальные данные; последовательность — подряд идущие закодированные данные, описывающие исходную подстроку; литерал — несжатый символ. Также стоит упомянуть, что для оценки алгоритмов сжатия используется такая численная характеристика, как коэффициент компрессии — размер исходных данных, деленный на размер сжатых.

2.1.1 LZ77

LZ77 — это алгоритм компрессии без потери данных, опубликованный в 1977 году Авраамом Лемпелем и Яковом Зивом [11]. Основной идеей алгоритма стало использование словарного подхода — разбиение данных на подстроки и замена их на индексы в словаре. LZ77, в отличие от алгоритма Хаффмана и его модификаций, не использует неравномерные коды для минимизации кодировки часто повторяющихся символов (энтропийное кодирование). Вместо этого алгоритм сжимает повторяю-

щиеся подстроки.

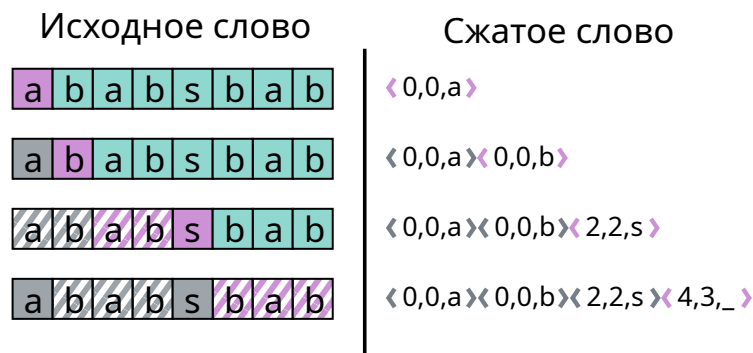


Рис. 1: Пример компрессии lz77.

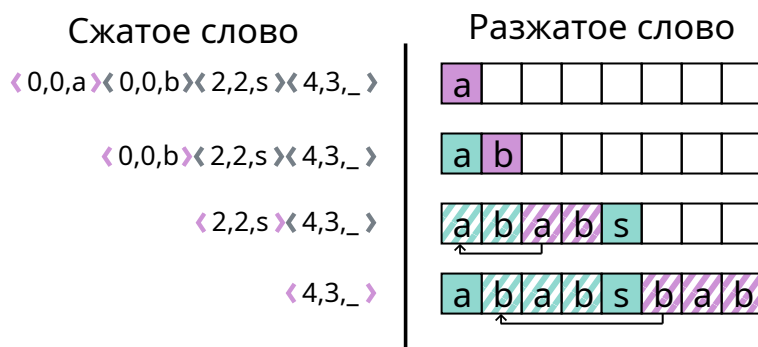


Рис. 2: Пример декомпрессии lz77.

Подход к сжатию, предлагаемый LZ77, подразумевает замену повторов на ссылки на позиции в исходной последовательности данных, с указанием количества литералов, которые следует скопировать с этой позиции. Формат последовательности, описывающий одну подстроку, состоит из двух чисел и одного литерала: $\langle \text{сдвиг}, \text{длина совпадения}, \text{литерал} \rangle$. Пусть текущая позиция в данных на каком-то шаге алгоритма описывается индексом i , тогда $\text{сдвиг} = k$ указывает на $(i-k)$ позицию в исходной строке. Длина совпадения хранит размер подстроки n , которую нужно будет скопировать из раскодированных данных с позиции $(i-k)$ на позицию i . Литерал вставляется на позицию $(i+n)$ после применения копирования. Подробные примеры компрессии и декомпрессии приведены на рисунках 1 и 2.

LZ77 практически не используется на практике, гораздо более популярными алгоритмами сжатия являются LZ4 и ZSTD. Оба алгоритма имеют эталонные реализации с открытым исходным кодом на языке программирования Си, а также предоставляют API в Linux kernel [10].

2.1.2 LZ4

LZ4 представляет собой определенный стандарт хранения сжатых данных, а также реализацию с открытым исходным кодом [4]. Важно отметить, что стандарт никак не специфицирует техническую реализацию алгоритма. Подобно своему предшественнику LZ77, LZ4 не использует энтропийное сжатие, а применяет словарный подход. Сжатые этим алгоритмом данные состоят из набора последовательностей, каждая из которых представляет собой четыре компонента: токен, литералы, сдвиг, длина совпадения. Одной из отличительных особенностей алгоритма является то, что возможно хранение нескольких литералов, а под длину совпадения может отводиться переменное количество байт. Чтобы поддерживать это, используется токен — служебный байт, в начале последовательности, применяемый для описания количества байт, отводимых под литералы и под длину совпадения. Сдвиг — означает то же самое, что и в алгоритме LZ77, на его хранение отводится 2 байта.

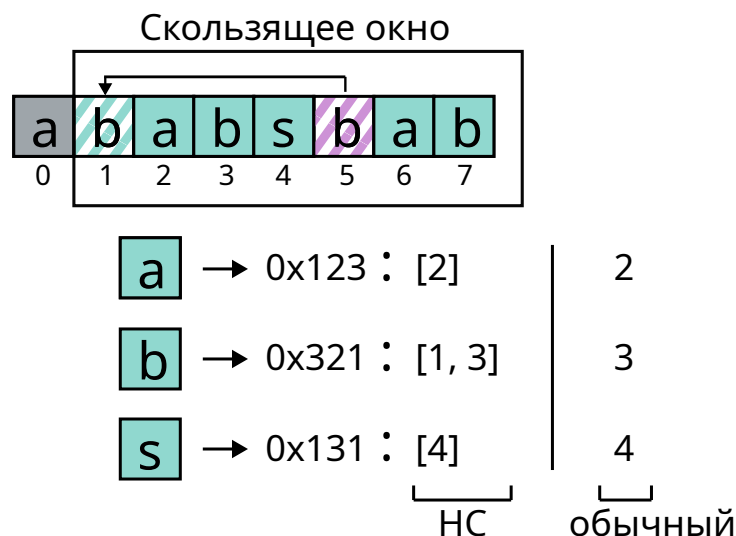


Рис. 3: Описание алгоритма lz4.

Основная идея алгоритма заключается в побайтовом считывании входной последовательности и постепенной генерации эквивалентного представления на основе встреченных ранее подстрок. Так, если полученный байт появился впервые, то он сохраняется в хэш-таблицу со своим индексом в исходной строке. В противном случае из хэш-таблицы достается индекс предыдущего появления этого символа и затем проис-

ходит побайтовое синхронное сравнение подпоследовательностей, начинающихся с текущего индекса и индекса последнего появления. Сравнение происходит до первого расхождения или пересечения индексов. Стоит обратить внимание, что на практике считывают сразу по 4 байта, а не по одному, поэтому использование хэшей имеет смысл. Для оптимизации времени, lz4 при поиске совпадающих подстрок использует подход скользящего окна, при котором данные сначала сканируются в определенной ограниченной области, после чего эта область перемещается. Соответственно совпадения проверяются только внутри этой области, в то время как более старые совпадения отбрасываются. Окно обычно имеет размер 65536 байт, что обусловлено максимальным значением сдвига (2 байта). Схематичный пример этого алгоритма приведен на рис. 3. На нем будет найдено две подстроки длиной три, начинающиеся с индексов один и пять.

Официальная реализация LZ4 предлагает два вида алгоритма: обычный и HC подходы. Первый вариант оптимизирует скорость сжатия, а второй — коэффициент компрессии. Это проявляется, например, в том, что обычный подход не сохраняет в хэш-таблице предыдущие индексы попавшегося ранее литерала, поэтому рассматриваются всегда последние совпавшие подстроки. В то время как HC подход сохраняет все появления и старается подобрать оптимальное совпадение на основе нескольких встретившихся ранее подстрок. В том числе из-за этого HC-алгоритм сможет найти совпадение строк `bab` в примере из рис. 3, а обычный — нет.

Немаловажным преимуществом LZ4 является использование параметра акселерации в функциях сжатия как HC варианта, так и обычного. Этот параметр используется для ускорения работы компрессора в ущерб оптимальности сжатия. Акселерация подразумевает перепрыгивание какого-то количества байт в случае, когда долгое время не находятся одинаковые подстроки. Соответственно, параметр акселерации влияет на число пропущенных байт.

Compressor	Ratio	Compression	Decompression
memcpy	1.000	13700 MB/s	13700 MB/s
LZ4 default (v1.9.0)	2.101	780 MB/s	4970 MB/s
LZO 2.09	2.108	670 MB/s	860 MB/s
QuickLZ 1.5.0	2.238	575 MB/s	780 MB/s
Snappy 1.1.4	2.091	565 MB/s	1950 MB/s
Zstandard 1.4.0 -1	2.883	515 MB/s	1380 MB/s
LZF v3.6	2.073	415 MB/s	910 MB/s
zlib deflate 1.2.11 -1	2.730	100 MB/s	415 MB/s
LZ4 HC -9 (v1.9.0)	2.721	41 MB/s	4900 MB/s
zlib deflate 1.2.11 -6	3.099	36 MB/s	445 MB/s

Рис. 4: Сравнение производительности алгоритмов сжатия.

В заключении стоит сказать, что LZ4 предполагается использовать в тех случаях, когда скорость компрессии и декомпрессии очень важна, а оптимальностью сжатия можно пожертвовать, это подтверждается и на официальных замерах производительности, сделанных разработчиками LZ4. Из сравнительной таблицы на рис. 2 видно, что обычный вариант LZ4 при коэффициенте сжатия в два раза обходит по скорости аналогичные решения в несколько раз. В качестве тестовых данных использовался датасет Silesia [7].

2.1.3 ZSTD

Еще одним не менее популярным решением с открытым исходным кодом является ZSTD (Zstandard). Это реализация одноименного стандарта RFC8878 [12]. RFC8878 представляет довольно гибкое описание формата сжатых данных и аналогично LZ4 не специфицирует техническую реализацию самого алгоритма. Ключевой особенностью ZSTD является использование двухэтапной компрессии, где первый этап — это сжатие с использованием словарного подхода, а второй этап — применение энтропийного кодирования для уплотнения полученных данных.

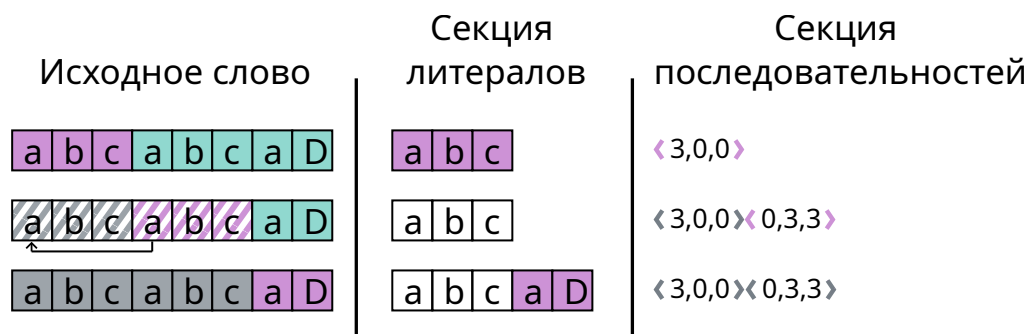


Рис. 5: Компрессия с алгоритмом ZSTD.

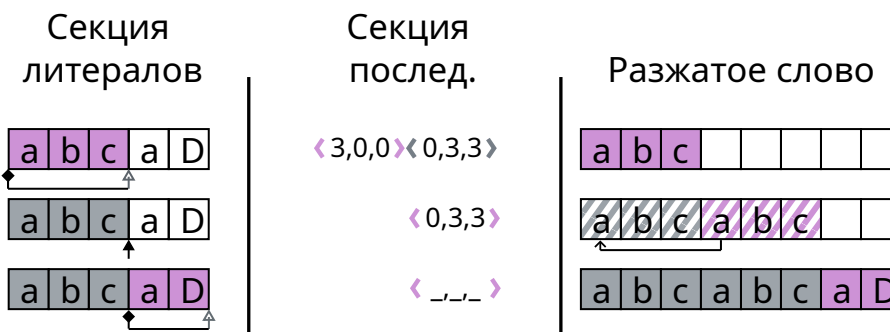


Рис. 6: Декомпрессия с алгоритмом ZSTD.

Преобразованные алгоритмом ZSTD данные представляются в виде блока, разбитого на две основные секции. Относящаяся к ним мета информация сохраняется в заголовке. Сначала идет секция литералов, представляющая из себя набор несжатых символов (причем между литералами нет никаких разделителей, они записываются подряд). Затем располагается секция последовательностей — набор ссылок на позиции в разжатых данных для применения словарного подхода. Размеры секций описываются в заголовке блока. Каждая последовательность состоит из трех компонент: **<количество литералов, сдвиг, длина совпадения>**. Количество литералов здесь отвечает за число байт, которые следует скопировать из секции литералов при декодировании текущей последовательности. При этом байты считываются не от начала секции, а от предыдущего считывания. Иными словами, если при декодировании в секции литералов лежит пять байт, и из нее было скопировано три байта, то дальше для копирования доступны только последние два байта. Подробные примеры компрессии и декомпрессии стадии словарного подхода представлены на рисунках 5 и 6 соответственно. Важной особенностью стандарта является то, что все непрочитанные литералы при

декомпрессии должны добавляться в конец разжатого слова. Это удобное свойство, позволяющее алгоритму ZSTD не производить негативное сжатие — сжатие, при котором эквивалентное представление занимает столько же или больше памяти, чем исходная строка.

Реализация ZSTD делает акцент на оптимизации коэффициента сжатия, а не скорости работы. В том числе из-за этого, например, при поиске совпадений выбирается наиболее длинное, после чего проверяется имеет ли смысл сжимать эту наибольшую строку через ссылку на позицию в строке, или выгоднее записать совпадение в секцию литералов. Попыткой оптимизировать процент сжатия данных объясняется и наличием второго этапа — энтропийного кодирования, который при компрессии, используя алгоритм Хаффмана, дополнительно сжимает секцию литералов. Кроме того, отдельные компоненты последовательностей тоже сжимаются, но уже посредством FSE. FSE [2] — это алгоритм энтропийного сжатия, который также, как и алгоритм Хаффмана, эффективно сжимает данные на основе вероятностей появления символов.

Compressor name	Ratio	Compression	Decompress.
zstd 1.5.6 -1	2.887	510 MB/s	1580 MB/s
zlib 1.2.11 -1	2.743	95 MB/s	400 MB/s
brothli 1.0.9 -0	2.702	395 MB/s	430 MB/s
zstd 1.5.6 -fast=1	2.437	545 MB/s	1890 MB/s
zstd 1.5.6 -fast=3	2.239	650 MB/s	2000 MB/s
quicklz 1.5.0 -1	2.238	525 MB/s	750 MB/s
lzo1x 2.10 -1	2.106	650 MB/s	825 MB/s
lz4 1.9.4	2.101	700 MB/s	4000 MB/s
lzf 3.6 -1	2.077	420 MB/s	830 MB/s
snappy 1.1.9	2.073	530 MB/s	1660 MB/s

Таблица 1: Сравнение алгоритмов компрессии.

Все вышеперечисленное значительно замедляет процесс компрессии и декомпрессии, однако дает существенный прирост в коэффициенте

компрессии. Подтверждением этого являются замеры, проведенные разработчиками ZSTD [9] на тестовом корпусе Silesia [7]. Результаты замеров представлены на рис. 1.

2.2 Bio struct

Прежде чем переходить к обзору существующих решений, стоит сделать акцент на реализации блочного стека в Linux kernel. Основной структурой, представляющей запросы ввода-вывода на блочное устройство, является `struct bio`. Внутри нее содержатся пользовательские данные, а также дополнительная мета информация. Важной особенностью является использование SG-буферов (Scatter-Gather) — это механизм представления непрерывного буфера через массив отдельных страниц памяти, каждая из которых дополнительно содержит смещение и размер, используемые для хранения части полезной нагрузки. Такой подход позволяет оптимизировать операции ввода-вывода в случаях, когда данные хранятся в разрозненных областях памяти.

Ниже приведены основные поля `struct bio`:

1. `bi_io_vec` — массив структур, описывающих SG-буфер;
2. `bi_vcnt` указывает количество структур внутри `bi_io_vec`;
3. `bi_iter` характеризует текущую позицию в операции ввода-вывода (какие данные следует записать или прочитать, с каким смещением и какого размера);
4. `bi_bdev` описывает блочное устройство, на которое пришел запрос;
5. `bi_end_io` — поле с функцией, которая должна будет вызваться в момент завершения операции ввода-вывода, связанной с текущим `struct bio`.

2.3 Обзор существующих решений

Существуют различные реализации блочных устройств, поддерживающих компрессию данных. Самыми популярными решениями являются `dm-vdo` [13] — модуль Linux kernel, а также реализация файловой системы ZFS — OpenZFS [6]. Оба решения имеют открытый исходный код.

2.3.1 `dm-vdo`

`Dm-vdo` — это модуль ядра Linux, опирающийся на инфраструктуру `device mapper`. Он позволяет создавать виртуальные блочные устройства, которые используют компрессию для оптимизации хранимых данных. `Device mapper` — это подсистема ядра Linux, предоставляющая абстрактный уровень для создания и управления блочными устройствами. Компрессия внутри `dm-vdo` реализована на основе алгоритм сжатия LZ4 с дефолтными параметрами акселерации. Блочные устройства, созданные этим модулем могут работать с блоками кратными 4096 (4K) или 512 байтам в зависимости от начальной конфигурации. Далее при описании особенностей модуля будет использоваться размер блока 4K, так как создатели рекомендуют устанавливать именно его.

Особенностью тома `vdo` является то, что под каждый запрос на запись выделяются дополнительные буфера, в которые производится копирование пользовательских данных. Так происходит из-за того, что в некоторых случаях запрос на запись может быть завершён успешно ещё до окончания его обработки, а значит и буфера могут быть освобождены раньше времени. Этот сценарий возможен, если `dm-vdo` гарантирует успешность запроса. Например, когда на нижележащем устройстве зарезервировано место под эту запись. Таким образом, даже до начала сжатия, пространство необходимое на обработку запроса удваивается.

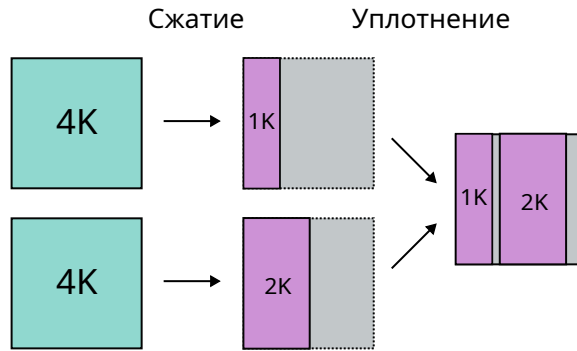


Рис. 7: Компрессия в dm-vdo.

За само сжатие в dm-vdo отвечает внутренний компонент — **packer**. Эта сущность осуществляет два основных этапа: сжатие данных и уплотнение. На первом этапе происходит непосредственное использование алгоритма компрессии LZ4. Второй же этап необходим из-за того, что данные после положительного сжатия будут точно меньше 4К, а само блочное устройство производит операции ввода-вывода только блоками ровно по 4К. Из-за этого возникает необходимость упаковывать несколько сжатых блоков в один. При этом dm-vdo позволяет сохранять в одном 4К блоке от 2 до 14 сжатых. Отсюда вытекает важное наблюдение, что если для сжатых данных не нашлись комплиментарных данных, способных поместиться в один 4К блок, то они будут сохранены в разжатом виде. Более подробно процесс уплотнения описан на рис. 7.

2.3.2 ZFS

ZFS — это файловая система, которая сочетает в себе функциональность файловой системы и менеджера логических томов. Её реализация OpenZFS [5] поддерживает создания виртуальных устройств хранения — **zvol**'ов, с блочным интерфейсом взаимодействия. Одним из параметров создания **zvol** является **compression**. **Compression** позволяет настроить прозрачное сжатие данных, для оптимизации дискового пространства. OpenZFS поддерживает довольно много алгоритмов сжатия, среди которых есть ZSTD и LZ4.

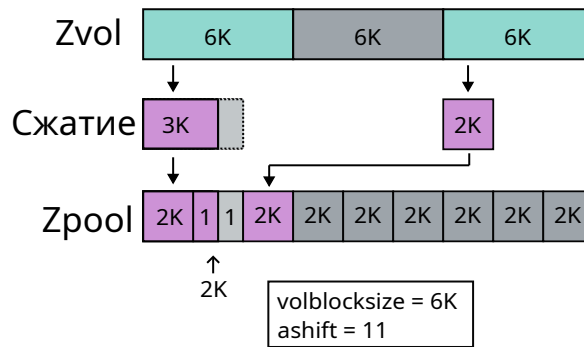


Рис. 8: Компрессия в zvol (zfs).

Сама компрессия данных в ZFS устроена достаточно наивна, для ее эффективной работы необходимым условием является наличие различия в размерах блоков, с которыми работает `zvol` и нижележащее устройство. Стоит уточнить, что `zvol`'ы создаются поверх `zpool` — логического объединения нескольких блочных устройств. Размер блоков, с которым работает `zpool`, называется `ashift` и задается либо пользователем, либо автоматически на основе базовых устройств. Кроме того пользователь также может указать размер блоков самого `zvol` — `volblocksize`, значение которого по умолчанию равняется 16K. Все это влияет на компрессию из-за специфичной реализации, при которой сжатые блоки данных округляются вверх до кратности `ashift`. При этом сжатие происходит всегда на блоках, соответствующих размеру `volblocksize`. Подробное описание процесса сжатия представлен на рис. 8.

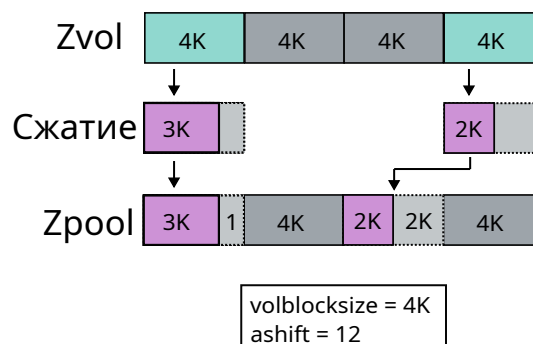


Рис. 9: Компрессия в zvol (zfs).

Такая реализация имеет существенный недостаток: если при настройке `volblocksize` будет равняться размеру блоков, который характери-

зует `ashift`, то несмотря на то, что сжатие данных будет производиться, реальная оптимизация хранимого пространства происходить не будет. Так как сжатый блок будет всегда заполняться нулями обратно до исходного размера. Более подробно эта проблема описана на рис. 9.

Проверка этого факта происходила на основе ZFS версии 2.2.6. Zpool был настроен с `ashift` равным 12 и 14, что соответствует размеру блоков 4К и 16К соответственно. Для создания использовалась команда `zpool create -o ashift=12`. Zvol имел начальную конфигурацию `volblocksize=4K` или `volblocksize=16K` в зависимости от используемого `ashift`. В качестве тестовых данных использовался датасет Silesia [7]. А проверка коэффициента сжатия осуществлялась через команду `zfs get compressratio`. В результате и на 4К и на 16К процент сжатия был нулевым (компрессия данных не происходила).

3 Архитектура решения

Для того чтобы иметь возможность сравнивать эффективность и проверять работоспособность различных реализаций блочной компрессии в ядре Linux было необходимо разработать блочное устройство, на базе которого можно было бы производить тестирование. Этот раздел подробно описывает архитектуру необходимого блочного устройства, а также требования, предъявленные к этой архитектуре.

3.1 Требования

При проектировании системы были выдвинуты два основных требования. Во-первых, для оценки эффективности того или иного подхода блочной компрессии было необходимо собирать статистическую информацию как о количестве сжатых блоках, так и об объеме сэкономленного пространства на всем устройстве. Во-вторых, для удобства интеграции новых подходов, вся система должна быть модульной. При этом модульность относится как к частям, отвечающим за непосредственное сжатие данных, так и к подсистеме трансляции логических адресов в физические. Стоило также разработать интерфейс конфигурации, чтобы была возможность пересоздавать блочное устройство без перезагрузки модуля.

3.2 Пользовательский интерфейс

Для реализации функциональности конфигурации блочного устройства из пространства пользователя были выбраны параметры модуля в совокупности с `sysfs`. `sysfs` — это виртуальная файловая система, которая предоставляет удобный способ взаимодействовать с Linux kernel. Она монтируется обычно в `/sys` и организована как иерархия директорий и файлов. Параметры модуля представляются файлами в `sysfs` и позволяют влиять на поведение модуля без необходимости перезагружать его.

Тестовое блочное обладает тремя параметрами: `bcomp_unmapper`,

`bcomp_mapper` и `bcomp_stats`. Первый параметр доступен только на запись и может быть использован для отвязки нижележащего блочного устройства. Два других параметра поддерживают как запись, так и чтение. `bcomp_mapper` позволяет создать блочное устройство с компрессией или получать информацию об уже созданном. Для получения информации о статистических данных используется `bcomp_stats`, кроме того этот параметр может быть использован для обнуления статистики.

3.2.1 Конфигурация нового блочного устройства

Листинг 1.: Команда конфигурации блочного устройства

```
$ echo -n "<BS> <COMP_TP> <CPRF> <DPRF> <MAP_TP> <UDEV>" \  
> bcomp_mapper
```

Стоит подробнее рассмотреть интерфейс создания блочного устройства. На листинге 1 для записи на `bcomp_mapper` используется утилита `echo` с параметром `-n`, который позволяет производить запись без символа перевода строки в конце аргумента. В качестве самого аргумента передается строка, состоящая из шести параметров. Первый параметр — `<BS>` устанавливает размер блока, которым оперирует конфигурируемое блочное устройство, например, 4К. `<COMP_TP>` параметризует алгоритм компрессии, используемый для сжатия данных. Также есть `<CPRF>` и `<DPRF>`. Они позволяют настраивать профили компрессии и декомпрессии внутри конкретного алгоритма. Например, `COMP_TP` может быть равен `lz4`, а `CPRF=16`, что соответствует HS реализации с нулевым параметром акселерации. Способ отображения логических адресов в физические задается параметр `<MAP_TP>` (физические адреса относятся к нижележащему устройству, в то время как пользователь пишет именно по логическим адресам). Последним значением передается путь до блочного устройства, поверх которого будет работать компрессия.

3.2.2 Статистика

Листинг 2.: Команда конфигурации блочного устройства

1	compressed_reqs_cnt_25: 1
2	compressed_reqs_cnt_50: 2
3	compressed_reqs_cnt_75: 3
4	compressed_reqs_cnt_99: 4
5	uncompressed_reqs_cnt: 5
6	all_reqs_cnt: 15 <i># общее количество блоков</i>
7	data_in_bytes: 4096*15
8	compressed_data_in_bytes: 9783

Статистика позволяет получить информацию об объеме сжатых данных, а также количеству блоков, которые удалось сжать. Важно уточнить, что если после применения алгоритма сжатия размер блока остался прежним или не уменьшился, то такой блок должен сохраняться в неизменном виде. Такой подход позволяет избежать лишних затрат на декомпрессию, а также гарантировать, что данные всегда будут помещаться в один рабочий размер — размер, с которым работает блочное устройство. Сами статистические данные представляются в виде восьми полей. Поля с первого по четвертое содержат в себе информацию о количестве блоков, размер которых после сжатия попадает в диапазон: 0%...<25%, 25%...<50%, 50%...<75% или 75%...<99%; где проценты берутся от размера несжатых данных. Также сохраняется информация о несжатых блоках. Кроме того, для вычисления коэффициента сжатия, выделено два отдельных поля с общим объемом данных до и после применения алгоритма сжатия (оба поля в байтах). Пример полей с наименованиями представлен на листинге 2

3.3 Архитектура

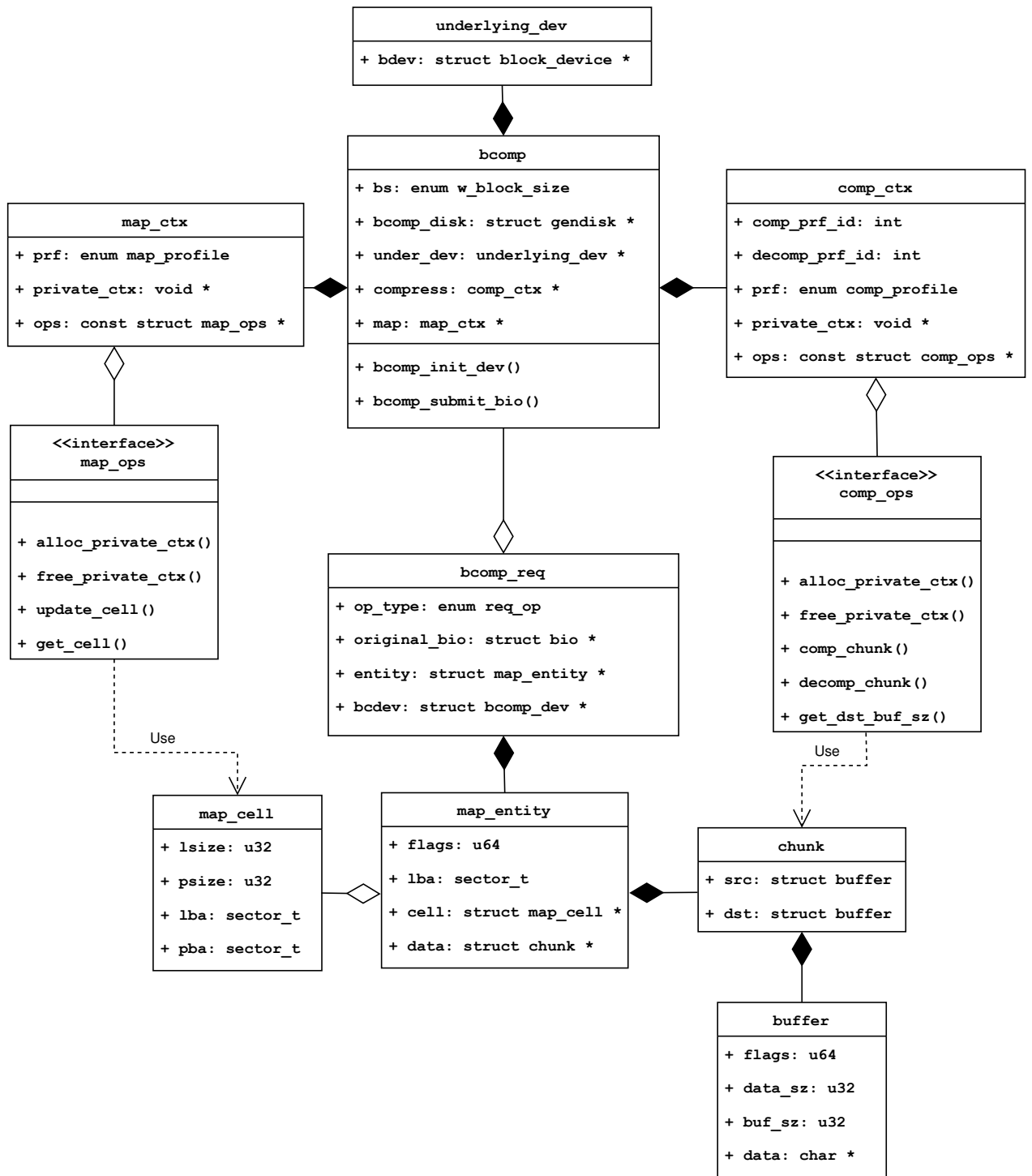


Рис. 10: UML-диаграмма устройства bcomp.

Архитектура разрабатывалась с учетом вышеперечисленных требований, поэтому упор сделан на низкую связность, которая позволяет обеспечить параметризуемость. UML-диаграмма этой архитектуры представлена на рис 10. Основными сущностями на ней являются `bcomp` и `bcomp_req`. Первая описывает созданное блочное устройство, а вторая представляет приходящие запросы. Немаловажными частями яв-

ляются `comp_ctx` и `map_ctx`, они представляют компоненты, отвечающие за компрессию данных и отображение логических адресов в физические соответственно. Для параметризации существуют интерфейсы `comp_ops` и `map_ops`. Реализация `comp_ops` позволяет использовать функции для аллокации и деаллокации приватной информации `comp_ctx`, а также предоставляет `comp_chunk()` и `decomp_chunk()`, инкапсулирующие внутри себя всю логику алгоритмов компрессии и декомпрессии. `Chunk` — это сущность, описывающая данные до сжатия и после сжатия. Она состоит из двух буферов: `src` хранит исходные данные, в то время как в `dst` записываются сжатые. Буфера также представляются отдельной сущностью. Интерфейс `map_ops` необходим для реализации различных подходов линейной и нелинейной адресации. Помимо функций для получения приватного контекста `map_ops` предоставляет `update_cell()` и `get_cell()`. Они необходимы для получения информации об отображении: `update_cell()` дает возможность обновить физический адрес, соответствующий логическому, в то время как `get_cell()` используется для получения физического адреса по логическому. Обе эти функции манипулируют такими структурами, как `map_cell`. Основной смысл `map_cell` заключается в сохранении сопоставления между логическим адресом в физический.

3.4 bcomp

Как было описано ранее, центральной сущностью является `bcomp`. Именно она содержит внутри себя `comp_ctx` и `map_ctx` с конкретными реализациями `comp_ops` и `map_ops`. Кроме того внутри `bcomp` хранится информация про нижележащее устройство, представленное `underlying_dev`. Тут же специализируется размер блоков, с которым будет работать блочное устройство. Основными функциями являются: `bcomp_init_dev()`, необходимая для инициализации нового блочного устройства с компрессией; а также функция `bcomp_submit_bio()`, которая используется блочным стеком Linux для отправки запросов, основанных на `bio struct`.

3.5 bcomp_req

Сущность `bcomp_req`, описывающая запрос внутри блочного устройства, состоит из типа запроса; конкретного экземпляра устройства, к которому относится; а также структуры `map_entity`. Она объединяет в себе `map_cell` и `chunk` текущего запроса. Важно заметить, что в текущей реализации блочное устройство не поддерживает IO-запросы с размера не равного `bcomp.bs`. Поэтому `bcomp_req` создается на каждый обрабатываемый блок размером `bcomp.bs`.

4 Реализация

В этом разделе описывается реализация блочного устройства для тестирования подходов к блочной компрессии. Текущая версия представляет собой модуль, способный конфигурировать одно блочное устройство. Кроме того в соответствии с требованиями добавлена поддержка обновления конфигурации через пересоздание `bcomp`-устройства. При этом перезагрузка всего модуля не требуется.

Реализована модульная архитектура с двумя видами сжатия и одним видом отображения: `empty_comp` — компрессия данных не происходит; `lz4_comp` — сжатие основанное на алгоритме `lz4` (здесь поддерживаются все возможные профили компрессии и декомпрессии). Отображение логических адресов в физические в текущей реализации происходит линейно (логический равен физическому), из-за чего фактической экономии пространства не происходит. Этот вид имеет смысл так как в дальнейшем его можно использовать для относительной оценки эффективности подходов с нелинейной адресацией.

На данный момент реализация обладает двумя основными ограничениями, которые в дальнейшем планируется устранить: во-первых, созданные блочные устройства не могут обрабатывать IO-запросы, чей размер не соответствует параметру `<BS>` из одноименного поля `struct bcomp`; во-вторых, поддержка параллельной обработки запросов также отсутствует.

Далее будут описаны некоторые технические особенности реализации `bcomp` модуля. С текущей реализацией можно ознакомиться в репозитории проекта [1].

4.1 Работа с памятью

Для начала стоит обратить внимание на управление памятью, так как она является важным ресурсом от которого зависит эффективность блочной компрессии. Основные затраты по памяти происходят при выделении буферов для сжатия, так как алгоритмы компрессии доступные в Linux kernel требуют наличия двух непрерывных буферов: `src` —

с исходными данными и `dst` — куда будет записан результат сжатия. И хотя за выделение одного из двух буферов (`src` или `dst` в зависимости от типа запроса) отвечает вышележащее устройство. Например, когда случается запрос на запись в функцию `bcomp_submit_bio()` приходит `bio` с `src` буфером, содержащем данные. Приходится аллоцировать и `src`, и `dst`. Так как буфера должны быть именно непрерывными, а не SG. Альтернативным подходом является модификация алгоритмов компрессии, однако на данный момент этот вариант не поддерживается.

Важно заметить, что текущая реализация для исключения случаев негативного сжатия хранит два вида блоков: с оригинальными данными и со сжатыми. Эта особенность позволяет оптимизировать работу с памятью при запросах на чтение. Поэтому в случае когда сжать данные не удалось и они хранятся в оригинальном виде, их считывание не требует выделения дополнительной памяти, достаточно воспользоваться пользовательским буфером. Хотя запросы на запись оптимизировать аналогичным образом невозможно из-за того, что определение негативного сжатия требует запуска алгоритма компрессии, что невозможно без выделенных буферов.

4.2 Обработка IO-запросов

Стоит подробнее рассмотреть графы потоков управления запросов на запись и на чтение. Рисунок 11 описывает запись, в то время как запрос на чтение изображен на рис. 12. Основная логика запроса на запись происходит до отправки данных на нижележащее устройство, именно здесь происходит выделение буферов, сжатие данных и обновление отображение из логических адрес в физические. В свою очередь чтение разбито на две основные части: до отправки запроса на нижележащее устройство и после него. В первом этапе определяется физический адрес и тип хранимого блока (сжатый или исходный). В зависимости от этого происходит аллокация памяти и конфигурируется новое `bio`. Второй этап происходит в функции завершения нового `bio`. Именно там данные разжимаются и копируются в пользовательский буфер.

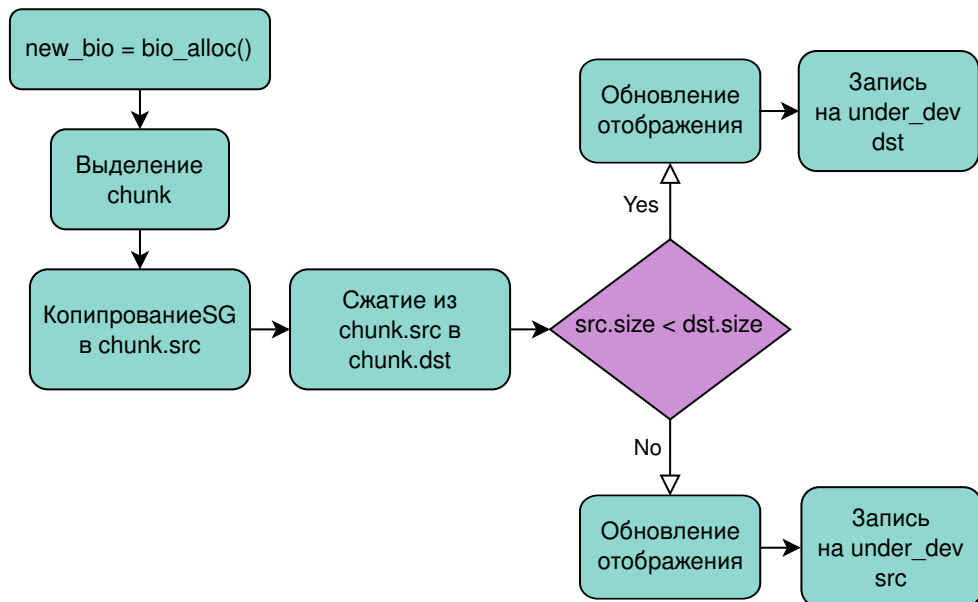


Рис. 11: Граф потока управления (запрос на запись).

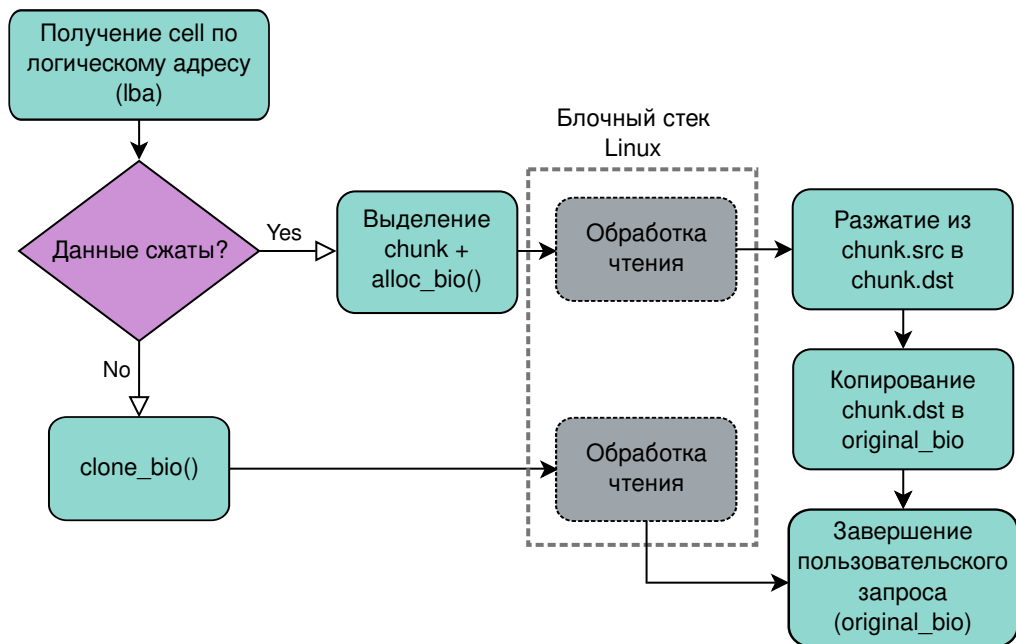


Рис. 12: Граф потока управления (запрос на чтение).

5 Тестирование

Тестирование было разбито на два основных этапа: первый заключался в проверки корректности запросов на запись и на чтение, в то время как на втором этапе собирались статистические данные.

5.1 Проверка корректности

Проверка корректности представляло собой системное тестирование со следующим сценарием:

1. конфигурируется блочное устройство;
2. запись на устройство;
3. чтение с устройства;
4. проверка, что записанные данные соответствуют прочитанным.

Этот сценарий применялся ко всем реализованным вариантам начальной конфигурации.

Для автоматизации тестирования использовались sh-скрипты и утилита fio. Fio [3] — это утилита, позволяющая генерировать запросы чтения и записи на блочное устройство без использования файловой системы.

5.2 Эксперимент

В текущей реализации тестирование представляло собой анализ алгоритма компрессии, интегрированного в тестовое блочное устройство. Поэтому на сконфигурированное блочное устройство производилась только запись, после чего происходил сбор статистик. Для записи использовалась утилита dd с параметром `oflag=direct`. Эта утилита, как и fio, позволяет совершать запись без использование файловой системы, однако в отличие от fio используется не сгенерированные данные, а пользовательские. Датасетом для тестирования был выбран корпус Silesia [7].

График коэффициента компрессии разных конфигурацию при фиксированном $\langle BS \rangle = 4K$ представлен на графике 13.

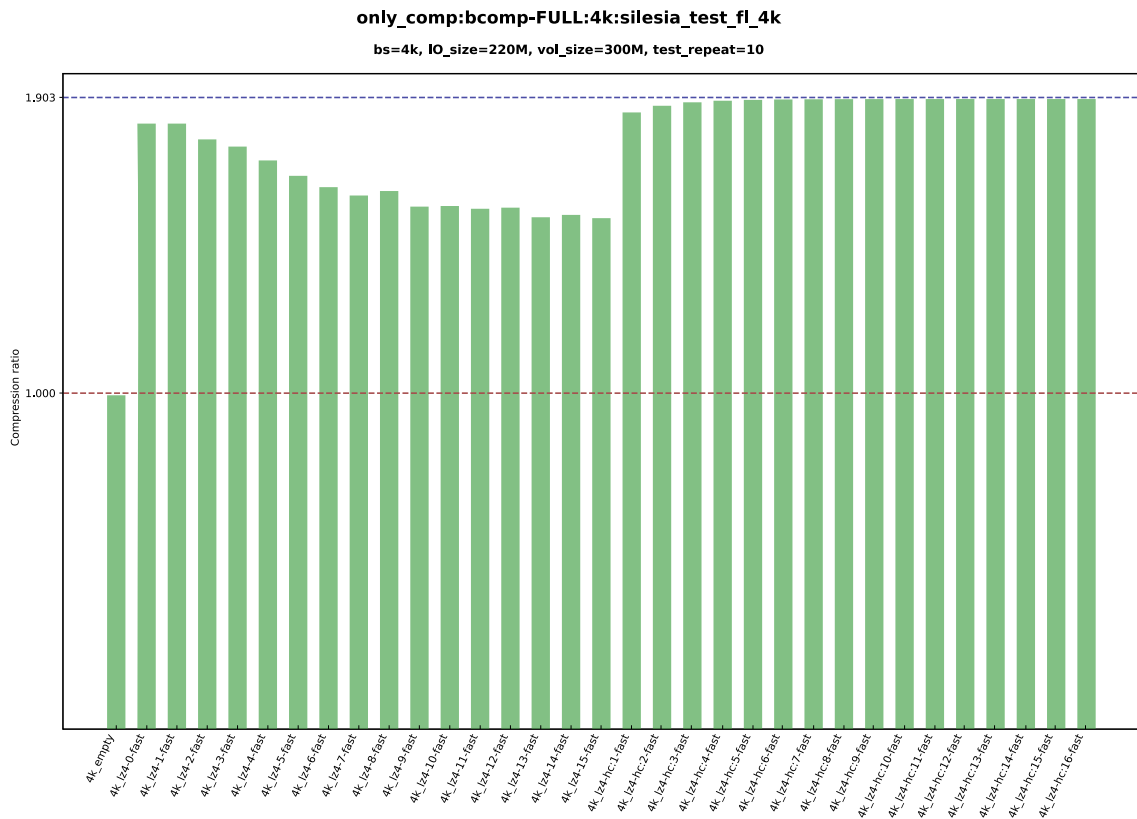


Рис. 13: График коэффициента сжатия для алгоритма lz4 с линейной адресацией.

Из графика можно сделать вывод, что наилучший коэффициент сжатия достигается при использовании НС алгоритма с максимальным коэффициентом акселерации, что в целом подтверждает информацию, описанную в документации LZ4. Кроме того стоит обратить внимание на таблицу 2, где представлены распределения сжатых блоков. Из нее видно, что наибольшее число блоков сжимаются примерно на 25 процентов. Кроме того, блоков, которые сжались менее чем в два раза значительно больше, чем блоков чей коэффициент сжатия больше двух.

Более подробно с результатами можно ознакомиться в тестовом репозитории [8].

	empty	lz4	lz4-hc
0% ≤ ... < 25%	0	2749	6197
25% ≤ ... < 50%	0	14591	13055
50% ≤ ... < 75%	0	22471	22591
75% ≤ ... < 100%	0	10271	9149
без сжатия	51742	1660	750

Таблица 2: Распределение сжатых блоков LZ4

Заключение

В ходе выполнения данной работы были достигнуты следующие результаты.

- Выполнен обзор алгоритмов компрессии без потерь, реализованных в ядре Linux, и существующих блочных устройств, использующих эти алгоритмы.
- Разработать архитектуру блочного устройства, позволяющего подключать алгоритмы компрессии LZ4 и ZSTD.
- Реализовать блочное устройство с компрессией на основе алгоритма LZ4 и линейной адресацией.
- Протестировать целостность реализованного блочного устройства.

Реализация и графики доступны в репозиториях [1] и [8].

В дальнейшем планируется реализовать неленейную адресацию, добавить поддержку алгоритма ZSTD, а также произвести сравнения прототипа с существующими открытыми решениями.

Список литературы

- [1] Bcomp sources. — 2024. — URL: <https://github.com/ItIsMrLaG/bio-comp-dev> (дата обращения: 2024-12-22).
- [2] FiniteStateEntropy. — 2024. — URL: <https://github.com/Cyan4973/FiniteStateEntropy> (дата обращения: 2024-12-22).
- [3] Fio documentation. — 2024. — URL: <https://fio.readthedocs.io/en/latest/index.html> (дата обращения: 2024-12-22).
- [4] LZ4 documentaion. — 2024. — URL: <https://github.com/lz4/lz4> (дата обращения: 2024-12-22).
- [5] The OpenZFS documentation, RAIDZ. — 2023. — URL: <https://openzfs.github.io/openzfs-docs/> (дата обращения: 2023-12-14).
- [6] OpenZFS sources. — 2024. — URL: <https://github.com/openzfs/zfs> (дата обращения: 2024-12-22).
- [7] Silesia dataset. — 2024. — URL: <http://sun.aei.polsl.pl/~sdeor/index.php?page=silesia> (дата обращения: 2024-12-22).
- [8] Utility for testing. — 2024. — URL: ... (дата обращения: 2024-12-22).
- [9] ZSTD documentaion. — 2024. — URL: <https://github.com/facebook/zstd> (дата обращения: 2024-12-22).
- [10] ZSTD documentaion. — 2024. — URL: <https://github.com/torvalds/linux> (дата обращения: 2024-12-22).
- [11] Ziv Jacob, Lempel Abraham. A universal algorithm for sequential data compression // IEEE Transactions on information theory. — 1977. — Vol. 23, no. 3. — P. 337–343.
- [12] Zstandard (RFC8878). — 2024. — URL: <https://datatracker.ietf.org/doc/html/rfc8878> (дата обращения: 2024-12-22).

- [13] dm-vdo documentation. — 2024. — URL: <https://docs.kernel.org/admin-guide/device-mapper/vdo.html> (дата обращения: 2024-12-22).