

Санкт-Петербургский государственный университет

Кафедра системного программирования

Группа 22.Б07-мм

Разработка блочного устройства реализующего дедупликацию в ядре Linux

Шалашнов Егор Дмитриевич

Отчёт по учебной 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
2. Обзор	6
2.1. Основные этапы дедупликации данных	6
2.2. Inline и offline дедупликация	10
2.3. Блочный уровень Linux kernel	10
2.4. Device mapper	12
2.5. Существующие решения	14
3. Решение	18
3.1. Логика дедупликации	19
4. Тестирование	21
4.1. Проверка корректности	21
4.2. Эксперимент	21
Заключение	24
Список литературы	25

Введение

В период с 2010 по 2020 годы общий объем данных/информации, созданных, полученных, скопированных и потребленных во всем мире, увеличился с 2 зеттабайт до 64.2 зеттабайт с прогнозируемым ростом вплоть до 181 зеттабайт к 2025 году [11]. Подобный экспоненциальный рост приводит к увеличению требований к системам хранения и обработки информации в плане оптимизации используемого дискового пространства. Исследования, проведенные компаниями Microsoft и EMC, показывают, что около 50 и 85 % данных в их системах хранения данных (СХД) соответственно являются избыточными [3]. В таких условиях, одним из решений проблемы оптимизации дискового пространства является дедупликация — процесс автоматического выявления дублирующихся блоков данных с последующей их заменой на ссылки на уникальные блоки.

Существует множество решений, предоставляющих возможность дедупликации: например, файловые системы OpenZFS [9] и ChunkFS [2] реализуют алгоритмы устранения дубликатов для блоков фиксированного и переменного размеров соответственно. Однако современные системы хранения данных (СХД), особенно в высоко нагруженных серверных средах, всё чаще базируются на ядре операционной системы Linux, где ключевой единицей обработки выступает блочное устройство. Это накладывает специфические требования — все манипуляции с данными происходят блоками строго равного размера, что создаёт необходимость интеграции дедупликации именно на этом уровне. Такой подход обеспечивает прозрачность для вышележащих слоёв инфраструктуры и позволяет достичь минимальных задержек критически важных для задач виртуализации, облачных сервисов, распределённых систем и прочих.

К сожалению, реализация блочного устройства с дедупликацией в пространстве ядра Linux сопряжено с множеством трудностей. Каждый этап алгоритма может быть реализован множеством различных способов, выбор которых затрудняется тем, что в отличие от пользова-

тельских решений, где допустимы компромиссы между производительностью и ресурсоемкостью, ядро требует строгой оптимизации алгоритмов под ограниченный контекст исполнения: отсутствие возможности выделения больших объёмов памяти, необходимость гарантировать атомарность операций при конкурентном доступе, минимизацию задержек даже при обработке миллионов блоков, необходимость сброса (и восстановления) метаданных на диск при отсутствии интерфейса файловой системы и многое другое. Влияют и требования, накладываемые на сами системы хранения данных — пользовательские данные должны храниться без потери содержимого, с осуществлением быстрого доступа.

Для исследования эффективности различных инженерных решений и трудностей, связанных с ними, появилась необходимость реализации блочного устройства с дедупликацией.

1 Постановка задачи

Целью данной работы является разработка прототипа блочного устройства, реализующего дедупликацию данных.

Для её выполнения были поставлены следующие задачи:

1.1 Осенний семестр

1. Провести обзор алгоритма дедупликации и существующих решений на базе ядра Linux.
2. Реализовать базовый поток управления алгоритма дедупликации в пространстве ядра.
3. Протестировать корректность полученного блочного устройства.

1.2 Весенний семестр

1. Разработать бэкенд метаданных, позволяющий осуществлять сброс на диск.
2. Произвести сравнение производительности с существующими решениями.

2 Обзор

Дедупликация данных — процесс исключения дублирующихся блоков данных с целью оптимизации используемого дискового пространства. Эта технология активно используется в системах резервного копирования, архивирования, первичных хранилищах данных, а также в оперативной памяти и SSD [10].

2.1 Основные этапы дедупликации данных

На рисунках 1 и 2 представлены упрощенные графы потоков управления запросов на чтение и запись соответственно.

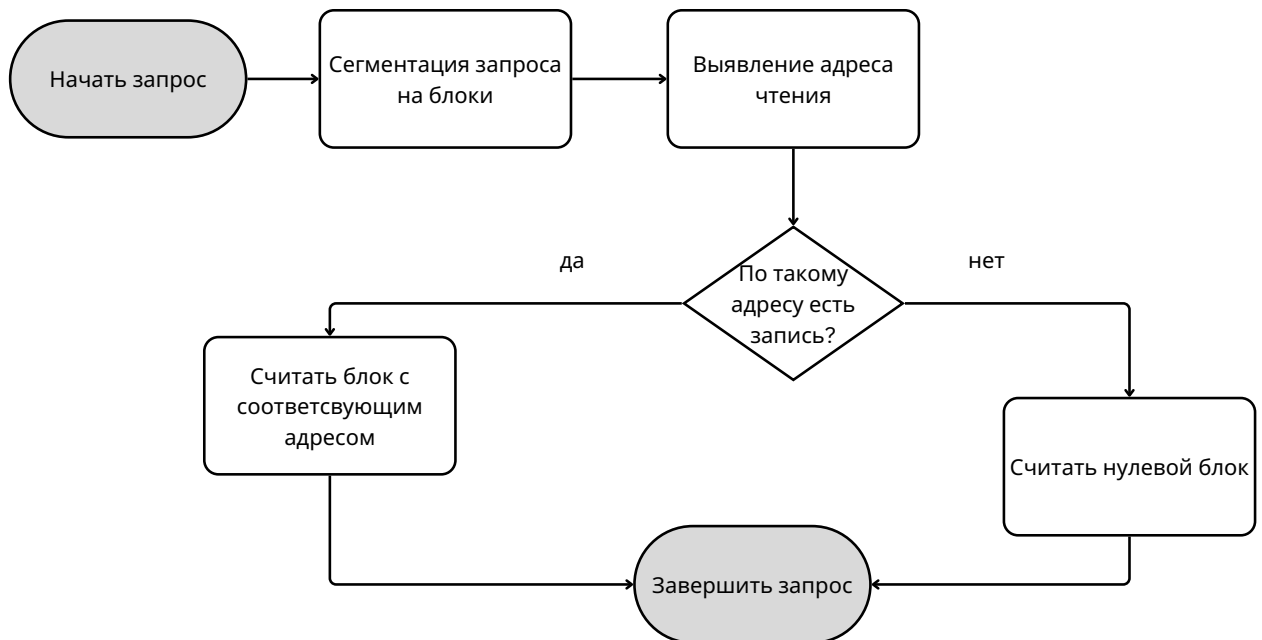


Рис. 1: Граф потока управления для запроса чтения

Запрос на чтение начинается с сегментации данных на блоки и поиска физических адресов блоков в списке уже записанных. Если адрес найден, система извлекает соответствующий блок из хранилища; в противном случае система считает что блок по такому адресу пуст. Для записи процесс усложняется: после сегментации данных вычисляется хэш записываемого блока, при помощи которого проверяется, является ли данный блок дубликатом. При обнаружении совпадения обновляется

счётчик ссылок и метаданные маппинга, в индексную структуру добавляется запись (адрес дубликата, адрес уникального блока). Если хэш отсутствует, блок сохраняется в хранилище, а в индексную структуру добавляется запись (хэш, адрес блока).

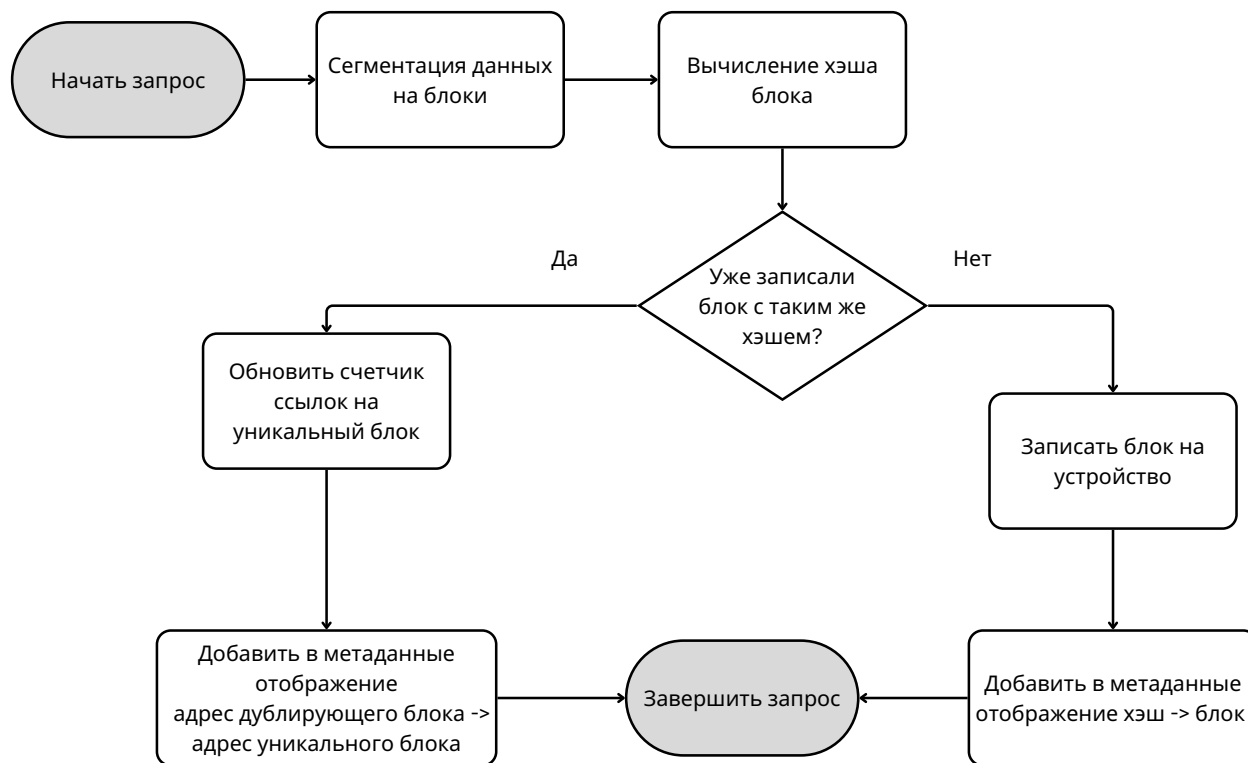


Рис. 2: Граф потока управления для запроса записи

2.1.1 Разбиение потока данных

Первоначальным этапом дедупликации является разбиение файла или потока данных на блоки, для каждого из которых формируется уникальный идентификатор (сигнатура, *fingerprint*). Наиболее простой метод — разбиение на блоки фиксированного размера (*Fixed-Size Chunking*, *FSC*). Однако данный подход подвержен проблеме сдвига границ блоков (*boundary-shift problem*): повторная запись набора данных с любой модификацией (вставкой, удалением) приводит к изменению границ блоков, что существенно снижает эффективность выявления дубликатов

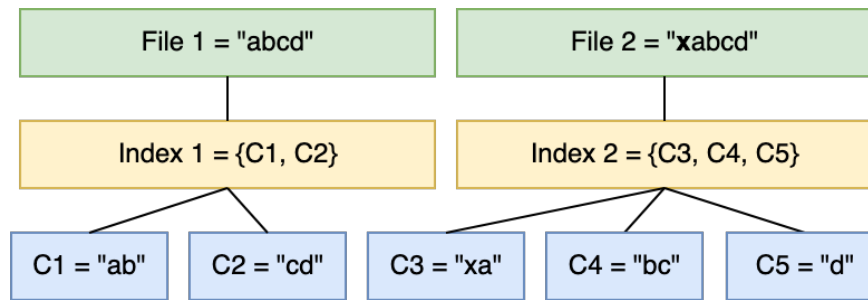


Рис. 3: boundary-shift problem

Для устранения этого недостатка были разработаны алгоритмы контентно ориентированного разбиения (Content-Defined Chunking, CDC), основанные на анализе содержимого блоков при помощи скользящего окна. CDC обеспечивает устойчивость границ блоков к локальным изменениям, сохраняя идентичность неизменённых сегментов, однако требует значительных вычислительных ресурсов, что ограничивает их применение в высокопроизводительных системах [3].

2.1.2 Вычисление уникальных идентификаторов блоков

Для идентификации дубликатов у всех блоков данных вычисляется значение некоторой (обычно криптографической) хэш-функции. Полученное значение сравнивается с уже сохранёнными хэшами, в случае нахождения совпадения производится побайтовое сравнение блоков. Однако некоторые системы, например LBFS, VENTI, DDFS, использующие криптографические хэш-функции семейства SHA, обходятся без побайтового сравнения, так как вероятность возникновения коллизии крайне мала (при обработке одного йотабайта данных и размере блока в 8 килобайт, вероятность составляет 10^{-8} у 160-битной SHA-1 и 10^{-144} у 512-битной SHA-512) [3].

2.1.3 Индексирование

Если рассматриваемый блок не является дубликатом, он записывается на диск по определенному адресу, в индексную структуру добавляется новая пара хэш — адрес блока. Также дедуплицирующие системы

должны поддерживать счетчики ссылок на записанные блоки, и отображение логический адрес — физический адрес для выделения свободных блоков на физическом устройстве и выполнения запросов на чтение.

При обработке больших объемов данных место, занимаемое этими структурами, достаточно быстро может превысить любые допустимые размеры оперативной памяти. Поэтому, при достижении определенного объема их содержимое сериализуется и сбрасывается на диск, что, в свою очередь, ставит вопрос о том, как эффективно организовать обращение к диску при поиске дубликатов, минимизируя замедление, вызванное большим временем доступа к диску. Снизить число обращений к диску помогают некоторые предположения о локальности данных.

Пространственная локальность (spatial locality) означает, что блоки, представленные в некотором файле (потоке данных), с высокой вероятностью будут встречаться в том же порядке в последующих файлах. Например, если за блоком A идут блоки B и C в первом потоке, то, как только блок A встретится в другом потоке, можно ожидать, что за ним последуют блоки B и C . Предположим, на диске хранятся (и поступили именно в таком порядке) отпечатки $\{3b, a7, 2f, 5c\}$ в структуре C , называемой главой, тогда, при необходимости обращения к отпечатку $3b$, мы можем выгрузить в оперативную память сразу все отпечатки главы C , и избежать лишних обращений к диску, как только поступят запросы на $a7$, $2f$ или $5c$.

Временная локальность (temporal locality) означает, что дублирующиеся блоки данных, как правило, появляются несколько раз в течение короткого временного интервала. Более конкретно, если блок данных A был записан, то с высокой вероятностью он будет записан снова несколько раз в ближайшем будущем. Временная локальность обычно используется путем реализации механизмов кэширования с политикой вытеснения Least Recently Used (LRU) [10].

Также для уменьшения числа обращений к диску можно использовать структуру данных, называемую фильтром Блума. Он позволяет за константное время проверить, принадлежит ли элемент множеству (возможны ложно-положительные ответы, но не ложно-отрицательные). Ес-

ли блок не нашелся в оперативной памяти, перед тем как обращаться к диску, система обращается к фильтру Блума, если окажется, что нужных нам данных там нет, можно сразу добавлять блок данных в индексную структуру.

2.2 Inline и offline дедупликация

Также стоит обратить внимание, что выявление и устранение дубликатов может производиться как до помещения данных в постоянное хранилище, так и после.

Inline дедупликация выполняется на критическом пути обработки запросов на запись. Inline дедупликацию можно встретить, например, в клиент-серверных решениях для уменьшения нагрузки на сеть: сначала клиент отправляет на сервер только сигнатуры блоков, затем сервер отвечает клиенту, указывая недостающие блоки, которых нет в его хранилище. Основным недостатком такого подхода является дополнительная задержка, которая вводится в процессе обработки запросов. В некоторых системах, этот накладной расход может быть неприемлем. Подавляющее большинство inline дедупликаторов применяет FSC для снижения накладных расходов.

Альтернативой inline дедупликации является offline дедупликация, при которой данные сначала записываются в хранилище, а затем сканируются в фоновом режиме для поиска и устранения дубликатов. Такой подход временно требует больше пространства и дополнительных ресурсов для сканирования хранилища, но не требует изменений на пути записи.

2.3 Блочный уровень Linux kernel

Каждая операция над блочным устройством задействует множество компонент ядра, самые важные из них показаны на рисунке 4.

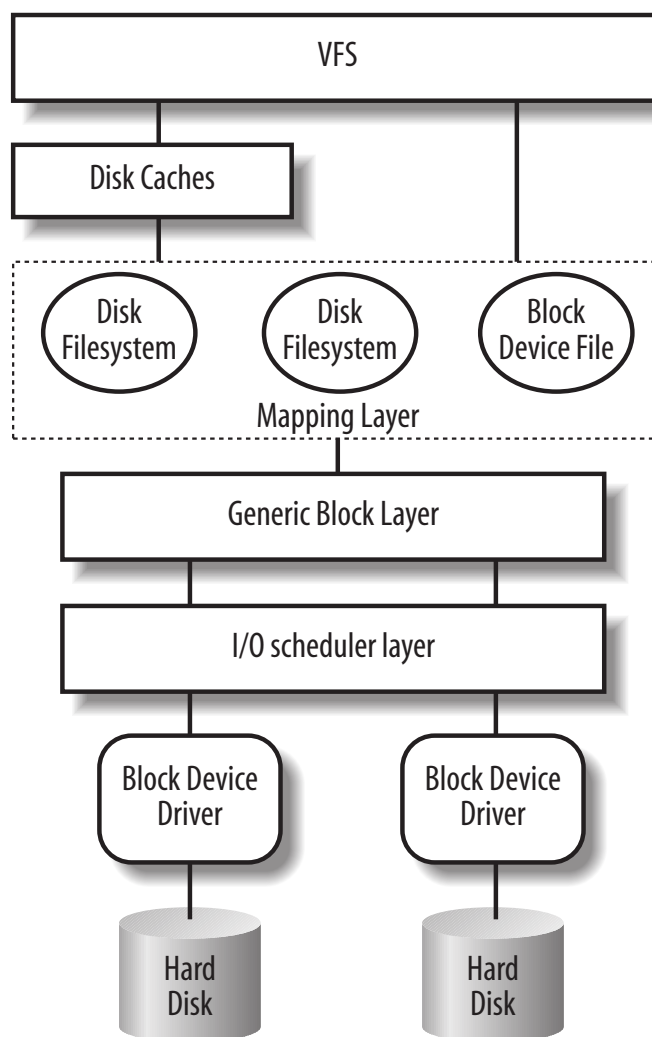


Рис. 4: Основные компоненты ядра задействованные в операциях над блочными устройствами. Источник: Understanding the linux kernel 3rd edition [1]

Для пояснения рассмотрим порядок обработки запроса чтения в ядре:

1. Системный вызов `read()` вызывает соответствующую функцию виртуальной файловой системы, передавая ей файловый дескриптор и смещение внутри файла.
2. Функция VFS анализирует доступны ли запрошенные данные в дисковом кэше. Часто обращение не требуется т.к. ядро сохраняет в оперативной памяти недавно прочитанные или записанные данные.

3. Слой отображения (mapping layer), определяет размер блока файловой системы и вычисляет расположение запрошенных данных в терминах логических номера блоков на диске.
4. Общий слой блочных устройства (generic block layer) формирует структуры блочного ввода-вывода (bio, block I/O).
5. Планировщик ввода-вывода (I/O scheduler), упорядочивает ожидающие запросы в соответствии с политиками ядра.
6. Драйверы блочных устройств выполняют непосредственную передачу данных, отправляя команды контроллерам дисков.

Основной структурой данных общего слоя блочных устройств является дескриптор текущей операции ввода-вывода блочного устройства — `struct bio` (англ. block I/O). Каждый объект `bio` включает в себя пользовательские данные, идентификатор области хранения на диске — начальный номер сектора и количество секторов, входящих в данную область, а также один или несколько сегментов, описывающих области памяти, задействованные в операции ввода-вывода.

2.4 Device mapper

Device mapper — фреймворк ядра Linux для создания виртуальных блочных устройств с логикой отображения секторов. Он позволяет строить сложные сценарии работы с данными (например шифрование данных "на лету" в `dm-crypt` или создание снапшотов в `dm-snapshot`), путем внедрения дополнительной логики в стандартный путь обработки `bio`, и перенаправления I/O запросов.

2.4.1 Обработка I/O запросов до поступления в модуль

Рассмотрим как происходит перенаправление запросов при помощи `device-mapper`. Когда приложение выполняет операцию ввода-вывода (например, запись в файл):

1. Файловая система формирует bio-запрос указывая логический сектор (LBA) для виртуального устройства
2. Блочный слой ядра передает bio в device-mapper
3. Device-mapper core:
 - Проверяет таблицу отображений (dm_table) связанную с виртуальным устройством
 - При необходимости разбивает BIO на части, например если запрос пересекает границы сегментов.
 - Для каждого результирующего bio:
 - Определяет целевой модуль (dm_target), ответственный за данный диапазон секторов
 - Вызывает функцию .map этого модуля

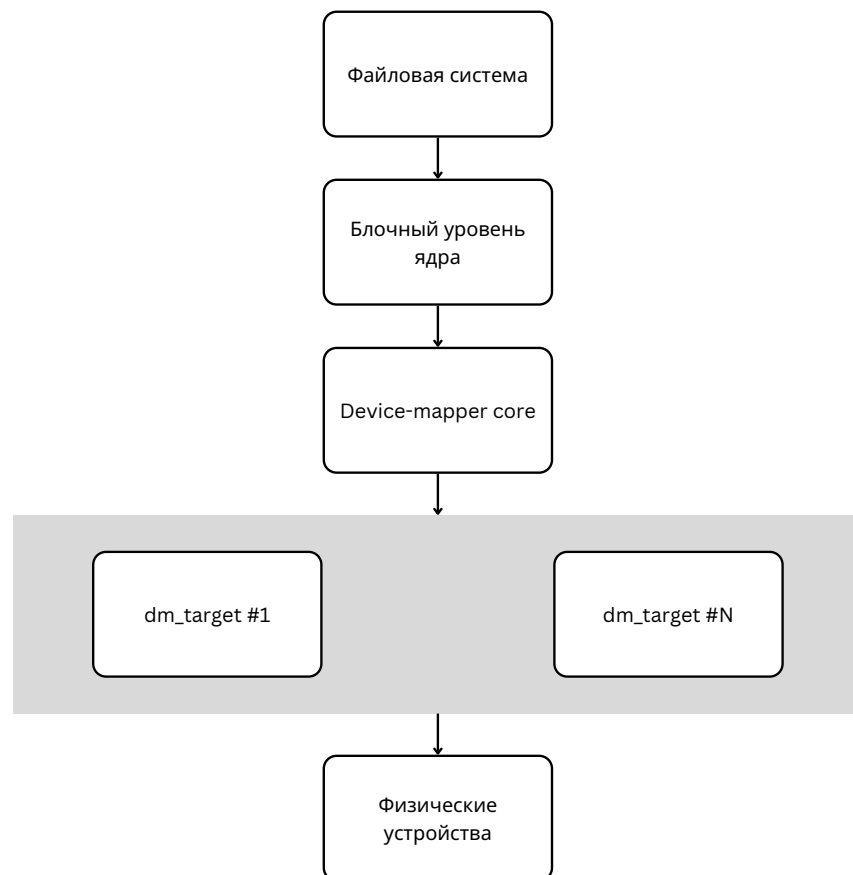


Рис. 5: Схема обработки запроса через device-mapper

2.4.2 Структура `target_type`

Структура `target_type` содержит указатели на callback-функции для управления жизненным циклом `dm-target`.

Основные поля:

- `.ctr` — указатель на конструктор таргета
- `.dtr` — указатель на деструктор таргета
- `.map` — указатель на функцию-обработчик `bio`, переданного из `dm-core`

2.5 Существующие решения

В контексте разработки драйвера блочного устройства, реализующего дедупликацию, в пространстве ядра Linux, нас интересуют системы `inline` дедупликации для первичного хранилища. В статьях [10] и [3] упоминается большое число (не менее 40) систем. Большинство из них направлены на дедупликацию в резервных хранилищах, ещё часть — в оперативной памяти. Около половины решений для первичного хранилища выполняет удаление дубликатов в `offline` режиме. В связи с этим, для дальнейшего анализа мы выделяем следующие системы: `OpenZFS`, `dmddedup`, `dm-vdo`.

2.5.1 OpenZFS

`OpenZFS` — это `open-source`¹ файловая система с поддержкой логических томов, снапшотов, компрессии и дедупликации. Она обеспечивает целостность данных и гибкость управления хранилищем, однако её реализация дедупликации долгое время считалась не рекомендуемой к использованию из-за высоких накладных расходов.

Дедупликация в `OpenZFS` использует таблицу `DDT` (`Deduplication Table`), основанную на `on-disk` хэш-таблице, называемой `ZAP`, и списке активных записей хранящемся в оперативной памяти (`Live entry list`).

¹<https://github.com/openzfs/zfs>

Запись в DDT содержит хеш (SHA-256), физическое расположение блока и счётчик ссылок. До версии 2.3.0 из-за особенностей реализации ZAP и live entry list, где даже для изменения одной записи требовалось перезаписывать целые блоки, возникали критические проблемы:

- Высокое потребление памяти: DDT требовала до 424 байт на каждую запись в оперативной памяти (Live Entry List), что быстро исчерпывало ресурсы;
- Фрагментация операций: каждая запись или удаление блока вызывало чтение и перезапись ZAP-блоков, увеличивая нагрузку на диск и создавая «бутылочное горлышко» в системе;

Начиная с версии 2.3.0 (14.01.2025) в OpenZFS включена функция fast dedup, решающая часть проблем старой реализации:

- Было введено журналирование операций (DDT Log), снижающее фрагментацию записей;
- Размер записей был снижен с 424 до 216 байт;

2.5.2 dmdedup

Dmdedup — open-source² исследовательский проект, предназначенный для сравнения производительности различных реализаций индексных структур в контексте дедупликации [7]. Система реализована как драйвер блочного устройства ядра Linux с использованием device-mapper интерфейса. Общий алгоритм работы совпадает с описанным в разделе 2.1. Пользователю на выбор предоставляется 2 индексные структуры: ingram хэш-таблица, и copy-on-write B+ дерево из библиотеки persisitnet-data. Для хэширования блоков данных можно использовать любую из функций представленных в библиотеке kernel crypto.

Dmdedup реализован для ядер Linux версии 3.19, 4.13, 4.8 и 4.19. Начиная с Ubuntu 20.04 версией ядра по умолчанию является 5.15³, что делает dmdedup неактуальным для современных систем.

²<https://github.com/dmdedup>

³<https://ubuntu.com/security/livepatch/docs/livepatch/reference/kernels>

2.5.3 dm-vdo

Начиная с версии 6.9, в ядро Linux включен драйвер блочного устройства dm-vdo (virtual data optimizer), реализующий дедупликацию, компрессию и тонкое выделение (thin provisioning) [6]. Dm-vdo поддерживает до 256 терабайт физического пространства, с логическим объёмом до 4 петабайт. Дедупликация достигает коэффициента 254:1 (каждый уникальный 4KB блок может быть представлен 254 ссылками).

Dm-vdo применяет индексную структуру, называемую UDS. блоки хэшируются с использованием 16 байтной murmurhash3 (криптографические функции не используются так как они вычисляются значительно дольше). Запись в индексной структуре состоит из имени блока и предполагаемого расположения данных в хранилище. Из-за возможности коллизий хэшей и того, что обновление индекса при перезаписи или удалении блока может быть слишком затратным, точность UDS не гарантируется. Поэтому vdo рассматривает записи в UDS лишь как подсказки и проверяет найденные блоки на идентичность побайтово.

Новые записи заносятся в структуру, называемую страницей (record page) в порядке их поступления. Страницы, в свою очередь, составляют так называемые главы (chapters). Новые записи добавляются в последнюю главу, называемую открытой (open chapter). Эта глава хранится в формате, оптимизированном для добавления и изменения записей, и ее содержание не финализируется, пока не исчерпается место для новых записей. Когда открытая глава заполняется, она "закрывается", и создается новая открытая глава для сбора записей.

Заккрытие главы преобразует ее в другой формат, оптимизированный для чтения. Преобразованная глава заносится в LRU кэш. При достижении определенного размера кэша, самая старая глава сбрасывается на диск. Каждый раз, когда по запросу хэша индексная структура возвращает запись не из open chapter, такая запись заносится в open chapter.

UDS применяет 2 хэш-таблицы с дельта-кодированными ключами (delta index) — chapter index и volume index. Volume index возвращает

главу в которой может лежать блок по его сигнатуре. Chapter index возвращает местоположение потенциального дубликата на диске.

В стандартной конфигурации UDS вмещает вплоть до 256 GB данных, то есть дубликат будет обнаружен только, если его предыдущее вхождение произошло в окне в 256 GB.

3 Решение

Текущая версия представляет собой модуль ядра Linux версии 6.10, реализующий inline дедупликацию блоков фиксированного размера, с использованием хэш-функции `xxhash64`, способный конфигурировать одно блочное устройство. Все метаданные хранятся в оперативной памяти, без возможности их сброса на диск и восстановления.

При помощи фреймворка `device-mapper` стандартный путь обработки блочного запроса ввода-вывода (`bio`) был дополнен этапами дедупликации. Для эффективного управления метаданными на базе красно-чёрного дерева реализованы три ключевые структуры:

- `pbn_manager` — отвечает за распределение свободных блоков на нижележащем устройстве хранения, обеспечивая выделение и освобождение физических адресов (PBN, Physical Block Number)
- `hash_pbn_memtable` — поддерживает отображение хешей блоков данных на их физическое расположение (хеш $\rightarrow$ PBN), что является основой для идентификации дубликатов,
- `lbn_pbn_memtable` — реализует связь между логическими адресами блоков (LBN, Logical Block Number) и физическими адресами на устройстве (LBN $\rightarrow$ PBN), обеспечивая доступ к дедуплицированным данным

Нынешняя реализация обладает двумя существенными ограничениями, которые в дальнейшем планируется устранить: во-первых, отсутствует возможность сброса метаданных на диск, что делает невозможным переиспользование устройства после перезапуска машины; во-вторых, отсутствует поддержка параллельной обработки запросов.

Далее будут описаны некоторые технические подробности реализации модуля `dedup`. С текущей реализацией можно ознакомиться в репозитории проекта [5].

3.1 Логика дедупликации

Пройдемся по шагам алгоритма дедупликации, описанного в разделе 2.1.

Сегментация потока данных производится до перехода в модуль, при помощи вызова функции `dm_set_target_max_io_len` в конструкторе, ограничивающей размер `bio`, поступающих в таргет, (в нынешней реализации это 4KB).

Функция `.map` (обработчик `bio` в `device mapper`), вызываемая `dm-core`, выполняется в атомарном контексте, то есть в ней запрещены операции, способные вызвать ожидание, прерывание или переключение контекста, в таком контексте код должен быть максимально быстрым. Так как логика дедупликации довольно ресурсоёмкая, обработка поступивших `bio` откладывается и переводится в контекст процесса при помощи `workqueue`.

На рисунках 6 и 7 представлены графы потоков управления запросов на чтение и запись соответственно.

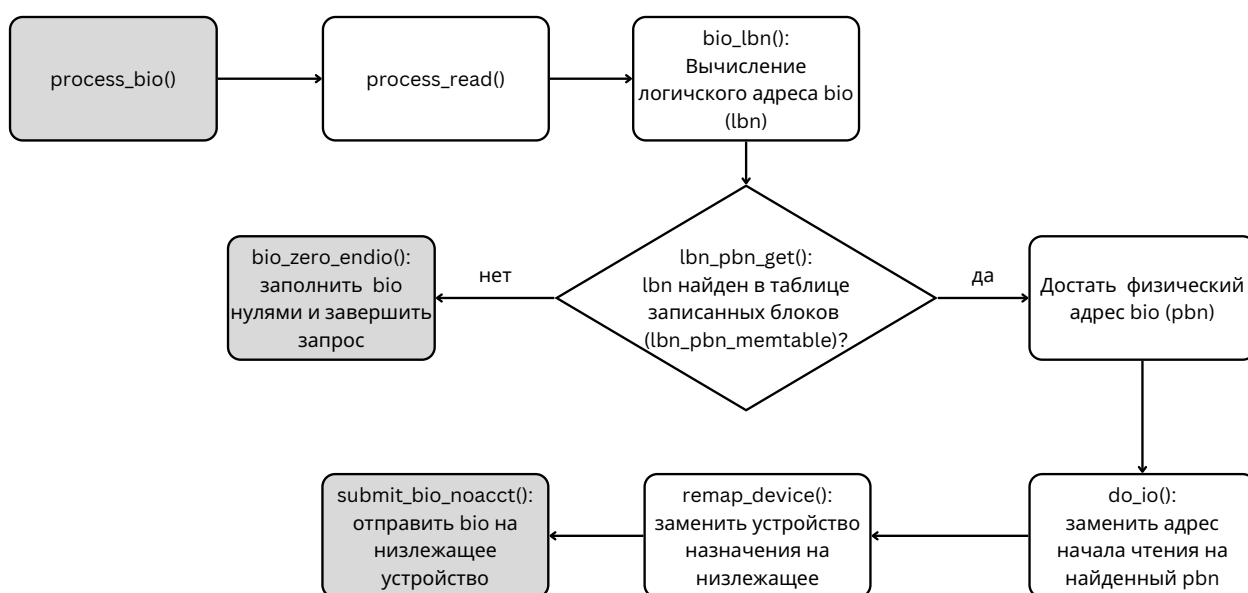


Рис. 6: Граф потока управления для запроса чтения

Вся логика дедупликации происходит до отправки запросов на нижележащее устройство, используется двухуровневую система адресации: логические адреса блоков (LBN) отображаются на физические (PBN) через таблицу `lbn_pbn_memtable`. При чтении определяется LBN, и ес-

ли он связан с PBN, данные считываются с физического блока, иначе возвращаются нули. Для записи вычисляется хеш данных, и если он совпадает с уже сохранённым, существующий PBN перепривязывается к текущему LBN, с обновлением счётчика ссылок `pbn_manager`. Для уникальных блоков выделяется отдельный PBN, а пара (хэш, PBN) сохраняется в `hash_pbn_memtable`. Менеджер физических блоков управляет выделением, освобождением и учётом ссылок на PBN.

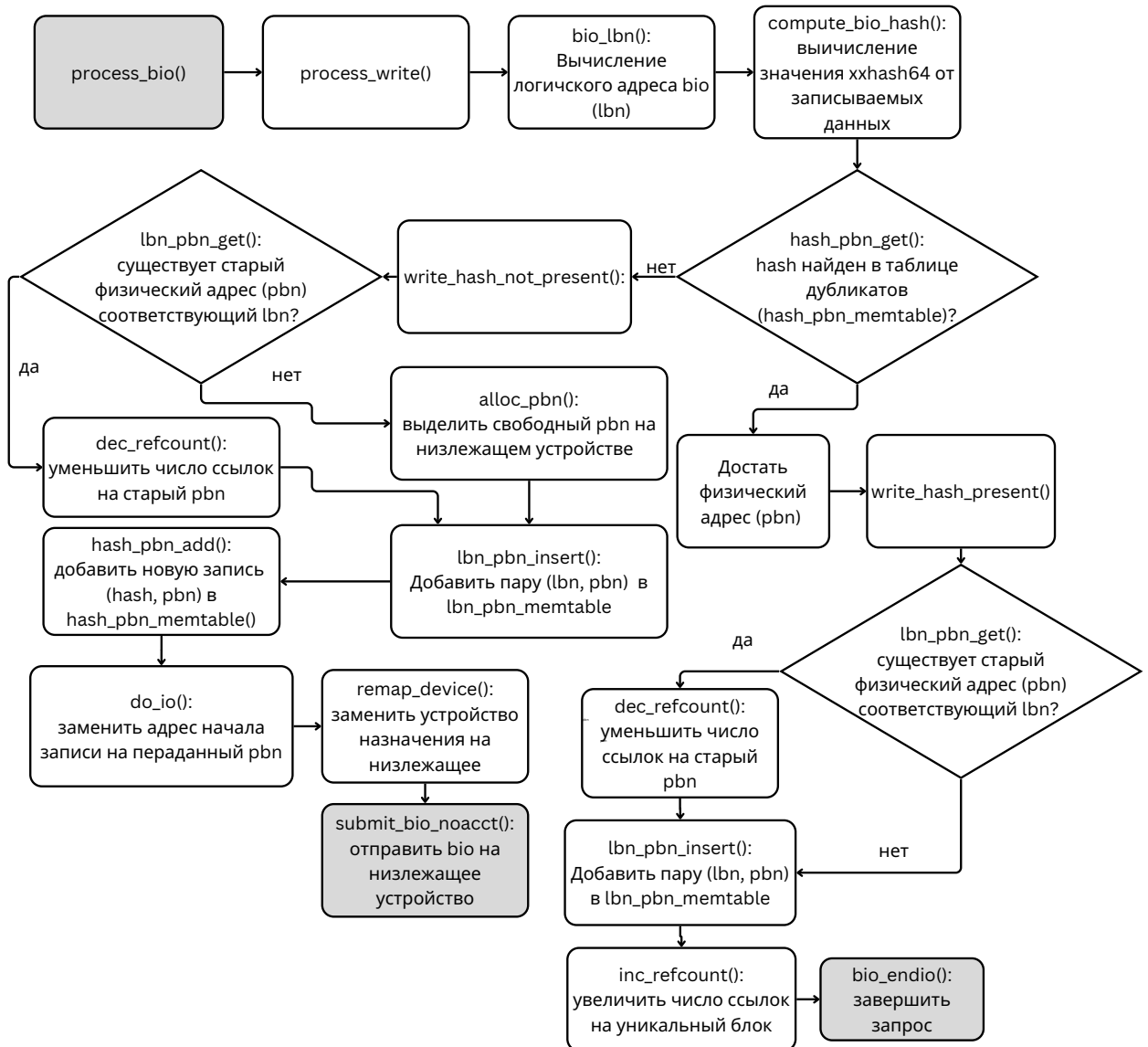


Рис. 7: Граф потока управления для запроса записи

4 Тестирование

Тестирование модуля состояло из двух этапов: проверка корректности запросов на запись и на чтение, а также сбор и анализ статистических данных о сэкономленном пространстве. Исходный код тестов доступен в репозитории проекта [5].

4.1 Проверка корректности

Проверка корректности представляла собой тестирование со следующим сценарием:

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

Для автоматизации тестирования применялись sh-скрипты и утилита fio. Fio (Flexible I/O Tester) — это утилита с открытым исходным кодом [8], предназначенная для тестирования производительности дисковых подсистем (HDD, SSD, NVMe), файловых систем и сетевых хранилищ. Она позволяет моделировать различные сценарии нагрузки, в том числе проверять корректность проводимых операций на синтетически сгенерированных данных.

4.2 Эксперимент

Тестирование представляло собой анализ алгоритма дедупликации интегрированного в блочное устройство. Для сбора статистики через sysfs был реализован доступ к переменным:

- `total_lbn` — число виртуальных блоков записанных на устройство
- `unique_pbn` — число физических блоков записанных на устройство

Sysfs — это виртуальная файловая система в Linux, предоставляющая иерархическое представление устройств, драйверов и параметров ядра через файлы и каталоги в /sys, позволяя взаимодействовать с ними из пользовательского пространства. Для записи использовалась утилита dd с параметром oflag=direct, позволяющая совершать запись пользовательских данных без использования файловой системы.

В качестве датасетов для тестирования были выбраны образы виртуальных машин различных дистрибутивов Linux⁴. На основе полученных значений были вычислены коэффициент дедупликации и общий объём сэкономленного пространства, по следующим формулам:

$$\text{Dedup ratio} = \frac{\text{total_lbn}}{\text{unique_pbn}},$$

$$\text{Saved Space} = (\text{total_lbn} - \text{unique_pbn}) * \text{block_size},$$

где block_size это размер блока данных (в нынешней реализации это 4096 байт). Результаты замеров представлены на графиках 8, 9 и таблице 1.

Таблица 1: Полученные результаты

Distribution	Total LBN	Unique PBN	Ratio
Ubuntu Server 24.04	794 000	651 529	1.22
CentOS 8.4	459 008	346 561	1.32
Debian 12.0	743 200	541 537	1.37
Fedora 39	1 129 904	1 014 439	1.11

Данные совпадают с полями logical_blocks_used и data_blocks_used полученными вызовом vdostats -verbose после записи в dm-vdo [4], что свидетельствует о корректности реализованного алгоритма дедупликации. Для более подробного изучения производительности и эффективности требуется дальнейшее исследование.

⁴<https://www.linuxvmimages.com/>

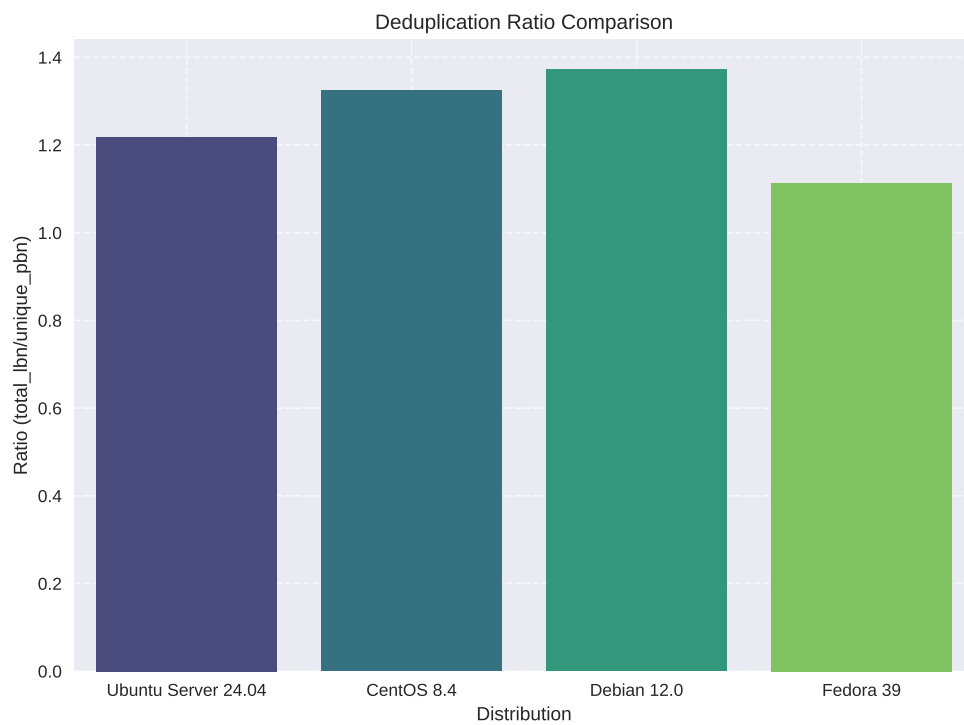


Рис. 8: Коэффициент дедупликации

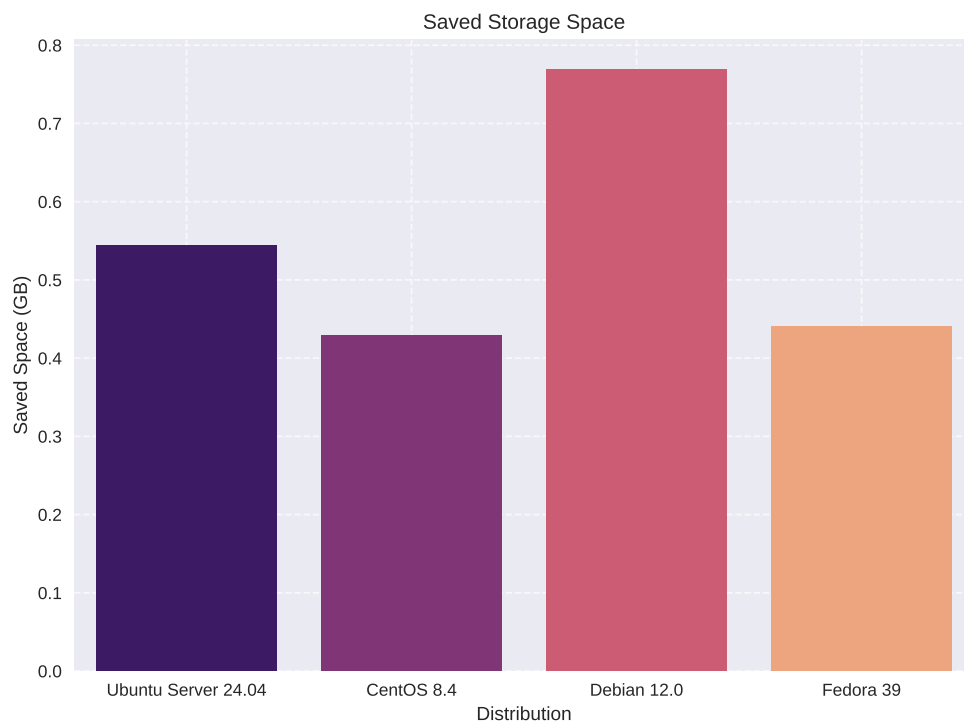


Рис. 9: Общее сэкономленное пространство

Заключение

В ходе выполнения данной работы были достигнуты следующие результаты:

- Выполнен обзор алгоритма дедупликации и существующих решений на базе ядра Linux.
- Реализован базовый поток управления алгоритма дедупликации в пространстве ядра.
- Протестирована корректность реализованного блочного устройства.

Реализация доступна в репозитории проекта [5]. В дальнейшем планируется реализовать сброс метаданных на диск, внедрить параллельную обработку запросов и произвести сравнение с существующими решениями.

Список литературы

- [1] Bovet Daniel, Cesati Marco. Understanding The Linux Kernel. — O'Reilly & Associates Inc, 2005. — ISBN: [0596005652](#).
- [2] ChunkFS github page. — URL: <https://github.com/Piletskii-Oleg/chunkfs> (дата обращения: 25 февраля 2025 г.).
- [3] A Comprehensive Study of the Past, Present, and Future of Data Deduplication / Wen Xia, Hong Jiang, Dan Feng et al. // [Proceedings of the IEEE](#). — 2016. — Vol. 104, no. 9. — P. 1681–1710.
- [4] Creating a deduplicated and compressed logical volume. — URL: https://docs.redhat.com/fr/documentation/red_hat_enterprise_linux/9/html/deduplicating_and_compressing_logical_volumes_on_rhel/creating-a-deduplicated-and-compressed-logical-volume_deduplicating-and-compressing-logical-volumes-on-rhel (дата обращения: 6 марта 2025 г.).
- [5] Dedup sources. — URL: <https://github.com/egshnov/dedup> (дата обращения: 3 марта 2025 г.).
- [6] Design of dm-vdo. — URL: <https://docs.kernel.org/admin-guide/device-mapper/vdo.html> (дата обращения: 29 ноября 2024 г.).
- [7] Dmddedup: Device Mapper Target for Data Deduplication / Vasily Tarasov, Deepak Jain, Geoff Kuenning et al. // Linux Symposium / Citeseer. — P. 83.
- [8] FIO documentation. — URL: <https://fio.readthedocs.io/en/latest/> (дата обращения: 4 марта 2025 г.).
- [9] OpenZFS documentation. — URL: https://openzfs.org/wiki/Main_Page (дата обращения: 29 ноября 2024 г.).

- [10] Paulo João, Pereira José. A Survey and Classification of Storage Deduplication Systems // [ACM Comput. Surv.](#) — 2014. — June. — Vol. 47, no. 1. — 30 p. — URL: <https://doi.org/10.1145/2611778>.
- [11] Taylor Petroc. Volume of data/information created, captured, copied, and consumed worldwide from 2010 to 2020, with forecasts from 2021 to 2025. — URL: <https://www.statista.com/statistics/871513/worldwide-data-created/> (дата обращения: 12 ноября 2024 г.).