Правительство Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего профессионального образования
«Санкт-Петербургский государственный университет»

Кафедра системного программирования

Демьяненко Илья Игоревич

Разработка модуля обнаружения последовательных запросов в системах хранения данных с блочным доступом

Дипломная работа

Допущена к защите.
Зав. кафедрой:
д. ф.-м. н., профессор Терехов А. Н.

Научный руководитель:
ст. преп. Луцив Д. В.

Рецензент:
Руководитель исследовательской лаборатории ООО "Рэйдикс", к. т. н. Лазарева С. В.

Санкт-Петербург
2015

SAINT-PETERSBURG STATE UNIVERSITY

Department of Software Engineering

Ilia Demianenko

Implementation of sequential access pattern detection module for block-level data storage

Graduation Thesis

Admitted for defence.
Head of the chair:
professor Andrey Terekhov

Scientific supervisor:
sen. lect. Dmitry Luciv

Reviewer:
Head of RAIDIX Research Lab, Ph. D. Svetlana Lazareva

Saint-Petersburg
2015

Оглавление

Введение **4**

1. Постановка задачи **6**

2. Обзор существующих решений **7**

2.1. C-Miner 7

2.2. Table-based Prefetching (TaP) 8

2.3. Алгоритм, используемый в ПО RAIDIX 9

2.4. Последовательности в мультимедиа 10

3. Предлагаемый алгоритм **12**

3.1. Используемые структуры данных 12

3.1.1. Запрос 12

3.1.2. Последовательность 12

3.1.3. Детектор последовательностей 14

3.2. Процесс обработки запроса 15

3.2.1. Поиск подходящих последовательностей 15

3.2.2. Добавление запроса в последовательность 15

3.2.3. Удаление запроса из последовательности 17

3.2.4. Создание новой последовательности 17

3.2.5. Асимптотическая сложность 18

4. Реализация **20**

4.1. Управление памятью 20

4.2. Структуры данных 20

4.3. Арифметика с фиксированной точкой 21

5. Апробация **22**

5.1. Функциональное тестирование 22

5.1.1. Последовательная нагрузка 23

5.1.2. Случайная нагрузка 23

5.1.3. Характерная для мультимедиа нагрузка 23

5.2. Тест пропускной способности 24

5.3. Нагрузочное тестирование 25

5.4. Выводы 26

Заключение **27**

Список литературы **28**

Введение

Объём данных, генерируемых человечеством изо дня в день, растёт экспоненциально. С одной стороны, сами данные увеличиваются в размерах: появляются новые форматы видео со всё большим разрешением, в суперкомпьютерных вычислениях обрабатываются растущие массивы данных, увеличивается число пользователей, запрашивающих эти данные. С другой стороны, накопители данных совершенствуются: растут в объёме жёсткие диски, приобретают распространение твёрдотельные накопители с улучшенным временем доступа и увеличенными скоростными характеристиками. Появляются новые интерфейсы для подключения накопителей, предоставляющие сниженные задержки и повышенную скорость работы.

Производительность системы хранения данных (СХД) характеризуют два основных параметра: время доступа и полоса пропускания. Оба они могут быть улучшены не только путём обновления аппаратной части, а ещё и усовершенствованием программной части. Одной из главных оптимизаций, призванных сократить время доступа и повысить пропускную способность, является применение технологии упреждающего чтения (read ahead). Она заключается в том, чтобы на основании уже запрошенных данных предугадывать, какие данные будут запрошены следующими, и переносить их с более медленных носителей информации, например, с жёстких дисков, на более быстрые, такие как оперативная память и твёрдотельные накопители, до того, как к этим данным обратятся (рис. 1). В большинстве случаев упреждающее чтение применяется при операциях последовательного чтения.

Особенную ценность упреждающее чтение представляет в системах хранения, используемых для работы с мультимедиа, а именно, в сервисах отдачи потокового видео множеству клиентов, а также при съёмке и монтаже видео высокого разрешения. Эти отрасли предъявляют высокие требования к производительности СХД именно при последовательном чтении одновременно с множества рабочих станций.

Работу алгоритма упреждающего чтения принято разделять на два этапа: обнару-

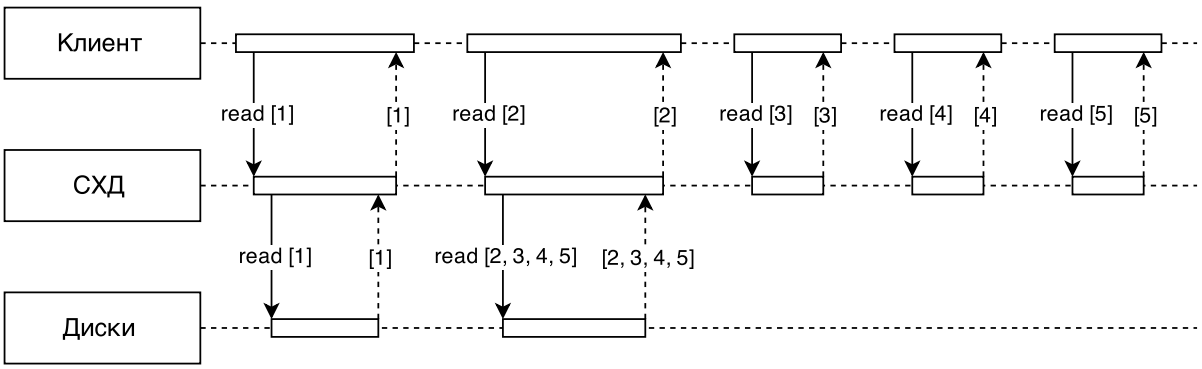


Рис. 1: Упреждающее чтение

жение запросов последовательного чтения в потоке всех запросов и принятие решения о том, нужно ли считывать данные наперёд, и в каком объёме. В данной работе пойдёт речь преимущественно о первом этапе.

Последовательное чтение, как правило, производится на одном файле или группе связанных между собой файлов. Однако, система хранения данных с блочным доступом не обладает сведениями о развёрнутых на ней файловых системах, и информации о запросах, которую можно использовать для упреждающего чтения, в ней имеется гораздо меньше, чем в СХД с файловым доступом. В блочных СХД известно время прихода каждого запроса, его адрес, а также размер запрашиваемых данных. Имя или адрес клиентского компьютера, от которого пришёл запрос, также известны, но отделять запросы друг от друга на их основе не рекомендуется, т.к. запросы, пришедшие от разных клиентов, на самом деле могут быть частями одной операции на распределённой файловой системе, входить в один блок данных, над которыми проводятся вычисления на кластере, или вообще исходить от одного компьютера, подключенного к СХД несколькими путями.

Обнаружение последовательных запросов может производиться как обработкой каждого запроса по отдельности, так и ведением более подробной истории, рассматривающей сразу группы запросов и объединяющей их в последовательности. Первый способ более легковесен и требует минимальных вычислительных ресурсов, но при этом не делает отличий между разными последовательностями, и соответственно, не может предоставить модулю упреждающего чтения подробной информации о связанных между собой операциях. Второй же способ более ресурсоёмок и может потребовать использования сложных структур данных, но имеет больше сведений о взаимосвязи запросов и может предоставлять расширенную информацию о последовательностях, например, её скорость, плотность, возраст, а также суммарный охват запрошенных адресов.

1. Постановка задачи

Целью данной работы является усовершенствование алгоритма упреждающего чтения, используемого в ПО RAIDIX, для лучшей поддержки паттернов, свойственных программам нелинейного видеомонтажа. Для осуществления этой цели были выделены следующие подзадачи:

- Провести обзор существующих алгоритмов обнаружения последовательных запросов.
- Разработать собственный алгоритм выделения последовательных операций среди запросов ввода-вывода блочного уровня.
- Реализовать модуль в виде набора функций на языке C, работающих в пространстве ядра, встроить реализованное решение в ПО RAIDIX.
- Провести функциональное и нагрузочное тестирование реализованного модуля и сравнить результаты с существующим алгоритмом.

2. Обзор существующих решений

Согласно классическому определению, при последовательных операциях адреса обращений монотонно возрастают, а разность между адресами соседних запросов равна размеру первого из них. Более сложные определения, используемые для нахождения последовательностей в реальных нагрузках, дополнительно рассматривают расширенный набор характеристик, включающий в себя:

- размеры запросов;
- время между поступлением соседних запросов (interarrival time);
- повторные чтения по адресам, запрошенным ранее;
- пропуск адресов (strided access).

Работа [1] содержит подробные описания различных метрик, которые можно использовать для обнаружения последовательных запросов.

2.1. C-Miner

Одним из наиболее интересных алгоритмов, используемых для упреждающего чтения, является C-Miner [3], основанный на поиске корреляций между адресами последовательно запрашиваемых блоков данных. Этот алгоритм обнаруживает семантические связи между запросами, такими как запрос метаданных файловой системы перед чтением соответствующих им файлов или поиск записи в индексе СУБД. Работа C-Miner состоит из следующих этапов:

1. В последовательности адресов запросов ввода-вывода производится обнаружение повторяющихся коротких подпоследовательностей, ограниченных окном некоторого размера.
2. Из полученных подпоследовательностей извлекаются суффиксы, которые затем комбинируются в соответствии с частотой встречаемости.
3. На основе извлечённых суффиксов и цепочек суффиксов генерируются правила вида $\{a \to b, a \to c, b \to c, ab \to c\}$, описывающие, запросы к каким блокам данных с наибольшей вероятностью предваряются запросами по адресам из коррелирующей области.
4. Полученные правила ранжируются в соответствии с их надёжностью.

Алгоритм C-Miner хорошо подходит для систем хранения, данные в которых редко обновляются, но запрашиваются часто и небольшими порциями, что позволяет

составить набор правил, который не устареет долгое время, а также не замедлит выполнение первых запросов, на момент прихода которых правила ещё отсутствуют. Недостатки этого подхода проявляются, когда данные живут недолго, а запрашиваются сразу большими участками, например, при многоступенчатой обработке видео высокого разрешения.

2.2. Table-based Prefetching (TaP)

Следующий алгоритм, представляющий интерес, называется TaP [8] и основан на применении хэш-таблицы для обнаружения операций потокового чтения. Он состоит из следующих шагов:

1. Для каждого поступающего запроса на чтение проверяется, есть ли запрашиваемые данные в кэше.

    1.1. Если данные содержатся в кэше, алгоритм проверяет, есть ли у них метка $prefetchTrigger$, и если она установлена, то производится упреждающее чтение с адреса поступившего запроса на расстояние $prefetchDegree$. Новая метка $prefetchTrigger$ выставляется одному из последних прочтённых адресов. Адреса блоков данных, вытесенных при этом из кэша, заносятся в хэш-таблицу.

    1.2. Если данные в кэше отсутствуют, проверяется наличие в хэш-таблице диапазона адресов от текущего запроса на расстояние $strideRange$.

        1.2.1. Если хотя бы один из адресов там присутствует, производится упреждающее чтение на расстояние $prefetchDegree$.

        1.2.2. Если искомые адреса отсутствуют в таблице, туда добавляется адрес, следующий за запрошенным, чтобы при обращении к нему активировать упреждающее чтение.

Таким образом, хэш-таблица представляет собой метаданные кэша второго уровня и содержит адреса, по которым в дальнейшем ожидается обращение. Вытеснение её элементов происходит по методу FIFO.

Алгоритм TaP хорошо подходит для обнаружения операций последовательного чтения, но имеет ряд недостатков:

- Его вычислительная сложность на каждый запрос линейно зависит от коэффициентов $prefetchDegree$ и $strideRange$. В сочетании со случайным доступом к памяти, свойственным хэш-таблицам, это нанесёт ущерб производительности при высокой нагрузке.

- Для активации упреждающего чтения достаточно двух запросов к последовательным адресам, и этот параметр не регулируется. Такая чувствительность

может оказаться критичной для производительности при наличии даже небольшой доли случайных запросов.

- Алгоритм предполагает, что все запрашиваемые блоки имеют одинаковый размер. В реальных СХД это не так, поэтому для адаптации придётся:
  - либо принять за размер блока длину сектора диска, что приведёт к многократному увеличению коэффициентов и росту вычислительной сложности алгоритма;
  - либо задать блокам достаточно большой размер, в этом случае возникнут сложности в работе с кэшом, потому что запрошенные блоки сразу вытесняются алгоритмом, а большой блок будет запрашиваться несколько раз.
- TaP не содержит сущности "последовательность", из-за чего параметры упреждающего чтения одинаковы для всех запросов.

2.3. Алгоритм, используемый в ПО RAIDIX

Программное обеспечение RAIDIX [6], для которого проведено данное исследование, использует свой алгоритм упреждающего чтения. Он основан на понятии диапазонов, соответствующих связным интервалам адресного пространства. Диапазоны обозначаются парами (lba_i, len_i), что соотетствует адресу и длине запроса, и отсортированы по lba. Они разделяются на случайные, длина которых меньше некоторого порога, и последовательные. Одновременно отслеживается до 128 случайных диапазонов и до 64 последовательных. Алгоритм работает следующим образом:

1. Для каждого приходящего запроса на чтение производится поиск ближайших диапазонов в радиусе $strideRange$.
   1.1. Если таковых не нашлось, создаётся новый случайный диапазон с адресом и длиной, соответствущим характеристикам запроса. При этом один из существующих диапазонов может быть вытеснен по LRU.
   1.2. Если нашёлся один диапазон, запрос добавляется в него с расширением интервала. Случайный диапазон при этом может быть преобразован в последовательный.
   1.3. Если нашлось два диапазона (слева и справа), они объединяются в один. Получившийся в результате этой операции диапазон также может быть преобразован в последовательный.
2. В случае, когда запрос оказался в последовательном диапазоне, для этого диапазона может быть произведено упреждающее чтение.

Алгоритм, использованный в RAIDIX, достаточно легковесен и хорошо работает для простых операций последовательного чтения, поддерживая при этом пропуски адресов размером до *strideRange*. Однако, он также имеет и недостатки:

- Количество отслеживаемых последовательностей жёстко ограничено. При его превышении будут постоянно создаваться новые диапазоны и удаляться старые. Увеличение соответствующих констант повлечёт за собой линейный рост вычислительной сложности алгоритма.

- Он не позволяет узнать, сколько последовательностей активны в данный момент. Если за всё время работы СХД было обнаружено 64 последовательности, они будут храниться до возникновения новых.

- Для каждой отслеживаемой последовательности известен только адрес последнего поступившего в неё запроса. Этого достаточно только для базовых алгоритмов упреждающего чтения.

- Константа *strideRange* не зависит от характеристик последовательностей. С одной стороны, она может оказаться слишком маленькой и спровоцировать "потерю" высокоскоростной последовательности при слишком большом пропуске. Если же она окажется слишком большой, то станет возможным сконструировать нагрузку, генерирующую значительный объём упреждающего чтения при небольшом количестве реально запрашиваемых клиентом данных.

2.4. Последовательности в мультимедиа

СХД под управлением ПО RAIDIX ориентированы на рынок мультимедиа, который устанавливает высокие требования к производительности, при этом нагрузка от видеоприложений имеет свои особенности.

Обработка видео является интенсивной как с точки зрения скорости ввода-вывода, так и с точки зрения объёмов вычислений, чему способствует появление новых форматов видео. Например, 1 секунда несжатого видео с разрешением 4K (4096×3112) и глубиной цвета 10 бит занимает 1.4 гигабайта дискового пространства, что является серьёзным объёмом данных для обработки. Чтобы полностью задействовать мощности рабочих станций, программы нелинейного видеомонтажа используют многопоточные вычисления, ввод-вывод при этом тоже становится параллельным. Это приводит к ситуации, когда каждый кадр видео обрабатывается отдельно в своём потоке, при этом несколько последовательных кадров считываются с СХД одновременно. Адреса запросов возрастают от кадра к кадру и в совокупности покрывают почти весь интервал адресного пространства, но из-за многопоточности ввода-вывода перемешиваются и не попадают под определение последовательных (рис. 2).

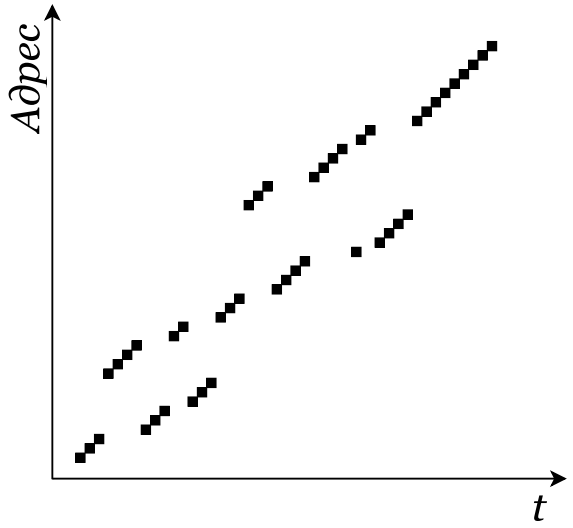


Рис. 2: Переупорядочивание запросов от мультимедийных приложений

Также некоторые приложения для видеомонтажа хранят каждый кадр видео в отдельном файле, из-за чего даже при последовательном хранении файловая система выравнивает начало каждого файла до некоторого ”круглого” значения, из-за чего между соседними кадрами возникают пропуски в адресном пространстве.

Эти свойства нагрузки, специфичной для мультимедиа-приложений, создают трудности алгоритмам обнаружения последовательных запросов, из-за чего существует необходимость в алгоритме, рассчитаном на работу с такими приложениями.

3. Предлагаемый алгоритм

3.1. Используемые структуры данных

Основной идеей предлагаемого алгоритма является приближение последовательностей линейной функцией $lba(t)$ на основе истории поступивших запросов за последние $timeout$ секунд. Для этого используются следующие сущности: $request$ – поступивший запрос, $sequence$ – последовательность запросов, $seqDetector$ – детектор последовательностей, главная сущность, которая обрабатывает все входящие запросы. Рассмотрим их подробнее.

3.1.1. Запрос

Запрос характеризуется следующими величинами:

- id – уникальный идентификатор запроса;
- t – время прихода;
- lba – адрес запрашиваемых данных;
- len – длина запрашиваемых данных.

3.1.2. Последовательность

Рассмотрим набор запросов, пришедших за последние $timeout$ секунд и относящихся к одной последовательности, упорядочив их по lba (рис. 3). Их можно разделить на три группы:

- Новые запросы, между которыми есть пропуски из-за того, что запросы с бóльшими адресами поступили раньше запросов с меньшими адресами.
- Устаревающие запросы, среди которых также есть пропуски, потому что запросы с бóльшими адресами устарели быстрее.
- Актуальные запросы, совокупность которых представляет собой интервал адресов с покрытием $minSeqDensity$. Покрытие определяется как отношение суммы длин запросов к разности между концом крайнего правого и началом крайнего левого запросов.

Основную часть информации дают актуальные запросы, поэтому для каждой последовательности поддерживается указатель $denseFirst$, указывающий на первый запрос плотного интервала, и $denseLast$, указывающий на последний запрос плотного

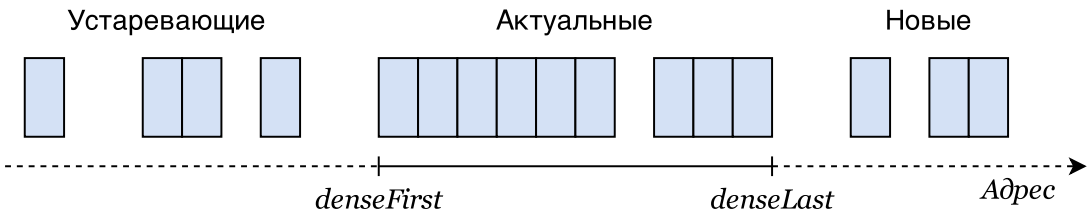


Рис. 3: Запросы в последовательности

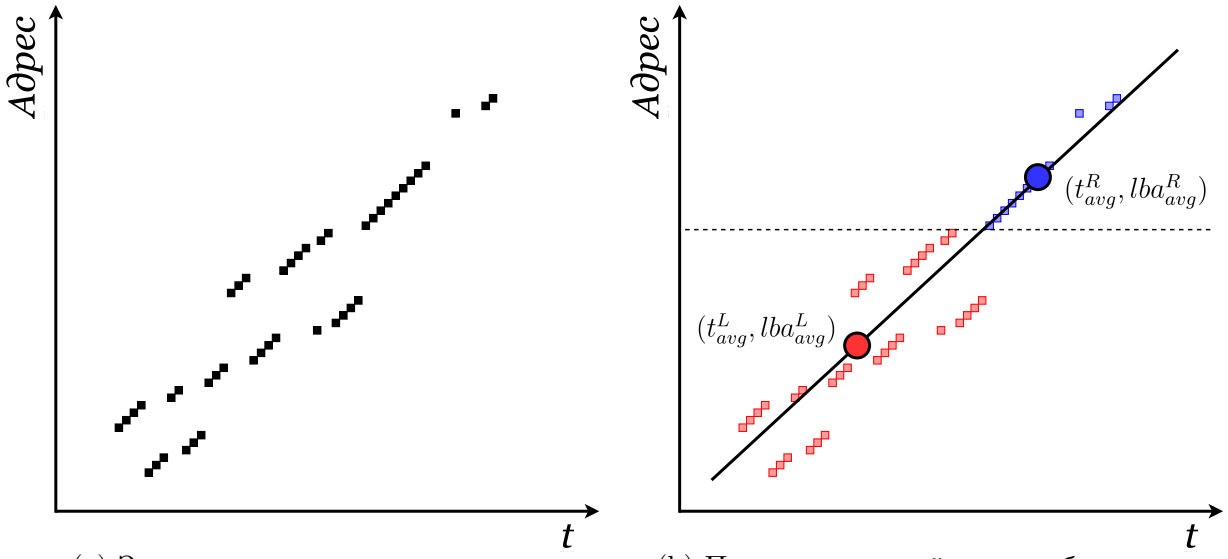
интервала. Также ведётся учёт величин $firstDenseLbaEver$ и $lastDenseLbaEver$, соответствующих наименьшему и наибольшему адресу среди запросов, когда-либо входивших в плотный интервал. Эти значения позволяют оценить "размах" последовательности. Наконец, среди хранимых в данный момент запросов вычисляется медианный адрес $median$.

Для хранения запросов в последовательности используется структура $requestsTree$, которая является augmented multimap и основана на сбалансированном двоичном дереве поиска. Запросы в ней упорядочены по адресам. Поскольку по одному адресу за интервал времени может прийти несколько запросов, то отображение из адресов в запросы не является инъективным, из-за чего и возникает необходимость в использовании multimap. То, что она является дополненной (augmented), позволяет оптимальным образом подсчитывать значения функций на группах запросов. В каждой вершине дерева хранятся результаты следующих функций от этой вершины и всех её потомков:

- $sumLen$ – суммарная длина запросов;
- $minLba$ – наименьший адрес запроса;
- $maxEnd$ – наибольший адрес конца запроса;
- $avgLba$ – средний адрес запроса;
- $avgTimestamp$ – среднее время прихода запроса.

Предподсчёт значений $sumLen$, $minLba$, $maxEnd$ позволяет вычислять плотность произвольного интервала запросов за $O(\log N)$, где N – количество запросов, хранимых в последовательности.

Описание поведения последовательности осуществляется с помощью приближения зависимости запрашиваемого адреса от времени линейной функцией $lba(t) = slope * t + offset$. Наклон функции $slope$ соответствует скорости рассматриваемой последовательности. Традиционные методы аппроксимации множества точек линейной функцией сложны вычислительно, поэтому был выбран другой, менее точный, но достаточно робастный (robust) метод. У вершины дерева запросов для левого и правого



(a) Запросы в последовательности (b) Построение линейного приближения

Рис. 4: Приближение последовательности линейной функцией

поддеревьев подсчитываются среднее время поступления запроса и средний адрес запроса. Результатом являются две точки (t^L_{avg}, lba^L_{avg}) и (t^R_{avg}, lba^R_{avg}), через которые проводится прямая (рис. 4). Расчёт средних значений не требует значительного количества вычислительных ресурсов благодаря обновлению значений соответствующих функций "на лету" и производится амортизированно за $O(1)$ операций при добавлении и удалении вершин из дерева. Стоит заметить, что использование такого подхода подразумевает наличие у корня дерева обеих дочерних вершин.

3.1.3. Детектор последовательностей

Детектор последовательностей отвечает за хранение величин, необходимых для работы алгоритма в целом. Он содержит:

- *chronoRequests* – связный список всех запросов, пришедших за последние *timeout* секунд и упорядоченных по времени поступления.
- *chronoSequences* – связный список отслеживаемых в данный момент последовательностей, упорядоченных по времени последнего обновления.
- *randRequestsTree* – multimap запросов, не отнесённых к какой-либо из отслеживаемых последовательностей. Для вершин подсчитываются те же функции, что и в последовательностях.
- *sequencesTree* – multimap последовательностей. Ключом является медианный адрес хранимых в последовательности запросов.

3.2. Процесс обработки запроса

Поступающие запросы обрабатываются в соответствии со следующим алгоритмом:

1. Запросы, возраст которых превышает $timeout$, удаляются.
2. Если после удаления старых запросов находятся последовательности размером менее $minRequestsInSeq$ запросов, эти последовательности также удаляются.
3. Запрос добавляется в список $chronoRequests$.
4. Производится поиск последовательностей, которым может соответствовать пришедший запрос.
5. Для каждой из последовательностей производится попытка добавления в неё запроса.
   - 5.1. Если запрос успешно добавлен в последовательность, производится попытка добавления в неё всех случайных запросов, адрес которых находится между lba нового запроса и $median$ последовательности. Обработка запроса завершается.
   - 5.2. Если запрос не соответствует условию добавления в каждую из последовательностей, он добавляется в $randRequestsTree$.
     - 5.2.1. Если из нового запроса и смежных с ним возможно составить последовательность, она создаётся и добавляется в $sequencesTree$ и список $chronoSequences$. Запросы удаляются из $randRequestsTree$.

3.2.1. Поиск подходящих последовательностей

Количество последовательностей, отслеживаемых алгоритмом, может быть достаточно большим, и большинство из них заведомо не могут быть продолжены поступившим запросом. Для первичного поиска подходящих последовательностей используется следующая эвристика: близкие по адресу последовательности могут быть продолжены запросом с большей вероятностью, чем дальние, поэтому будем рассматривать $seqSearchArea$ последовательностей, $median$ которых ближе всего к адресу запроса. В $sequencesTree$ обнаруживается $\lceil seqSearchArea/2 \rceil$ последовательностей с $median$, меньшим, либо равным lba запроса, и $\lfloor seqSearchArea/2 \rfloor$ последовательностей с большим $median$.

3.2.2. Добавление запроса в последовательность

На определение принадлежности нового запроса последовательности влияют различные факторы.

С одной стороны, установление этого факта упрощается наличием линейной функции, описывающей поведение последовательности. Вводится ограничение на степень переупорядочивания запросов в ходе обработки, а именно, предполагается, что пришедший запрос либо входит в её плотный интервал, либо отстоит от ближайшего конца плотного интервала не более, чем на $predictionWindow$ секунд:

$$lba_{denseFirst} \leq lba_i \leq lba(t_{denseLast} + predictionWindow)$$

$lba(t)$ здесь означает линейное приближение последовательности. Приведённое условие справедливо для восходящих последовательностей, но может быть модифицировано и для нисходящих. Для последовательностей, наклон которых отрицателен, оно будет выглядеть так:

$$lba(t_{denseFirst} + predictionWindow) \leq lba_i \leq lba_{denseLast}$$

С другой стороны, запросы обычно поступают не равномерно, а всплесками, причём время между приходом запросов будет небольшим (порядка единиц микросекунд) внутри группы, и более существенным между разными группами. Если последовательность состоит только из одной группы, то сильно заниженная разница во времени между приходом запросов приведёт к сильно завышенному углу наклона линейной функции.

Для иллюстрации возьмём поток, состоящий из запросов размером 1 МБ и имеющий среднюю скорость в 1 ГБ/с. Запросы будут поступать группами по 16 штук с интервалом 10 мкс между соседними запросами внутри группы. Последовательностью, включающей в себя только одну группу, будет прочитано 16 МБ за 150 мкс, и при наивном подсчёте её скорость (мгновенная) составит $\frac{16 \cdot 10^{-3}}{150 \cdot 10^{-6}} = 106{,}(6)$ ГБ/с, что на два порядка превышает реальную скорость.

Во избежание ложных срабатываний на новых последовательностях вводится ограничение на расстояние до них:

$$maxDistance = (lastDenseLbaEver - firstDenseLbaEver) \cdot minSizeMultiplier$$

С помощью $maxDistance$ вводится второе условие:

$$lba_{denseFirst} \leq lba_i \leq lba_{denseLast} + maxDistance$$

для восходящих последовательностей, и, соответственно,

$$lba_{denseFirst} - maxDistance \leq lba_i \leq lba_{denseLast}$$

для нисходящих.

Запрос, удовлетворяющий обоим условиям, включается в последовательность. По-

сле его добавления производится попытка расширения интервала между $denseFirst$ и $denseLast$ на случай, если новый запрос закрывает собой пропуск и увеличивает плотный интервал.

3.2.3. Удаление запроса из последовательности

При удалении запроса производятся следующие действия:

1. Запрос удаляется из $requestsTree$.
2. Определяется положение удаляемого запроса по отношению к плотному интервалу.
   - 2.1. Запрос находился снаружи интервала. Обработка заканчивается.
   - 2.2. Запрос принадлежит интервалу, и после его удаления плотность интервала превышает $minSeqDensity$. Обработка заканчивается.
   - 2.3. Запрос находился внутри интервала, и после его удаления плотность стала ниже $minSeqDensity$. Интервал делится на две части по месту удаления запроса, плотным объявляется больший из них. Если интервал содержит меньше, чем $minRequestsInSeq$ запросов, последовательность удаляется.
3. В конце обработки производится пересчёт изменившихся значений.

3.2.4. Создание новой последовательности

Попытка создания новой последовательности, включающей в себя новый запрос, происходит следующим образом:

1. Если $randRequestsTree$ содержит менее, чем $minRequestsInSeq$ запросов, создание последовательности невозможно.
2. Создаются указатели $first$ и $last$, указывающие на новый запрос. Они считаются соответственно началом и концом плотного интервала последовательности. Изначальная плотность интервала равна единице.
3. Определяется, какой из двух граничащих с интервалом запросов может быть включён в интервал так, чтобы результирующая плотность не опустилась ниже $minSeqDensity$. Если под это условие подходят оба запроса, включается тот, плотность интервала с которым больше. Операция повторяется, пока в интервале не окажется $minRequestsInSeq$ запросов.
   - 3.1. Если удалось найти достаточно запросов, из них создаётся новая последовательность.

3.2. Если на одной из итераций добавление запроса в интервал влечёт за собой чрезмерное снижение плотности, последовательность не создаётся.

3.2.5. Асимптотическая сложность

Количество операций, выполняемых для каждого запроса, меняется в зависимости от характера нагрузки, что не позволяет провести точную оценку сложности каждого этапа алгоритма. Тем не менее, среднее число операций на каждый запрос поддаётся оценке, поэтому имеет смысл амортизированная асимптотическая сложность.

Трудоёмкость вычислений зависит от текущего размера структур данных, хранящих запросы. Предсказать его заранее нельзя, но можно оценить сверху как $O(timeout \cdot iops)$, где $iops$ (input/output operations per second) – предполагаемая нагрузка, измеренная в запросах в секунду. Поскольку $timeout$ является константой, то оценка превращается в $O(iops)$. Этой величиной оценивается число запросов, находящихся в каждой последовательности, число отслеживаемых последовательностей, а также число хранимых в данный момент времени случайных запросов.

Оценим сложность каждого этапа алгоритма:

1. Каждый запрос удаляется единожды за время своего существования. Выбор самого старого запроса из связного списка производится за $O(1)$.

   1.1. Если удалённый запрос был случайным, удаление его из $randRequestsTree$ требует $O(\log iops)$, согласно приведённой выше оценке.

   1.2. Если же он входил в последовательность, при удалении его оттуда затрачивается $O(\log iops)$ операций на удаление из дерева, $O(1)$ операций на обновление указателей и пересчёт прямой и $O(\log iops)$ операций на пересчёт медианы.

2. Удаление запроса может спровоцировать удаление не более, чем одной последовательности, что также производится за $O(\log iops)$ операций.

3. Добавление запроса в связный список занимает $O(1)$ операций.

4. Поиск подходящих последовательностей включает в себя нахождение ближайшей последовательности в дереве и составление списка последовательностей, что оценивается в $O(\log iops + seqSearchArea)$ операций. $seqSearchArea$ – константа, поэтому оценка сокращается до $O(\log iops)$.

5. Добавление запроса в последовательность состоит из нескольких действий:

   5.1. Проверка соответствия запроса линейному приближению, а также размаху последовательности включает в себя $O(1)$ операций.

5.2. Каждый запрос может быть включён в плотный интервал один раз за время своего существования, поэтому обновление $denseFirst$ и $denseLast$ происходится за $O(1)$.

5.3. Обновление медианы производится за $O(\log iops)$ операций.

6. Если запрос не удалось добавить ни в одну последовательность, добавление его в $randRequestsTree$ занимает $O(\log iops)$ операций.

7. Попытка создания новой последовательности занимает $O(minRequestsInSeq)$ операций, что эквивалентно $O(1)$.

Можно заметить, что сложность каждого этапа алгоритма не превышает $O(\log iops)$ операций, а поскольку этапов константное число, то итоговая сложность также составляет $O(\log iops)$ операций.

4. Реализация

Предложенный алгоритм был реализован на языке C в виде набора функций пространства ядра (kernel space) Linux.

4.1. Управление памятью

При программировании высоконагруженных систем в пространстве ядра важна предсказуемость времени работы. При написании программ, работающих в пространстве пользователя (user space), а также при использовании языков высокого уровня обычно используется динамическая аллокация памяти. Динамические аллокаторы не дают гарантий на время, затрачиваемое при выделении памяти, поэтому во встроенных системах используется статическая аллокация.

Реализация предлагаемого алгоритма использует пулы памяти для хранения запросов и последовательностей. Максимальное количество iops, которое может обрабатывать аппаратная конфигурация, известно заранее, из чего можно рассчитать необходимый объём пула запросов $requestsPoolSize$. Число одновременно отслеживаемых последовательностей известно на этапе выработки требований к системе, из чего следует ограничение $sequencesPoolSize$. Если при добавлении запроса в одном из пулов нет свободного места, производится вытеснение элементов по стратегии LRU. Использование пулов исключает задержки, связанные с динамической аллокацией памяти, и предоставляет гарантии по её потреблению.

Обычно структуры данных, предоставляемые стандартными библиотеками языков программирования, подразумевают динамическое выделение вспомогательных структур. Например, для хранения структур $Request$ в связном списке требуется вспомогательная структура $ListNode$, состоящая из указателей на соседние $ListNode$ и указателя на $Request$. Этот подход плохо совместим со статическим выделением памяти.

При разработке в пространстве ядра Linux со статическим выделением памяти используются интрузивные (intrusive) структуры данных, вспомогательная информация для которых хранится непосредственно в пользовательских структурах. Для приведённого выше примера это означает, что $ListNode$ будет одним из полей структуры $Request$. Таким образом, пропадает необходимость в отдельном выделении памяти для вспомогательных структур, и упрощается управление памятью в целом.

4.2. Структуры данных

Ядро Linux предоставляет базовый набор структур данных. В качестве связного списка была использована структура list. Готового решения, предоставляещего интерфейс augmented multimap, в ядре нет, но присутствует реализация двоичного красно-чёрного дерева, которая и была принята за основу. Реализация модуля производится

под версию ядра 2.6.32 (ПО RAIDIX основано именно на ней), в то время как реализация дополненных красно-чёрных деревьев появилась начиная с версии 2.6.33, из-за чего пришлось портировать структуру данных с более актуальной версии.

Красно-чёрное дерево изначально предназначается для хранения уникальных элементов, что противоречит идее multimap. Для решения этой проблемы на запросах вводится два отношения порядка. Строгое отношение сравнивает запросы сначала по адресу, затем по уникальному идентификатору и используется для расположения запросов внутри дерева. Слабое же отношение использует только адреса запросов, вводя классы эквивалентности.

4.3. Арифметика с фиксированной точкой

Предложенный алгоритм оперирует не только целочисленными значениями, но и дробными, такими как плотность интервала, средний адрес запроса в поддереве, скорость последовательности. Обычно для операций над вещественными значениями используется арифметика с плавающей точкой, но её поддержка присутствует не во всех процессорных архитектурах, из-за чего использование вещественной арифметики в пространстве ядра ограничено [5]. ПО RAIDIX разрабатывается преимущественно под архитектуру x86-64, но не исключено будущее использование его на других платформах, поэтому код модулей ядра должен быть кроссплатформенным.

Для работы с дробными числами была реализована арифметика с фиксированной точкой. Числа в ней имеют размер 64 бита, 32 из которых используются под целую часть и 32 – под дробную. Основные требования к точности, обеспечиваемой дробной частью, следуют из вычисления среднего адреса запросов поддерева. В них используется адрес корня поддерева и средние адреса в его левом и правом поддеревьях, взвешенные в соответствии с размером этих поддеревьев:

$$lba_i^{avg} = \frac{count_L}{count_L + count_R + 1} \cdot lba_L^{avg} + \frac{count_R}{count_L + count_R + 1} \cdot lba_R^{avg} + \frac{1}{count_L + count_R + 1} \cdot lba_i$$

При этом минимально возможный вес слагаемого составляет $(timeout \cdot iops)^{-1}$, для чего хватит 32 бит даже при высоких нагрузках.

32 бита целой части позволяют представлять адреса до 2 ТБ (2^{41} байт) при размере секторов в 512 байт, чего недостаточно для покрытия всего адресного пространства СХД. Однако, средние значения подсчитываются среди запросов одной последовательности, о которых заранее известно, что их адреса близки друг к другу. Поэтому каждой последовательности назначается базовый адрес, равный адресу первого запроса последовательности, а среднее считается среди смещений относительно базового адреса. Для покрытия адресов внутри одной последовательности достаточно 32 бит адресного пространства.

5. Апробация

Было проведено тестирование реализованного модуля и сравнение его с существующим алгоритмом выделения последовательностей, который используется в ПО RAIDIX. Далее существующий алгоритм будет обозначаться как *range*, а предлагаемый – как *detector*.

Были выбраны следующие параметры алгоритма:

- $requestsPoolSize = 1\,000\,000$ (запросов);
- $sequencesPoolSize = 1\,000$ (последовательностей);
- $timeout = 10$ (секунд);
- $predictionWindow = 10$ (секунд);
- $minSizeMultiplier = 5$;
- $minSeqDensity = 0.9$;
- $minRequestsInSeq = 40$ (запросов);
- $seqSearchArea = 7$ (последовательностей).

5.1. Функциональное тестирование

Для оценки качества обнаружения последовательных операций алгоритмом был выбран набор метрик:

- α – доля ошибок первого рода, когда случайный запрос был классифицирован как последовательный.
- β – доля ошибок второго рода, когда последовательный запрос был принят за случайный.
- ARI (adjusted Rand index) [4] – оценка качества разделения набора запросов на последовательности. Набору запросов, сформированных тестирующей программой, были сопоставлены метки последовательностей, сначала тестируемым алгоритмом, затем человеком. Если запрос был распознан алгоритмом как случайный, ему сопоставляется уникальная метка, которая больше нигде не встречается. Таким образом были получены два разбиения одного и того же множества запросов: тестируемое и эталонное. Метрика ARI показывает степень близости двух разбиений. Её значение для двух случайных разбиений близко к нулю, полному совпадению соответствует единица.

5.1.1. Последовательная нагрузка

Цель первого теста состояла в том, чтобы удостовериться в отсутствии регрессии по сравнению с существующим алгоритмом при простой нагрузке. Для этого была выбрана утилита `fio` (Flexible I/O tester) [10]. С её помощью были сгенерированы 4 последовательных потока с непересекающимися адресами, состоящие из запросов по 1 МБ. Запросы внутри каждого потока никак не перемешивались, их адреса были расположены по возрастанию. Поскольку нагрузка была полностью последовательной, тестирование на ней позволяет получить показатели β и ARI, но не α. Получившаяся выборка насчитывает 240 000 запросов.

Алгоритм	β	ARI
range	22,3%	0,70
detector	0,2%	0,98

Таблица 1: Последовательная нагрузка `fio`

В таблице 1 отражены результаты тестирования. Заметно не только отсутствие регрессии по сравнению с существующим алгоритмом, но и существенно лучшие показатели как по числу ложноотрицательных срабатываний, так и по качеству отделения последовательностей друг от друга.

5.1.2. Случайная нагрузка

Второй тест проводился с целью выявления ложноположительных срабатываний α. С помощью `fio` было сгенерировано 1 000 000 случайных запросов размером 4 КБ с адресами, равномерно распределёнными по всему объёму тома (1 ТБ).

Алгоритм	α
range	0,0%
detector	0,0%

Таблица 2: Случайная нагрузка `fio`

Как показывает таблица 2, оба алгоритма не допустили ни одного ложноположительного срабатывания, а значит, по этому параметру регрессии тоже не произошло.

5.1.3. Характерная для мультимедиа нагрузка

Сравнение качества обнаружения последовательностей на нагрузке, характерной для видеоприложений, было проведено на бенчмарке `sw_io_perf_tool` от компании Autodesk [2]. Эта утилита примечательна тем, что генерирует нагрузку с помощью тех же средств, что и программы для монтажа видео от этой компании, такие, как Flame и Smoke. Поскольку для функционирования утилиты требуется файловая система, на СХД была развёрнута XFS [9].

В данном тесте эмулировалось чтение видео разрешением 4096×3112 и глубиной цвета 10 бит в 8 потоков. Всего было сформировано 364 000 запросов. Такая нагрузка является полностью последовательной, из-за чего, как и в первом тесте, представляется возможным получить только показатели β и ARI.

Алгоритм	β	ARI
range	53,9%	0,05
detector	0,5%	0,99

Таблица 3: Последовательная нагрузка `sw_io_perf_tool`

Результаты тестирования приведены в таблице 3. Алгоритм *range* классифицировал более половины запросов как случайные, в то время как предлагаемый алгоритм ошибся только в 0,5% случаев. Разделение последовательностей, согласно метрике ARI, похоже на случайное для *range* и близко к эталонному для нового алгоритма.

5.2. Тест пропускной способности

Для оценки влияния, которое оказывает на пропускную способность СХД замена алгоритма обнаружения последовательностей, было проведено тестирование с использованием утилиты `frametest` [7]. Эта утилита создана компанией SGI и предназначена для тестирования производительности СХД при потоковой отдаче видео.

В ходе теста одновременно запускалось 1, 4 или 12 экземпляров `frametest`, эмулируя работу с СХД нескольких клиентов. Каждый процесс производил чтение 500 кадров видео разрешением 4K из отдельной директории. Обработка набора кадров каждым процессом производилась в 1, 2, 3 или 4 потока, количество потоков здесь характеризует степень переупорядочивания запросов. Аппаратная конфигурация включала в себя процессор Intel Xeon E5620 и 12 ГБ оперативной памяти. Чтение производилось с массива RAID-6 с 16 дисками и 6 ГБ кэш-памяти, который находился под управлением ПО RAIDIX 4.3.

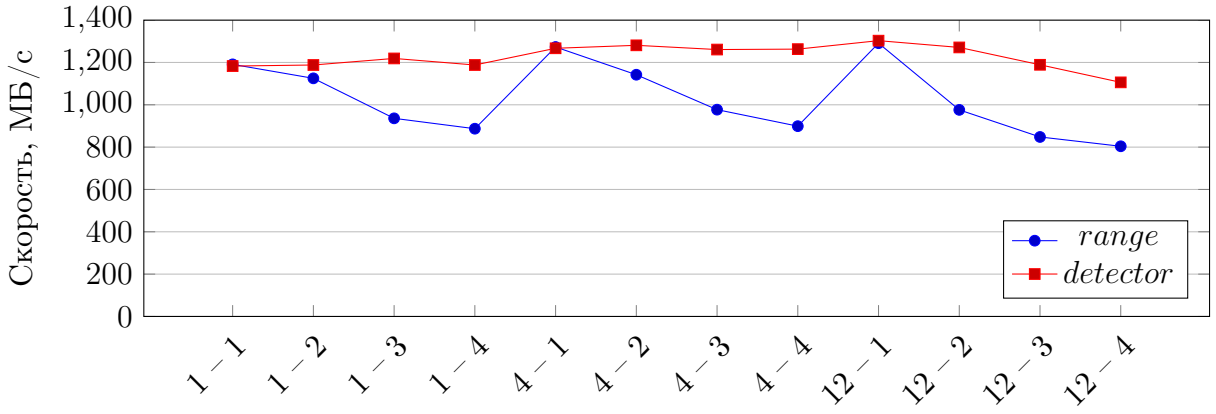


Рис. 5: Многопоточное тестирование утилитой `frametest`

На рисунке 5 изображена зависимость пропускной способности СХД в зависимости от числа запущенных экземпляров `frametest` и количества потоков обработки. Оба алгоритма одинаково хорошо справляются с любым количеством процессов при маленькой степени переупорядочивания запросов. С повышением же числа потоков *detector* начинает выигрывать у *range*, причём разница в скорости достигает 40%.

5.3. Нагрузочное тестирование

При создании нового алгоритма важно иметь представление о границах его применимости. Проблема, возникающая при нагрузочном тестировании, заключается в том, что общая производительность дисков ограничена, в связи с чем нет возможности создать нагрузку большими блоками с произвольной интенсивностью. Однако, поскольку оба алгоритма оперируют метаданными запросов, размер данных при этом не имеет значения. Это позволяет проводить тест блоками меньшего размера, снижая такиим образом нагрузку на диски.

Тестирование проводилось с использованием `fio`. Утилита формировала последовательность запросов одинакового размера с максимальной скоростью, которую позволяла СХД. Размер запросов варьировался между 4 КБ и 1 МБ. В качестве аппаратного обеспечения использовалась та же система, что и в предыдущем тесте, а RAID-массив состоял из 17 дисков с 6 ГБ кэш-памяти.

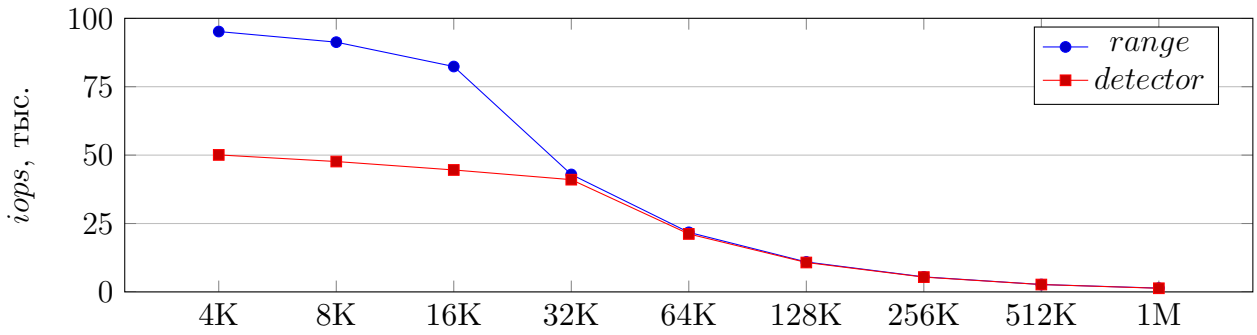


Рис. 6: Зависимость *iops* от размера блока

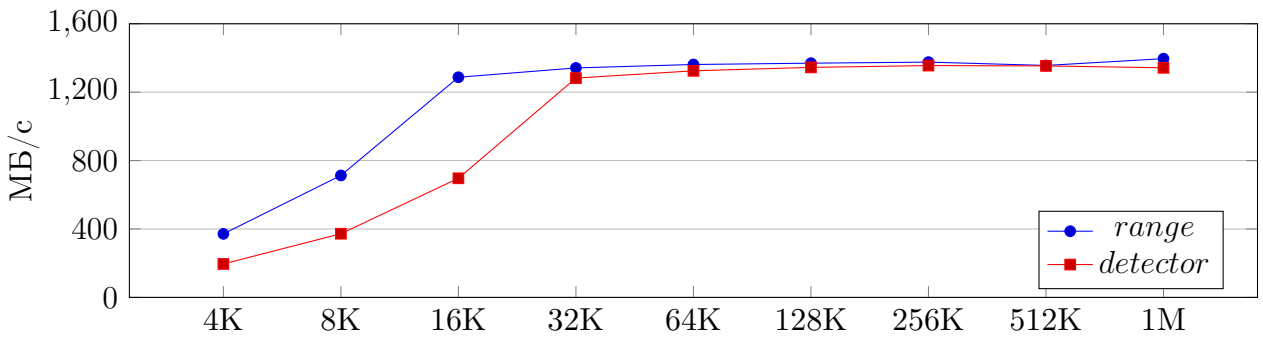


Рис. 7: Зависимость скорости от размера блока

На рисунках 6 и 7 изображена зависимость производительности СХД (в *iops* и МБ/с) от размера блока для обоих алгоритмов. Из рисунка 6 видно, что предел производительности *detector* составляет 50 000 *iops*, что почти в два раза ниже, чем аналогичный предел для *range*. При увеличении размера блока более значительную роль начинает играть производительность дисков, и начиная с запросов размера 64 КБ показатели алгоритмов сравниваются.

Использование нового алгоритма предполагается на задачах мультимедиа, где все операции производятся блоками размером по 1 МБ. Исходя из этого, его предельная пропускная способность составляет 50 ГБ/с, что эквивалентно скорости СХД из 350 жёстких дисков.

Данный тест показывает, что *detector* может замедлить работу системы хранения при нагрузках, требующих обработки большого числа запросов в секунду, но хорошо справляется там, где требуется обеспечить максимальную пропускную способность при чтении данных большими блоками.

5.4. Выводы

Функциональное сравнение алгоритмов показало, что на традиционных последовательных нагрузках *detector* справляется с задачей обнаружения последовательностей не хуже *range*, а на специфичных для мультимедиа типах нагрузки показывает существенно лучшие результаты.

Тест, проведённый утилитой `frametest`, показал, что улучшения в работе алгоритма несут не только теоретический характер, но и отражаются на практике, увеличивая пропускную способность СХД до 40%.

Слабой стороной алгоритма является его бóльшая вычислительная сложность, что ограничивает его применение в сферах, где имеет значение количество обрабатываемых в секунду запросов, а не пропускная способность системы. Нагрузки, характерные для мультимедиа, предъявляют требования именно к пропускной способности системы, поэтому в этой области ограничение алгоритма не является существенным.

Заключение

В ходе данной работы были достигнуты следующие результаты:

- Изучен принцип работы существующих алгоритмов обнаружения последовательных запросов.
- Разработан новый алгоритм выделения последовательных операций среди запросов ввода-вывода блочного уровня.
- Алгоритм реализован на языке C и интегрирован с модулями ядра ПО RAIDIX.
- Проведено функциональное и нагрузочное тестирование реализованного модуля, а также сравнение его с существующим алгоритмом.

Реализованный модуль показал лучшие результаты в целевой области по сравнению со своим предшественником. Идеологически алгоритм не привязан к какому-либо окружению или аппаратной конфигурации и может использоваться в любых СХД с блочным доступом.

Список литературы

[1] Assert(!Defined(Sequential I/O)) / Cheng Li, Philip Shilane, Fred Douglis et al. // Proceedings of the 6th USENIX Conference on Hot Topics in Storage and File Systems. — HotStorage'14. — 2014. — P. 10–10.

[2] Autodesk. Testing storage and filesystem Performance using the swioperftool command // Autodesk Knowledge Network. — 2014. — URL: `http://goo.gl/OB5ics` (online; accessed: 03.06.2015).

[3] C-Miner: Mining Block Correlations in Storage Systems / Zhenmin Li, Zhifeng Chen, Sudarshan M. Srinivasan, Yuanyuan Zhou // Proceedings of the 3rd USENIX Conference on File and Storage Technologies. — FAST'04. — 2004. — P. 13–13.

[4] Comparing partitions // Journal of Classification. — 1985. — Vol. 2, no. 1.

[5] Love Robert. Linux Kernel Development. — 3rd edition. — Addison-Wesley Professional, 2010. — ISBN: 0672329468, 9780672329463.

[6] RAIDIX. Официальный сайт // RAIDIX. — 2015. — URL: `http://raidix.ru` (дата обращения: 03.06.2015).

[7] SGI. FRAMETEST(1) // Techpubs Library. — 2015. — URL: `http://goo.gl/xApoLV` (online; accessed: 03.06.2015).

[8] TaP: Table-based Prefetching for Storage Caches / Mingju Li, Elizabeth Varki, Swapnil Bhatia, Arif Merchant // Proceedings of the 6th USENIX Conference on File and Storage Technologies. — FAST'08. — 2008. — P. 6:1–6:16.

[9] Wang Randolph, Wang Olph Y., Anderson Thomas E. xFS: A Wide Area Mass Storage File System // In Fourth Workshop on Workstation Operating Systems. — 1993. — P. 71–78.

[10] fio. Flexible I/O tester // GitHub. — 2015. — URL: `https://github.com/axboe/fio/blob/master/README` (online; accessed: 03.06.2015).