Санкт-Петербургский государственный университет

Кафедра системного программирования

Васенина Анна Игоревна

Исследование и разработка адаптивного объединения запросов для RAID-массива на основе твердотельных накопителей

Бакалаврская работа

Научный руководитель:
д. ф.-м. н., профессор Терехов А. Н.

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

Санкт-Петербург
2019

SAINT-PETERSBURG STATE UNIVERSITY

Software Engineering

Vasenina Anna

Research and development of adaptive request merging for SSD-based RAID

Graduation Thesis

Scientific supervisor:
professor Andrey Terehov

Reviewer:
Head of RAIDIX R&D department, Ph. D. Svetlana Lazareva

Saint-Petersburg
2019

Оглавление

Введение **4**

1. Постановка задачи **7**

2. Обзор **8**

2.1. RAIDIX ERA 8

2.2. Тестирование производительности СХД 8

2.3. Планировщики запросов ввода-вывода в Linux 10

2.4. Технологии анализа входящей нагрузки 11

3. Анализ эффективности объединения запросов **14**

3.1. Тестирование при помощи FIO 14

3.2. Тестирование при помощи VDbench 17

3.3. Выводы 17

4. Алгоритм **18**

4.1. Детектор последовательностей 19

4.2. Детектор скоростей 20

5. Внедрение алгоритма в систему RAIDIX ERA **22**

6. Тестирование алгоритма **24**

6.1. Сравнение с ранее полученными результатами 24

6.2. Тестирование с изменяющейся нагрузкой 25

Заключение **27**

Список литературы **28**

Введение

В современном мире рост количества информации с каждым годом набирает темп, что отражается на развитии технологий хранения данных. Для организации долговременного хранения данных применяются различные устройства хранения данных (УХД) [20]. Помимо объема, существенной характеристикой УХД является производительность. Современные твердотельные накопители способны работать существенно быстрее, чем их предшественники. Значительным прорывом в производительности УХД стало появление нового стандарта доступа к твердотельным накопителям – NVMe [8]. Этот стандарт за счет подключения накопителей по шине PCI Express и специально разработанным под современные многопроцессорные системы набором команд, существенно увеличивает скорость обработки запросов к накопителю.

Помимо усовершенствования УХД, увеличение производительности системы может достигаться за счет объединения отдельных УХД в систему хранения данных (СХД). Одной из технологий организации СХД является технология RAID-массивов [4]. RAID-массив объединяет УХД при помощи полос данных. Каждая полоса данных состоит из равных фрагментов (стрипов), количество которых равно количеству УХД в RAID-массиве. Каждый стрип хранится на своем УХД, что обеспечивает возможность параллельного чтения и записи информации на УХД в составе RAID-массива. В зависимости от требований к надежности СХД, в полосы данных могут быть добавлены элементы избыточности, позволяющие восстанавливать данные при отказе дисков. В зависимости от требований к производительности, надежности и стоимости СХД, применяются RAID-массивы разных уровней [12], отличающихся структурой полос данных. Наиболее широкое распространение в промышленности получили уровни 0, 1, 5, 6 и их комбинации.

RAID-массив уровня 0 не поддерживает избыточности и служит для увеличения скорости обработки запросов. RAID-массив уровня 1 использует зеркалирование, то есть все диски являются полными копиями друг друга. RAID-массивы уровней 5 и 6 используют добавление

к полосе данных контрольных сумм – синдромов, распределенных по всем дискам RAID-массива. Объем дискового массива при этом уменьшается не вдвое, как при зеркалировании, а на количество синдромов, умноженное на объем УХД в составе RAID-массива. Такая экономичность является особенно значимой для накопителей стандарта NVMe, до сих пор сохраняющих высокую стоимость на рынке устройств хранения данных.

Одним из основных недостатков синдромных RAID-массивов являются тяжеловесные операции чтения/записи (в дальнейшем RMW) [1]. Эти атомарные операции возникают при записи в полосу данных блока данных размером меньше длины полосы данных, в дальнейшем будем называть такие блоки «маленькими». При этом одна операция записи превращается в следующую последовательность операций: чтение информации из адресуемого блока и соответствующих ему синдромов, пересчет контрольных сумм, запись адресуемого блока и новых контрольных сумм.

При последовательной записи маленькими блоками количество RMW-операций может быть сокращено при помощи объединения запросов, так как если запросы отправляются на выполнение не по одному, а всей полосой данных, то контрольные суммы нужно посчитать всего один раз. В операционной системе Linux задача объединения запросов возлагается на планировщики запросов ввода-вывода [11]. Однако предоставляемой планировщиками функциональности недостаточно для решения задачи сокращения количества RMW-операций, поскольку в то время как планировщики пытаются объединить хоть какое-то количество запросов, для выигрыша в производительности необходимо накопление запросов на всю длину полосы данных.

Особенности работы СХД на основе твердотельных накопителей требуют появления новых решений в области программного обеспечения, удовлетворяющих требованиям потребителей. Одним из таких решений стала представленная в 2018 году российской компанией RAIDIX программно-определяемая СХД, ориентированная на работу с твердотельными накопителями – RAIDIX ERA [10], благодаря lockless архи-

тектуре параллельных вычислений показавшая отличные результаты на синтетических тестах [14].

Реализация алгоритма объединения запросов для RAID, представленная в RAIDIX ERA, намеренно устанавливает некоторое время ожидания для запросов на запись. В течение этого времени запрос не передается на диски, а ожидает поступления новых запросов в эту полосу данных. По истечении времени ожидания, если это возможно, последовательные запросы объединяются в один запрос,с длиной равной длине полосы данных, что позволяет существенно выиграть в производительности, несмотря на дополнительное время ожидания. Тем не менее, принудительное ожидание для каждого запроса является неприемлемым, поскольку объединение запросов необходимо только для небольшой доли возможных паттернов нагрузки. Таким образом, появляется необходимость в алгоритме, позволяющем автоматически определять, в каких случаях необходимо применять объединение запросов.

1. Постановка задачи

Целью работы является создание подсистемы, управляющей объединением входящих запросов записи на основании анализа входящей нагрузки. Для достижения этой цели в рамках работы были сформулированы следующие задачи.

- Изучить возможные стратегии обнаружения последовательных запросов к СХД.
- Проанализировать эффективность объединения последовательных запросов на различных паттернах нагрузки.
- Разработать алгоритм, управляющий объединением запросов.
- Реализовать и внедрить алгоритм в систему RAIDIX ERA.
- Выполнить тестирование алгоритма.

2. Обзор

2.1. RAIDIX ERA

Система RAIDIX ERA является программно-определяемой СХД, предназначенной для высокопроизводительных накопителей. Система ориентирована на использование RAID-массивов на базе контрольных сумм. Первая версия системы была выпущена в сентябре 2018 года, весной 2019 года на рынок была выпущена версия 2.0 и в настоящий момент ведется активная работа над версией 3.0, включающей в себя в том числе поддержку гибридных уровней RAID-массивов и новую технологию восстановления данных. Система RAIDIX ERA эффективно использует потенциал твердотельных накопителей для создания быстрого и отказоустойчивого RAID-массива. Система содержит два компонента: управляющую утилиту и модуль для ядра Linux.

Управляющая утилита написана на языке python версии 3.4. Ее задачей является обеспечение удобного взаимодействия с пользователем. Для выполнения этой задачи утилита выполняет преобразование удобных для человека команд управления RAID-массивами в команды, используемые модулем ядра Linux, и проверку корректности аргументов команды.

Модуль для ядра Linux обеспечивает создание и управление RAID-массивами, доступными в виде локальных блочных устройств. Блочным устройством в операционной системе Linux называется особый вид файла, обеспечивающий интерфейс доступа к устройству через обмен с ним блоками данных. Такая организация СХД позволяет использовать файловые системы без каких-либо дополнительных затрат.

2.2. Тестирование производительности СХД

Обычно тестирование производительности СХД проводится двумя различными путями: с использованием файловой системы или нагрузкой непосредственно на блочное устройство. В данной работе мы будем использовать второй вариант, так как это позволяет получить более

«чистые» результаты с точки зрения влияния на производительность сторонних факторов помимо тестируемого кода.

Для тестирования блочных устройств в операционной системе Linux существует два распространенных инструмента: Oracle VDbench [9] и Flexible I/O Tester (FIO) [17]. Оба инструмента позволяют генерировать различные типы нагрузки в соответствии с заданным конфигурационным файлом. При этом VDbench предоставляет широкие возможности для проведения сложных тестов и эмуляции паттернов нагрузки реальных приложений. В рамках данной работы VDbench использовался для реализации одного из паттернов нагрузки SNIA (Storage Networking Industry Association, ассоциации индустрии сетей хранения данных), описанном в [13], а также для эмулирования некоторых паттернов нагрузки реальных систем: базы данных, файлового сервера, VDI и веб-сервера.

Паттерн тестирования SNIA нуждается в адаптации для твердотельных накопителей, так как минимальный размер запроса к твердотельному накопителю составляет 4 килобайта, а представленный паттерн включает запросы меньшего размера. Для твердотельных накопителей будем считать, что все запросы длины меньшей, чем 4 килобайта, преобразуются в запросы длиной 4 килобайта. Такое распределение размеров блоков (рис. 1) назовем неоклассическим паттерном тестирования.

BS	Neoclassic		Four		Sixty Four		Four+Largeseq		Four+Neoseq	
	Sequential	Random	Sequential	Random	Sequential	Random	Sequential	Random	Sequential	Random
4	29	31	56	55	30	30		55	29	55
8	33	31	33	31	33	31		31	33	31
16	6	5	6	5	6	5		5	6	5
32	5	5	5	5	31	5		5	5	5
48		1		1		1		1		1
56		1		1		1		1		1
60		2		2		27		2		2
64	22	20					81		22	
128	3	2					11		3	
256	2	2					8		2	

Рис. 1: Распределение размеров блоков в неоклассическом паттерне тестирования

Для тестирования типов нагрузки, не предполагающих наличия в одном потоке блоков разного размера, будем использовать FIO. Помимо генерирования чистых типов нагрузки, таких как последовательная или

случайная запись, FIO позволяет тестирование простых комбинаций типов нагрузки и возможность последовательного запуска сценариев тестирования.

2.3. Планировщики запросов ввода-вывода в Linux

Многие из планировщиков задач Linux в ходе своей работы выполняют объединение запросов. В области планирования постоянно ведутся исследования и появляются новые разработки. Увеличение скорости накопителей привело к переходу от единой очереди (blk-sq), используемой такими планировщиками как noop, deadline и cfq, к технологии blk-mq [7], заменяющей одну очередь на количество очередей, соответствующих количеству ядер центрального процессора. Некоторые из планировщиков также сталкиваются с проблемой диапазона длины объединения запросов. В статье [5] предлагается в качестве диапазона длины запросов, в пределах которого осуществляется объединение, использовать физический параметр SSD-диска – размер его буффера чтения/записи.

Для решения вопроса использования планировщика запросов в системе RAIDIX ERA было проведено тестирование предела производительности планировщиков запросов. Тестирование проводилось на блочном устройстве с отсутствующей логикой обработки запроса: на каждый поступающий запрос это блочное устройство сразу возвращало команду успешного завершения запроса. Такой тест позволяет узнать скорость поступления запросов в блочное устройство, а следовательно и скорость планировщика запросов. Тестирование производилось при помощи FIO в количество потоков 32, каждый из которых имел глубину очереди запросов 32. Тестирование проводилось для всех чистых видов нагрузки и для всех стандартных планировщиков запросов Linux: noop, deadline [3], cfq [2], none-mq, mq-deadline и kyber [6]. Результаты тестирования в тысячах IOPS представлены на рис. 2.

Как можно увидеть по результатам тестирования, использование планировщиков ввода-вывода, даже noop и none-mq, не выполняющих

Тип нагрузки	none	blk-sq			blk-mq		
		noop	deadline	cfq	none-mq	mq-deadline	kyber
Случ. чтение	7273	348	323	3240	3130	352	2971
Случ. запись	5411	350	322	3296	2983	352	2924
Посл. чтение	7021	345	321	190	3079	355	2920
Посл. запись	5111	346	320	192	2952	352	2875

Рис. 2: Результаты тестирования планировщиков ввода-вывода Linux. Результаты представлены в тысячах IOPS

перестановок и объединения запросов, существенно снижает производительность системы, по сравнению с использованием блочного устройства без планировщика (none). Из планировщиков, выполняющих объединение запросов, наилучший результат для последовательной записи показал mq-deadline. При этом данная производительность становится лимитом для блочного устройства, в связи с чем данное решение было признанно непригодным для нашей системы. Поэтому было принято решение о разработке собственного алгоритма объединения последовательных запросов записи на основе красно-черных деревьев и управляющего этим процессом алгоритма на основе анализа входящей нагрузки.

2.4. Технологии анализа входящей нагрузки

В данной работе мы рассматриваем задачу анализа входящей нагрузки. Подобная задача возникает также как составная часть других технологий систем хранения данных, таких как SSD-кэширование, приоритизация запросов и упреждающее чтение. Рассмотрим подробнее анализаторы нагрузки из этих решений.

SSD-кэширование

Статья [16] предлагает путем внедрения детектора случайного доступа к памяти решение таких проблем SSD-кэша для высокопроизводительных вычислений, как необходимость большого объема SSD-дисков, а также гонки скоростей вычисления и сохранения информации в постоянное хранилище на жестких дисках. При этом все запросы

разделяются на группы определенного размера. Если группа идентифицируется как случайная, то ее запись пойдет через SSD-кэш, если же группа запросов определена как последовательная, то запись пойдет непосредственно на HDD-диски, минуя SSD-кэш. Это позволяет избегать главной проблемы производительности жестких дисков – долгого механического перемещения головок при случайном доступе к данным.

Для определения группы запросов как случайной или последовательной, используется приведенный ниже алгоритм.

1. Запросы внутри группы сортируются по логическому адресу (LBA).
2. Для каждого запроса считается метрика random factor (RF), которая иллюстрирует частоту движения дисковой головки. Она равняется единице, если следующий в группе адрес равен длине запроса, прибавленной к текущему адресу. В противном случае она равняется нулю.
3. Для группы запросов считается S, сумма RF.
4. Для группы считается процент RF от общего числа переходов ($p = S/(N-1)$, где N — число запросов в группе).
5. В зависимости от показателя p и некоторой константы, определяется, считать группу запросов случайной или последовательной.

Алгоритмы потокового анализа трафика

В решении RAIDIX, ориентированном на HDD, используется алгоритм [15], классифицирующий входящие запросы в реальном времени, что позволяет разделять запросы от разных приложений и выделять среди них более приоритетные. Алгоритм построен на основе сбора информации о каждом запросе и анализа этой информации при помощи классифицирующих деревьев (Random Forest). Основным недостатком подобного подхода для нашей задачи является необходимость пропускать через определяющий алгоритм все входящие запросы, которых

при работе с накопителями стандарта NVMe становится во много раз больше, чем при работе с HDD-дисками.

Алгоритмы упреждающего чтения и предварительной загрузки страниц

Также детекторы входящей нагрузки являются частью задач, ориентированных на ускорение работы СХД за счет предугадывания, к каким данным будет осуществлен доступ в следующий момент времени. Исследования на эти темы уже проводились студентами кафедры системного программирования [18] [19] и содержат обзор некоторых из таких алгоритмов. Однако для нашей задачи детекторы из этого раздела оказываются недостаточно ориентированными на определение типа нагрузки, так как для задачи упреждающего чтения главным является поиск взаимосвязи между поступающими запросами, а не определение типа нагрузки на систему.

Актуальность работы

Представленные в этом разделе технологии анализа входящей нагрузки разработаны в расчете на небольшое количество операций ввода-вывода в секунду. Их применение в онлайн-режиме для высокопроизводительного RAID-массива на базе твердотельных накопителей не представляется возможным, так как такой массив должен быть способен обрабатывать миллионы операций ввода-вывода в секунду.

3. Анализ эффективности объединения запросов

Для решения задачи автоматизации управления объединением запросов, были проведены тесты производительности на различных паттернах нагрузки. Для тестирования использовался массив RAID 6 на 6 NVMe с размером стрипа 16 килобайт. Соответственно, в полосе данных RAID-массива данные занимают 64 килобайта, а контрольные суммы 32 килобайта.

В этой главе приведены результаты тестирования производительности описанного RAID-массива для двух конфигураций: с объединением запросов и без объединения запросов. В таблицах, представляющих результаты тестирования, используется цветовая схема зеленый-желтый-красный, отображающая зеленым цветом области выигрыша конфигурации в производительности, желтым – разницу между конфигурациями в пределах погрешности измерений и красным – области проигрыша конфигурации в производительности. По горизонтали изменяется количество потоков, по вертикали — глубина очереди. Результаты тестов приведены в мегабайтах в секунду. Каждая серия тестов сопровождается кратким анализом.

В начале главы приводятся результаты для случайной и последовательной нагрузки в чистом виде. Вторая половина главы посвящена изучению влияния фоновой нагрузки на эффективность объединения запросов. Тестирование проводится при помощи инструментов FIO и VDbench.

3.1. Тестирование при помощи FIO

В первую очередь выполним тесты, подтверждающие существенное падение производительности при случайной записи. Таблицы, приведенные на рис. 3 демонстрируют производительность случайной записи блоком размера 4 килобайта. Результаты показывают падение производительности до двух раз при использовании объединения запросов. Та-

ким образом, проведенные измерения на практике подтверждают предположение о неэффективности объединения запросов при случайном доступе.

Без объединения запросов

Глубина очереди \ Количество потоков	1	8	32
1	27.2	170	392
2	48.9	293	634
4	68.9	425	880
8	108	598	1157
16	143	823	1493
32	259	1136	1681

С объединением запросов

Глубина очереди \ Количество потоков	1	8	32
1	8.95	67.6	227
2	17.8	128	360
4	34.3	226	569
8	64	359	878
16	106	556	1248
32	149	830	1563

Рис. 3: Результаты тестирования производительности случайной записи для размера блока 4 килобайта. Результаты представлены в мегабайтах

блоков

Размер блока 4 KiB

Без объединения запросов

Глубина очереди \ Количество потоков	1	8	32
1	29	206	550
2	53.1	339	806
4	70.1	453	1018
8	104	581	1203
16	154	760	1473
32	233	988	1742

С объединением запросов

Глубина очереди \ Количество потоков	1	8	32
1	22.4	165	477
2	26.2	187	567
4	33.6	238	709
8	46.9	314	875
16	592	3363	5594
32	661	4306	5582

Размер блока 8 KiB

Без объединения запросов

Глубина очереди \ Количество потоков	1	8	32
1	46.8	251	610
2	68	399	900
4	82.9	500	1103
8	127	669	1383
16	196	883	1789
32	319	1176	2052

С объединением запросов

Глубина очереди \ Количество потоков	1	8	32
1	30	203	599
2	29	227	694
4	47	311	896
8	753	3974	5980
16	944	5186	5847
32	1419	5610	5813

Размер блока 16 KiB

Без объединения запросов

Глубина очереди \ Количество потоков	1	8	32
1	44.6	291	750
2	56.1	370	935
4	89.8	516	1202
8	153	736	1590
16	246	1042	2015
32	395	1401	2012

С объединением запросов

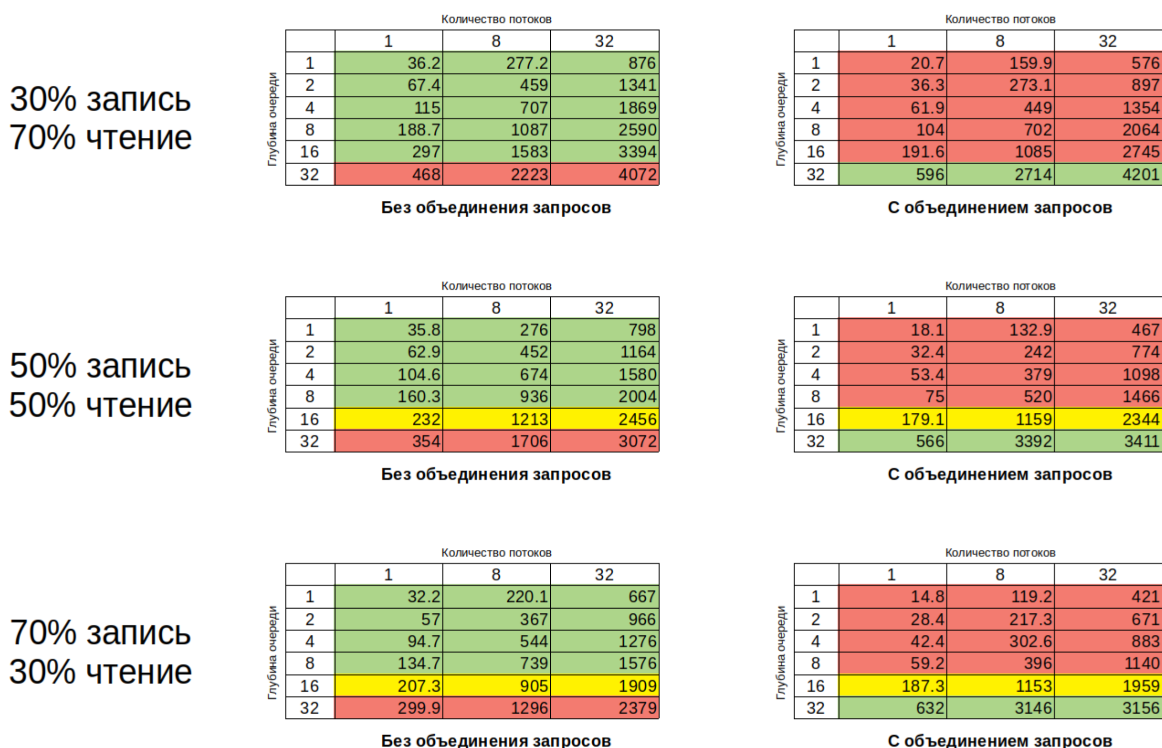
Глубина очереди \ Количество потоков	1	8	32
1	36.3	228	694
2	45.6	308	896
4	917	4595	4961
8	1195	5457	4888
16	1789	5670	5360
32	2446	5859	4493

Рис. 4: Результаты тестирования производительности (Мбайт/с) последовательной записи для размеров блоков 4, 8, 16 килобайт

Далее была проведена серия тестов для последовательной записи. Результаты, представленные на рис. 4, показали, что для эффективности объединения запросов недостаточно наличия последовательной нагрузки. Из результатов, представленных для разных размеров блоков, можно сделать вывод, что объединение запросов начинает давать

выигрыш в производительности, когда размер блока, умноженный на глубину очереди, достигает значения, равного длине данных в полосе данных, обеспечивая достаточную скорость заполнения полосы данных.

Последняя серия тестов, проведенная при помощи инструмента FIO, исследует влияние фоновой нагрузки чтения на эффективность объединения запросов. Тесты проводились только для одного размера блока (4 килобайта), но для разных соотношений чтения и записи. И чтение, и запись были последовательными. Результаты, представленные на рис. 5, показывают что при фоновой нагрузке чтения объединение запросов становится эффективным только при глубине очереди в 32 запроса, в то время как при простой последовательной записи для этого было достаточно глубины очереди в 16 запросов. Предполагаем, что такой эффект возникает из-за недостаточной скорости поступления запросов в полосу данных.



30% запись 70% чтение

Без объединения запросов

Глубина очереди \ Количество потоков	1	8	32
1	36.2	277.2	876
2	67.4	459	1341
4	115	707	1869
8	188.7	1087	2590
16	297	1583	3394
32	468	2223	4072

С объединением запросов

Глубина очереди \ Количество потоков	1	8	32
1	20.7	159.9	576
2	36.3	273.1	897
4	61.9	449	1354
8	104	702	2064
16	191.6	1085	2745
32	596	2714	4201

50% запись 50% чтение

Без объединения запросов

Глубина очереди \ Количество потоков	1	8	32
1	35.8	276	798
2	62.9	452	1164
4	104.6	674	1580
8	160.3	936	2004
16	232	1213	2456
32	354	1706	3072

С объединением запросов

Глубина очереди \ Количество потоков	1	8	32
1	18.1	132.9	467
2	32.4	242	774
4	53.4	379	1098
8	75	520	1466
16	179.1	1159	2344
32	566	3392	3411

70% запись 30% чтение

Без объединения запросов

Глубина очереди \ Количество потоков	1	8	32
1	32.2	220.1	667
2	57	367	966
4	94.7	544	1276
8	134.7	739	1576
16	207.3	905	1909
32	299.9	1296	2379

С объединением запросов

Глубина очереди \ Количество потоков	1	8	32
1	14.8	119.2	421
2	28.4	217.3	671
4	42.4	302.6	883
8	59.2	396	1140
16	187.3	1153	1959
32	632	3146	3156

Рис. 5: Результаты тестирования производительности (Мбайт/с) последовательной записи для блока 4 килобайт и различных процентов

3.2. Тестирование при помощи VDbench

Тестирование сложных паттернов нагрузки, приближенных к реальной нагрузке систем хранения данных, проводилось при помощи инструмента VDbench. При этом проводилось тестирование 5 вариаций неоклассического паттерна (Neoclassic, Four, Sixty Four, Four+Largeseq, Four+Neoseq) и четырех эмуляций нагрузки реальных систем (база данных, файловый сервер, VDI и веб-сервер). Результаты, представленные на рис. 6 показывают, что на всех сложных паттернах объединение запросов дает негативный результат в производительности, так как нагрузка других типов преобладает над последовательной записью.

	Без объединения запросов	С объединением запросов
Neoclassic	2021.61	1722.22
Four	972.53	714.67
Sixty Four	1386.13	1188.56
Four+Largeseq	2668.43	2457.04
Four+Neoseq	1368.54	1254.06
База данных	1029.37	789.37
Файловый сервер	2065.74	1667.17
VDI	440.86	249.66
Веб-сервер	5427.48	5361.73

Рис. 6: Результаты тестирования производительности (Мбайт/с) для сложных паттернов нагрузки

3.3. Выводы

Из проведенного тестирования можно сделать вывод о неэффективности объединения запросов в следующих случаях.

1. При случайной записи.
2. При недостаточной скорости поступления запросов.

Таким образом, наша управляющая подсистема должна уметь определять приведенные выше паттерны нагрузки для того, чтобы не применять объединение запросов, когда оно не дает выигрыша в производительности.

4. Алгоритм

Задачей управляющего алгоритма является адаптация под различные типы нагрузки, заключающаяся в использовании объединения запросов только тогда, когда это дает преимущество в производительности. В реальных условиях нередко встречается ситуация, когда на разные части RAID-массива действуют разные типы нагрузки. Ввиду этого, определять необходимость объединения запросов выгодно не для всего RAID-массива, а для некоторых его областей.

За область примем некоторое фиксированное количество полос данных, расположенных в RAID-массиве друг за другом. Для каждой области будем принимать независимое решение об объединении запросов, базирующееся на анализе поступающей в область нагрузки записи. При этом важно оценивать, имеет ли нагрузка последовательный характер и достаточную скорость. Для конкретного запроса при этом управляющий алгоритм можно описать схемой, представленной на рис. 7.

В случае, если в области отсутствует последовательная нагрузка, или скорость этой последовательности недостаточна, объединение запросов будет проигрывать в производительности, поэтому управляющий алгоритм должен отправить запросы на выполнение как только запросы поступили в RAID-массив. В случае же, когда нагрузка последовательна и скорость последовательности является достаточной, область переходит в состояние постоянного накопления поступающих запросов, которые она отправляет на выполнение только по истечении максимального времени ожидания, определяемого требованиями к системе или когда поступившие запросы формируют полную полосу данных и их можно объединить, тем самым избежав RMW-операций, которые пришлось бы выполнять при обработке запросов по отдельности.

Наиболее сложными в алгоритме являются шаги, определяющие принадлежность запроса последовательности в область и скорость последовательности. Решение этих задач производится детектором последовательностей и детектором скоростей. Рассмотрим их устройство подробнее.

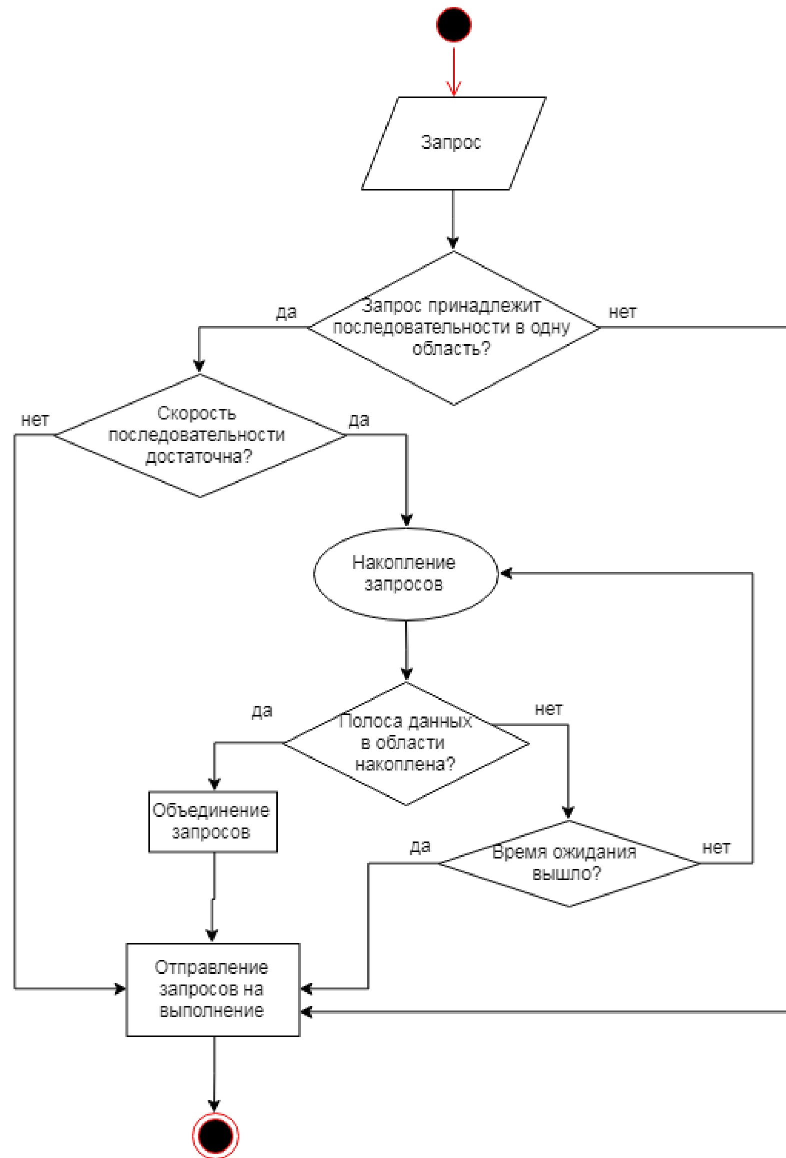


Рис. 7: Схема алгоритма

4.1. Детектор последовательностей

Для запроса ввода-вывода, адресованного RAID-массиву, по адресу можно определить соответствующую ему полосу данных. В случае, если запрос попадает в несколько полос данных, он будет разбит и разные части продолжат обработку независимо друг от друга, поэтому в дальнейших рассуждениях будем считать, что запросу соответствует одна полоса данных.

При поступлении запроса в полосу данных производится поиск в оперативной памяти структуры, описывающей эту полосу данных и, в случае отсутствия такой структуры, создается новая. Введем в эту

структуру битовую карту, в которой будем отмечать области, в которые поступали запросы записи (рис. 8). Каждому биту в битовой карте при этом соответствует 4 килобайта (минимальная длина запроса к твердотельному накопителю).

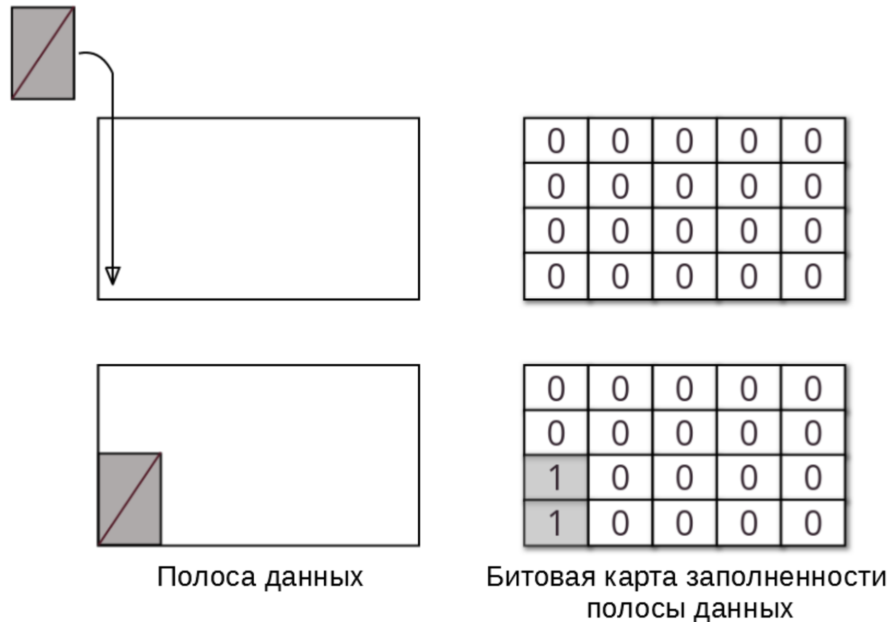


Рис. 8: Процесс заполнения битовой карты заполненности полосы данных

Структура, описывающая полосу данных, существует только пока существуют запросы, адресованные в эту полосу данных. Поэтому, если за время ее существования, битовая карта заполненности полосы данных полностью заполнилась единицами, то мы можем с высокой степенью уверенности утверждать, что в область ведется последовательная запись.

4.2. Детектор скоростей

На базе битовой карты заполненности полосы данных, представленной в описании детектора последовательностей, мы можем так же определять, на сколько процентов заполнилась полоса данных с момента создания структуры, описывающей полосу данных.

Наше время ожидания для запроса t всегда варьируется между нулем и максимумом, задаваемым требованиями к системе. При этом t

равное нулю отвечает за отсутствие ожидания для запроса. Так что для описания детектора скоростей будем считать, что t всегда больше нуля. Мы считаем, что скорость поступления запросов в полосу данных в идеальной ситуации равномерна. Черная полоса на графике на рис. 9 обозначает минимальную скорость, с которой запросы должны поступать в полосу данных, чтобы мы успели накопить полную полосу данных до истечения максимального времени ожидания.

Теперь оценим скорость поступления запросов для двух примеров (рис. 9): для точки (t1,a) текущая скорость выше нашей теоретически необходимой, а для точки (t2,b) ниже. Таким образом, для точки (t1,a) мы делаем вывод, что успеем накопить полную полосу данных и должны продолжить ожидание. Для точки (t2, b), напротив, накопить полную полосу данных мы не сможем, поэтому ждать дольше не имеет смысла. Будем называть этот случай неудачным ожиданием.

Если бы скорость заполнения полосы данных всегда была постоянна, то мы бы могли на основании одного неудачного ожидания определять отсутствие необходимости объединения запросов из-за недостаточной скорости. На практике же скорость поступления запросов не постоянна и функция заполненности полосы данных приобретает ступенчатый вид (рис. 10), что не дает нам однозначно определять эффективность объединения запросов на основании одного измерения. Поэтому решение о неприменении объединения запросов принимается при наличии некоторой доли неудачных ожиданий в области.

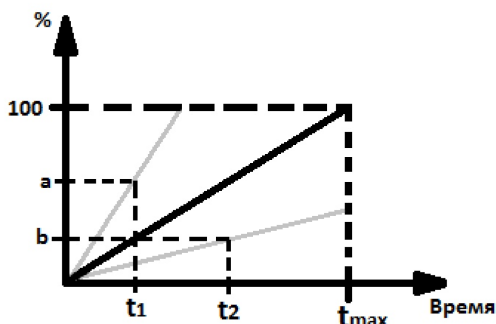


Рис. 9: Теоретическая оценка необходимой скорости поступления запросов

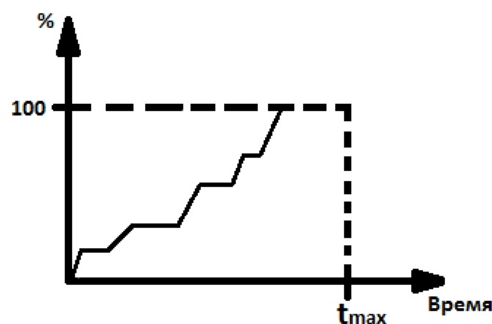


Рис. 10: Скорость заполнения полосы данных в условиях, приближенных к реальным

5. Внедрение алгоритма в систему RAIDIX ERA

Для реализации разделения RAID-массива на области была введена битовая карта, в которой для каждой области есть бит, управляющий объединением запросов. Он выставляется в 1, если объединение запросов необходимо производить и в 0, если объединение запросов производить не нужно. При этом информации о каждой области мы можем доверять только определенный промежуток времени, поэтому для каждого бита также хранится время его последнего обновления. Общая схема показана на рис. 11.

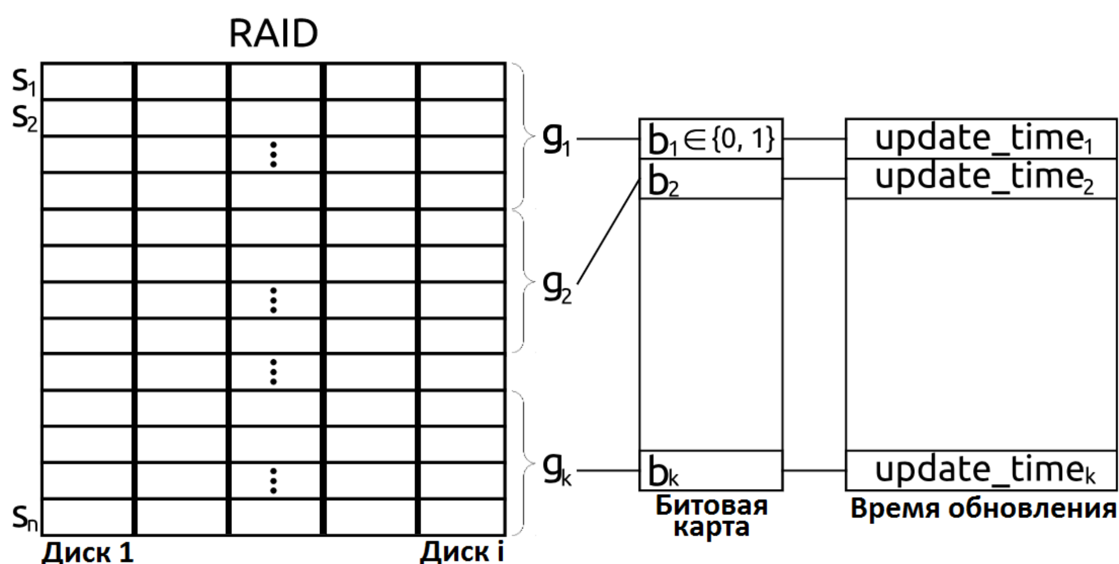


Рис. 11: Разделение RAID-массива на области

Для реализации детектора последовательностей и детектора скоростей, представленных в главе 4, было написано три функции на языке C. Через каждую из этих функций запрос проходит в процессе обработки RAID-массивом.

1. Функция, определяющая актуальность бита в управляющей битовой карте $bitmap_clear$.

2. Функция, следящая за скоростями и временным лимитом $merge_check$.

3. Функция, определяющая последовательную запись *seq_detect*

Функция *bitmap_clear* оценивает, насколько давно было последнее обновление управляющего бита, и, если управляющий бит равен 1 и при этом был выставлен давнее установленного времени, то функция выставляет управляющий бит в 0, отключая объединение запросов в области. После этого, если управляющий бит равен 1, начинается накопление запросов. Функция *merge_check* обрабатывает полосы данных в состоянии накопления запросов. При этом поступающие запросы не отправляются на выполнение, а сохраняются в буфере накопления до тех пор, пока не будет накоплена полоса данных или не будет принято решение об отправлении запросов на выполнение без объединения с фиксацией при этом неудачного ожидания.

Отправление запросов на выполнение по отдельности может быть произведено по двум причинам. Певая причина – это неудачное ожидание, при котором текущей скорости заполнения не хватает на накопление полной полосы данных до истечения максимального времени ожидания. Вторая причина – это истечение максимального времени ожидания несмотря на то, что скорость заполнения оценивалась как достаточная. Отслеживанием скоростей и временного лимита, а также отправлением накопленных полос данных на объединение занимается функция *merge_check*. Помимо этого, если в области произошло слишком много (в процентном отношении) неудачных ожиданий, функция *merge_check* обозначит область как область без объединения запросов.

После проделанных операций, если в полосе данных, в которую пришел запрос, битовая карта заполненности полосы данных полностью состоит из единиц, то функция *seq_detect* определит, что запись ведется последовательно.

6. Тестирование алгоритма

В этой главе приводятся результаты тестов производительности для объединения запросов, которое регулируется при помощи управляющей подсистемы, представленной в главах 4 и 5. Будем называть объединение запросов под управлением представленной подсистемы адаптивным объединением запросов. Для демонстрации того, что предлагаемый в этой работе управляющий алгоритм справляется с задачей выбора наилучшей стратегии объединения запросов, в таблицах проводится сравнение трех конфигураций: без объединения запросов, с объединением запросов и с адаптивным объединением. Приведенные таблицы демонстрируют, что результаты адаптивного объединения во всех случаях близки к более выгодной стратегии (находятся в пределах погрешности измерений в 20%).

Тестирование проводится на RAID-массиве с такими же параметрами, как и в главе 3. Для апробации управляющего алгоритма была выбрана нагрузка с размером блока 4 килобайта. В таблицах, представляющих результаты тестирования, также используется цветовая схема зеленый-желтый-красный, отображающая зеленым цветом области выигрыша конфигурации в производительности, желтым – разницу между конфигурациями в пределах погрешности измерений и красным – области проигрыша конфигурации в производительности. В первой части главы приводится сравнение результатов, полученных с адаптивным объединением запросов, с результатами, полученными в главе 3. Во второй части приводятся тесты с изменяющейся нагрузкой и демонстрируется, что управляющий алгоритм способен быстро изменять стратегию, подстраиваясь под изменяющуюся нагрузку.

6.1. Сравнение с ранее полученными результатами

Для случайной (рис. 12) и последовательной (рис. 13) записи мы видим, что управляющий алгоритм выбирает нужную стратегию, и адаптивное объединение запросов не теряет в производительности там, где объединение запросов без управляющей подсистемы показывало отри-

цательные результаты.

Количество потоков

Глубина очереди	1	8	32
1	27.2	170	392
2	48.9	293	634
4	68.9	425	880
8	108	598	1157
16	143	823	1493
32	259	1136	1681

Без объединения запросов

Количество потоков

Глубина очереди	1	8	32
1	8.95	67.6	227
2	17.8	128	360
4	34.3	226	569
8	64	359	878
16	106	556	1248
32	149	830	1563

С объединением запросов

Количество потоков

Глубина очереди	1	8	32
1	27.3	162	387
2	52.1	285	634
4	66.7	420	891
8	106	603	1172
16	159	830	1469
32	241	1115	1685

С адаптивным объединением

Рис. 12: Результаты тестирования производительности (Мбайт/с) случайной записи для размера блока 4 килобайта

Количество потоков

Глубина очереди	1	8	32
1	29	206	550
2	53.1	339	806
4	70.1	453	1018
8	104	581	1203
16	154	760	1473
32	233	988	1742

Без объединения запросов

Количество потоков

Глубина очереди	1	8	32
1	22.4	165	477
2	26.2	187	567
4	33.6	238	709
8	46.9	314	875
16	592	3363	5594
32	661	4306	5582

С объединением запросов

Количество потоков

Глубина очереди	1	8	32
1	30.5	192	608
2	53.2	310	868
4	79.5	445	1018
8	106	556	1219
16	583	3255	5417
32	660	4319	5304

С адаптивным объединением

Рис. 13: Результаты тестирования производительности (Мбайт/с) последовательной записи для размера блока 4 килобайта

Для сложной нагрузки, тестируемой через VDbench (рис. 14), мы можем также видеть, что управляющий алгоритм выбирает верную стратегию и не объединяет запросы во всех тестах.

	Без объединения запросов	С объединением запросов	С адаптивным объединением
Neoclassic	2021.61	1722.22	1988.38
Four	972.53	714.67	921.06
Sixty Four	1386.13	1188.56	1466.78
Four+Largeseq	2668.43	2457.04	2677.37
Four+Neoseq	1368.54	1254.06	1350.05
База данных	1029.37	789.37	1012.23
Файловый сервер	2065.74	1667.17	1910.22
VDI	440.86	249.66	434.33
Веб-сервер	5427.48	5361.73	5314.09

Рис. 14: Результаты тестирования производительности (Мбайт/с) для сложных паттернов нагрузки

6.2. Тестирование с изменяющейся нагрузкой

Для того, чтобы продемонстрировать, насколько быстро управляющая подсистема адаптируется под изменяющуюся нагрузку, был проведен тест при помощи инструмента FIO, в котором каждые 30 секунд

изменялся характер нагрузки: с 1 по 30 секунды последовательная запись с глубиной очереди 32, с 30 по 60 секунды случайная запись с глубиной очереди 64, с 60 по 90 секунды последовательная запись с глубиной очереди 16, с 90 по 120 секунды комбинация чтения и записи в соотношении 70 к 30 с глубиной очереди 16. Показатели производительности собирались с блочного устройства каждую секунду при помощи утилиты blktrace.

Представленный на рис. 15 график изменения производительности показывает, что управляющий алгоритм тратит около 15 секунд на то, чтобы определить необходимость объединения запросов, а в случае, когда объединение запросов не нужно, адаптируется почти мгновенно.

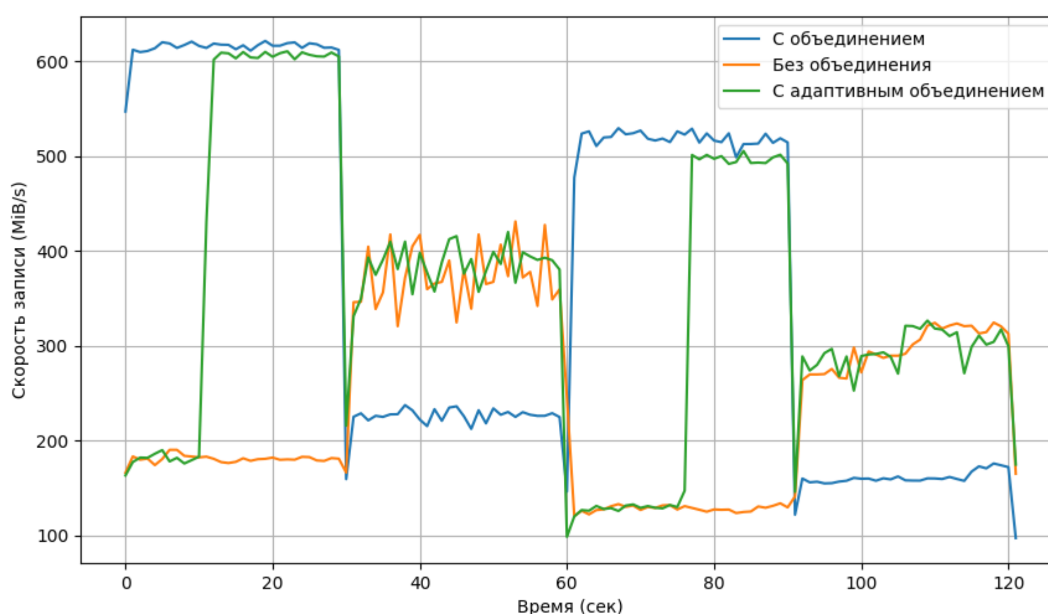


Рис. 15: Тестирование производительности при изменяющейся нагрузке в области

Заключение

В ходе данной работы были получены следующие результаты.

- Выполнен обзор существующих решений для обнаружения последовательных запросов в различных технологиях СХД.
- Проведено сравнение работы системы RAIDIX ERA в режиме объединения последовательных запросов и без него.
- Разработан алгоритм, управляющий объединением запросов. Алгоритм основывается на использовании битовых карт.
- Управляющий алгоритм реализован на языке C и внедрен в систему RAIDIX ERA.
- Производительность алгоритма протестирована с помощью стандартных инструментов тестирования блочных устройств FIO и Oracle VDbench.

Список литературы

[1] Abhishek Singhal Rob Van der Wijngaart Peter Barry. Atomic Read Modify Write Primitives for I/O Devices. — Intel White Paper. — 2008. — URL: https://pdfs.semanticscholar.org/0650/da144a2200e62af4efbfb84c58116c5d65cf.pdf (online; accessed: 08.05.2019).

[2] Complete Fairness Queueing. — URL: https://www.kernel.org/doc/Documentation/block/cfq-iosched.txt (online; accessed: 08.05.2019).

[3] Deadline Task Scheduling. — URL: https://www.kernel.org/doc/Documentation/scheduler/sched-deadline.txt (online; accessed: 08.05.2019).

[4] EMC Education Services. От хранения данных к управлению информацией. — 2016.

[5] Jaehong Kim Sangwon Seo Dawoon Jung Jin-Soo Kim, Huh Jaehyuk. Parameter-Aware I/O Management for Solid State Disks (SSDs). — 2010. — URL: http://csl.skku.edu/papers/CS-TR-2010-329.pdf (online; accessed: 08.05.2019).

[6] Kyber I/O scheduler. — URL: https://github.com/torvalds/linux/blob/master/block/kyber-iosched.c (online; accessed: 08.05.2019).

[7] Matias Bjørling Jens Axboe† David Nellans† Philippe Bonnet. Linux Block IO: Introducing Multi-queue SSD Access on Multi-core Systems. — 2013. — URL: http://kernel.dk/blk-mq.pdf (online; accessed: 08.05.2019).

[8] NVM Express Base Specification. — 2018. — URL: https://nvmexpress.org/wp-content/uploads/NVM-Express-1_3c-2018.05.24-Ratified.pdf (online; accessed: 08.05.2019).

[9] Oracle Vdbench. — URL: https://www.oracle.com/technetwork/server-storage/vdbench-downloads-1901681.html (online; accessed: 08.05.2019).

[10] RAIDIX ERA. — URL: https://www.raidix.ru/products/era/ (online; accessed: 08.05.2019).

[11] Robert Love. Linux Kernel Development, Third Edition. — 2010.

[12] SNIA. Common RAID Disk Data Format Specification. — 2008. — URL: https://www.snia.org/sites/default/files/SNIA_DDF_Technical_Position_v2.0.pdf (online; accessed: 08.05.2019).

[13] SNIA Emerald Power Efficiency Measurement Specification. — URL: https://www.snia.org/sites/default/files/technical_work/Emerald/SNIA_Emerald_Power_Efficiency_Measurement_Specification_V3_0_3.pdf (online; accessed: 08.05.19).

[14] Smirnov Dmitrii. RAIN: Reinvention of RAID for the World of NVMe // FMS 2018. — 2018. — URL: https://www.flashmemorysummit.com/English/Collaterals/Proceedings/2018/20180808_NVME-202-2_Smirnov.pdf (online; accessed: 08.05.2019).

[15] Svetlana Lazareva Ilia Demianenko. Automatic request analyzer for QoS enabled storage system // CEE-SECR '15 Proceedings of the 11th Central Eastern European Software Engineering Conference in Russia. — 2015. — URL: https://dl.acm.org/citation.cfm?id=2855670 (online; accessed: 08.05.19).

[16] Xuanhua Shi Ming Li Wei Liu Hai Jin Chen Yu Yong Chen. SSDUP: A Traffic-Aware SSD Burst Buffer for HPC Systems // Proceedings of ICS '17, Chicago, IL, USA, June 14-16, 2017, 1. — 2017. — URL: http://grid.hust.edu.cn/xhshi/paper/ssdup-ics.pdf (online; accessed: 08.05.2019).

[17] fio - Flexible I/O tester // github. — 2018. — URL: https://github.com/axboe/fio (online; accessed: 08.05.2019).

[18] А. С. Бутрова. Алгоритмы предварительной подгрузки для рабочей нагрузки с большой долей случайных запросов в СХД // СПбГУ. Курсовая работа. — 2016. — URL: http://se.math.spbu.ru/SE/YearlyProjects/spring-2016/index_html (online; accessed: 08.05.2019).

[19] И. И. Демьяненко. Разработка модуля обнаружения последовательных запросов в системах хранения данных с блочным доступом // СПбГУ. Дипломная работа. — 2015. — URL: http://se.math.spbu.ru/SE/diploma/2015 (online; accessed: 08.05.2019).

[20] Р. В. Климов. Исследование методов оптимизации нагрузки восстановления распределенных систем хранения данных на базе корректирующих кодов. — URL: https://www.psuti.ru/ru/science/dissertation_councils/announcements/klimov-roman-vladimirovich-dissertaciya-na-soiskanie (online; accessed: 08.05.19).