Министерство науки и высшего образования Российской Федерации

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ЯДЕРНЫЙ УНИВЕРСИТЕТ «МИФИ» (НИЯУ МИФИ)

ИНСТИТУТ ИНТЕЛЛЕКТУАЛЬНЫХ КИБЕРНЕТИЧЕСКИХ СИСТЕМ
КАФЕДРА КИБЕРНЕТИКИ

На правах рукописи

УДК 004.912

ЕРОФЕЕВ В.С.

РАЗРАБОТКА ПРОГРАММНЫХ СРЕДСТВ ДЛЯ ОПТИМИЗАЦИИ ВЗАИМОДЕЙСТВИЯ С ФАЙЛОВОЙ СИСТЕМОЙ

Выпускная квалификационная работа бакалавра

Направление подготовки 09.03.04 Программная инженерия

Выпускная квалификационная
работа защищена
«___»_____20 г.
Оценка _____
Секретарь ГЭК _____

г. Москва

2021

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ
УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
Национальный исследовательский ядерный университет «МИФИ»

Институт
интеллектуальных кибернетических систем

Кафедра №22 «Кибернетика»

Направление подготовки 09.03.04 Программная инженерия

Пояснительная записка

к ВКР на тему:

Разработка программных средств для оптимизации взаимодействия с файловой системой

Группа Б17-514

Студент _____ (подпись) Ерофеев В.С. (ФИО)

Руководитель _____ (подпись) Маренков А.В. (ФИО)

Научный консультант _____ (подпись) _____ (ФИО)

Москва 2021

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

Национальный исследовательский ядерный университет «МИФИ»

Институт	Интеллектуальных кибернетических систем	Кафедра	№22 «Кибернетика»
		Группа	Б17-514
Специальность (направление)	Программная инженерия (09.03.04)	«Утверждаю» Зав. кафедрой	
		_____ (подпись)	Загребаев А.М.
		«___» _____» 20__г.	

ЗАДАНИЕ НА ДИПЛОМНУЮ РАБОТУ

(выпускную квалификационную работу ВКР)

1. Фамилия, имя, отчество студента
   Ерофеев Владимир Сергеевич _____
   (ФИО) (подпись)

2. Тема работы (ВКР) « Разработка программных средств для оптимизации взаимодействия с файловой системой »

3. Срок сдачи студентом готовой работы: 11 июня 2021 г.

4. Место выполнения 22 кафедра

5. Руководитель работы
   Маренков Александр Владимирович | ассистент | _____
   (ФИО) (уч. степень, должность) (подпись)

6. Соруководитель работы от НИЯУ МИФИ
   _____ _____ _____
   (ФИО) (уч. степень, должность) (подпись)

7. Консультант работы
   _____ _____ _____
   (ФИО) (уч. степень, должность) (подпись)

1. Аналитическая часть

1.1. Изучение и анализ научной литературы в области бенчмаркинга файловых систем с целью выделения рекомендаций по построению бенчмарков.
1.2. Анализ современных бенчмарков с целью выделения сильных и слабых сторон для последующего учета в разработке.
1.3. Анализ факторов, влияющих на производительность файловых систем, применительно к оптимизации взаимодействия с ними.

2. Теоретическая часть

2.1. Описание сценария использования разрабатываемого ПО как метода решения задачи оптимизации
2.2. Анализ требований к ПО
2.3. Описание модели паттерна нагрузки
2.4. Описание метода обеспечения идентичных условий запуска тестов
2.5. Описание метода обеспечения достижения системой стабильности при тестировании
2.6. Описание метода реализации факторов, влияющих на производительность файловых систем
2.7. Выбор целевой метрики производительности
2.8. Описание модели выбранной целевой метрики производительности
2.9. Описание метода повышения точности результатов

3. Технологическая часть

3.1. Разработка общего алгоритма работы бенчмарка
3.2. Проектирование тестирующего компонента бенчмарка
3.3. Проектирование систематизирующего компонента бенчмарка

4. Практическая часть

4.1. Реализация тестирующего компонента бенчмарка
4.2. Реализация автоматических тестов для процесса снятия измерений
4.3. Реализация систематизирующего компонента бенчмарка
4.4. Реализация интерфейсов ввода и вывода бенчмарка
4.5. Реализация компонента для построения графиков
4.6. Тестирование бенчмарка и компонента для построения графиков

Дата выдачи задания «06» февраля 2021 г.

Реферат

Пояснительная записка содержит 57 страниц с учетом приложения, 11 рисунков, 4 листинга кода.

Количество использованных источников – 34.

Ключевые слова – бенчмарк, файловая система, тестирование, оптимизация.

Целью данной работы является разработка исследовательского программного обеспечения для оптимизации взаимодействия с файловой системой.

В первом разделе приводится анализ научных исследований в области бенчмаркинга файловых систем и современных бенчмарков, а также рассматриваются факторы, влияющие на производительность файловых систем. По результатам анализа приводятся выводы применительно к разработке собственного бенчмарка.

Во втором разделе описываются результаты анализа требований к разрабатываемым программным средствам, производится теоретическая проработка используемых моделей и методов, а также осуществляется проектирование бенчмарка.

В третьем разделе рассматриваются результаты реализации программных средств и приводится отчет о тестировании.

Содержание

Введение

На сегодняшний день распространено большое число операционных систем, каждая из которых позволяет более удобно решать отдельные спектры задач. По этой причине все большее число пользователей приходят к необходимости иметь под рукой сразу несколько установленных операционных систем. Есть несколько вариантов решения проблемы, один из которых – использовать виртуальную машину.

Виртуальная машина – это программный компьютер, который предоставляет все возможности физических компьютеров [1]. Иными словами, это программа, которая выполняет роль физического компьютера для другой программы. Известны такие реализации виртуальных машин как VirtualBox, Parallels Desktop, и другие. Виртуальные машины позволяют внутри одной операционной системы установить еще несколько операционных систем и работать с ними одновременно.

Одним из основных критериев качества виртуальной машины является производительность. Она определяет область ее применения: например, для запуска современных игр нужна высокая производительность; а также влияет на пользовательский опыт: медленно работающая программа может вызвать недовольство пользователя. Поэтому вполне актуальной является проблема повышения производительности виртуальных машин.

В зависимости от решаемой пользователем задачи узким местом для виртуальной машины может стать работа с памятью, или, конкретнее, взаимодействие с файловой системой [2]. С одной стороны, причиной этому является то, что накопители данных в принципе медленнее процессоров, а с другой – то, что некоторые современные файловые системы являются проприетарными и содержат неизвестные внешнему пользователю оптимизации, которые могут существенно влиять на производительность. Однако закрытость таких систем не препятствует их изучению путем тестирования методом «черного ящика». Именно за счет этого мы предполагаем исследовать поведение файловых систем, после чего определять, при каких параметрах взаимодействия с ними производительность наиболее высока. Для реализации данного подхода необходимо разработать ПО, которое позволило бы проводить эксперименты над файловыми системами и находить оптимальные параметры взаимодействия. Таким образом, целью данной работы является разработка программных средств для оптимизации взаимодействия с файловой системой.

В первом разделе пояснительной записки будет проведен анализ научных исследований в области бенчмаркинга файловых систем и современных бенчмарков. В результате будут сделаны выводы применительно к разработке собственного ПО. Также в

разделе будут разделены на категории и рассмотрены факторы, которые влияют на производительность файловых систем.

Во втором разделе будет осуществлен анализ требований, проведена теоретическая проработка реализуемых в ПО моделей и методов, а также произведено проектирование бенчмарка как части разрабатываемого программного комплекса.

В третьем разделе будут приведены результаты реализации ПО с описанием полученных алгоритмов работы и результаты тестирования.

1. Анализ инструментов бенчмаркинга файловых систем и влияющих на их производительность факторов

Профилирование файловой системы, заключающееся в измерении ее производительности при разных условиях работы, возможно только в случае, когда система находится в действующем состоянии, то есть исполняет некоторую нагрузку или, проще говоря, решает некоторую задачу. Чтобы получившиеся измерения были сравнимы, нагрузка во всех случаях должна быть одной и той же. Здесь на помощь приходят бенчмарки.

Бенчмарк файловой системы – контрольная задача, использующаяся для измерения производительности. Существует значительное число бенчмарков разных видов. Мы их рассмотрим и сделаем выводы о том, каким должен быть разрабатываемый нами бенчмарк. Также в данном разделе будут выделены и проанализированы факторы, влияющие на производительность файловых систем, с целью выбора наиболее пригодных для целей оптимизации.

1.1. Анализ научных исследований и современных разработок в области бенчмаркинга файловых систем

1.1.1. Анализ бенчмаркинга конца 90-х и начала 2000-х

Бенчмаркинг файловых систем был особенно популярной темой для научных статей в конце 90-х и начале 2000-х годов. Инструменты того времени имели множество недостатков, что отобразилось в нескольких трудах по их анализу.

В работе A. Traeger'a и др. были изучены 106 релевантных статей и проанализированы положительные и отрицательные стороны упомянутых в них бенчмарков [3]. Была выявлена следующая категоризация:

- *Макробенчмарки* — бенчмарки, нацеленные на имитацию реальной нагрузки путем выполнения большого количества различных операций;
- *Воспроизводящие бенчмарки* — бенчмарки, которые в качестве теста воспроизводят нагрузку, полученную в ходе какой-либо реальной работы;
- *Микробенчмарки* — бенчмарки, исполняющие лишь очень узкий набор операций (обычно 1-2).

Для получения наилучших результатов оценки производительности эффективным будет создание системы комплексной оценки производительности за счет использования результатов бенчмарков различных категорий.

Также были выделены несколько существенных факторов, влияющих на результаты:

- *Состояние кэша.* Кэш бывает «холодный» или «прогретый», т. е. очищенный и только что запущенный или уже заполненный данными и выполнивший достаточное для стабильной работы число операций;
- *Зональная постоянная угловая скорость (ZCAV)*, означающая хранение дисковыми накопителями большего количества секторов на внешних дорожках, чем на внутренних. В современном мире встречается на большинстве жестких дисков (HDD);
- *Состаренность компьютерной системы*. Только что установленная, «чистая» файловая система будет заметно отличаться по своим показателям от системы, проработавшей несколько месяцев или лет, поэтому важно уметь «состарить» систему до состояния, соответствующего наиболее реалистичной ситуации;
- *Запущенные вторичные процессы.* Работающие в фоне не основные процессы способны заметно повлиять на результаты оценки, поэтому должны быть либо отключены, либо учтены.

Данная статья является частично устаревшей, поэтому не все ее выводы в данный момент могут быть применимы. В частности, современные твердотельные накопители (Solid-State Drive или SSD) не обладает зональной постоянной угловой скоростью, т. к. используют принципиально другой механизм хранения и обработки данных.

Для корректного бенчмаркинга рекомендуется следовать трем принципам. Во-первых, необходимо удостовериться, что все запуски проводятся в идентичных условиях. Во-вторых, для увеличения точности каждый тест должен быть запущен некоторое количество раз, определяемое либо среднеквадратичным отклонением, либо доверительным интервалом. В-третьих, тесты должны длиться достаточно долго, чтобы можно было говорить о стабильном состоянии системы на протяжении большей части времени.

При представлении результатов следует использовать доверительные интервалы. Для их вычисления важно использовать распределение Стьюдента, а не нормальное, если количество запусков каждого отдельного теста меньше 30. Чтобы обеспечить воспроизводимость эксперимента, рекомендуется четко фиксировать состояние всех известных при проведении тестов параметров системы и запуска.

Применительно к нашей работе, мы можем использовать предложенную категоризацию при разработке собственного бенчмарка, а также учитывать все три указанные принципа бенчмаркинга. Использовать доверительные интервалы при представлении результатов может оказаться неудобным для глаза и неоправданно сложным в реализации, но данную рекомендацию также стоит учесть. Важными являются советы

использовать распределение Стьюдента и фиксировать состояние всех известных при проведении тестов параметров системы и запуска. Первый из них мы постараемся учесть. Второй переработаем в вывод о том, что для корректного сравнения и агрегации любых тестовых результатов, а также повышения точности их дальнейшей интерпретации следует по возможности максимально фиксировать состояние среды и других оказывающих влияние на производительность условий или параметров работы, а также стремиться к тому, чтобы они совпадали при всех тестовых запусках, которые затем будут сравниваться между собой или объединяться. Указанные факторы влияния на производительность, помимо ZCAV, следует рассмотреть.

1.1.2. Задача-ориентированный бенчмаркинг

Заслуживающие внимания идеи по внедрению новых подходов к построению инструментов бенчмаркинга были предложены в труде M. Seltzer и других [4]. Представлены три методологии бенчмаркинга: векторная, воспроизводящая и гибридная. В каждой из них производительность измеряется относительно конкретной задачи.

В рамках векторной методологии оценка производительности получается в результате скалярного произведения двух векторов: системного вектора и вектора задачи. Системный вектор хранит данные о результатах проведения микробенчмарков, характеризующих производительность базисных операций (например, последовательной записи или случайного чтения). Вектор задачи определяется весовыми коэффициентами, соответствующими базисным операциям.

Воспроизводящая методология применима тогда, когда нагрузка задачи по базисным операциям зависит от входных данных. В таком случае фиксируется профиль нагрузки и генерируется соответствующая стохастическая нагрузка, которая в дальнейшем используется как бенчмарк воспроизводящего типа.

Гибридная методология совмещает в себе лучшее из двух предыдущих подходов. Оценка производительности определяется так же как в векторной методологии, и системный вектор остается без изменений, но для вычисления вектора задачи анализируется профиль ее нагрузки и соответственно полученным данным проставляются весовые коэффициенты.

Заметим, что гибридная методология содержит один мало продуманный аспект. Она не учитывает ситуацию, когда вектор задачи не сможет адекватно представить нагрузку. В таком случае точность оценки производительности окажется снижена. В качестве примера, представим следующую ситуацию. Файловая система имеет 4 базисные операции – последовательное чтение, последовательная запись, случайное чтение, случайная запись. Оцениваемая нагрузка представляет собой случайно смешанный набор из данных четырех

операций. Даже зная доли каждой из них, вектор задачи не сможет отразить ее смешанный характер. По этой причине важно обращать внимание на выбор базисных микробенчмарков для системного вектора при использовании методологии. Однако предложенную гибридную методолгию мы примем к сведению.

1.1.3. Гибридная методология: формулировка задач и требований к бенчмарку

Реализация бенчмарка по гибридной методологии была осуществлена в труде D. Tang [5]. Рассмотрим, как для него разрабатывались требования. Изначально производится деление бенчмарков на категории, близкие к уже нами изученным:

- *По типу нагрузки*: синтетические бенчмарки и бенчмарки приложений;
- *По масштабу*: макро- и микробенчмарки.

Бенчмарки приложений состоят из программ и утилит, которые в действительности используются пользователями. Например, компилятор C или интерпретатор Lisp. В то же время синтетические бенчмарки генерируют искусственную нагрузку и оценивают производительность системы по ней. Они хорошо масштабируются, но хуже оценивают реальную производительность.

Макробенчмарки оценивают систему в целом и могут использовать нагрузку любого типа. Микробенчмарки вместо реальной нагрузки моделируют лишь очень узкие ее сегменты. Одной из сложностей, с которой сталкиваются пользователи инструментов последнего типа, является легкость некорректной интерпретации результатов. Например, рост показателей по одному из тестов может расцениваться как успех оптимизации, однако при этом не будут учтены другие тесты либо аспекты системы, которые не оцениваются в рамках бенчмарка, хотя их производительность упадет. С другой стороны, микробенчмарки очень удобны для точечного выделения слабых сторон в файловой системе.

Приведем краткие выводы по данным категориям. В идеале, для оптимизации программы нужно запускать соответствующий ей бенчмарк приложения. Однако решение данной задачи потребует снятия профиля нагрузки программы и построения воспроизводящего бенчмарка по нему. Во-первых, некоторые программы обладают динамическим, т. е. изменяющимся со временем и в зависимости от входных факторов, профилем нагрузки, поэтому зафиксировать его однозначно невозможно. Во-вторых, задачи снятия профиля нагрузки программы и построения по нему воспроизводящего бенчмарка являются крайне сложными, и их реализаций в открытом доступе обнаружено не было. Исходя из недостатка ресурсов для их самостоятельной разработки, мы вынуждены выбрать более простой путь. Мы не будем отталкиваться от оптимизируемой программы, а сконцентрируемся на исследовательских возможностях разрабатываемого ПО, что позволит с большей эффективностью тестировать и анализировать взаимодействие

с файловой системой. По этой причине выберем синтетический тип нагрузки. Адаптируя аспект масштаба к нашей работе, где на верхнем уровне в качестве оцениваемой системы выступает взаимодействие некоторой абстрактной программы с файловой системой, можно отнести разрабатываемое ПО к макробенчмаркам. Однако привычнее все же в качестве системы рассматривать файловую систему в целом, тогда бенчмарк для оценки производительности взаимодействия с ней правильнее считать микробенчмарком, поэтому мы остановимся именно на этом варианте.

Далее автор заходит издалека и задается вопросом о том, какие существуют причины использовать бенчмарки. В качестве ответа еще в 1972 H. C. Lucas сформулировал три возможных мотивации [6]:

- *Оценка выборки.* Какая система из предложенных для меня лучше всего подойдет?
- *Мониторинг производительности.* Как я могу настроить имеющуюся систему, чтобы улучшить ее производительность?
- *Прогноз производительности.* Как хорошо покажет себя данная идея для системы?

И заключил, что в первом случае бенчмарки очень удобны, во втором — приемлемы, а для третьего — непригодны. Продолжая его рассуждения, D. Tang выделяет две группы пользователей: покупатели, которым интересны макробенчмарки приложений, и архитекторы систем, которые предпочтут комбинацию синтетических микро- и макробенчмарков в силу их большей гибкости и настраиваемости.

При формулировке требований к разрабатываемому бенчмарку автор использует в качестве базиса работу P. M. Chen [7], в которой выделен следующий набор критериев правильной оценки систем ввода-вывода. Бенчмарк должен быть

- *Предписывающим*, т. е. указывать системным архитекторам на места для улучшения;
- *Ограниченным только вводом-выводом.* Бенчмарк должен тестировать только систему ввода-вывода, но не, например, процессор;
- *Масштабируемым* с развитием технологий*;*
- *Сравниваемым* при применении на разных системах*;*
- *Универсальным,* т. е. применимым для моделирования широкого спектра нагрузок;
- *Четко специфицированным:* во всех аспектах должна быть абсолютная ясность.

Чтобы применить данные критерии к оценке файловых систем достаточно лишь уточнить второй пункт: бенчмарк должен быть ограниченным только файловой системой.

Завершающей проблемой при выработке требований к бенчмарку является выбор метрики оценки системы. Самыми частыми являлись на момент написания работы и являются до сих пор пропускная способность и задержки при выполнении операций.

Мы примем к сведению указанные варианты метрик производительности и переймем себе следующие требования: разрабатываемый бенчмарк должен быть ограниченным только файловой системой и масштабируемым.

1.1.4. Гибридная методология: разделение на набор микробенчмарков и характеризатор нагрузки на примере инструмента dtangbm

В качестве реализации бенмчарка по описанным выше требованиям (см. раздел 1.1.3) D. Tang представила инструмент dtangbm [5]. Согласно авторскому подходу, при проектировании бенчмарка целесообразным является его разделение на две части: набор микробенчмарков и характеризатор нагрузки, каждая из которых собирает некоторый набор статистик. Первый из них описывает файловую систему, а второй — целевую нагрузку. Используя полученные данные, можно оценить производительность файловой системы. Чтобы повысить интерпретируемость результатов тестирования, набор микробенчмарков должен обладать наибольшей полнотой.

Dtangbm делится на набор микробенмчарков, называемый fsbench, и характеризатор нагрузки. Fsbench имеет 4 фазы, основными из которых являются

- *опциональная фаза начальных измерений*, в рамках которой определяются пропускная способность и время записи и чтения, обусловленные аппаратной составляющей устройства;
- *обязательная фаза начальных измерений*, оценивающая размер буферного кэша, кэша атрибутов и кэша транслированных имен;
- *фаза микробенчмарков*, собирающая статистику по производительности для будущего применения совместно с характеризатором нагрузки;

Характеризатор нагрузки представляет собой Perl скрипт, который принимает входные данные в виде результата использования модифицированной утилиты мониторинга nfswatch, и анализирует его, генерируя следующие статистики:

- процент вызовов операций read, write, create, delete, mkdir, getattr, lookup и других;
- процент последовательных и случайных чтений;
- процент последовательных и случайных записей;
- среднее число открытых файлов и его среднеквадратичное отклонение.

В рамках разработанного бенчмарка общая оценка производительности вычисляется путем, аналогичным используемому при определении среднего числа циклов в процессорной инструкции. В этом случае число циклов, необходимое для конкретной инструкции, умножается на частоту появления этой инструкции, а затем суммируется по всем инструкциям. Переводя в термины гибридной методологии, производительность

базисной операции умножается на частоту ее появления в целевой задаче и суммируется по всем базисным операциям.

В конечном итоге dtanbgm предназначен для решения задачи оценки выборки, т. е. выбора оптимальной системы из набора предложенных. Нашей же целью при построении бенчмарка является мониторинг производительности. Таким образом, dtangbm представляет интерес для изучения лишь в целях знакомства с авторской методологией.

1.1.5. Обзор бенчмарка файловых систем CrystalDiskMark

Рассмотрим CrystalDiskMark версии 7.0.0, 2020 года выпуска [9]. Он содержит тесты на скорость записи и чтения по 4 паттернам:

- SEQ1M Q8T1 - последовательный доступ с 8 очередями (Queues), 1 потоком и размером блока 1 МБ;
- SEQ1M Q1T1 - последовательный доступ с 1 очередями, 1 потоком и размером блока 1 МБ;
- RND4K Q32T16 - случайным доступ с 32 очередями, 16 потоками и размером блока 4 КБ;
- RND4K Q1T1 - случайный доступ с 1 очередью, 1 потоком и размером блока 4 КБ.

После знакомства с ними возникает вопросы, касающиеся параметра Queues и содержания тестов, но они остаются нерешенными в силу отсутствия какой-либо документации.

Присутствует возможность определить профиль бенчмарка:

- *Стандартный;*
- *Пиковая производительность;*
- *Реальная производительность;*

и добавить смешанную нагрузку с указанным соотношением операций записи и чтения. Что означает каждый профиль не объясняется.

Из плюсов можно выделить настраиваемость большинства параметров:

- *Количество запусков* определяет, сколько раз будет запущен каждый отдельный тест;
- *Размер теста.* Может быть от 16 МиБ до 64 ГиБ;
- *Единица измерения.* Доступными опциями являются MB/s, GB/s, IOPS, µs.
- *Вид тестовых данных.* Можно заполнить содержимое либо случайными данными, либо значениями 0x00;
- *Интервал времени.* Градация от 0 секунд до 10 минут;
- *Глубина очереди* (Queues). В степенях двойки от 1 до 32;

- *Количество потоков*. От 1 до 64.

Результаты тестирования разделены по паттернам нагрузки: чтение, запись, комбинация чтения и записи. Визуализация результатов представлена в виде заполняющейся линейной шкалы на фоне каждого отображаемого результата, что не несет полезной смысловой нагрузки. При повторном проведении тестов предыдущие значения метрик теряются, однако в качестве выхода из положения можно воспользоваться возможностью сохранить результаты в отдельный файл.

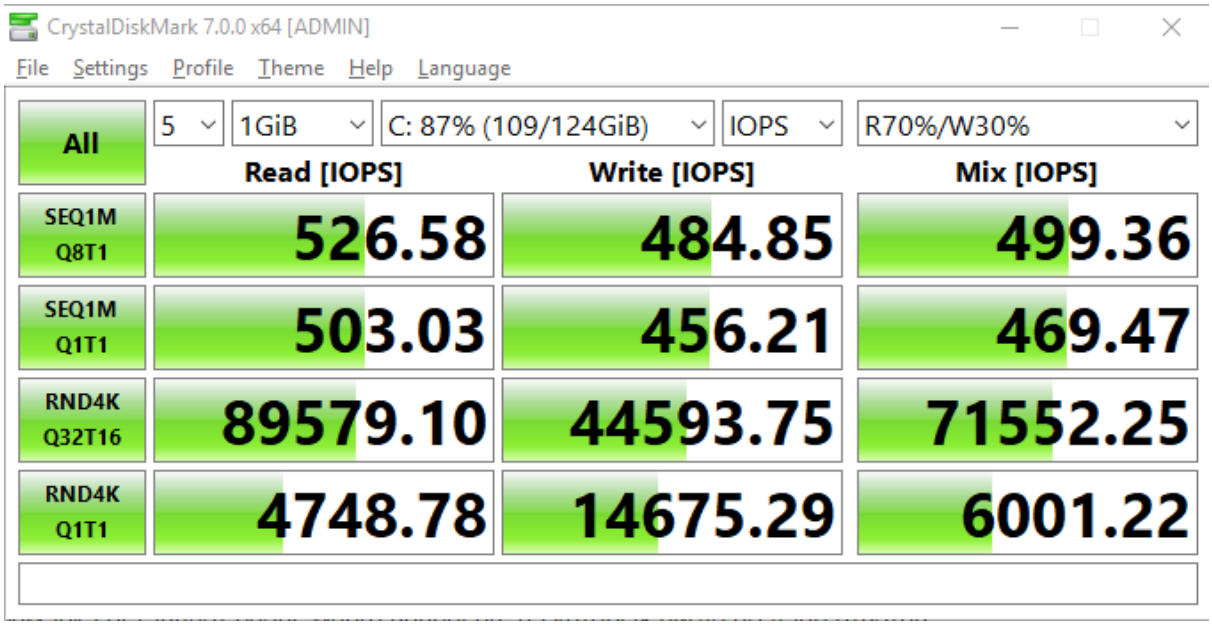


Рис. 1 – Бенчмарк CrystalDiskMark v7.0.0

В качестве сильных сторон бенчмарка выделим достаточно неплохие возможности настройки и удобный интерфейс. Учтем при проектировании собственного бенчмарка увиденные здесь факторы нагрузки и идею систематического тестирования с разделением на несколько микробенчмарков (паттернов нагрузки).

1.1.6. Обзор бенчмарка файловых систем IOzone

IOzone это кроссплатформенный бенчмарк с открытым исходным кодом для широкопрофильного тестирования файловых систем [10]. Он представляет собой консольное приложение, которое способно оценивать производительность операций последовательных записи, чтения, записи в уже созданный файл, повторного чтения, случайных записи и чтения, и нескольких других.

Из настраиваемых параметров можно выбрать принудительную очистку кэша, размеры блока и файла при тестировании, а также количество запущенных потоков либо процессов и ряд других факторов, менее интересных в рамках данного исследования. Однако функции “разогрева” для стабилизации нагрузки перед снятием показателей не предусмотрено. Результаты тестов выводятся в стандартный вывод.

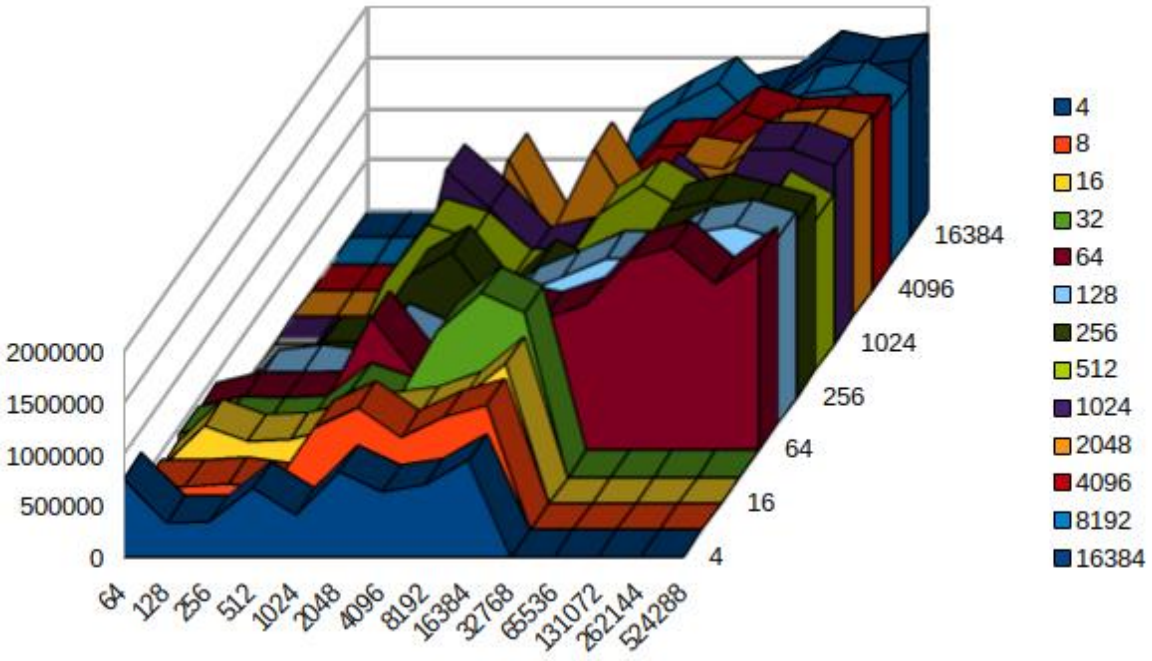


Рис. 2 – Визуализация результатов запуска бенчмарка IOzone

В ходе тестирования оценивается производительность при варьировании параметров размера блока и размера файла, что помещает получаемые результаты в трехмерную плоскость. Несмотря на то, что инструмент не содержит встроенного генератора графиков, в нем реализована компенсирующая опция, позволяющая сгенерировать отчет, совместимый с Excel, с помощью которого можно легко представить полученные данные в виде многомерной диаграммы. Результат работы представляется в виде пропускной способности, выраженной в килобайтах в секунду (КБ/с).

Применительно к целям данной работы, IOZone демонстрирует неплохие возможности систематического тестирования, которые стоит расширить до варьирования и других факторов. Полученная с помощью Excel по результатам бенчмаркинга диаграмма является простым примером того, как можно визуализировать результаты, который мы учтем.

1.1.7. Обзор инструмента ATTO Disk Benchmark

Одним из широко известных инструментов оценки производительности для Windows является Disk Benchmark, разработанный компанией ATTO [11]. Он обладает приятным графическим интерфейсом, чем напоминает CrystalDiskMark. Также у них есть и общая негативная сторона: отсутствие полноценной документации, хотя в Disk Benchmark есть краткое описание варьируемых параметров и получаемых результатов. Однако после его прочтения остается много открытых вопросов. В данном бенчмарке присутствуют всего 2 типа тестируемой нагрузки: запись и чтение, но ни одна из них в деталях не описана, поэтому невозможно в точности понять, что именно и каким образом тестируется.

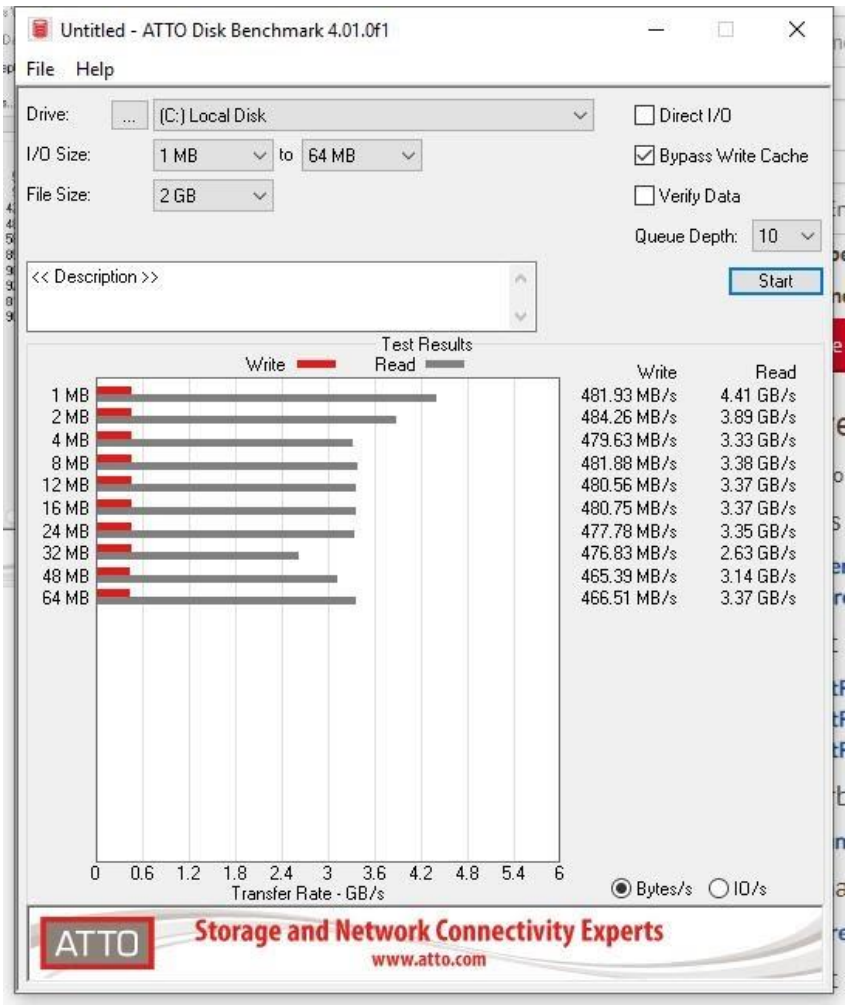


Рис. 3 – Бенчмарк ATTO Disk Benchmark

Суть тестирования заключается в варьировании размера блока в указанном диапазоне и построении по полученным результатам сравнительной диаграммы, что напоминает графики, которые строятся на основе отчетов в IOzone. Однако Disk Benchmark уступает в размерности результирующего пространства, что является существенным недостатком.

В качестве определяемых опций доступны выбор размера файла тестирования, пропуск кэша при записи, прямой ввод-вывод, функция верификации данных, а также указание размера очереди. Вновь отсутствует возможность “разогрева” перед снятием показателей. В качестве единицы измерения результатов можно выбрать либо байт в секунду (Б/с), которая автоматически будет приведена к МБ/с или ГБ/с, либо операций ввода-вывода в секунду (IO/s или IOPS). Демонстрируют ли данные показатели средний или пиковый результат, неясно.

Disk Benchmark привлекает своим интерфейсом, но как инструмент оценки производительности предоставляет достаточно скудные возможности. Типов тестируемой нагрузки всего 2 и они фиксированы, а также нигде в деталях не описаны, как и результирующие метрики. Еще одним существенным недостатком является отсутствие

возможности выбора параметров, варьируемых в ходе тестирования. Таким образом, в качестве конкурентноспособного преимущества Disk Benchmark обладает только графическим интерфейсом.

1.1.8. Другие бенчмарки и исследования

В ходе работы также были изучены дополнительная литература [5-7] и бенчмарки IOMeter [12] и Filebench [13]. Результаты их анализа не позволили сделать существенных выводов, поэтому в данном разделе не приводятся.

1.2. Анализ факторов, влияющих на производительность файловых систем

Нашей главной целью является оптимизация взаимодействия с файловой системой. Для ее достижения мы разрабатываем исследовательское ПО, с помощью которого будет удобно тестировать взаимодействие с файловой системой и анализировать результаты. Одной из его компонент будет являться бенчмарк, предназначенный для тестирования взаимодействия с файловой системой при разных уровнях факторов. В данном разделе описаны результаты выделения таких факторов, их категоризации и анализа на применимость в бенчмарке.

1.2.1. Категоризация факторов влияния

Множество различных факторов может оказывать влияние на производительность файловых систем. Их можно разделить на следующие группы:

- *Фиксированные факторы*. Сюда входят: используемая аппаратура, операционная система и параметры, определяемые при ее настройке, а также некоторое системное программное обеспечение (например, драйверы);
- *Изменяемые параметры операционной и файловой систем* включают те опции, которые могут быть изменены в ходе работы (например, размер блока);
- *Тестируемый паттерн использования файловой системы* подразумевает то, какой характер имеет нагрузка (например, чтение со случайным паттерном доступа к памяти);
- *Харатеристики нагрузки* – параметры нагрузки, которые можно изменять, не меняя тестируемого паттерна нагрузки (например, средний размер запроса).

Исходя из целей работы, для добавления в бенчмарк нам необходимо выделить те факторы, которые возможно и целесообразно изменять изнутри программы, либо те, которые необходимо учитывать при оптимизации.

Таким образом, аппаратные характеристики устройства, драйверы и неизменяемые параметры операционной системы исключаются в силу невозможности их программно

изменить. Однако их все же следует учесть при анализе результатов, так как некоторые явления резких изменений производительности могут быть ими обусловлены. В частности, особый интерес представляют модель накопителя данных и его характеристики, т. к. он может содержать внутренние оптимизации (например, DRAM-буфер), и оперативная память, используемая для кэширования.

Тестируемый паттерн использования файловой системы (паттерн нагрузки) зависит от решаемой программой задачи и не может быть изменен пользователем, но оказывает крайне существенное влияние на производительность. По этой причине исследовать взаимодействие с файловой системой стоит с учетом паттерна нагрузки, но рассматривать его стоит как контролируемый, но неуправляемый фактор.

Параметры, характеризующие нагрузку, по определению могут быть изменены внутри программы, поэтому представляют наибольший интерес. В качестве дополнения к ним могут выступить изменяемые параметры операционной и файловой систем. Обе категории факторов стоит рассмотреть и выделить те из них, которые изменять целесообразно.

1.2.2. Тестируемые паттерны использования файловой системы

Уточним, что понимается под паттерном нагрузки. Пусть есть некоторая программа, которая обращается к ФС. Обращение к файловой системе характеризуется двумя параметрами: операцией – чтения или записи (есть и другие операции, но в данной работе мы не будем их рассматривать) и паттерном доступа к памяти – последовательным или случайным. При последовательном доступе происходят обращения к последовательным участкам памяти. При случайном доступе – к случайным, т. е. адресно независимым, участкам памяти. Тогда нагрузка данной программы задается конечным набором обращений. Очевидно, обращение на каждом шаге задается в зависимости от текущих нужд программы и не может быть предугадано. Однако то, в каком порядке и какие именно осуществляются обращения оказывает крайне сильное влияние на производительность, поэтому иметь какое-то представление о нагрузке необходимо. Для этого мы вводим понятие паттерна нагрузки, которое означает некоторую законченную модель, стремящуюся отразить нагрузку программы за счет выделения в ней закономерностей выбора операций и паттерна доступа к памяти. Таким образом, паттерн нагрузки можно представить, например, как соотношение в нагрузке 4 возможных видов обращений: последовательной записи, последовательного чтения, случайной записи, случайного чтения.

Чаще всего для микробенчмарков используются искусственные паттерны нагрузки, получаемые путем комбинации операций чтения и записи с варьированием соотношения

между ними и формата доступа к памяти (последовательный или случайный доступ), хотя возможен и более основательный подход (см. бенчмарк dtangbm, пункт 1.1.4). Важными являются вырожденные случаи, когда тестируют, например, только операцию последовательного чтения.

В нашем случае, следует включить в бенчмарк возможность варьирования паттерна нагрузки, с целью либо приблизить его к реальному, либо сформировать набор базисных операций для применения в гибридной методологии или другой подобной. При наличии возможности, внимание следует уделить теоретической проработке модели паттерна нагрузки для повышения ее точности отражения реальных нагрузок.

1.2.3. Анализ изменяемых параметров операционной и файловой систем

Из рассмотренных бенчмарков и работ [1-7] был выделены два диапазона факторов. Первый диапазон определяет изменяемые параметры файловой системы: размер блока, состаренность, опция журналирования, опция записи времени последнего доступа (atime для Linux и last access для Windows), планировщик ввода-вывода (только Linux).

Большинство файловых систем оперируют с данными не побайтово, а блоками фиксированного размера. В таком случае любой ввод-вывод осуществляется только в выбранной размерности или кратных ей. Это может замедлить некоторые процессы. Например, для записи 256 байт при размере блока в 4 КБ потребуется провести больше работы, чем необходимо. Если подобные записи преобладают в нагрузке, то производительность будет существенно ниже. Т. к. при проведении операций ввода-вывода задействуется множество слоев стэка хранения, возможно появление аномалий в производительности в силу удачных или неудачных комбинаций параметров разных слоев. Из практического опыта известно, что размер блока является одним из ключевых факторов, обуславливающих такие комбинации, поэтому его следует учитывать при анализе. Однако программу не следует оптимизировать за его счет, т. к. это может сильно повлиять на работу остальной системы, что всегда нежелательно и в большинстве случаев недопустимо.

Состаренность файловой системы означает загрязненность дискового пространства в силу его длительного использования таким образом, что наблюдается стабильная разность в производительности по сравнению с только что установленной системой. Одним из основных факторов состаренности является фрагментация дискового пространства (появление пропусков между данными как результат многократных дописываний и удалений данных). Сделать систему «моложе» возможно только с помощью ее переустановки с форматированием диска. В рамках программного обеспечения можно только состарить ее еще больше, однако такая динамика скорее нежелательна, поэтому в бенчмарк данный фактор включать мы не будем.

Во многих файловых системах существует опция журналирования. Она заключается в записи наиболее важных или даже всех изменений в системе в журнал с целью повышения отказоустойчивости. Это дает возможность восстанавливать данные, потерянные при неудачном завершении работы. Однако пользователю необходимо регулярно платить за это цену в виде потери производительности, так как каждую операцию необходимо логировать. К моменту написания статьи во многих файловых системах есть возможность отключения опции журналирования. Данный параметр для изменения требует перемонтирования файловой системы, поэтому имеет немного более низкий приоритет включения в бенчмарк, чем более простые в изменении факторы.

В системах ext4, NTFS и некоторых других у всякого файла существует атрибут времени последнего доступа. И хотя лишь небольшое количество программ требует его постоянного обновления, по умолчанию часто устанавливается опция, к этому обязывающая. Несмотря на это, ее можно отключить в ходе работы либо переключить в оптимизирующий режим, что улучшит производительность, но приведет к невозможности работы зависимых процессов. Данная опция существенно влияет на работоспособность других приложений в системе, поэтому оптимизация за ее счет не целесообразна, а значит в бенчмарк на данном этапе ее включать нет необходимости.

Планировщик ввода-вывода это утилита операционной системы, которая группирует файловые операции, чтобы оптимизировать последующую работу накопителя данных. В операционной системе Linux его можно изменить прямо во время работы и это может существенно повлиять на производительность. Оптимизировать за счет данного параметра не рекомендуется, т. к. это может повлиять на скорость работы других процессов в системе. По этой причине мы не будем включать данный фактор в бенмчарк, хотя при возможности его следует учесть в анализе, т. к.он является одним из ключевых.

1.2.4. Анализ характеристик нагрузки

Второй диапазон выделенных факторов определяет характеристики нагрузки:

- *Средний размер запроса,*
- *Количество используемых файлов,*
- *Средний размер файла,*
- *Глубина очереди,*
- *Количество потоков\процессов,*
- *Количество одновременно открытых файловых дескрипторов,*
- *Приоритет процесса(-ов),*

- *Состояние тестируемых файлов* (например, ftruncate, непреаллоцированые, sparse файлы),
- *Состояние кэша* (очищенный, загрязненный, «разогретый», прямой ввод-вывод),
- *Используемый программный интерфейс* (например, POSIX, WinAPI, libaio).

Рассмотрим каждый параметр в отдельности.

Средний размер запроса представляет собой то количество данных, которое необходимо прочитать или записать за один запрос, то есть за один вызов соответствующей функции программного интерфейса. Фактор может крайне сильно влиять на производительность, при этом являясь относительно простым в изменении, поэтому его важно включить в бенчмарк.

Количество используемых файлов определяет то, сколько файлов будет использовано при единократном тестировании. При маленьком количестве файлов предполагается, что почти всю нагрузку будет составлять работа с содержимым. При большом количестве файлов на скорость работы начнут значительнее влиять операции над метаданными. Однако тестирование работы с несколькими файлами в большей степени расширит применимость бенчмарка, чем повлияет на возможности оптимизации, поэтому у данного фактора низкий приоритет на включение.

Средний размер файла означает то, сколько места занимают используемые при тестировании файлы в среднем. Файловые системы могут использовать оптимизации, делящие файлы по размеру на большие и маленькие, поэтому при варьировании данного параметра предполагается резкое изменение производительности. В случае тестирования с одним файлом его размер совпадает со средним размером. Тогда анализ данного фактора сводится к определению значения перехода от маленького файла к большому. Т. к. скорее всего бенчмарк не будет предусматривать возможность работы с несколькими файлами, размер файла останется варьируемым только как характеристика тестируемой среды, а не отдельный фактор.

Глубина очереди может трактоваться двумя способами: как количество отправляемых за один раз запросов и как количество поддерживаемых запросов в очереди планировщика ввода-вывода. В рамках данной работы мы остановимся на первом варианте. Различные значения параметра могут влиять на производительность за счет соответствия или несоответствия с внутренними механизмами системы (например, с процессором) и планировщиком. Эмпирически выявлено, что данный фактор часто вызывает аномалии, поэтому он является достаточно перспективным для анализа. Глубина очереди относительно проста в изменении и не оказывает прямого влияния на другие запущенные приложения, поэтому ее следует включить в бенчмарк.

Количество потоков\процессов характеризует параллельность при тестировании. В большинстве случаев можно выявить значение данного фактора, когда процессор и стэк хранения будут оптимально загружены, что обеспечит повышение производительности. Данный параметр стоит включить в бенчмарк.

При распараллеливании задач иногда интересно открыть файл с разными флагами или просто с несколькими отдельными указателями. Данную ситуацию в общем виде можно обозначить как одновременный доступ к файлу с помощью нескольких дескрипторов, а в качестве параметра взять их количество. Разные системы предоставляют разные возможности. В некоторых открытие нескольких файловых дескрипторов на самом деле будет означать создание нескольких ссылок на один дескриптор. В других системах будут доступны не только множественные дескрипторы, но и спецификация в них разных флагов работы с файлами. В силу разнообразия предоставляемых взаимодействий фактор обуславливает широкий простор для появления аномалий производительности, поэтому он является перспективным для анализа. С точки зрения оптимизации, мы будем считать, что работа с несколькими дескрипторами является более редкой, чем работа с единственным дескриптором, поэтому обозначим данному фактору более низкий приоритет на включение, чем остальным, более распространенным факторам.

Операционные системы вынуждены делить ограниченное число ресурсов между множеством программ. Для удобства решения этой задачи в них используются приоритеты, количественно описывающие важность тех или иных процессов. Соответственно, способствуя выделению большего количества ресурсов конкретному процессу, данный фактор повышает его производительность. Приоритет процессов в рамках данной работы не представляет существенного интереса.

Скорость выполнения той или иной операции с данными зависит в том числе от состояния целевого файла. Например, большинство современных файловых систем осуществляют фактическое выделение памяти для файла только при наличии такой необходимости. Так, если отправить запрос на запись в только что созданный файл, то перед его выполнением системе будет необходимо аллоцировать сам файл, что повлечет за собой дополнительные задержки. Или если нужно записать данные в середину файла, а не в начало, то предварительно потребуется отыскать ее местоположение. Также производительность операций может зависеть и от других характеристик файла. Особый интерес представляет ситуация с файлами, которые были преаллоцированы функцией ftruncate(). Она вызывает изменение фактического расположения данных в хранилище, из-за чего может оказать негативный эффект на скорость, например, последовательного чтения. Влияние состояния тестового файла на производительность является достаточно

сложным для рассмотрения и оптимизации фактором, поэтому имеет низкий приоритет на включение.

Состояние кэша задает то, в каком виде он находится в начале тестирования и используется ли вообще. Кэш может быть:

- *очищенным*, т. е. не содержать каких-либо данных,
- *загрязненным*, т. е. содержать остаточные данные от предыдущих операций,
- *«разогретым»*, т. е. содержать данные, которые релевантны нагрузке при тестировании.

Разные состояния способны обуславливать существенную разницу в производительности, поэтому кэш является одним из наиболее важных факторов. Однако его оптимизация изнутри программы представляется мало возможной, поэтому на данном этапе его необязательно включать в бенчмарк.

Также при взаимодействии с файловой системой может использоваться прямой ввод-вывод, что означает полный пропуск кэша при выполнении операций. Это оказывается полезно в случае, когда он плохо выполняет свою работу, в частности, когда он имеет высокий процент промахов. Данный фактор также сам по себе может являться индикатором низкой эффективности кэша, если скорости при работе с его применением и без не имеют первостепенных различий. Прямой ввод-вывод довольно прост в применении и не оказывает влияния на работу других программ, поэтому является перспективным фактором. Его следует включить в бенчмарк.

Используемый программный интерфейс определяет набор функций, с помощью которых осуществляются взаимодействие с ФС. Так как разные интерфейсы предоставляют разные возможности, а также зачастую реализуют одни и те же возможности разным образом, то используемый API (Application Programming Interface) является одним из основополагающих факторов влияния на производительность файловой системы. Однако данный фактор далеко не всегда возможно и целесообразно изменять внутри программы, поэтому он не высокий приоритет на включение.

1.3. Выводы

На основе проанализированной литературы был выделен ряд рекомендаций. В процессе бенчмаркинга следует руководствоваться 3-мя принципами: обеспечить идентичные условия запуска тестов, запускать тест достаточное число раз для повышения точности и обеспечить достижение системой стабильности перед снятием измерений. Разрабатываемый бенчмарк должен способствовать соблюдению данных принципов и удовлетворять следующим требованиям: быть масштабируемым и ограниченным только файловой системой.

Также был сделан вывод о том, что для корректного сравнения и агрегации тестовых результатов, а также для повышения точности их интерпретации следует фиксировать состояние среды тестирования и других, оказывающих влияние на производительность, факторов, а также стремиться к тому, чтобы они совпадали при всех тестовых запусках, которые затем будут сравниваться между собой или объединяться.

Были рассмотрены несколько методологий тестирования. По результатам анализа одной из них было решено, что при разработке стоит сконцентрироваться на исследовательских возможностях разрабатываемого ПО, а не оптимизации конкретной программы. Также стало ясно, что наиболее подходящим к целям работы и нашим возможностям является категория синтетических микробенчмарков, поэтому разрабатываемый инструмент должен принадлежать ей.

В работе также был рассмотрен целый ряд современных бенчмарков. В общем смысле, т. к. все они разрабатывались не под нужды нашего метода, а просто для оценки файловых систем, их нельзя применить в данной работе. Однако на основе их анализа был выделен ряд функциональных возможностей, которыми должен обладать разрабатываемый бенчмарк. Сюда входят возможности: выбора паттерна нагрузки, варьирования факторов нагрузки, систематического тестирования, предварительного разогрева, построения двух- и трехмерных графиков по результатам тестирования, а также требование к ясности метода тестирования. Также из них были выделены идея разделения бенчмарка на несколько микробенчмарков и несколько вариантов выбора метрики производительности (пропускная способность, IOPS, задержки с разделением на среднее значение и пиковое).

Совместно из рассмотренной литературы и бенчмарков было выделено множество факторов, оказывающих влияние на производительность файловой системы. Затем была проведена категоризация, разделившая их на 4 группы: фиксированные факторы, изменяемые параметры операционной и файловой систем, тестируемый паттерн использования файловой системы, и характеристики нагрузки. Некоторые группы факторов в общем и конкретные факторы в частности были отдельно проанализированы на возможность и целесообразность включения бенчмарк. Большинству из них тем или иным образом были присвоены приоритеты. Таким образом, возможность выбора паттерна нагрузки является обязательной для включения в бенчмарк. Также необходимым является включение таких характеристик нагрузки как средний размер запроса и глубина очереди. При наличии возможности, следует включить факторы многопоточности, прямого ввода-вывода и выбора API взаимодействия с файловой системой. Из изменяемых параметров операционной и файловой систем лишь некоторые факторы могут быть включены в бенчмарк, но все они имеют невысокий приоритет в силу большéй сложности при

оптимизации на их основе, чем при оптимизации на основе характеристик нагрузки. Фиксированные факторы не имеет смысла включать в бенчмарк.

1.4. Цели и задачи выпускной квалификационной работы

Целью данной работы является разработка исследовательского программного обеспечения для оптимизации взаимодействия с файловой системой. Для ее достижения необходимо выполнить следующие задачи:

1. Сформулировать предполагаемый метод решения задачи оптимизации.
2. Сформулировать высокоуровневые требования к разрабатываемому ПО.
3. Описать модели и методы, которые будут реализованы или использоваться в ПО.
4. Спроектировать ПО.
5. Реализовать ПО.
6. Протестировать ПО.

2. Анализ требований и теоретическая проработка используемых в работе моделей и методов

2.1. Анализ требований к разрабатываемым программным средствам

2.1.1. Описание сценария решения задачи оптимизации с помощью разрабатываемого ПО

Определим сценарий решения задачи поиска оптимальных параметров взаимодействия с файловой системой следующим образом ([14]):

1. Определить паттерн нагрузки оптимизируемой программы.
2. Определить факторное пространство F, внутри которого будет осуществляться оптимизация.
3. Определить параметры исследования: параметры разогрева, параметры среды тестирования, параметры тестирования.
4. Провести тестирование и сохранить результаты.
5. По результатам тестирования получить график зависимости производительности от уровней факторов (т. е. визуализировать результирующее пространство R = F + 1, где дополнительной размерностью является измеренная производительность).
6. Проанализировать полученный график и сделать выводы об оптимальных параметрах работы.

По завершению поиска оптимальных параметров, останется самостоятельно применить полученные выводы для оптимизации конкретной программы.

Таким образом, разрабатываемое ПО должно позволять выполнять шаги 4 и 5 данного алгоритма и облегчать выполнение шага 6. Для удобства, разделим ПО на два компонента – бенчмарк, который будет отвечать за шаг 4, и визуализатор, который будет отвечать за шаг 5 и помогать в выполнении шага 6.

2.1.2. Требования к бенчмарку

Сформулируем требования к бенчмарку. Бенчмарк должен:

1. Уметь генерировать нагрузку ФС с указанным паттерном.
2. Уметь подготавливать среду тестирования.
3. Поддерживать возможность «разогрева».
4. Снимать измерения, которые пойдут в результат, только после завершения периода «разогрева».
5. Предоставлять возможность варьировать один или два фактора из поддерживаемых, т. е. систематически тестировать на каждом уровне из указанного пользователем множества.

6. Предоставлять возможность задавать статичные уровни тех факторов, которые не участвуют в варьировании.
7. Поддерживать факторы: размер запроса, глубина очереди, прямой ввод-вывод.
8. Позволять проводить тестирования производительности с заданными параметрами среды, разогрева, снятия измерений и др.
9. Поддерживать возможность замены API обращения к ФС.
10. Снимать измерения некоторой метрики производительности (конкретная метрика определяется далее).
11. Предоставлять возможность запускать один и тот же тест несколько раз с усреднением результатов.

2.1.3. Требования к визуализатору

Сформулируем требования к визуализатору. Визуализатор должен:

1. Уметь считывать результаты тестирования бенчмарком.
2. Уметь преобразовывать считанные результаты во внутреннее представление.
3. Уметь строить по преобразованным результатам двух- или трехмерные столбчатые диаграммы.

На любой диаграмме, построенной визуализатором, должны быть отражены:

- среднее значение метрики производительности за все время тестирования,
- среднеквадратическое отклонение метрики.

Объясним выбор данный показателей. Во-первых, при тестировании в любом случае должна присутствовать основная метрика, отражающая общую производительность системы. В качестве такой метрики выступает среднее значение измеряемой метрики производительности. Альтернативой могла бы быть медиана, но мы считаем, что среднее лучше, т. к. при тестировании предполагается достаточно большая выборка, что обеспечит достаточную точность и компенсирует потенциальные аномалии, с которыми призвана бороться медиана.

Во-вторых, при визуальном анализе полезно иметь некоторую характеристику разброса, которая бы позволила оценить стабильность основной метрики. Для среднего в качестве такой характеристики мы выбрали среднеквадратическое отклонение, а не дисперсию, т. к. оно имеет ту же размерность, что и основная величина, т. е. они сравнимы между собой и могут быть одновременно представлены на графике.

2.2. Теоретическая проработка разрабатываемого ПО

Приведем результаты теоретической проработки разрабатываемого ПО и методов его использования в порядке следования требований.

2.2.1. Модель паттерна нагрузки

В рамках данной работы мы остановимся на модели паттерна нагрузки, определяемой двумя параметрами: операцией (чтение или запись) и характером доступа (последовательный и случайным). Таким образом, бенчмарк можно будет использовать для микробенчмаркинга по 4 простейшим профилям нагрузки – последовательное чтение, последовательная запись, случайное чтение, случайная запись.

2.2.2. Метод обеспечения идентичных условий запуска тестов

Для того чтобы можно было запускать один и тот же тест несколько раз и агрегировать результаты, необходимо обеспечить идентичность условий запуска тестов. Также это нужно для того, чтобы можно было сравнивать результаты тестов с разными уровнями факторов между собой. Для этого необходимо с одной стороны – обеспечить одинаковые условия среды тестирования, а с другой стороны – свести к минимуму внешнее влияние.

Чтобы свести к минимуму внешнее влияние, следует каждый раз перед началом тестирования выполнять следующую процедуру. Перезагрузить компьютер. Подождать несколько минут, чтобы убедиться, что все процессы, запускающиеся при старте системы, успешно выполнились. Не запуская никакого дополнительного ПО и по возможности отключив работающие в фоне процессы, начать тестирование.

Обеспечение одинаковых условий среды тестирования поручим бенчмарку. Для этого не нужно внедрять новый функционал, но нужно проследить за тем, чтобы среда тестирования, которую подготавливает бенчмарк перед каждым запуском, зависела только от входных данных и не содержала случайных процессов. Тогда мы сможем говорить о том, что при одних и тех же входных данных, бенчмарк подготовит одни и те же условия среды тестирования.

При задании среды тестирования бенчмарк должен отвечать за решение следующих задач:

- создание файлов,
- открытие файлов,
- заполнение файлов данными.

Работа с файлом в целом задается несколькими параметрами: путь к файлу, размер файла, флаг удаления существующего файла и консольный скрипт подготовки среды. Путь к файлу включает в том числе имя самого файла. Этот параметр влияет на расположение файла в дереве каталогов, что позволяет выбирать с одной стороны – накопитель данных, на который попадет файл (например, флеш-накопитель или SSD), а с другой стороны – файловую систему, которая будет отвечать за его хранение. Таким образом, мы можем

тестировать любую установленную файловую систему и любую подключенную к компьютеру аппаратуру. Размер файла является числовым параметром, который передается файловой системе. Флаг удаления существующего файла нужен для тех случаев, когда мы хотим протестировать работу с уже существующим файлом. Это может быть полезно в случае, когда данный файл аллоцирован некоторым нетривиальным образом (например, sparse-файлы). Консольный скрипт подготовки среды представляет собой строку, которая предполагает содержание в ней скрипта на Bash, который запускается изнутри программы. Данный параметр может использоваться для дополнительной внепрограммной подготовки тестируемой среды.

Открытие файлов всегда сопряжено с передачей ряда параметров, называемых флагами, которые определяют возможности последующей работы и изначальное состояние файла. Так, для создания файла, нужно добавить соответсвующий флаг при вызове функции открытия файла. Помимо этого, важно при открытии файла указать флаги, которые бы обеспечили возможность чтения и записи в файл текущему пользователю системы, т. е. данной программе. Также нужно предусмотреть ситуацию, когда открываемый файл был создан вне программы. В таком случае, пользователь самостоятельно должен обеспечить доступ к работе с данным файлом программе. В нашем случае, при невозможности работы следует только обработать ошибку и прекратить тестирование.

То, какими данными заполнен файл, может влиять на скорость работы с ним, поэтому важно также реализовать данный аспект. Два стандартных вида данных, которыми можно заполнить файл, являются нули и случайные данные. Из практического опыта известно, что операционные системы могут оптимизировать файлы, заполненные нулями, сжимая их, что не всегда может быть ожидаемым для пользователя. Чтобы этого избежать, перед тестированием файл должен заполняться случайными данными. Заполнение файла изнутри программы другим видом данных не предусматривается, однако это можно сделать, если вынести подготовку файла в отдельный скрипт и передать его в бенчмарк качестве параметра. Чтобы это реализовать, заполнение файла случайными данными должно происходить только при условии установленного флага удаления файла. Т. е. будем считать, что если файл не удаляется, то пользователь сам отвечает за его заполнение.

2.2.3. Метод обеспечения достижения системой стабильности

Чтобы повысить точность результатов, во время тестирования необходимо обеспечивать достижение системой стабильного состояния, прежде чем приступать к снятию измерений. Решим проблему за счет введения периода «разогрева». Будем производить тестирование итерационно. На каждой итерации будем оценивать среднее и

среднеквадратическое отклонение накопившихся с начала тестирования данных. Условием достижения стабильности будем считать выполнение уравнения:

$$std \leq coef * mean, \quad (1)$$

где std – несмещенная оценка среднеквадратического отклонения, $coef$ – некоторый коэффициент, $mean$ – несмещенная оценка среднего (среднее арифметическое). В качестве первого приближения $coef$ возьмем 0.15.

Очевидно, что при некоторых параметрах работы условие достижения стабильности в принципе не будет выполняться. Для такого случая введем временное ограничение на достижение стабильности, равное 30 секундам. Данный параметр, так же как и $coef$, можно будет изменить.

После достижения системой стабильности, будем отбрасывать накопленные вспомогательные измерения и начинать этап основных измерений.

2.2.4. Метод реализации факторов

В ПО мы должны поддержать три фактора, влияющих производительность: размер запроса, глубину очереди и прямой ввод-вывод. Опишем то, каким образом они будут реализованы.

Размер запроса будет реализован как статический параметр, задающий размер буфера, в котором хранятся данные, которые будут записываться в файл, либо который передается файловой системе, для записи в него считываемых из файла данных. Разброса в данном случае не предполагается.

Глубина очереди будет реализована как еще один статический параметр, который определяет то, сколько операций будет вложено в одно обращение к памяти, т. е. в один вызов функции обращения к памяти. Соответственно, это число также означает то число буферов, которые необходимо будет передать этой функции для записи или чтения данных по каждой из вложенных операций. Разброс глубины очереди в рамках одного тестирования не предполагается.

Прямой ввод-вывод будет представлять собой булевую переменную, значение True которой будет означать то, что тестируемый файл необходимо открыть с некоторым флагом, который бы обеспечил прямой ввод-вывод. Стоит заметить, что данные флаги разнятся для разных операционных систем. В случае значения False, данный флаг при открытии файла использоваться не должен.

2.2.5. Выбор целевой метрики производительности

Рассмотрим фазу снятия измерений. В качестве целевой метрики производительности возьмем задержки (latency) при выполнении операций. Из этого

автоматически будет следовать, что операции будут выполняться синхронно и вызывающий поток будет блокироваться вплоть до их завершения.

Задержки должны будут фиксировать только время выполнения обращений к файловой системе. Некоторые современные системы имеют настолько высокую производительность, что инструменты измерения времени C++ не могут зафиксировать соответствующие задержки. Чтобы это компенсировать, будем объединять обращения к памяти в группы. В качестве первого приближения возьмем число обращений равное 10. Далее будем называть этот параметр размером группы обращений. Мотивация к такому выбору состоит в том, что практический опыт показал точность измерений в 1 мкс, а максимальную производительность – 3 обр. \ мкс.

Рассмотрим теперь фазу анализа результатов. Предполагается, что большинству пользователей будет проще воспринимать такую метрику производительности, более высокое значение которой соответствовало бы более высокой производительность, т. к. она более интуитивна. Но в случае задержек это не так, т. к. чем меньше задержки, тем больше производительность. Поэтому мы при анализе должны использовать одну из двух других метрик производительности – количество операций ввода-вывода в секунду (IOPS) или пропускную способность. Пропускная способность отражает объем передаваемых данных в секунду. Хотя эти метрики очень похожи, IOPS является менее предпочтительной, так как имеет размерность «операций в секунду», что делает невозможным, например, сравнение результатов тестов с разным размером запроса. Пропускная способность с размерностью «байт в секунду» уже учитывает размер запроса и глубину очереди, поэтому результаты с такой размерностью могут сравниваться между собой.

2.2.6. Модели задержек и пропускной способности

Теоретическую модель задержек зададим следующим образом. Пусть размер группы обращений к файловой системе равен N, количество обработанных за тест групп обращений – M, а i-ая операция из j-ой группы имеет задержку t_i, $i = \overline{1..N}, j = \overline{1..M}$. Тогда суммарная задержка j-ой группы обращений будет равна:

$$lat_j = \sum_{i=1}^{N} t_i. \quad (2)$$

Пропускная способность для той же группы при условии, что глубина очереди равна qd (queue depth), а размер запроса равен rs (request size) получается как произведение количества обращений в группе на количество операций в обращении (глубину очереди) и объем передаваемых данных за одну операцию (размер запроса), деленное на задержку в секундах:

$$Throughput_j = \frac{N * qd * rs}{lat_j}. \quad (3)$$

Минимальную и максимальную пропускную способность тогда можно вычислить как минимум и максимум по всем групповым пропускным способностям соответственно:

$$\min Throughput = \min_{j=\overline{1..M}} Throughput_j, \quad (4)$$

$$\max Throughput = \max_{j=\overline{1..M}} Throughput_j. \quad (5)$$

Для получения средних задержек необходимо взять среднее арифметическое по всем задержкам:

$$\text{avg}\, lat = \overline{lat} = \frac{1}{M}\sum_{M=1}^{M} lat_j. \quad (6)$$

Средняя пропускная способность будет вычисляться аналогичным образом:

$$\text{avg}\, Throughput = \overline{Throughput} = \frac{1}{M}\sum_{M=1}^{M} Throughput_j = \frac{1}{M}\sum_{M=1}^{M} \frac{N * qd * rs}{lat_j}. \quad (7)$$

Среднеквадратическое отклонение для обеих величин может быть получено по формуле несмещенной оценки ([15]), в общем случае имеющей вид:

$$S_n = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})^2}, \quad (8)$$

где S_n – оценка среднеквадратического отклонения, n – количество измерений, x_i – i-ое измерение, $\bar{x}$ – среднее арифметическое всех измерений.

Среднее и среднеквадратическое отклонение для задержек будет использоваться при разогреве, а для пропускной способности – при представлении результатов тестирования.

2.2.7. Метод повышения точности результатов

Производительность файловой системы при тестировании зависит от множества неконтролируемых факторов (например, сторонняя деятельность операционной системы). Одним из следствий этого является возможность возникновения случайных неконтролируемых возмущений, которые могут существенно повлиять на результат. Поэтому есть необходимость снизить данное влияние. Для этого предлагается запускать один и тот же тест несколько раз и усреднять групповые пропускные способности.

Опишем подробнее алгоритм усреднения. Пусть тест запускался N раз, тогда у нас есть столько же массивов с групповыми пропускными способностями. Построим новый массив, где на i-ой позиции будет находиться среднее арифметическое i-ых элементов в первых N массивах, т. е. напрямую усредним каждое измерение по всем запускам.

Для этого в принципе нужно было бы гарантировать сравниваемость результатов на i-ой позиции. Но мы считаем, что это уже гарантировано, т. к. различие выполненных во время i-го измерения операций не препятствует сравниваемости результатов, потому что такую вариацию предполагает сам тестируемый паттерн нагрузки.

Помимо этого, важно, чтобы не оставалось неусредненных измерений, т. е. чтобы число измерений совпадало по всем запускам теста. Для этого будем делать следующее. Пусть первый запуск теста определяется временным ограничением пользователя (например, 100 секунд). Тогда по его окончанию мы сможем определить длительность теста в количестве снятых измерений, взяв размер результирующего массива. Зная, что число измерений совпадает с числом итераций, выставим соответствующее ограничение на число итераций для последующих запусков. Таким образом, мы гарантируем равенство числа измерений по всем запускам.

В дополнение к этому добавим, что при многократном тестировании с разными уровнями факторов, следует запускать тесты в случайном порядке [16]. Это позволит, при наличии продолжительных внешних случайных возмущений, распределить их по всему факторному пространству, т. е. будут изменяться результаты случайных, пространственно разделенных тестов. В противном случае влияние бы было оказано на подряд идущие тесты, что могло создать в результате иллюзию особенности соответствующих точек факторного пространства. Помимо этого, такой подход позволит скомпенсировать влияние случайных возмущений на производительность в каждой конкретной точке. Это произойдет за счет того, что повторные запуски одного и того же теста будут иметь больший шанс не оказаться под влиянием возмущений, т. е. смогут приблизить конечный усредненный результат к реальному.

2.3. Проектирование бенчмарка

Разделим бенчмарк на две составляющие – ядро, отвечающее за тестирование, и экспериментатор, отвечающий за пользовательский интерфейс и проведение экспериментов, т. е. систематических тестов с варьированием факторов. Далее рассмотрим общий алгоритм работы бенчмарка – его будет реализовывать экспериментатор, а также построим диаграмму классов для обеих составляющих. Алгоритм тестирования и результаты разработки будут приведены в следующем разделе по факту реализации.

2.3.1. Общий алгоритм работы бенчмарка

Алгоритм работы бенчмарка в общем виде должен выглядеть следующим образом (см. рис. 4). После запуска программа должна считать из консоли описание эксперимента. Для этого ей необходимо выводить поочередно названия всех параметров эксперимента в консоль и считывать их значения, которые будет вводить пользователь. Затем нужно

проверить на корректность получившееся описание эксперимента. В случае обнаружения ошибок в описании следует оповестить об этом пользователя и завершить работу программы. Далее необходимо случайным образом определить порядок выполнения тестов, чтобы обеспечить большую точность результатов, и подготовить массив, в котором будут храниться промежуточные необработанные результаты. Затем необходимо провести тестирование в указанном порядке. По его завершению, необходимо обработать полученные промежуточные результаты и вывести уже готовые результаты эксперимента в консоль. На этом работа программы должна завершиться.

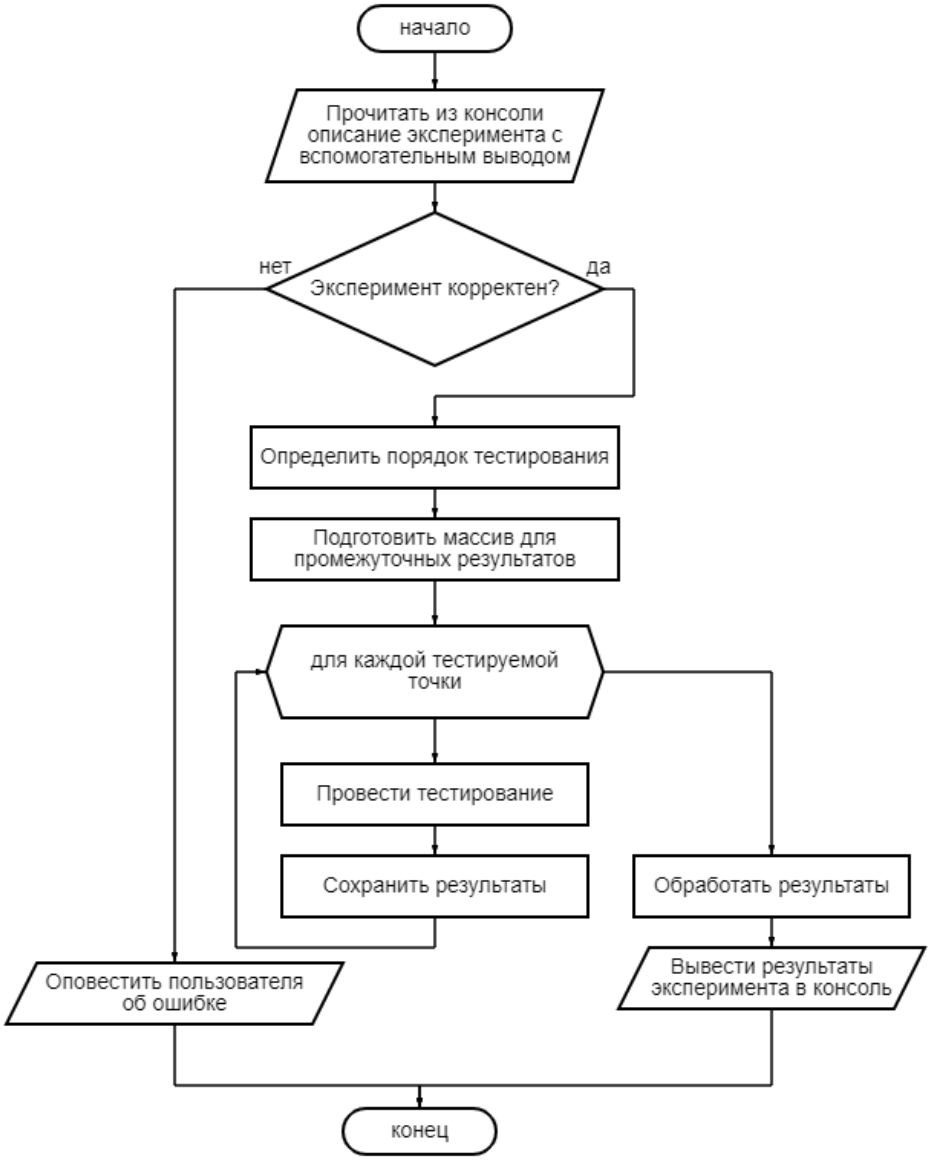


Рис. 4 – Блок-схема общего алгоритма работы бенмчарка

Далее перейдем к описанию составляющих бенчмарка.

2.3.2. Описание диаграммы классов ядра бенчмарка

Бенчмарк будет представлять собой консольное приложение, написанное на C++. Диаграмма классов ядра бенчмарка представлена на рисунке 5. Основным является класс

TBenchmark. Он имеет два публичных метода – конструктор, который принимает на вход некоторые параметры, и Benchmark(), который при вызове производит тестирование и возвращает массив задержек. Также введен приватный вспомогательный метод PrepareEnvironment(), который при вызове производит подготовку среды и вернет файловый дескриптор открытого в процессе подготовки файла.

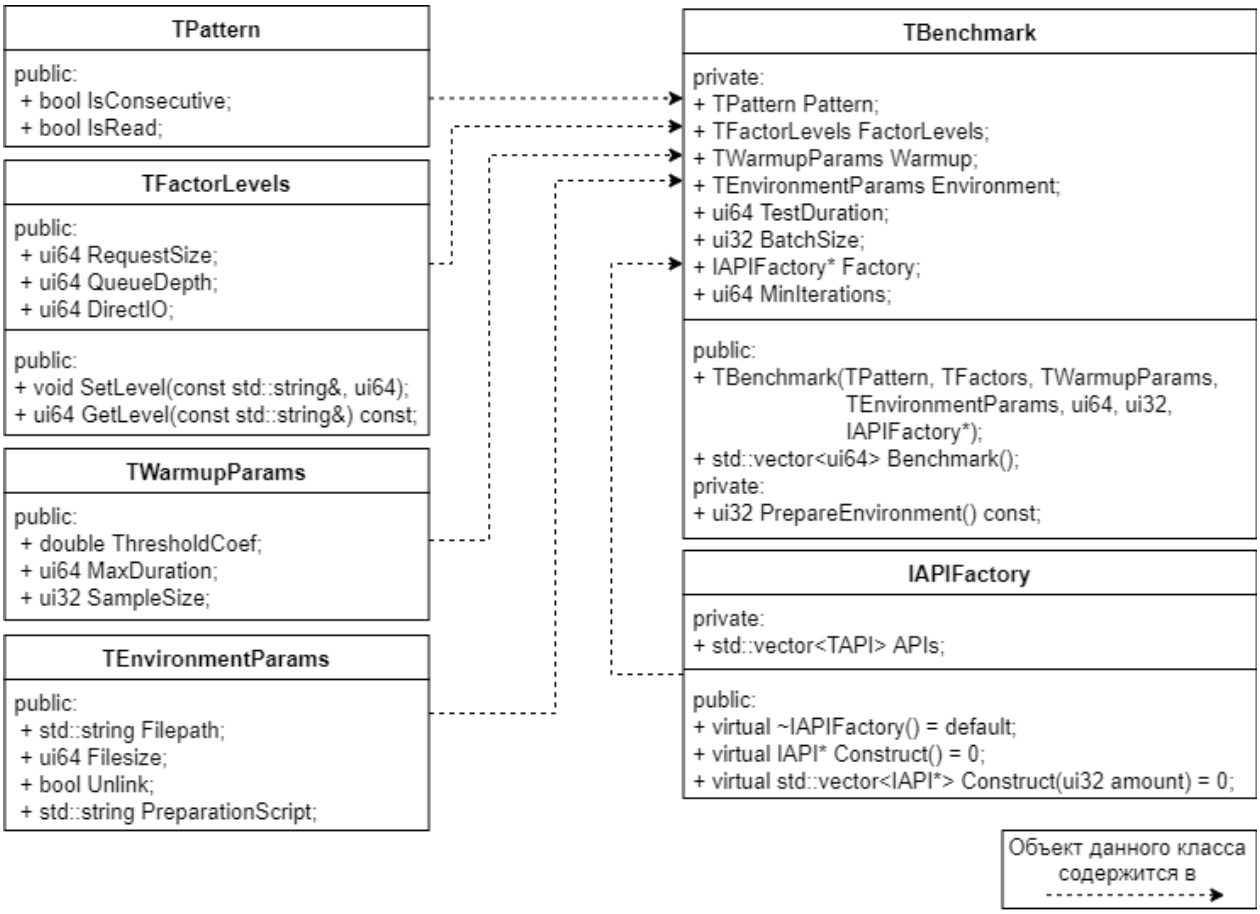


Рис. 5 – Диаграмма классов ядра бенчмарка

Используются следующие параметры бенчмарка: паттерн нагрузки, уровни факторов, параметры разогрева, условия среды, длительность тестирования, размер группы. Для большинства параметров тестирования созданы отдельные классы: TPattern, TFactors, TWarmupParams, TEnvironmentParams, и IAPIFactory. Рассмотрим их подробнее.

Класс TPattern содержит два флага IsConsecutive (является последовательным) и IsRead (является чтением). Установленный первый флаг показывает, что доступ будет осуществляться к последовательным участкам памяти. В противном случае – к случайным. Второй флаг определяет то, какая операция будет выполняться – чтение или запись.

Класс TFactors хранит поле на каждый фактор. В данном случае их три – размер запроса, глубина очереди, прямой ввод-выод. Помимо этого, данный класс обладает функцией SetLevel(), которая позволяет по заданному строковому названию фактора,

установить для него уровень, и функцией GetLevel(), которая аналогичным образом позволяет извлечь уровень конкретного фактора.

Класс TWarmupParams обладает тремя полями: коэффициентом «разогрева», максимальной длительностью разогрева и размером выборки. Хотя ранее уже было выбрано число 10 в качестве размера выборки, соответствующий параметр все равно введен, так как с одной стороны, это является хорошей практикой, с другой стороны – предусматривает вполне вероятный случай, когда размер выборки изменится.

Класс TEnvironmentParams содержит три поля: путь к файлу, размер файла, скрипт подготовки флаг пересоздания файла, скрипт подготовки файла. Путь определяет место, в котором будет создан файл. Помимо этого, он может указывать на уже существующий файл, если есть необходимость протестировать именно его. Это может быть полезно в случае, например, когда нужно заполнить файл какими-то особенными начальным данными. Для этой же цели присутствует флаг пересоздания файла. Если он установлен, то файл по указанному пути будет удален и создан заново. Смысл поля «размер файла» соответствует названию.

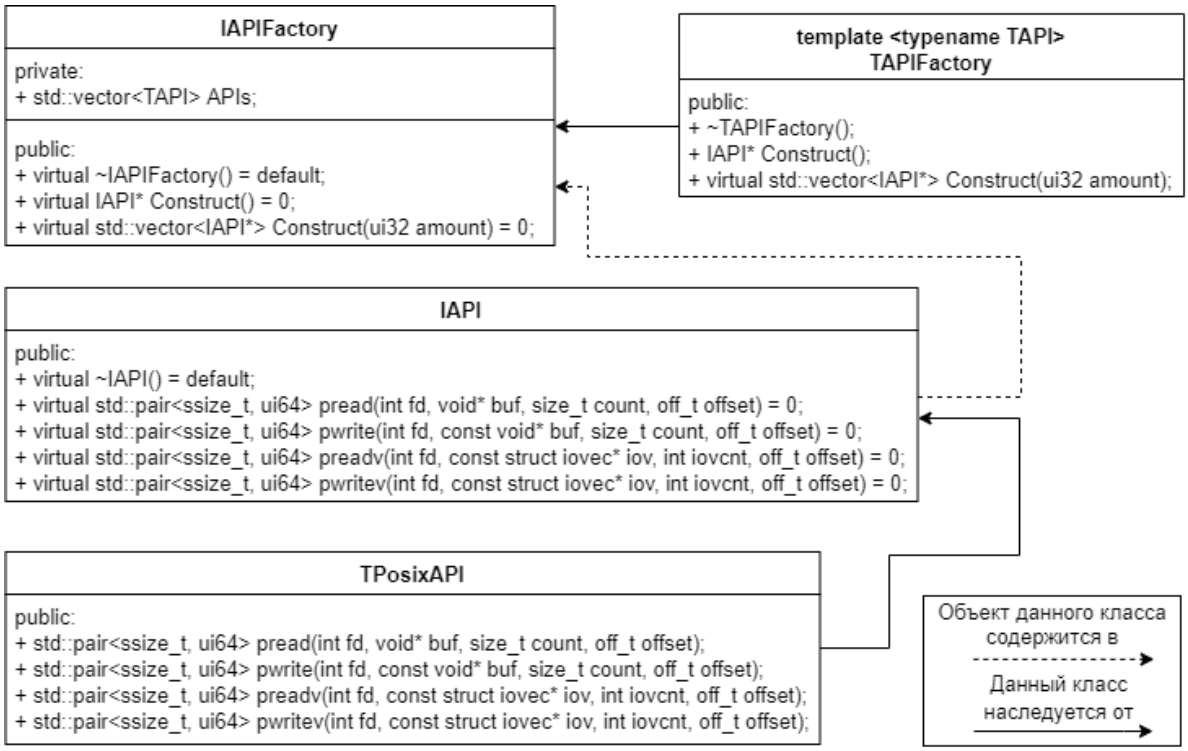


Рис. 6 – Диаграмма классов реализации абстрактного API

Для того чтобы предусмотреть возможность замены интерфейса общения с ФС был введен абстрактный класс IAPI (рис. 6), который задает 4 функции – 2 позиционных операции (pread, pwrite) и 2 позиционных векторных операции (preadv, pwritev) [17]. Они принимают в качестве аргументов файловый дескриптор, буфер для данных в том или ином виде, и сдвиг в файле, по которому будет выполняться операция. В качестве результата они

возращают пару чисел: количество обработанных байт данных и получившуюся задержку. Интерфейс IAPI реализуется классом TPosixAPI, который использует функции обращения к памяти, удовлетворяющие стандарту POSIX [18].

Для того чтобы предусмотреть возможность введения фактора многопоточности (для этого необходимо предотвращать состояния гонки, т.е. иметь собственный объект на каждый поток [19, 20]) и добавление автоматических тестов, создан интерфейс IAPIFactory и реализующий его шаблонный класс TAPIFactory, которые несут ответственность за создание, хранение и удаление объектов классов, наследующихся от IAPI. Первый задает, а второй реализует функцию Construct() с двумя перегрузками. В одном случае она возвращает один объект, во втором случае – массив объектов, размер которого задается аргументом функции $amount$. Данные объекты существуют независимо от возвращенных указателей на них до следующего вызова любой из перегрузок функции Construct(). Шаблонность класса TAPIFactory позволяет статически реализовывать интерфейс IAPIFactory для любого класса, наследующегося от интерфейса IAPI.

Дополнительно, введен параметр MinIterations, в который записывается количество итераций, которое было выполнено во время первого запуска бенчмарка. Все последующие запуски будут продолжаться как минимум до тех пор, пока не осуществят такое же количество итераций. Это нужно для реализации метода повышения точности результатов (см. раздел 2.2.7).

2.3.3. Описание диаграммы классов экспериментатора

Перейдем к описанию класса экспериментатора (см. рис. 7). Он обладает похожим на TBenchmark набором полей. Во-первых, он хранит те же общие параметры тестирования: паттерн нагрузки, параметра разогрева, параметры среды, длительность теста, размер группы. Помимо них, он хранит массив со всеми тестируемыми комбинациями уровней факторов – FactorLevels. Также, т. к. экспериментатор отвечает за многократное воспроизведение тестов, он хранит число воспроизведений – Replays. Чтобы при большом количестве факторов точно знать, какие из них варьируются, а какие заданы на единственном уровне, хранится массив VaryingFactors, в котором содержатся названия варьируемых факторов. В заключение, для хранения фабрик API, указатели на которые передаются в бенчмарк при его создании, добавлен изменяемый массив APIFactories.

В публичном доступе класс содержит конструктор, основную функцию экспериментирования Experiment(), и три внешних вспомогательных функции GetPattern(), GetFactorLevels() и GetVaryingFactors(), которые позволяют узнавать снаружи экспериментатроа о тестируемом паттерне нагрузки, тестируемых комбинациях уровней факторов и о названиях варьируемых факторов.

```
TExperimenter
---------------------------------------------------------------------------
private:
+ TPattern Pattern;
+ std::vector<TFactorLevels> FactorLevels;
+ TWarmupParams Warmup;
+ TEnvironmentParams Environment;
+ ui64 TestDuration;
+ ui32 BatchSize;
+ ui32 Replays;
+ std::vector<std::string> VaryingFactors;
+ mutable std::vector<TAPIFactory<TPosixAPI>> APIFactories;
---------------------------------------------------------------------------
public:
+ TExperimenter(TPattern, std::vector<TFactorLevels>&&,
                const TWarmupParams&,
                const TEnvironmentParams&,
                ui64, ui32, ui32,
                const std::vector<std::string>&);

+ std::vector<std::pair<ui64, ui64>> Experiment() const;

+ TPattern GetPattern() const;
+ const std::vector<TFactorLevels>& GetFactorLevels() const;
+ const std::vector<std::string>& GetVaryingFactors() const;

private:
+ std::vector<ui32> GenerateOrder() const;
+ std::vector<TBenchmark> CreateBenchmarks() const;
+ std::vector<ui64> ConvertToThroughput(const std::vector<ui64>&, ui32);
```

Рис. 7 – Диаграмма класса экспериментатора

Конструктор реализован стандартным образом за тем исключением, что паттерн нагрузки передается копированием, т. к. содержит полей всего на 2 байта, что меньше чем размер ссылки (4 байта), и комбинации уровней факторов передаются универсальной ссылкой, т. е. перемещаются. Последнее является оптимизацией для предотвращения копирования массива, который в перспективе может иметь достаточно большой размер.

Основная функция экспериментирования Experiment() работает согласно общему алгоритму работы бенчмарка (см. раздел 2.3.1). На выход она передает массив пар чисел – среднего и среднеквадратического отклонения пропускной способности на каждую тестируемую точку (комбинацию). Порядок результатов в возвращаемом массиве соответствует порядку комбинаций в массиве FactorLevels.

Функции GetPattern(), GetFactorLevels() и GetVaryingFactors() позволяют узнать некоторые параметры эксперимента, но не дают доступа к их изменению за счет передачи данных копированием или по ссылке на константу. Они будут использоваться при выводе результатов эксперимента, который будет осуществляться вне экспериментатора. Подробно способ вывода будет описан после реализации в следующем разделе.

Также экспериментатор содержит три приватные функции: GenerateOrder(), CreateBenchmarks() и ConvertToThroughput(). Первая из них должна определять порядок тестирования, создавая массив индексов. Каждый индекс определяет тестируемые элемент в массиве FactorLevels. Для реализации нескольких воспроизведений, каждый индекс хранится такое же число раз, сколько и должен воспроизводиться тест. Вторая функция, CreateBenchmarks(), отвечает за создание массива бенчмарков на каждую комбинацию уровней факторов. Последняя функция, ConvertToThroughput(), принимает на вход массив задержек и индекс соответствующего элемента в массиве FactorLevels. Получив оттуда данные о размере запроса и глубине очереди, она конвертирует полученные задержки в пропускные способности и возвращает новый массив. Все три указанные функции являются вспомогательными и используются только внутри функции Experiment().

2.4. Выводы

В данном разделе был сформулирован сценарий решения задачи оптимизации с помощью разрабатываемых программных средств. На основе этого и выводов из первого раздела были получены высокоуровневые требования.

Затем были описаны модели паттерна нагрузки, а также задержек и пропускной способности, которые будут использованы при разработке ПО. Также были описаны методы обеспечения идентичных условий запуска тестов, обеспечения достижения системой стабильности, реализации факторов и повышения точности результатов, которые по большей части будут реализованы в программном обеспечении, и частично – рекомендуются к исполнению со стороны пользователя при использовании разработанных программных средств.

На основе полученных требований, описанных моделей и методов, и предыдущего опыта прототипирования бенчмарка было осуществлено проектирование. В ходе него были описаны: общий алгоритм работы бенчмарка, а также диаграммы классов ядра бенчмарка и экспериментатора.

На этом мы данный раздел завершается. Далее будут приведены результаты реализации и тестирования.

3. Реализация и тестирование ПО

3.1. Реализация бенчмарка

Согласно проекту были разработаны ядро бенчмарка и экспериментатор. Дополнительно, были реализованы интерфейсы ввода и вывода. В данном разделе мы опишем алгоритмы работы основных функций бенчмарка, а также приведем некоторые технические детали реализации. Полный листинг кода разработанного ПО представлен в репозитории GitHub по ссылке: https://github.com/veefore/benchmark.

3.1.1. Реализация функции Benchmark(): этап подготовки

Рассмотрим реализацию функции Benchmark(). На рисунке 8 представлен этап подготовки перед тестированием. Вначале для удобства создаются псевдонимы параметров тестирования и осуществляется подготовка среды за счет вызова функции PrepareEnvironment(). Затем объявляются массивы умных указателей [21, 22], которые будут отвечать за управление памятью, хранящей буферные (обрабатываемые) данные. Затем в зависимости от глубины очереди (queue depth или кратко qd) происходит аллокация данных для соответствующего массива.

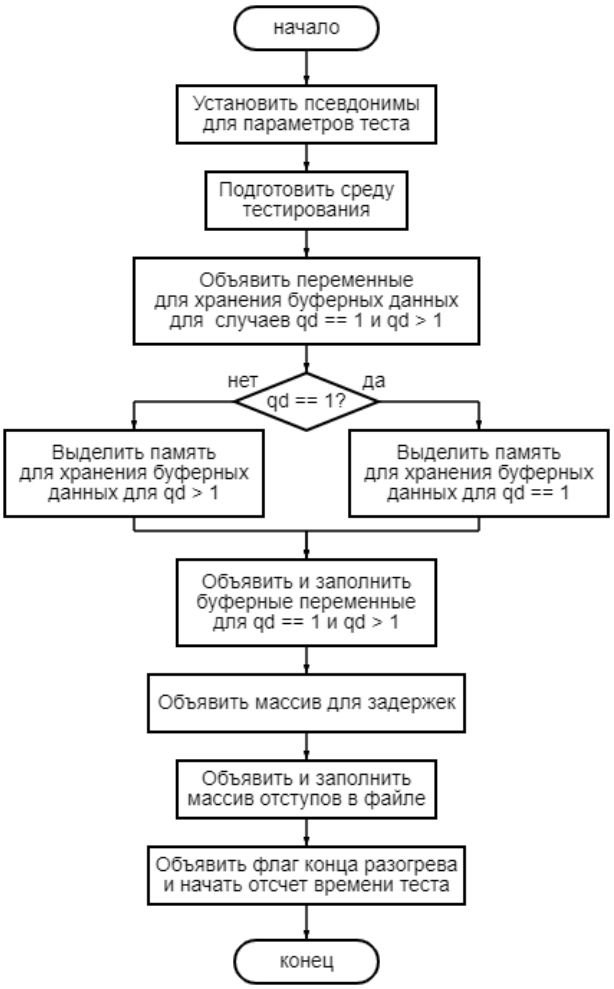


Рис. 8 – Блок-схема алгоритма работы функции Benchmark(), этап подготовки

После этого объявляются и заполняются указателями сами буферы, объявляется массив для задержек, объявляется и заполняется массив отступов (отступ определяет позицию в файле, по которой будет выполняться операция), определяется флаг конца разогрева со значением истинности «ЛОЖЬ» и начинается отсчет времени теста. На этом завершается подготовительный этап.

Рассмотрим отдельно код реализации массивов для буферных данных, т. к. он нетривиален в понимании (см. листинг 1). Объясним применение буферов. Буферы передаются в качестве аргументов при вызове функций обращений к памяти Read() или Write(). В случае чтения функция запишет в буферы считанные из файла данные, а в случае записи – возьмет из буферов данные и запишет их в файл. Хотя мы употребляем слово «буферы» во множественном числе, реальное количество буферов совпадает с размером очереди и может быть равно 1. Добавим также, что для реализации обращений к ФС с глубиной очереди > 1 используется специальный тип данных – struct iovec (массив операций ввода-вывода [23]), который хранит сразу множество буферов и их размеров. Зная это, мы можем перейти к рассмотрению кода.

В строках 3 и 4 объявляются массивы умных указателей bufferPtrs и iovPtrs размера, равного размеру группы. Данные используют умные указатели и создаются в принципе по следующей причине. В обоих случаях буферы в функции read() и write() передаются в сыром виде, т. е. чистыми указателями, и поэтому хранятся таким же образом. Это означает, что они не контролируют ту память, на которую указывают. Здесь может возникнуть достаточно частая проблема, когда-либо программист забудет явно освободить выделенную память, либо код просто не дойдет до исполнения кода освобождения памяти в силу, например, выброшенного ранее исключения. В таком случае произойдет утечка памяти, что в некоторых случаях может привести к аварийному завершению работы всей операционной системы. Чтобы предотвратить данную и другие подобные проблемы в языке C++ была введена техника программирования RAII (“Resource Acquisition Is Initialization”) [24, 25]. Простыми словами, она обязывает привязывать жизненный цикл захваченного общедоступного ресурса ко времени жизни конкретного объекта. Зная, что объект всегда удалится [26], мы в таком случае гарантируем и освобождение ресурсов. В нашем случае в качестве ресурсов выступает память на стеке [27], а в качестве конкретных объектов – умные указатели (конкретнее, std::unique_ptr), предназначенные конкретно для данных целей. В строке 6 выделяется отдельный массив iovData, хранящий данные типа uint32_t [28]. Он нужен для хранения сырых данных, указатели на которые будут потом записаны внутри объектов типа struct iovec в массиве iovPtrs.

```
1.      const ui64 bufSize = ceil((long double)rs / sizeof(ui32)); // Размер буфера
2.      // ~ Массивы для хранения данных, используемых в буферах при выполнении
   операций
3.      std::vector<std::unique_ptr<ui32[]>> bufferPtrs(BatchSize); // qd == 1
4.      std::vector<std::unique_ptr<struct iovec[]>> iovPtrs(BatchSize); // qd > 1
5.      // Вспомогательный массив хранения для случая qd > 1
6.      std::vector<std::unique_ptr<ui32[]>> iovData(BatchSize * qd);
7.
8.      // Аллокация данных в массивах
9.      for (ui32 i = 0; i < BatchSize; i++) {
10.         if (qd == 1) {
11.             bufferPtrs[i].reset(new ui32[bufSize]);
12.         } else if (qd > 1) {
13.             iovPtrs[i].reset(new struct iovec[qd]);
14.             for (ui32 j = 0; j < qd; j++) {
15.                 iovData[i * qd + j] = std::unique_ptr<ui32[]>(new
   ui32[bufSize]);
16.                 iovPtrs[i][j].iov_base = iovData[i * qd + j].get();
17.                 iovPtrs[i][j].iov_len = rs;
18.             }
19.         }
20.     }
21.
22.     // ~ Объявление и заполнение массивов непосредственно буферов.
23.     std::vector<void *> bufs(BatchSize);
24.     std::vector<const struct iovec*> iovs(BatchSize);
25.     for (ui32 i = 0; i < BatchSize; i++) {
26.         bufs[i] = bufferPtrs[i].get();
27.         iovs[i] = iovPtrs[i].get();
28.     }
```

Лист. 1 – Код подготовки буферов в функции Benchmark()

В строках 8 -20 приведено то, как три вышеуказанных массива заполняются данными, выделяемыми на куче. В случае глубины очереди, равной 1, это очевидно. В случае большей глубины очереди сначала выделяется несколько объектов типа struct iovec и записываются в массив iovPtrs. Затем на каждый записанный объект выделяется память для сырых данных и записывается в массив iovData. После этого указатель на выделенную память и ее размер записываются в соответствующий объект в массиве iovPtrs.

Подготовка буферов завершается выделением и заполнением массивов с непосредственно буферами, которые далее будут передаваться непосредственно в функции Read() и Write() (строки 22-28).

3.1.2. Реализация функции Benchmark(): этап тестирования

Перейдем к этапу тестирования (см. рис. 9). Непосредственное тестирование происходит внутри цикла while, пока выполняется хотя бы одно из условий: разогрев еще

не завершен, время теста еще не вышло, снято недостаточно измерений. В теле цикла происходит следующее. Сначала объявляются переменные для сохранения текущих задержек и количества обработанных за группу операций байт данных. Затем выполняется генерация основной нагрузки: по указателю на объект IAPI в зависимости от паттерна нагрузки вызывается соответствующая функция обращения к памяти – Read() или Write(). Одна из двух перегрузок функции выбирается исходя из глубины очереди.

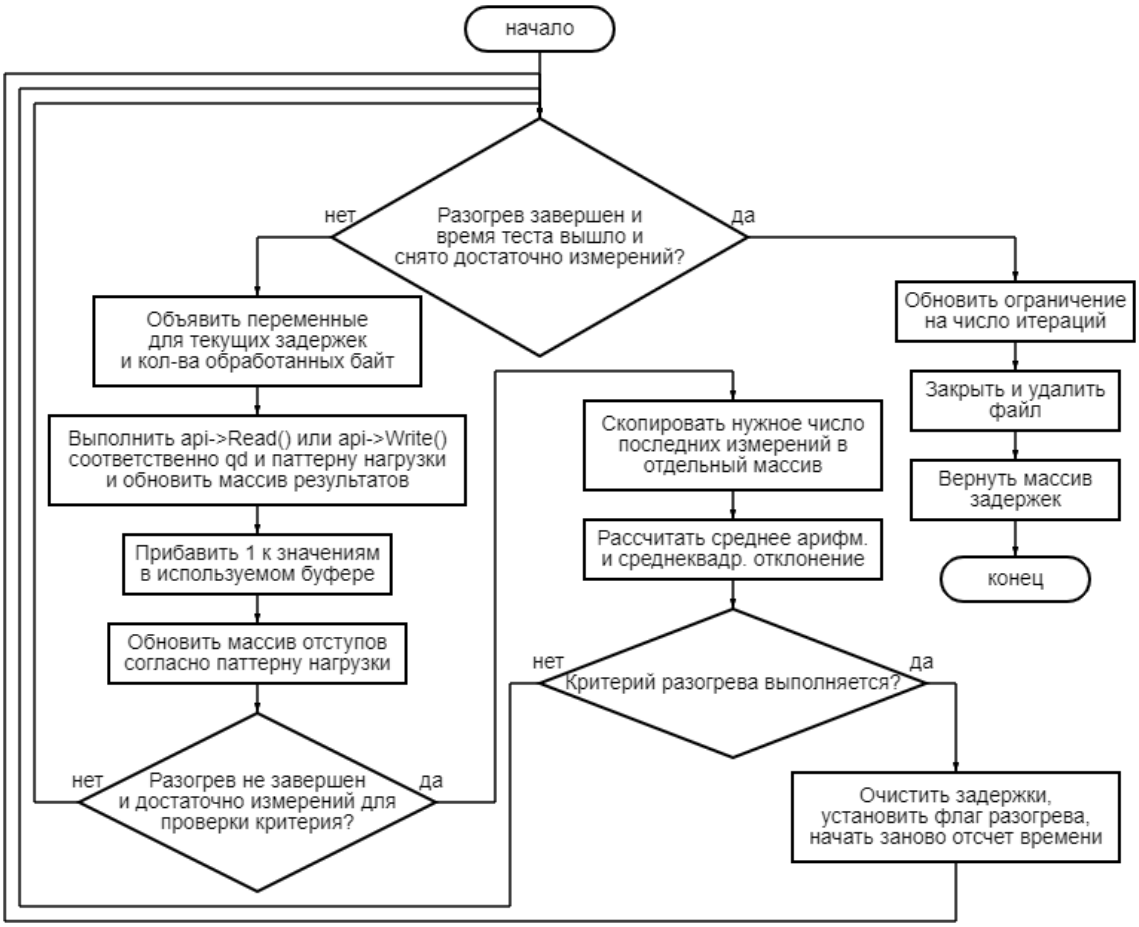


Рис. 9 – Блок-схема алгоритма работы функции Benchmark(), этап тестирования

После такого как обращение к памяти завершено, нужно обновить результаты для подготовки к следующей итерации. Для этого мы, во-первых, прибавим ко всем буферным значениям единицу. Это позволит избежать возможных оптимизаций хранения файла, использующих тот факт, что он состоит из одинаковых значений. Во-вторых, мы обновим массив отступов. В случае последовательного доступа к памяти – сдвинем отступы вперед на размер последнего обращения, а в случае случайной – сгенерируем новые отступы случайным образом. Тело цикла завершается блоком обработки критерия разогрева. В случае, если разогрев уже завершен или нам просто не хватает измерений для проверки критерия – пропустим блок. В противном случае войдем в блок, выделим нужное количество последних измерений задержек исходя из параметра разогрева «размер

выборки», сгенерируем по ним статистику в виде среднего арифметического и среднеквадратического отклонения, и проверим на выполнение критерий разогрева. Если критерий выполнен, то очистим массив накопленных задержек, установим флаг конца разогрева, и начнем заново отсчет тестового времени. Таким образом, время основного тестирования оказывается независимым от времени разогрева и удовлетворяет параметру длительности тестирования. Здесь завершается тело цикла.

После тела основного цикла выполняется следующая проверка: если ограничение на количество минимальных измерений при тестировании не установлено (равно значению по умолчанию), то установим его равным текущему число измерений. Благодаря данному обновлению и соответствующему условию в цикле while все тесты после первого будут обязаны длиться достаточно долго, чтобы число измерений в них было не меньше, чем число измерений в самом первом тесте. Так мы сможем потом агрегировать результаты. Помимо данной проверки, осуществляются также закрытие файлового дескриптора и удаление тестового файла. На этом выполнение функции завершается, и она возвращает массив задержек.

3.1.3. Реализация абстракции API

Отметим отдельно, что запланированная абстракция API взаимодействия с ФС также была реализована с некоторыми модификациями относительно проекта. В частности, в основном интерфейсе IAPI* теперь 4 изначальных функции (pread(), pwrite(), preadv(), pwritev()) стали приватными. Вместо них в публичном доступе теперь находятся методы Read() и Write() в двух перегрузках, которые используют те же 4 базисных функции, содержат Для реализации интерфейса создан указанный в проекте класс TPosixAPI. Указанные функции он переопределяет с использованием следующих функций: read(), write(), readv(), writev(), lseek() [29, 30]. Пример этого для той перегрузки функции Write(), которая реализует очередь, представлен в листинге 2.

```
1. std::pair<ssize_t, ui64> IAPI::Write(int fd, const
   std::vector<const struct iovec*>& iovs, int iovcnt, const
   std::vector<off_t>& offsets) {
2.     auto start = Nhrc::now();
3.     ssize_t bytesProcessed = 0;
4.     for (ui32 i = 0; i < iovs.size(); i++)
5.         bytesProcessed += pwritev(fd, iovs[i], iovcnt,
   offsets[i]);
6.     auto end = Nhrc::now();
7.     return {bytesProcessed, Duration(start, end)};
8. }
9.
```

```
10.     ssize_t TPosixAPI::pwritev(int fd, const struct iovec* iov,
   int iovcnt, off_t offset) {
11.         lseek(fd, offset, SEEK_SET);
12.         return ::writev(fd, iov, iovcnt);
13.     }
```

Лист. 2 – Реализация функций IAPI::Write() и TPosixAPI::pwritev()

Внутри функции IAPI::Write() осуществляется вызов операции pwritev() в количестве, равном размеру группы, и измерение времени выполнения функции от ее начала до конца. Функция TPosixAPI::pwritev() реализует позиционную векторную (т. е. с учетом очереди) запись за счет предварительного вызова lseek(), меняющей отступ в файле, и следующего за ним вызова библиотечной функции writev(), осуществляющий векторную запись по уже сохраненному в файле отступу.

3.1.4. Реализация функции PrepareEnvironment()

Функция PrepareEnvironment осуществляет необходимую для тестирования подготовку файла и возвращает файловый дескриптор [31]. Алгоритм ее работы приведен на рисунке 10. Рассмотрим его подробнее.

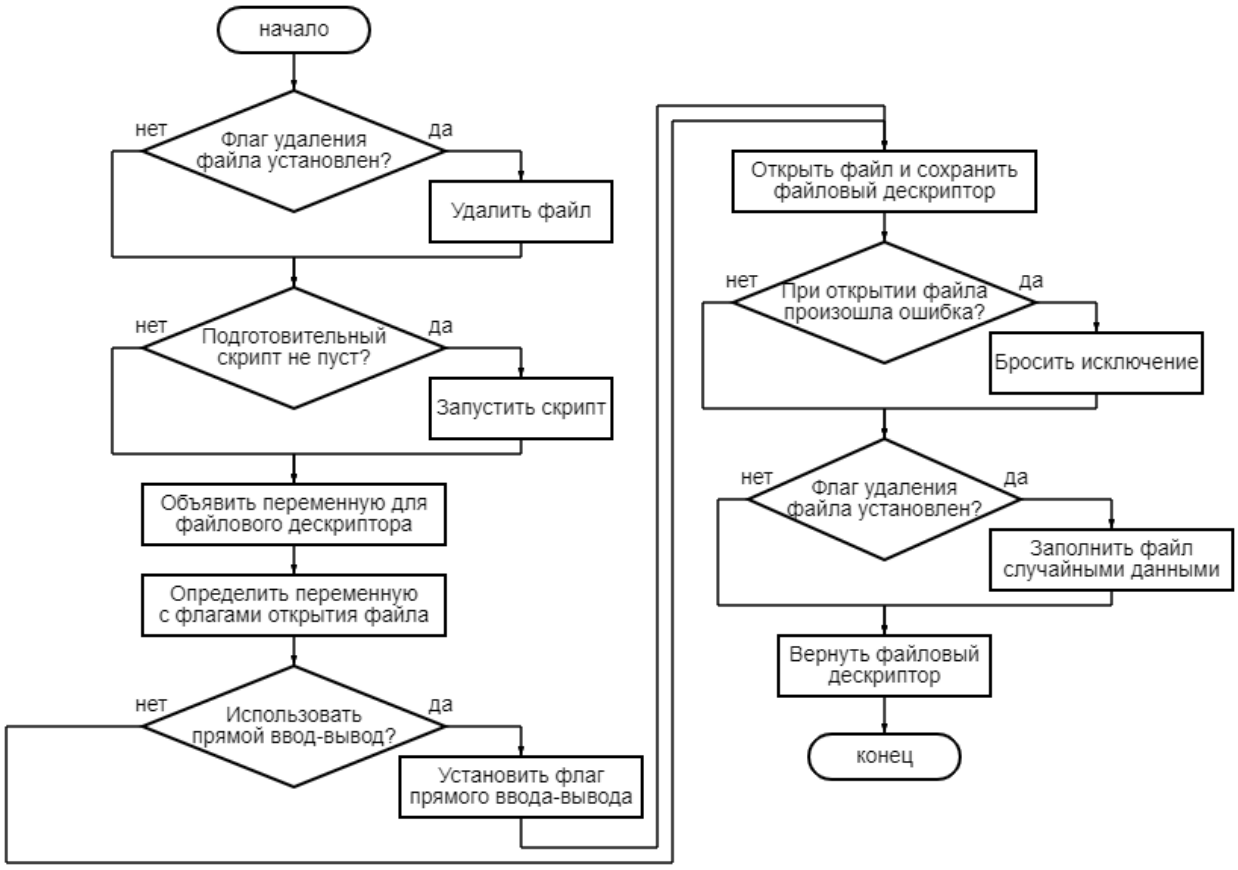


Рис. 10 – Блок-схема алгоритма работы функции PrepareEnvironment()

В первую очередь функция проверяет, установлен ли флаг удаления файла. Если да, то файл удаляется. Затем, если подготовительный скрипт не пуст, он запускается и программа ждет его выполнения. После этого остается только открыть файл с нужными флагами и заполнить его данными. Для открытия файла заводятся две переменные – для файлового дескриптора и для флагов. В качестве базисных флагов используются O_RDWR, позволяющий обращаться к файлу для чтения или записи, и O_CREAT, указывающий на то, что при отсутствии файла его необходимо создать [32]. В случае, если уровень фактора «прямой ввод-вывод» установлен равным единице, то добавляется также флаг O_DIRECT для Linux или F_NOCACHE для MacOS. Права доступа к файлу при его создании задаются отдельным флагом S_IRWXU, который означает предоставление прав на чтение, запись и исполнение владельцу файла, т. е. пользователю, от имени которого был запущен бенчмарк. Далее программа пытается открыть файл по заданному пользователем пути с указанными флагами с помощью функции open(). В случае, если попытка открытия файла оказывается неудачной, то выбрасывается исключение. Затем, если установлен флаг удаления файла, нужно заполнить открытый файл случайными данными. Для этого используются тестовый API обращения к ФС. После этого функция завершается и возращает полученный файловый дескриптор.

3.1.5. Реализация экспериментатора

Экспериментатор был реализован согласно проекту с только тем отличием, что вне класса была введена дополнительная функция AddVectors(), которая позволяет поэлементно прибавить к одному массиву другой. Если результирующий массив изначально пуст, то она сначала заполнит его нулями в том же количестве, сколько элементов в прибавляемом массиве.

Рассмотрим основную функцию экспериментатора Experiment() (см. листинг 3). Она не принимает аргументов, т. к. все необходимые данные о параметрах эксперимента уже сохранены в полях класса. Начало работы происходит с инициализации массива testResults, хранящего все тестовые результаты. Он получает размер, равный количеству тестовых точек (комбинаций уровней факторов), и заполняется пустыми подмассивами для групповых пропускных способностей. Далее определяется порядок выполнения тестов. Его и функцию GenerateOrder() мы рассмотрим отдельно. Затем на каждую тестовую точку создается бенчмарк и записывается в общий массив. После этого выполняется непосредственно тестирование: в указанном порядке вызываются бенчмарки, результирующие массивы задержек конвертируются в массивы пропускных способностей и поэлементно прибавляются к уже накопленным результатам тестов для той же тестовой точки с помощью функции AddVectors(). Затем все элементы во всех подмассивах массива

testResults делятся на количество воспроизведений теста, преобразуясь из суммы всех результатов в их среднее арифметическое. Далее для каждого подмассива вычисляются две характеристики: среднее и среднеквадратическое отклонение, и записываются в финальный массив. После чего функция завершается и возвращает финальный массив пар «среднее – среднее арифметическое», вычисленных для каждой тестовой точки.

```
1.      std::vector<std::pair<ui64, ui64>> TExperimenter::Experiment() const {
2.      std::vector<std::vector<ui64>> testResults(FactorLevels.size(),
   std::vector<ui64>());
3.      std::vector<ui32> order = GenerateOrder();
4.      std::vector<TBenchmark> benchmarks = CreateBenchmarks();
5.
6.      for (ui32 i = 0; i < order.size(); i++) {
7.          auto result = benchmarks[order[i]].Benchmark();
8.          AddVectors(testResults[order[i]], ConvertToThroughput(result,
   order[i]));
9.      }
10.     for (auto& result : testResults)
11.         for (auto& value : result)
12.             value /= Replays;
13.
14.     std::vector<std::pair<ui64, ui64>> resultStatistics(testResults.size());
15.     for (ui32 i = 0; i < testResults.size(); i++)
16.         resultStatistics[i] = Statistics(testResults[i]);
17.     return resultStatistics;
18. }
```

Лист. 3 – Реализация функции Experiment()

Опишем механизм определения порядка. Конструктор экспериментатора получает на вход и сохраняет массив тестируемых комбинаций уровней факторов, которые мы называем точками. Каждая точка однозначно определяется индексом в данном массиве. Исходя из этого мы можем задать порядок выполнения тестов в виде последовательности индексов. Но мы хотим учесть в порядке не только тестируемые точки, но и все отдельные запуски теста в каждой конкретной точке. Так как порядок запусков для нас не имеет значения, то мы каждый запуск одного и того же теста можем задавать одним и тем же числом – все тем же индексом в массиве тестируемых точек. Таким образом, результирующий порядок выполнения запусков тестов можно задать в виде множества подряд идущих чисел, каждое из которых означает запуск теста в точке, определяемой текущим числом взятым в качестве индекса в массиве комбинаций. Именно такая модель реализована в коде и используются внутри функции GenerateOrder(). Сама функция лишь создает соответствующий массив индексов и перемешивает его функцией std::shuffle().

3.1.6. Реализация интерфейсов ввода и вывода бенмарка

Для бенчмарка был реализован пользовательский интерфейс, который позволяет описывать эксперимент текстом из командной строки. Он активируется при запуске программы. Реализация интерфейса в общем проста и состоит из множества функций, осуществляющий сначала вывод названия некоторого параметра эксперимента, а затем считывание введенных данных из командной строки с обработкой возможных ошибок. Отдельного упоминания заслуживает лишь функция считывания уровней факторов ReadFactorLevels(), внутри которой содержится дополнительный алгоритм генерации всех возможных комбинаций уровней факторов. Он работает следующим образом (см. листинг 4). В начале объявляется и заполняется словарь factorsMap, который каждому названию фактора ставит в соответствие массив всех его возможных уровней. Затем создается массив для комбинаций, состоящий из одного пустого элемента. Далее по очереди для каждого фактора осуществляем декартово произведение множества уже имеющихся комбинаций со множеством уровней текущего фактора. Пример описания эксперимента приведен в приложении А.

```
1. std::unordered_map<std::string, std::vector<ui64>> factorsMap;
2. /* Здесь вырезан код, осуществляющий считывание данных в
   factorsMap */
3. // Сгенерировать все возможные комбинации уровней факторов
4. std::vector<TFactorLevels> combinations(1);
5. for (const auto [factor, levels] : factorsMap) {
6.     ui32 n = combinations.size();
7.     ui32 m = levels.size();
8.     combinations.resize(n * m);
9.     // Сопоставить каждой комбинации уже обработанных факторов
10.    // каждый возможный уровень текущего фактора.
11.    for (ui32 i = 0; i < n; i++)
12.        for (ui32 j = 0; j < m; j++) {
13.            combinations[i + n * j] = combinations[i]; //
   Присвоение излишне при j == 0
14.            combinations[i + n * j].SetLevel(factor,
   levels[j]);
15.        }
16.    }
```

Лист. 4 – Фрагмент функции ReadFactorLevels(): генерация комбинаций уровней факторов

Также для бенчмарка был реализован интерфейс вывода результатов. Он в некотором известном порядке выводит данные о паттерне нагрузки, количестве тестируемых факторов, количестве тестовых точек, а затем выводит информацию об

уровнях варьируемых факторов и результатах измерений в виде среднего и среднего арифметического пропускной способности в каждой тестовой точке. Пример результатов эксперимента приведен в приложении 1.

3.1.7. Реализация визуализатора

Для построения графиков по результатам тестирования был реализован консольный скрипт на Python, который мы называем визуализатором. Внутри него для построения двух- и трехмерных столбчатых диаграмм используются библиотеки matplotlib и numpy [33, 34]. При запуске скрипт считывает из консоли путь к файлу с результатами тестов, преобразовывает хранимые в нем сырые данные во внутреннее представление и строит по ним соответствующие графики. Примеры результатов работы визуализатора вы можете видеть в приложении Б и на рисунке 11.

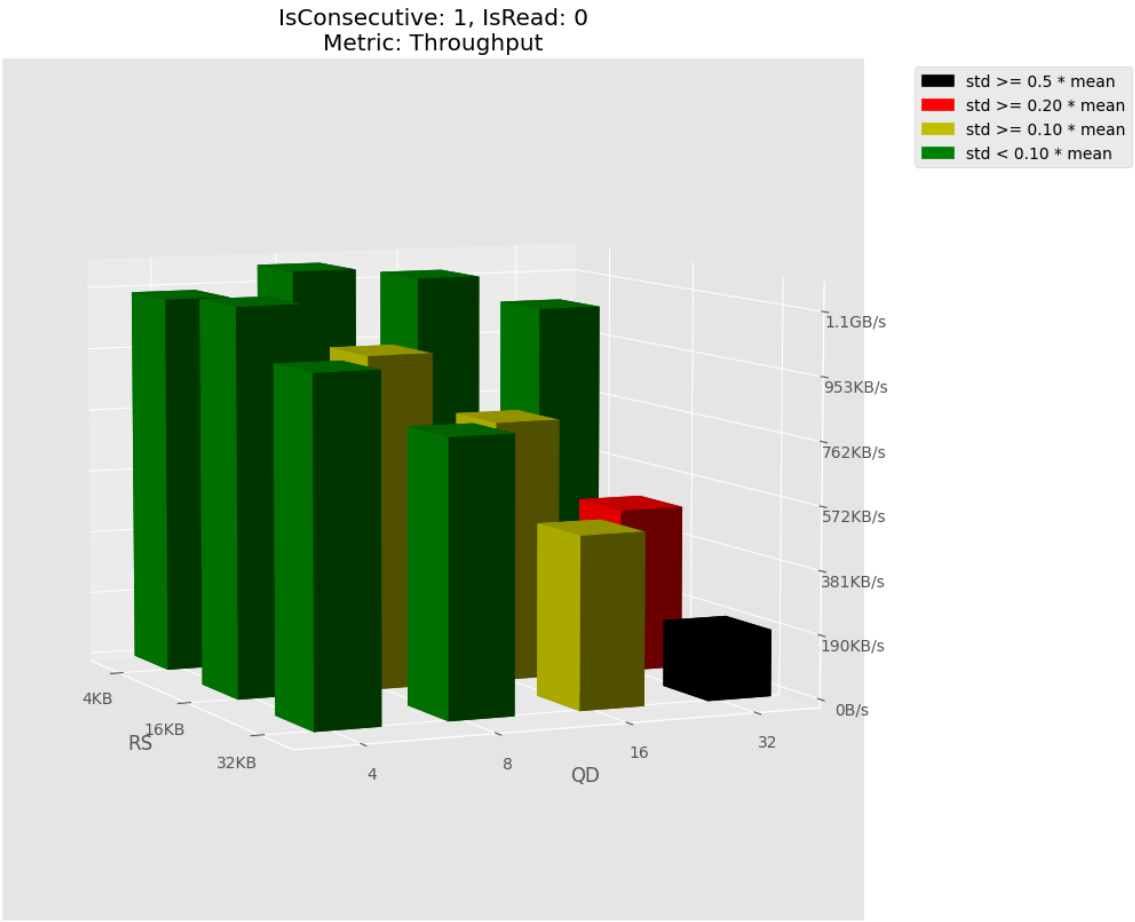


Рис. 11 – Пример диаграммы результатов эксперимента

Рассмотрим рисунок подробнее. На нем представлено трехмерное пространство со множеством столбцов, соответствующих некоторым тестовым точкам. Уровни факторов, определяющие точку для конкретного столбца, определяются по осям горизонтальной плоскости. В данном случае здесь варьировались факторы размер запроса и глубина очереди. Высота столбца определяется получившейся средней пропускной способностью.

Ее можно приблизительно определить по отметкам на вертикальной оси. У данной оси нет названия, так как оно уже отражено вверху открывшегося окна напротив поля «Метрика». Еще выше него находится описание тестируемого паттерна нагрузки через два соответствующих параметра. Справа от диаграммы расположена легенда. На ней каждому возможному цвету столбца ставится в соответствие некоторое относительное ограничение, которому должно удовлетворять среднеквадратическое отклонение, чтобы соответствующий столбец имел данный цвет. Если выполняются несколько ограничений, то цвет определяется по самому строгому из них. На этом закончим рассмотрение результатов реализации ПО и перейдем к описанию результатов тестирования.

3.2. Тестирование ПО

Чтобы проверить работоспособность бенчмарка, он запускался вручную с различными вариантами значений параметров. Варьировались: паттерн нагрузки, количество и уровни тестируемых факторов, параметры разогрева, параметры среды, длительность теста, размер группы. Все запуски завершались либо успешно, либо в аварийном порядке с соответствующим сообщением о выброшенном исключении, если тестировалась обработка крайних случаев.

Дополнительно, были разработаны автоматические тесты для проверки корректности процесса снятия измерений бенчмарка. Суть тестов заключалась в подстановке искусственного интерфейса взаимодействия с ФС вместо реального. В нем реализовывались вместо реальных операций – функции-заглушки, которые возвращают известные нам задержки. Таким образом тестировались случаи, когда:

- Возвращается единственная конкретная задержка;
- Возвращаемые задержки меняются в известном порядке;
- Возвращаемая задержка выбирается случайным образом.

Все запуски тестов завершились успешно. Копию вывода с результатами тестирования можно видеть в приложении В.

Тестирование визуализатора происходило вручную путем запуска бенчмарка с разными параметрами эксперимента и построения графиков по получившимся результатам. Ошибок выявлено не было: во всех протестированных случаях при корректных результатах визуализатор строит корректные двух- или трехмерные диаграммы.

Таким образом, мы делаем вывод, что разработанные программные средства работают корректно.

3.3. Выводы

В результате работы была получена реализация ядра бенчмарка на C++. Он позволяет тестировать производительность файловой системы с заданными уровнями

факторов, паттерном нагрузки и интерфейсом взаимодействия с файловой системой, а также поддерживает функции разогрева и настраиваемой подготовки среды тестирования.

Над ядром бенчмарка реализована надстройка в виде экспериментатора. Он позволяет проводить тестирование во множестве точек пространства факторов с различным числом повторений тестов и усреднением результатов.

Для удобной работы с бенчмарком реализованы текстовые интерфейсы ввода и вывода. Ввод параметров тестирования осуществляется через стандартный ввод, а вывод результатов через стандартный вывод.

Для визуализации результатов тестирования реализован скрипт на Python, который строит по ним двух- и трехмерные столбчатые диаграммы. Для повышения информативности столбцы окрашиваются в некоторый цвет в соответствии с оценкой среднеквадратического отклонения, полученной по тестовым данным.

Для тестирования ядра бенчмарка были реализованы автоматические тесты с подстановкой специального интерфейса взаимодействия с файловой системой с известной скоростью работы. Экспериментатор и скрипт для построения диаграмм были протестированы вручную на множестве различных входных данных.

Заключение

В рамках выпускной квалификационной работы изучался вопрос повышения производительности виртуальных машин за счет оптимизации взаимодействия с памятью. Для этого было проведен анализ научных исследований в области бенчмаркинга файловых систем и современных бенчмарков, а также были рассмотрены те факторы, которые влияют на производительность файловых систем. На основе этого были сделаны выводы применительно к разработке собственного ПО и сформулирована цель работы.

Целью данной работы являлась разработка исследовательского программного обеспечения для оптимизации взаимодействия с файловой системой. Для этого были сформулированы предполагаемый метод решения задачи оптимизации с помощью нашего ПО и выработаны высокоуровневые требования. Далее были описаны модели и методы, которые предполагалось реализовать в ПО, а также было осуществлено проектирование бенчмарка как главного компонента программного комплекса. По результатам проектирования бенчмарк был реализован на C++ и протестирован. В дополнение к нему, был разработан и проверен на практике компонент для построения графиков на Python, называемый визуализатором. Таким образом, цель данной работы можно считать достигнутой.

В рамках дальнейшего развития темы можно выбрать один из двух путей развития. Первый путь предполагает анализ конкретной файловой системы на широкой выборке аппаратуры для выделения общеприменимых выводов о ее скрытых закономерностях и оптимальных параметрах работы, которые затем можно будет внедрить напрямую в код виртуальных машин. Для расширения возможностей тестирования при выборе первого пути целесообразно будет улучшить текущее ПО за счет усовершенствования модели паттерна нагрузки, поддержки новых API и добавления новых факторов.

Второй путь предполагает улучшение ПО таким образом, чтобы оно могло быть встроено напрямую в виртуальную машину и осуществляло оптимизацию самостоятельно и динамически. В качестве возможных нововведений на данном пути можно выделить: автоматическое определение паттерна нагрузки, накопление истории и предугадывание паттернов нагрузки, автоматическая оптимизация параметров взаимодействия с файловой системой в соответствии с текущим или ожидаемым паттерном нагрузки, и др.

Список использованной литературы

1. WMware Glossary. – URL: https://www.vmware.com/ru/topics/glossary/content/virtual-machine.html.
2. Goampaolo D. Practical File System Design with Be File System. – Morgan Kaufmann Publishers, 1999.
3. Traeger A., Zadok E., Joukov N., and Wright, C. A Nine Year Study of File System and Storage Benchmarking // ACM Transactions on Storage, Vol. 4, No. 2, Article 5, 2008.
4. Seltzer M.I., Krinsk, D., Smith K.A., Zhang, X. The case for application-specific benchmarking. // In Proceedings of the IEEE Workshop on Hot Topics in Operating Systems (HOTOS), Rio Rica, AZ, 1999, с. 102-107.
5. Tang D. Benchmarking filesystems // Tech. Rep. TR-19-95, Harvard University, 1995.
6. Lucas H.S. Jr. Performance Evaluation and Monitoring // Computing Surveys, September 1972, с. 79-91.
7. Chen P.M., Patterson D.A. A New Approach to I/O Benchmarks – Adaptive Evaluation, Predicted Performance // UCB/Computer Science Dept. 92/679, University of California at Berkeley, March 1992.
8. Cao Z., Tarasov V., Tiwari S., Zadok E. Towards Better Understanding of Black-Box Auto-Tuning: A Comparative Analysis for Storage Systems // Proceedings of the 2018 Annual USENIX Technical Conference (ATC'18), USENIX Association, 2018.
9. CrystalDiskMark documentation. – URL: https://crystalmark.info/en/software/crystaldiskmark/.
10. IOzone documentation. – URL: http://www.iozone.org/docs/IOzone_msword_98.pdf.
11. ATTO Utilities Installation and Operation Manual. – URL: https://www.atto.com/software/files/notes/PRMA-0267-000MD.pdf, с. 146.
12. Iometer User’s Guide. – URL: https://sourceforge.net/p/iometer/svn/HEAD/tree/trunk/IOmeter/Docs/Iometer.pdf.
13. Filebench wiki. – URL: https://github.com/filebench/filebench/wiki.
14. Г.И. Красовский, Г.Ф. Филаретов. Планирование эксперимента. Минск: Изд-во БГУ, 1982, 302 с.
15. DataLearning: лекции по математической статистике. – URL: http://datalearning.ru/index.php/mathstat.
16. Montgomery D.C. Design and Analysis of Experiments. – Wiley, 2012.
17. cppreference.com: abstract class. – URL: https://en.cppreference.com/w/cpp/language/abstract_class.
18. POSIX.1-2008 – IEEE Standard for Information Technology. – URL:

https://pubs.opengroup.org/onlinepubs/9699919799.2008edition/.
19. Williams A.D. C++ Concurrency in Action: Practical Multithreading. – Manning Publications Co., 2012.
20. Barney, B. POSIX Threads Programming. / Blaise Barney, Lawrence Livermore National Laboratory. – URL: https://computing.llnl.gov/tutorials/pthreads/.
21. Microsoft documentation: Smart pointers (Modern C++). – URL: https://docs.microsoft.com/en-us/cpp/cpp/smart-pointers-modern-cpp?view=msvc-160.
22. Документация Microsoft: Класс unique_ptr. – URL: https://docs.microsoft.com/ru-ru/cpp/standard-library/unique-ptr-class?view=msvc-160.
23. Ccplusplus.com: struct iovec iov. – URL: http://www.ccplusplus.com/2012/02/struct-iovec-iov.html.
24. C++ Core Guidelines E.6: Use RAII to prevent leaks. – URL: https://github.com/isocpp/CppCoreGuidelines/blob/master/CppCoreGuidelines.md#e6-use-raii-to-prevent-leaks.
25. Bjarne Stroustrup's C++ Style and Technique FAQ: RAII. – URL: https://www.stroustrup.com/bs_faq2.html#finally.
26. IBM Documentation: Stack unwinding (C++ only). – URL: https://www.ibm.com/docs/en/zos/2.3.0?topic=only-stack-unwinding-c.
27. ProgrammerInterview.com: What’s the difference between a stack and a heap? – URL: http://www.programmerinterview.com/data-structures/difference-between-stack-and-heap/.
28. cppreference.com: Fixed width integer types. – URL: https://en.cppreference.com/w/cpp/types/integer.
29. Linux manual pages. – URL: https://man7.org/linux/man-pages/index.html.
30. Linux man pages, section 2: system calls. – URL: https://linux.die.net/man/2/.
31. Документация IBM: Работа с дескрипторами файлов. – URL: https://www.ibm.com/docs/ru/aix/7.1?topic=volumes-using-file-descriptors
32. Linux manual pages: open(2). – URL: https://man7.org/linux/man-pages/man2/open.2.html.
33. Matplotlib documentation. – URL: https://matplotlib.org/.
34. NumPy v1.20 Manual. – URL: https://numpy.org/doc/stable/.

Приложение А. Примеры ввода и вывода бенчмарка

Пример описания эксперимента

```
1.  veefore@veefore-HP:~/github/benchmark$ ./run
2.  Please, enter experiment decription.
3.  Reading workload pattern.
4.  Memory access is consecutive (random otherwise) [0 or 1]: 0
5.  Memory access operation is read (write otherwise) [0 or 1]: 0
6.  Reading factors.
7.  Number of factors: 2
8.  Reading factor.
9.  Factors are:
10. "RS" for Request Size
11. Recommended range: [512B, 512MB] (Input value in Bytes!)
12. Default: 64KB
13. "QD" for Queue Depth
14. Recommended range: [1, 256]
15. Default: 8
16. "DIO" for Direct IO
17. Range: {0, 1}
18. Default: 0
19. Factor name ["RS", "QD", "DIO"]: RS
20. Factor levels count: 4
21. Factor level #1: 1024
22. Factor level #2: 4096
23. Factor level #3: 16384
24. Factor level #4: 65536
25. Reading factor.
26. Factors are:
27. "RS" for Request Size
28. Recommended range: [512B, 512MB] (Input value in Bytes!)
29. Default: 64KB
30. "QD" for Queue Depth
31. Recommended range: [1, 256]
32. Default: 8
33. "DIO" for Direct IO
34. Range: {0, 1}
35. Default: 0
36. Factor name ["RS", "QD", "DIO"]: QD
37. Factor levels count: 4
38. Factor level #1: 1
39. Factor level #2: 4
40. Factor level #3: 16
41. Factor level #4: 64
42. Reading warmup parameters.
43. Threshold coefficient (double): 0.15
44. Max duration (ms): 1000
45. Sample size (used in calculating mean and std): 100
46. Reading environment parameters.
47. Filepath: testfile
48. Filesize (MB): 1024
49. Unlink file at filepath [0 or 1]: 1
50. Preparation script:
51. Test duration (ms): 5000
52. Batch size: 10
53. Replays: 2
```

Пример результатов эксперимента

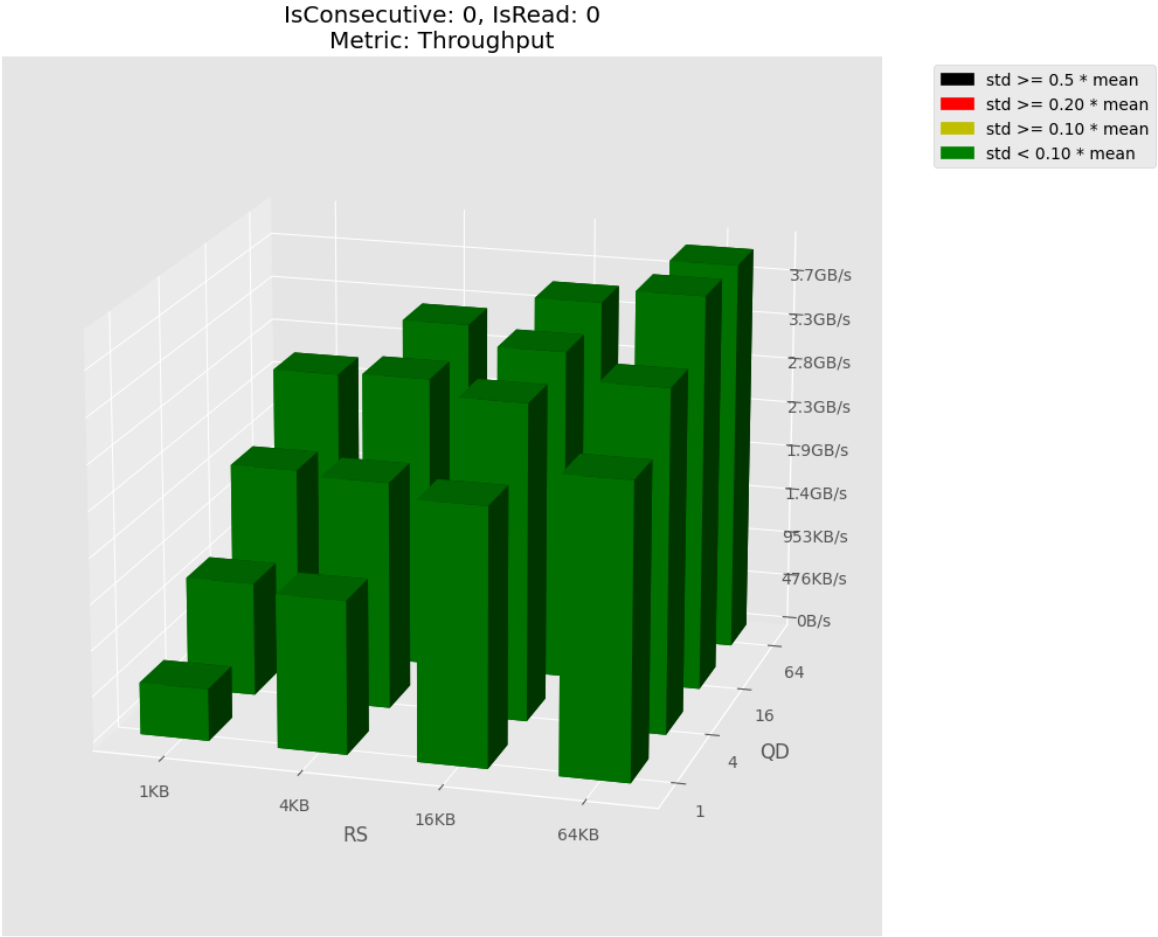
```
1.  0 (Доступ к последовательным участкам памяти отключен)
2.  0 (Тестируется операция записи)
3.  32 (Всего протестировано 32 точки)
4.  2 (Варьировалось 2 фактора)
5.  RS (Первый варьируемый фактор в первой точке)
6.  4096 (Уровень первого фактора)
7.  QD (Второй варьируемый фактор в первой точке)
8.  1 (Уровень второго фактора)
9.  1771784626 (Средняя пропускная способность в байтах в первой точке)
```

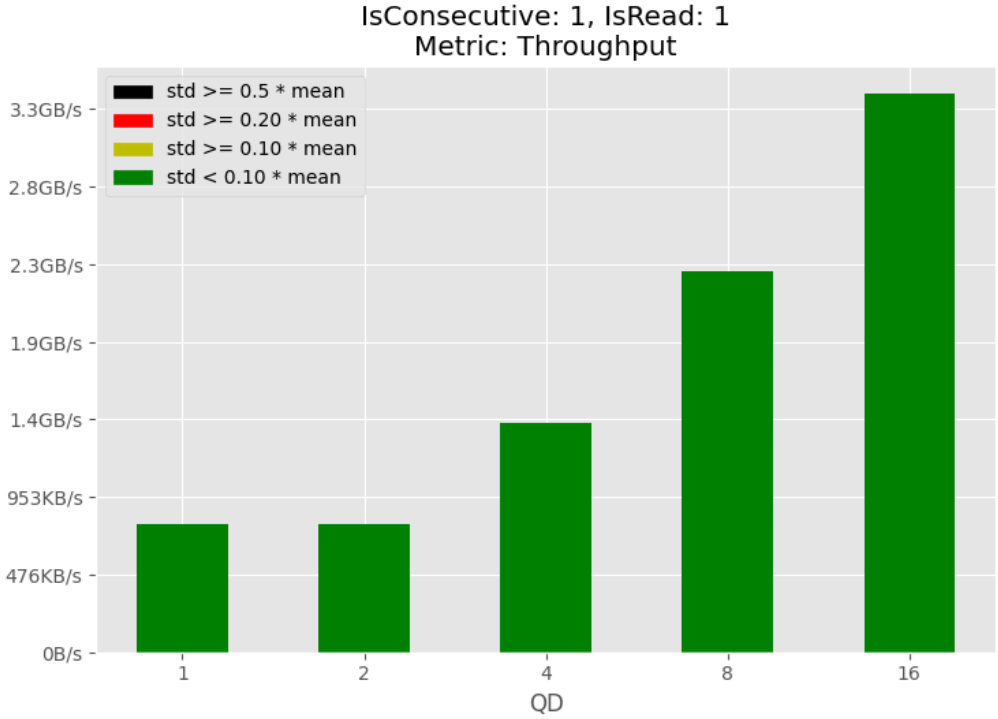
10. 4254189 (Среднеквадратическое отклонение пропускной способности в байтах в первой точке)
11. RS (Первый варьируемый фактор во второй точке)
12. 4096 (...)
13. QD
14. 4
15. 2621464941
16. 12787215
17. RS
18. 4096
19. QD
20. 8
21. 1780064140
22. 26431761
23. RS
24. 4096
25. QD
26. 16
27. 3134464217
28. 22365662
29. RS
30. 8192
31. QD
32. 1
33. 2093382071
34. 11725592
35. RS
36. 8192
37. QD
38. 4
39. 2871949094
40. 31706245
41. RS
42. 8192
43. QD
44. 8
45. 3185569696
46. 19874726
47. RS
48. 8192
49. QD
50. 16
51. 3647870963
52. 17963950
53. RS
54. 16384
55. QD
56. 1
57. 3090063666
58. 13516917
59. RS
60. 16384
61. QD
62. 4
63. 2309689094
64. 12509344
65. RS
66. 16384
67. QD
68. 8
69. 2718328873
70. 40923318
71. RS
72. 16384
73. QD
74. 16
75. 3793761164
76. 48004834
77. RS
78. 32768

79. QD
80. 1
81. 3599770556
82. 11078914
83. RS
84. 32768
85. QD
86. 4
87. 2508547600
88. 29441999
89. RS
90. 32768
91. QD
92. 8
93. 3449458576
94. 38651092
95. RS
96. 32768
97. QD
98. 16
99. 3955926131
100. 22288012
101. RS
102. 65536
103. QD
104. 1
105. 3947678615
106. 21929191
107. RS
108. 65536
109. QD
110. 4
111. 2665364017
112. 90477942
113. RS
114. 65536
115. QD
116. 8
117. 3295395945
118. 83950949
119. RS
120. 65536
121. QD
122. 16
123. 4206682105
124. 66293644
125. RS
126. 2048
127. QD
128. 1
129. 1186656858
130. 1799415
131. RS
132. 2048
133. QD
134. 4
135. 1461677138
136. 9225982
137. RS
138. 2048
139. QD
140. 8
141. 1783175715
142. 12804909
143. RS
144. 2048
145. QD
146. 16
147. 2724758170

148. 22227923
149. RS
150. 1024
151. QD
152. 1
153. 672587705
154. 4647402
155. RS
156. 1024
157. QD
158. 4
159. 1364787479
160. 8613799
161. RS
162. 1024
163. QD
164. 8
165. 1418641421
166. 8706766
167. RS
168. 1024
169. QD
170. 16
171. 2005440220
172. 23559571
173. RS
174. 512
175. QD
176. 1
177. 265227695
178. 6028836
179. RS
180. 512
181. QD
182. 4
183. 817057398
184. 7503503
185. RS
186. 512
187. QD
188. 8
189. 1198989431
190. 7101025
191. RS
192. 512
193. QD
194. 16
195. 1673051816
196. 5920575

Приложение Б. Примеры диаграмм, построенных визуализатором





Приложение В. Результаты общего тестирования

Testing with batch size: 1

Fixed latency test. Latency: 15 us.

Reference mean: 15

Result mean: 15

Reference std: 0

Result std: 0

[✓] Test passed.

Switching latency test. Latencies: {20 us, 30 us, 40 us}.

Reference mean: 30

Result mean: 30

Reference std: 8

Result std: 8

[✓] Test passed.

Random latency test. Latencies: {10 us, 20 us, 60 us}.

Reference mean: 30

Result mean: 30

Reference std: 21

Result std: 21

[✓] Test passed.

Testing with batch size: 5

Fixed latency test. Latency: 15 us.

Reference mean: 75

Result mean: 75

Reference std: 0

Result std: 0

[✓] Test passed.

Switching latency test. Latencies: {20 us, 30 us, 40 us}.

Reference mean: 150

Result mean: 149

Reference std: 8

Result std: 8

[✓] Test passed.

Random latency test. Latencies: {10 us, 20 us, 60 us}.

Reference mean: 150

Result mean: 150

Reference std: 48

Result std: 48

[✓] Test passed.

Testing with batch size: 10

Fixed latency test. Latency: 15 us.

Reference mean: 150

Result mean: 150

Reference std: 0

Result std: 0

[✓] Test passed.

Switching latency test. Latencies: {20 us, 30 us, 40 us}.

Reference mean: 300

Result mean: 300

Reference std: 8

Result std: 8

[✓] Test passed.

Random latency test. Latencies: {10 us, 20 us, 60 us}.

Reference mean: 300

Result mean: 300

Reference std: 68

Result std: 68

[✓] Test passed.

Test passed: 9/9