Санкт-Петербургский государственный электротехнический университет
“ЛЭТИ” им. В. И. Ульянова (Ленина)
(СПбГЭТУ “ЛЭТИ”)

Направление подготовки: 09.03.01 “Информатика и вычислительная техника”
Профиль: “Организация и программирование вычислительных и информационных систем”

Факультет: Компьютерных технологий и информатики
Кафедра: Вычислительной техники

К защите допустить:
Заведующий кафедрой
д. т. н., профессор _____ М. С. Куприянов

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА БАКАЛАВРА

Тема: “Анализ алгоритмов реализации программной транзакционной памяти в компиляторе Clang/LLVM”

Студент _____ В. В. Ивашкин

Руководитель
к. т. н., доцент _____ А. А. Пазников

Консультант
по разработке и стандартизации
программных средств
к. э. н., доцент _____ М. А. Косухина

Консультант от кафедры
к. т. н., доцент, с. н. с. _____ И. С. Зуев

Санкт-Петербург
2023 г.

Санкт-Петербургский государственный электротехнический университет "ЛЭТИ" им. В. И. Ульянова (Ленина)
(СПбГЭТУ "ЛЭТИ")

Направление: 09.03.01 "Информатика и вычислительная техника"
Профиль: "Организация и программирование вычислительных и информационных систем"
Факультет компьютерных технологий и информатики
Кафедра вычислительной техники

УТВЕРЖДАЮ
Заведующий кафедрой ВТ
д. т. н., профессор
(М. С. Куприянов)
"___" _____ 2023 г.

ЗАДАНИЕ
на выпускную квалификационную работу

Студент Ивашкин Валерий Владимирович Группа № 9305

1. Тема Анализ алгоритмов реализации программной транзакционной памяти в компиляторе Clang/LLVM

(утверждена приказом №_____ от_____)

Место выполнения ВКР: Кафедра вычислительной техники СПбГЭТУ «ЛЭТИ» _____

2. Объект и предмет исследования:

Объект исследования – транзакционная память. Предмет исследования – алгоритмы программной транзакционной памяти, реализованные в компиляторе Clang/LLVM.

3. Цель:

Анализ алгоритмов программной транзакционной памяти в компиляторе Clang/LLVM.

4. Исходные данные:

Научные статьи с описаниями транзакционной памяти, библиотека, реализующая алгоритмы в компиляторе Clang/LLVM.

5. Содержание:

Обзор программной транзакционной памяти, описание алгоритмов из библиотеки llvm-transmem, сравнение производительности при разных параметрах алгоритмов.

6. Технические требования:

64-разрядный двухъядерный или с большим количеством ядер процессор.

4 ГБ ОЗУ.

Место на жестком диске до 13 ГБ.

Компилятор Clang/LLVM до 10 версии.

7. Дополнительные разделы:

Разработка и стандартизация программных средств.

8. Результаты:

Пояснительная записка, программный код тестов на языке программирования C++.

Дата выдачи задания
«04» марта 2023 г.

Дата представления ВКР к защите
«22» июня 2023 г.

Студент _____ В. В. Ивашкин

Руководитель _____ А. А. Пазников

Санкт-Петербургский государственный электротехнический университет

“ЛЭТИ” им. В. И. Ульянова (Ленина)
(СПбГЭТУ “ЛЭТИ”)

Направление: 09.03.01 “Информатика и вычислительная техника”
Профиль: “Организация и программирование вычислительных и информационных систем”
Факультет компьютерных технологий и информатики
Кафедра вычислительной техники

УТВЕРЖДАЮ
Заведующий кафедрой ВТ
д. т. н., профессор
(М. С. Куприянов)
“___” _____ 2023 г.

КАЛЕНДАРНЫЙ ПЛАН
выполнения выпускной квалификационной работы

Тема Анализ алгоритмов реализации программной транзакционной памяти в компиляторе Clang/LLVM

Студент Ивашкин Валерий Владимирович Группа № 9305

№ этапа	Наименование работ	Срок выполнения
1	Обзор литературы по теме работы	05.03.2023-15.03.2023
2	Изучение библиотеки алгоритмов	16.03.2023-01.04.2023
3	Внедрение библиотеки в Clang/LLVM	02.04.2023-25.04.2023
4	Разработка тестов производительности	26.04.2023-15.05.2023
5	Тестирование алгоритмов	16.05.2023-24.05.2023
6	Оформление пояснительной записки	25.05.2023-16.06.2023
7	Предварительное рассмотрение работы	16.06.2023-18.06.2023
8	Представление работы к защите	22.06.2023

Руководитель
к. т. н., доцент _____ А. А. Пазников
Студент _____ В. В. Ивашкин

РЕФЕРАТ

Пояснительная записка содержит: 42стр., 11рис., 3 таблицы, 2 приложения.

Объектом исследования является программная транзакционная память.

Предметом исследования являются алгоритмы программной транзакционной памяти в компиляторе Clang/LLVM.

Цель работы: исследование принципов работы транзакционной памяти, анализ алгоритмов реализации программной транзакционной памяти в компиляторе Clang/LLVM, сравнение производительности данных алгоритмов.

В процессе выполнения исследования была проведена детальная работа по описанию транзакционной памяти и изучению нескольких алгоритмов, которые позволяют реализовать ее в программном виде. Для анализа этих алгоритмов на языке программирования C++ были разработаны и проведены тесты, использующие многопоточность.

По итогу была решена задача анализа алгоритмов реализации программной транзакционной памяти в компиляторе Clang/LLVM.

Данная работа может быть использована для ознакомления с принципами работы программной транзакционной памяти и различиями между разными алгоритмами, которые используются в разных компиляторах и библиотеках.

ABSTRACT

In the course of the study, a detailed work was performed on the description of transactional memory and the study of several algorithms that allow to implement it in software form. To analyze these algorithms, tests using multithreading were developed and conducted in a programming language C++.

As a result, the task of analyzing algorithms for the implementation of software transactional memory in the Clang/LLVM compiler was solved.

This work can be used to get acquainted with the principles of software transactional memory and the differences between different algorithms that are used in different compilers and libraries.

СОДЕРЖАНИЕ

ОПРЕДЕЛЕНИЯ, ОБОЗНАЧЕНИЯ И СОКРАЩЕНИЯ

Clang – транслятор C-подобных языков

LLVM – набор компиляторов из языков высокого уровня, системы оптимизации, интерпретации и компиляции в машинный код.

STM – Software Transactional Memory, программная транзакционная память.

Deadlock – взаимоблокировка, ситуация, когда несколько процессов находятся в состоянии бесконечного ожидания ресурсов, занятых друг другом.

Race condition – состояние гонки, ситуация, которая возникает, когда два и более потока имеют доступ к общей переменной.

Метаданные – информация о другой информации, раскрывающая сведения о признаках и свойствах, характеризующих какие-либо сущности, позволяющие автоматически искать и управлять ими в больших информационных потоках.

ВВЕДЕНИЕ

Все больше распространяются многоядерные вычислительные системы, для которых необходимы многопоточные приложения. Основная проблема многопоточности – организация масштабируемого доступа параллельных потоков к разделяемым структурам данных. Одним из решений данной проблемы является транзакционная память, которая призвана упростить и оптимизировать работу с разделяемыми структурами данных.

Традиционные подходы к синхронизации памяти, такие как использование блокировок, могут быть сложными в реализации и подвержены проблемам, таким как взаимоблокировка и состояние гонки. Транзакционная память предлагает более декларативный подход к синхронизации, который позволяет программисту определить критические секции кода в виде транзакций, внутри которых выполняются операции чтения и записи памяти. Эти транзакции могут быть автоматически управляемыми и гарантировать атомарность и изоляцию операций.

Транзакционная память может быть реализована как аппаратное или программное средство. В случае аппаратной реализации, процессор предоставляет дополнительные инструкции для начала, фиксации и отката транзакций, а также аппаратное обнаружения конфликтов. В программной реализации, транзакционная память предоставляется через специальные библиотеки или расширения языка программирования.

Библиотека llvm-transmem [1] предоставляет систему, реализующую алгоритмы STM с различными методами и стратегиями обнаружения и разрешения конфликтов. Основная цель библиотеки – облегчение работы с транзакционной памятью для обычных разработчиков и разработчиков компиляторов путём запрета явных прерываний транзакции и заменой лексической области видимости на лямбда-выражения [2].

Цель работы состоит в изучении и анализе алгоритмов, применяемых для реализации программной транзакционной памяти в компиляторе Clang/LLVM.

Объектом исследования является транзакционная память. Предметом исследования – алгоритмы программной транзакционной памяти в компиляторе Clang/LLVM.

В первом разделе работы приводятся основные сведения о транзакционной памяти, характеристиках алгоритмов транзакционной памяти. Во втором разделе описываются алгоритмы, реализованные в библиотеке llvm-transmem, рассматриваются их основные различия. Третий раздел содержит проведение тестирования алгоритмов и сравнение результатов. Четвертый раздел посвящен разработке и стандартизации программных средств.

1 Программная транзакционная память

Программная транзакционная память является концепцией и технологией, которая предоставляет альтернативный подход к управлению совместным доступом к данным в многопоточных приложениях. Вместо использования блокировок и критических секций, STM позволяет программистам определять транзакции, которые могут выполняться параллельно и обеспечивают атомарность и изоляцию данных.

Для реализации программной транзакционной памяти необходим компилятор, поддерживающий конструкции для создания транзакционных секций, и библиотека, в которой есть необходимые функции для работы с транзакционной памятью.

Алгоритмы работы программной транзакционной памяти реализуются в библиотеке и различаются по:

— способу хранения информации о состоянии защищаемых областей памяти;

— политике обновления объектов в памяти;

— стратегии обнаружения конфликтов;

— методу разрешения конфликтов.

1.1 Хранение информации о состоянии защищаемых областей памяти.

Конфликтные операции возникают, когда несколько транзакций пытаются изменить одни и те же области памяти одновременно. Чтобы обнаружить такие конфликты, нужно отслеживать изменения состояния используемых областей памяти. Для этого информация о состоянии может быть соответствовать областям памяти различной степени гранулярности.

На сегодняшний день используется два уровня гранулярности: уровень программных объектов (object-based STM) и уровень слоев памяти (word-based STM).

На уровне программных объектов (объектно-ориентированная STM), информация о состоянии относится к отдельным объектам данных, которые могут быть изменены во время транзакции. Каждый объект хранит метаданные, такие как версия или временная метка, которые отслеживают изменения состояния объекта. Это позволяет обнаруживать конфликты, когда две транзакции пытаются изменить один и тот же объект.

На уровне слоев памяти (словесно-ориентированная STM), информация о состоянии относится к отдельным словам или байтам памяти. Вся память разбивается на блоки фиксированного размера, и каждый блок содержит информацию о своем состоянии, например, версию или временную метку. Таким образом, система может отслеживать изменения состояния каждого блока памяти и обнаруживать конфликты, когда транзакции пытаются изменить один и тот же блок.

Оба уровня гранулярности имеют свои преимущества и недостатки и могут быть применены в разных сценариях в зависимости от требований приложения и характеристик системы. Рассмотрим преимущества и недостатки более подробно.

1.1.1 Объектно-ориентированная STM.

Преимущества объектно-ориентированной STM:

1. Более высокая абстракция: уровень объектов позволяет работать с более высокоуровневыми абстракциями, такими как переменные, структуры данных и объекты классов. Это делает программирование с использованием транзакций более удобным и позволяет использовать привычные концепции объектно-ориентированного программирования.

2. Более точное определение конфликтов: поскольку информация о состоянии хранится непосредственно в объектах, система управления транзакционной памятью может более точно определить конфликтные операции

между транзакциями. Это позволяет обнаруживать больше потенциальных конфликтов и уменьшает вероятность нежелательных эффектов.

Недостатки объектно-ориентированной STM:

1. Больший объем данных: хранение информации о состоянии на уровне каждого объекта может потребовать дополнительного объема памяти. Если приложение работает с большим количеством объектов, это может привести к увеличению расхода памяти и негативно сказаться на производительности.

2. Сложность управления объектами: уровень объектов требует более сложного управления объектами и их состоянием. Необходимо обеспечить синхронизацию доступа к объектам и обновление соответствующих метаданных при изменении состояния. Это может потребовать дополнительной работы по проектированию и реализации.

1.1.2 Словесно-ориентированная STM.

Преимущества словесно-ориентированной STM:

1. Более эффективное использование памяти: уровень слов позволяет более эффективно использовать память, поскольку информация о состоянии хранится на уровне блоков памяти, а не в каждом объекте отдельно. Это может привести к сокращению накладных расходов на хранение метаданных и более компактной структуре данных.

2. Более простая реализация: словесно-ориентированная STM обычно имеет более простую реализацию, поскольку нет необходимости отслеживать состояние каждого отдельного объекта. Это может сделать систему более легковесной и производительной.

Недостатки словесно-ориентированной STM:

1. Меньшая гибкость: уровень слов может быть менее гибким в управлении конфликтами. Это может ограничить возможности для определения стратегий разрешения конфликтов или приоритета операций.

2. Сложности в обработке некоторых типов данных: в словесно-ориентированной STM манипулирование слоями памяти может быть сложнее, особенно при работе с некоторыми сложными типами данных или структурами.

3. Возникновение ложных конфликтов.

Когда используется программная транзакционная память, метаданные, которые отслеживают состояние и конфликты между потоками, обычно занимают гораздо меньше места в памяти, чем само линейное адресное пространство процесса. Это может привести к ситуации, когда несколько потоков выполняют транзакции, обращаясь к разным областям памяти, но используя одни и те же метаданные о состоянии. Если хотя бы один поток выполняет операцию записи, возникает ложный конфликт [5].

Ложный конфликт в данном случае означает, что, хотя потоки обращаются к разным участкам памяти, система транзакционной памяти рассматривает их как конфликтные из-за использования общих метаданных. Даже если на самом деле нет гонки за данными или взаимодействия между потоками, эти ложные конфликты приводят к откату транзакций, аналогично настоящим конфликтам. Это влияет на производительность программы, так как время выполнения увеличивается из-за откатов.

Таким образом, ложные конфликты являются нежелательными эффектами в программной транзакционной памяти, которые могут возникать из-за использования общих метаданных при выполнении операций чтения и записи в разных областях памяти. Устранение или снижение ложных конфликтов является одной из задач оптимизации программной транзакционной памяти и может включать в себя применение более точных методов обнаружения конфликтов, изменение структуры данных или применение других оптимизаций для сокращения откатов и повышения эффективности выполнения транзакций.

1.2 Обновление объектов в памяти.

Политика обновления объектов в памяти определяет, когда изменения объектов в рамках транзакции будут записаны в память. Существуют две основные политики обновления: ленивая (lazy) и ранняя (eager).

Ленивая политика обновления объектов в памяти означает, что все операции над объектами откладываются до момента фиксации транзакции. Вместо того чтобы немедленно записывать изменения в память, они регистрируются в специальном журнале (redo log). При фиксации транзакции эти операции выполняются из журнала в отложенном виде. Это позволяет упростить процедуры отмены и восстановления, так как все изменения могут быть отменены вместе. Однако операция фиксации будет занимать больше времени.

Ранняя политика обновления означает, что все изменения объектов немедленно записываются в память. При этом в журнале отката (undo log) регистрируются все выполненные операции с памятью. Если возникает конфликт с другой транзакцией, используется журнал отката для восстановления оригинального состояния модифицированных областей памяти. Ранняя политика обновления обеспечивает быструю операцию фиксации, так как изменения применяются немедленно, но процедура отмены может занимать больше времени, так как требуется восстановление всех изменений.

1.3 Обнаружение конфликтов.

Стратегия обнаружения конфликтов определяет тот момент во времени, когда алгоритм обнаружения конфликта будет активирован.

Существуют две стратегии обнаружения конфликтов: отложенная (lazy conflict detection) и пессимистичная (eager conflict detection).

Отложенная стратегия обнаружения конфликтов означает, что алгоритм обнаружения конфликтов запускается только на этапе фиксации транзакции. Во время выполнения транзакции никакие проверки на конфликты не

выполняются. Это позволяет избежать дополнительных расходов на обнаружение конфликтов во время выполнения, но имеет недостаток в виде большого временного интервала между возникновением конфликта и его обнаружением. Если конфликт возникает в конечной стадии транзакции, то все предыдущие операции транзакции были выполнены напрасно.

Пессимистичная стратегия обнаружения конфликтов предполагает запуск алгоритма обнаружения конфликтов при каждой операции обращения к памяти. Это позволяет обнаруживать конфликты намного раньше, ещё до момента фиксации транзакции. Такой подход гарантирует, что любой конфликт будет обнаружен немедленно и позволяет принять соответствующие меры для его разрешения. Однако пессимистичная стратегия может привести к дополнительным накладным расходам из-за постоянных проверок на конфликты и в некоторых случаях может привести к увеличению числа откатов транзакций.

2 Алгоритмы реализации STM в библиотеке llvm-transmem

Библиотека llvm-transmem предоставляет систему, реализующую алгоритмы STM в компиляторе Clang/LLVM. Для добавления STM используется преобразование промежуточного кода LLVM, который создаётся с помощью фронтенда Clang. После этого создается объектный код. Получившийся код компонуется с файлом требуемого алгоритма. Схема имплементации приведена на рисунке 2.1.

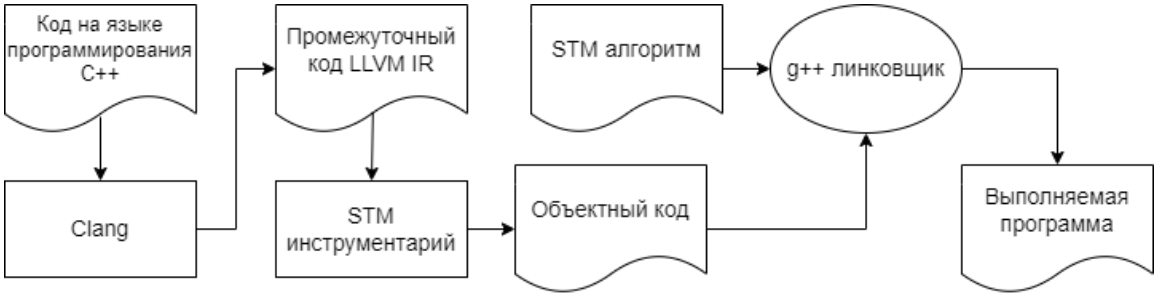

Рисунок 2.1 – Схема имплементации алгоритма

В данной работе будут рассмотрены алгоритмы, представленные в библиотеке llvm-transmem:

- TinySTM: WriteThrough;
- TinySTM: WriteBack;
- TinySTM: CTL;
- TML_Eager;
- TML_Lazy.

2.1 Описание алгоритма TinySTM: WriteThrough

Чтобы отслеживать состояние защищаемых областей введена структура “orec” что означает Ownership record, то есть запись владения. В этом алгоритме используются структуры ORECTABLE, которые хранят записи владения и временные метки всех потоков. Запись владения создается перед началом взаимодействия с данными.

Также этот алгоритм характеризуется ранней политикой обновления объектов в памяти и пессимистичной стратегией обнаружения конфликтов.

Алгоритм выполнения транзакций представлен на рисунке 2.2

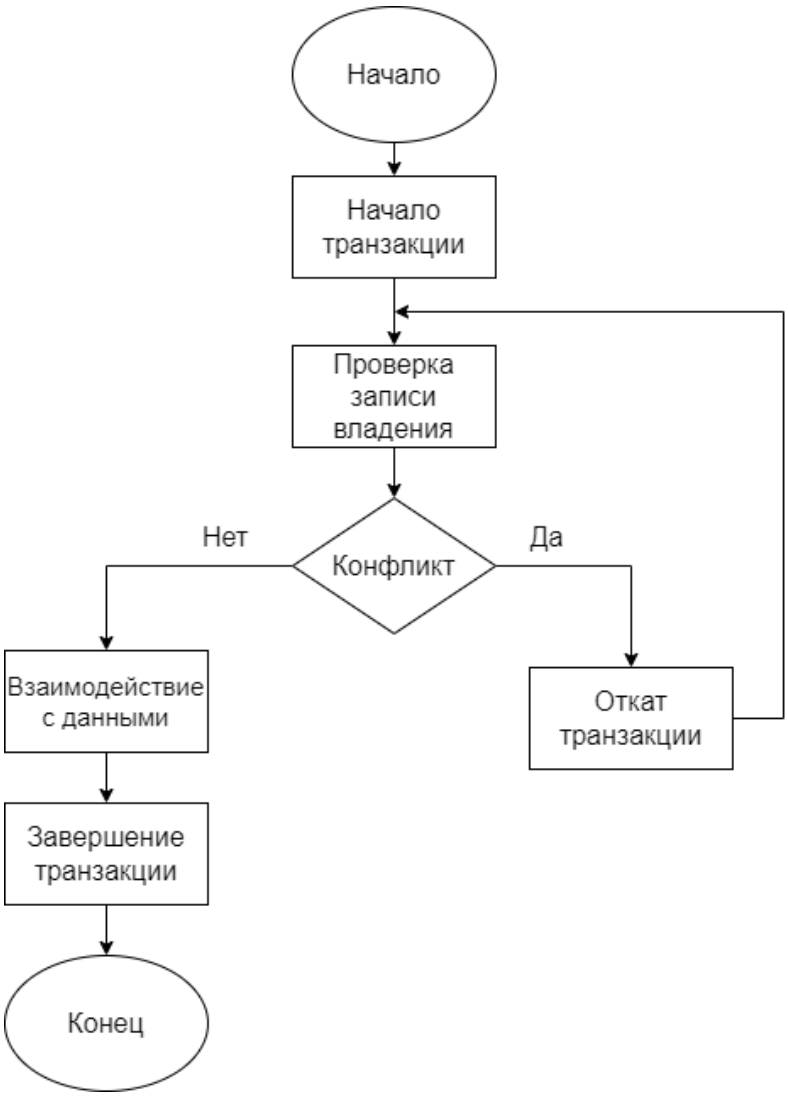

Рисунок 2.2. – Алгоритм выполнения транзакции

2.2 Описание алгоритма TinySTM: CTL

Данный алгоритм, наоборот, характеризуется ленивой политикой обновления объектов в памяти и отложенной стратегией обнаружения конфликтов.

Алгоритм выполнения транзакций представлен на рисунке 2.3.

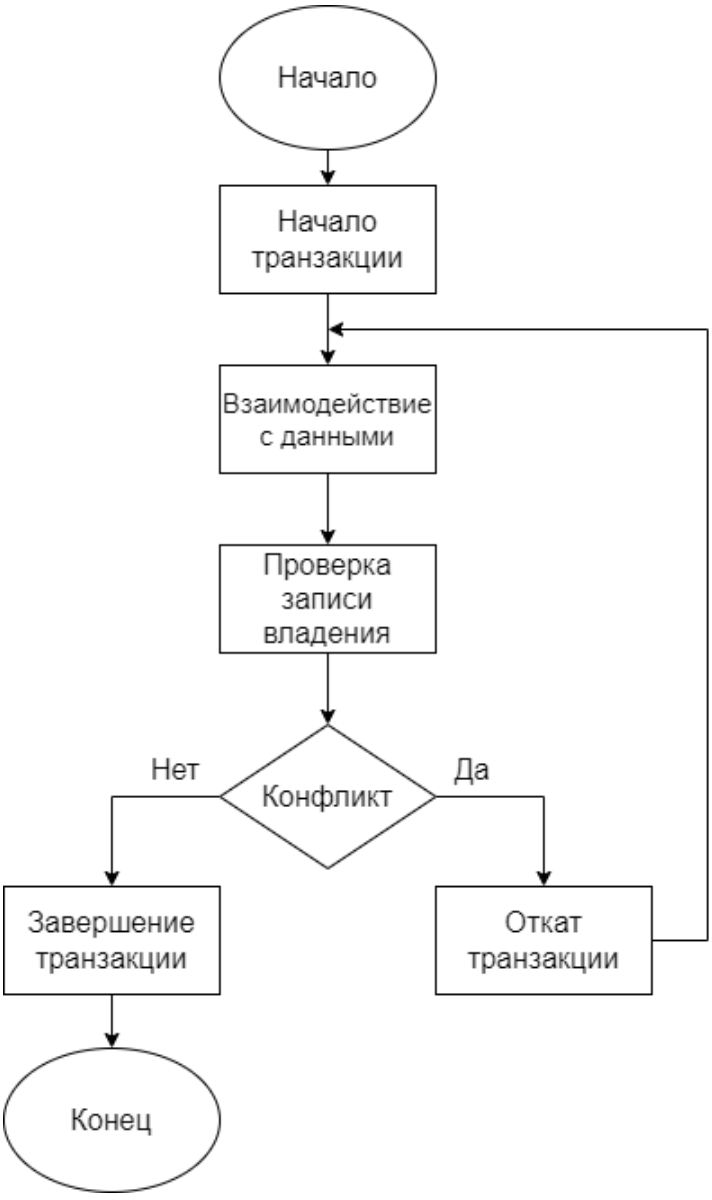

Рисунок 2.3 – Алгоритм выполнения транзакции

2.3 Описание алгоритма TinySTM: WriteBack

Алгоритм использует ленивую политику обновления и комбинирует отложенную и пессимистичную стратегию обнаружения конфликтов.

Алгоритм выполнения транзакции представлен на рисунке 2.4.

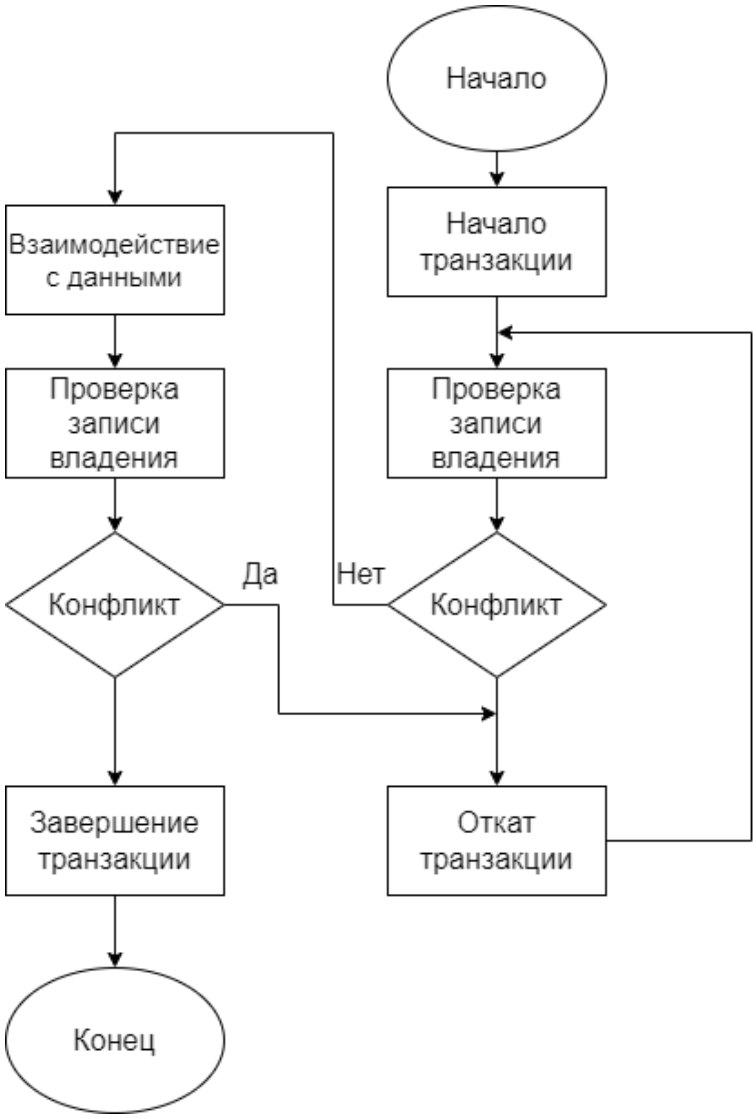

Рисунок 2.4 – Алгоритм выполнения транзакции

2.4 Описание алгоритма TML: Eager

В отличие от предыдущих алгоритмов, в этом алгоритме отслеживание блокировки происходит путем добавления глобальной переменной. В остальном этот алгоритм похож на TinySTM: WriteThrough. Используется ранняя политика обновления объектов в памяти и пессимистичная стратегией обнаружения конфликтов.

Алгоритм выполнения транзакции представлен на рисунке 2.5.

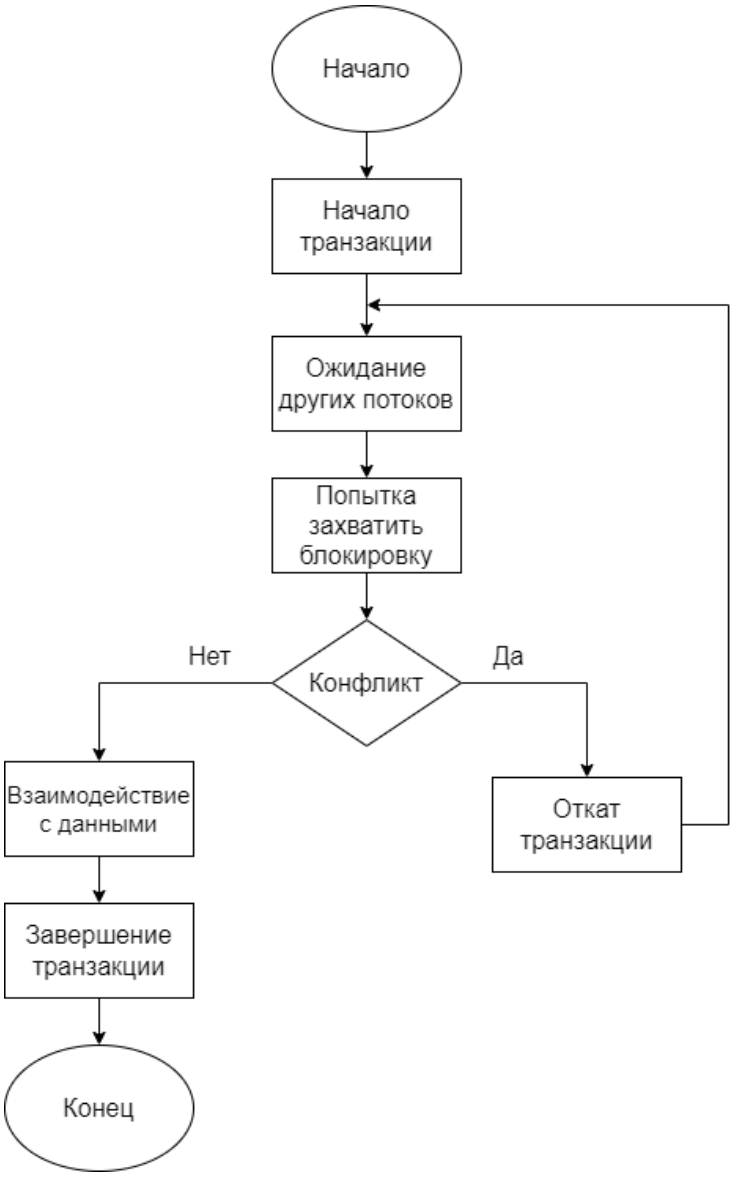

Рисунок 2.5 – Алгоритм выполнения транзакции

2.5 Описание алгоритма TML: Lazy

В этом алгоритме также используется глобальная переменная для отслеживания блокировок, но, в отличие от TML: Eager, используется ленивая политика объектов в памяти и отложенная стратегия обнаружения конфликтов.

Алгоритм выполнения транзакции представлен на рисунке 2.6.

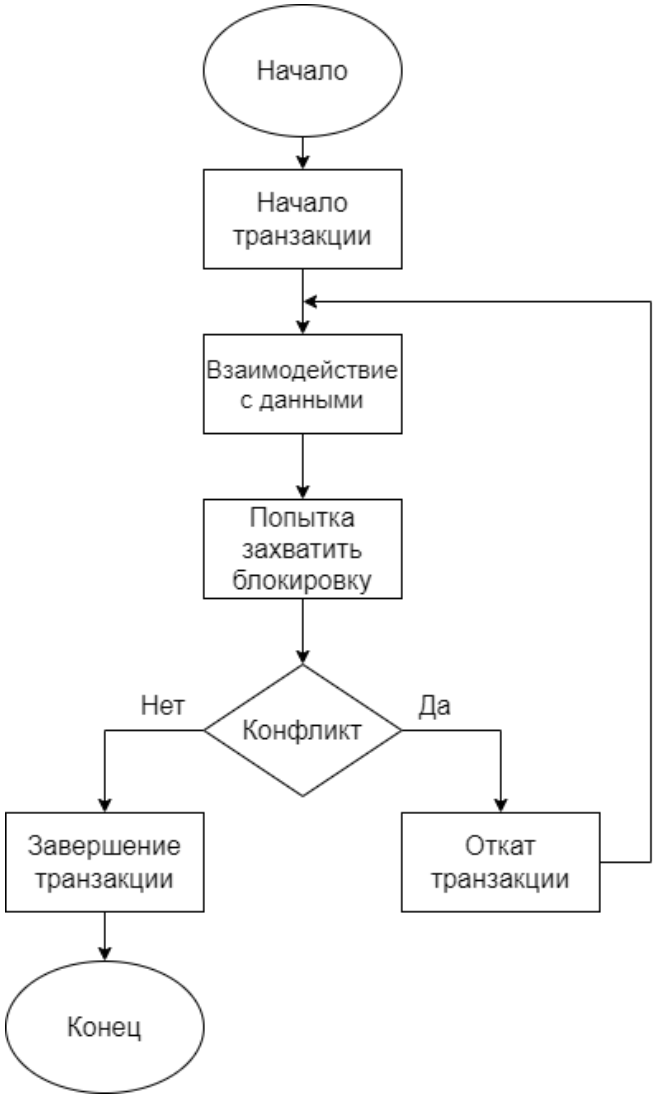

Рисунок 2.6 – Алгоритм выполнения транзакции

3 Проведение тестирования рассматриваемых алгоритмов

Для проведения тестирования на языке программирования C++ была написана программа, которая содержит многопоточную реализацию функций суммирования, вставки в массив, двусвязный список и дерево.

3.1 Реализация программы для анализа алгоритмов STM

Основу программы составляет класс NumbersTest, код которого приведён в приложении А. NumbersTest содержит функцию генерации последовательности случайных чисел и функцию создания потоков, которые параллельно исполняют функцию worker. Функция worker переопределяется в классе теста, где определена транзакционная секция.

Пример кода на C++ для реализации теста параллельного суммирования:

```
class ArraySumTest: public NumbersTest {
public:
    virtual void setup() {
        m_sharedSum = 0;
    }
    /*
    *  Check in case STM algorithm make wrong result
    */
    virtual bool check() {
        size_t refSum = 0;
        for(size_t i = 0; i < m_inputSize; i++) {
            refSum += m_input[i];
        }
        return (refSum == m_sharedSum);
    }
protected:
    /*
    * Function that threads execute in parallel
    * start&&end - points at part of sequience data for exact thread
    */
    virtual void worker(size_t start, size_t end) {
        size_t localSum = 0;
        for(size_t i = start; i < end; i++) {
            localSum += m_input[i];
        }
        //First way to express transaction
        /*TX_BEGIN{
            m_sharedSum += localSum;
        } TX_END;*/

        //Second way to express transaction
        {
            TX_RAII;
            m_sharedSum += localSum;
```

```
            }
        }

        size_t m_sharedSum;
};
```

3.2 Анализ результатов тестирования

Эксперименты проводились на вычислительной системе оборудованной 4-ядерным процессором AMD Ryzen 5 2500U и объемом кеш-памяти L1 – 384 Кбайт, L2 – 2 Мбайт, L3 – 4 Мбайт.

Использовалась виртуальная машина с объёмом оперативной памяти 4 Гбайт, операционной системой Ubuntu 20.04. Компилятор Clang/LLVM без флагов оптимизации. Так как использовалась виртуальная машина, максимальное логических процессоров будет равно 4 [9], что повлияет на результаты.

Тестирование проводилось для 1, 2, 4, 8, 16 потоков. Количество выполненных в каждом тесте транзакций равно 1 миллиону. В качестве показателя эффективности использовалось количество операций в милисекунду (Ops/ms).

На рисунке 3.1 представлен график зависимости Ops/ms от количества потоков для алгоритма вставки в массив.

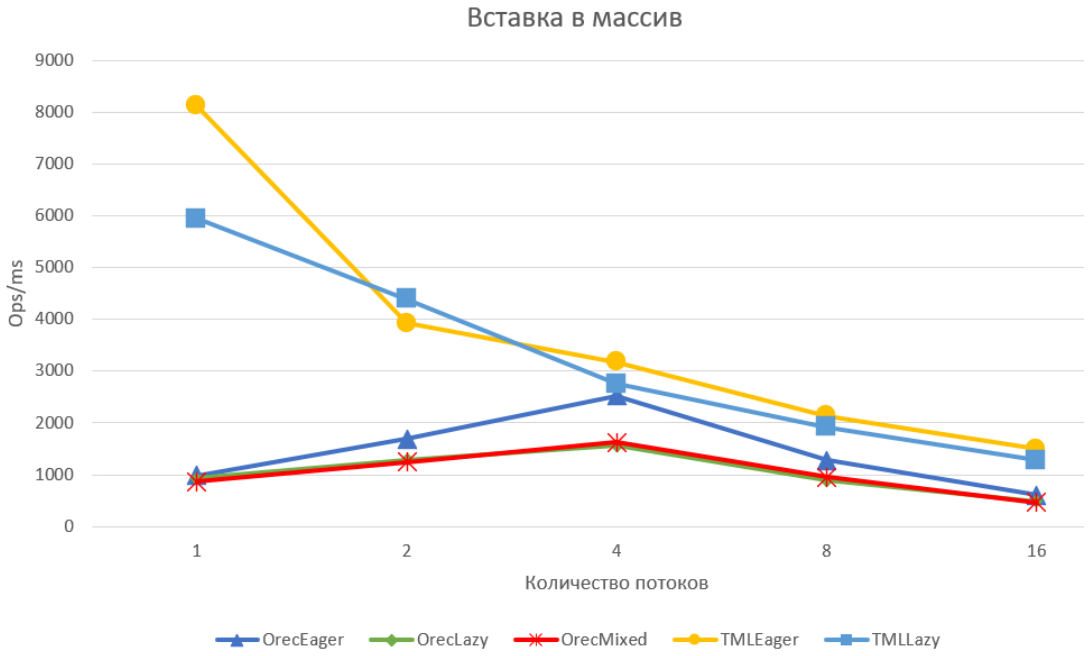

Рисунок 3.1 – график зависимости для вставки в массив

Здесь и в дальнейшем алгоритмы TinySTM:WriteThrough, TinySTM: WriteBack, TinySTM:CTL будут обозначаться как OrecEager, OrecMixed, OrecLazy соответственно.

Как видно из графика, TMLEager и TMLLazy сильно выигрывают в производительности на одном и двух потоках. Однако с увеличением числа потоков, производительность падает, в отличие от алгоритмов, использующих записи владения. Это может быть связано с тем, что они используют единую блокировку на все потоки. Пока Orec выполняет работу с данными, TML ожидают освобождения блокировки, прежде чем приступить к работе.

Помимо этого, можно наблюдать, что алгоритм OrecEager быстрее других Orec алгоритмов, которые используют ленивую политику обновления объектов в памяти. С большим шансом это связано с тем, что данный алгоритм тратит меньше ресурсов для того, чтобы завершить транзакцию.

На рисунке 3.2 представлен график зависимости Ops/ms от количества потоков для параллельного сложения.

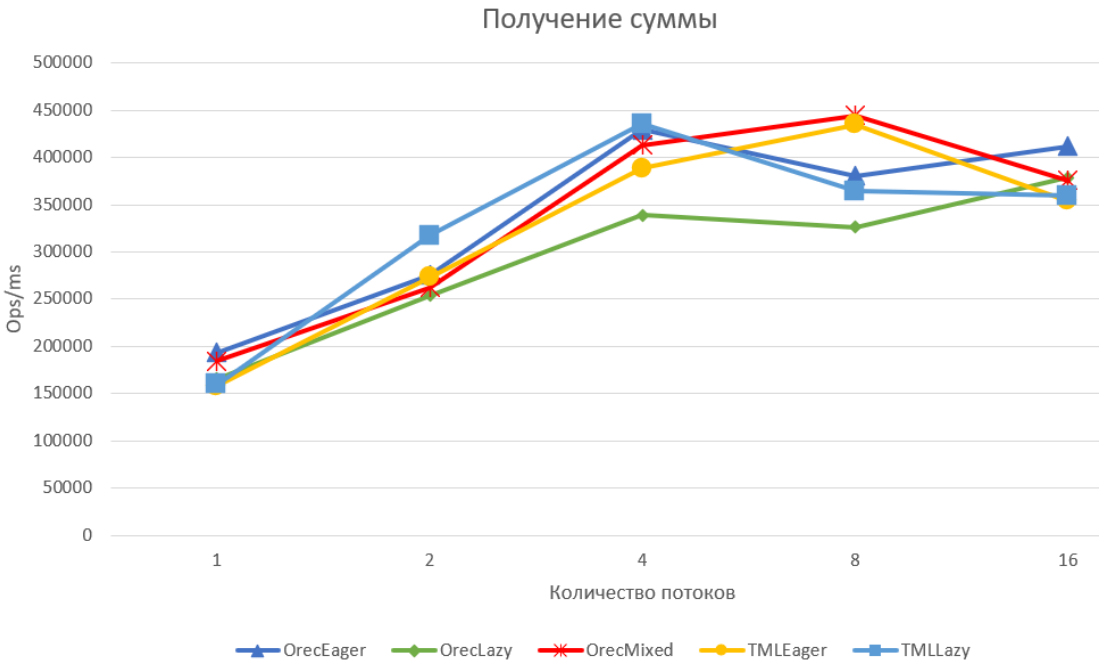

Рисунок 3.2 – график зависимости для суммы

По графику заметно, что алгоритм OrecMixed, быстрее OrecLazy благодаря тому, что, в отличие от него он также использует и пессимистичную стратегию обнаружения конфликтов. Благодаря этому он быстрее обнаруживает конфликты и не тратит ресурсы на выполнение операции. Это особенно заметно на большем количестве потоков, когда конфликты могут возникать чаще. В то время как OrecEager на меньшем количестве потоков выигрывает в скорости за счет того, что использует раннюю политику обновления объектов в памяти.

В целом, при использовании 8 потоков алгоритмы с ленивой политикой обновления объектов в памяти оказались хуже тех, кто использует раннюю политику. Я предполагаю, что это связано с тем, что выполнение redo log`a сравнимо с затратами на основные вычисления.

На рисунке 3.3 представлен график зависимости Ops/ms от количества потоков для вставки значения в хеш-таблицу для двух алгоритмов с ленивой политикой обновления объектов в памяти.

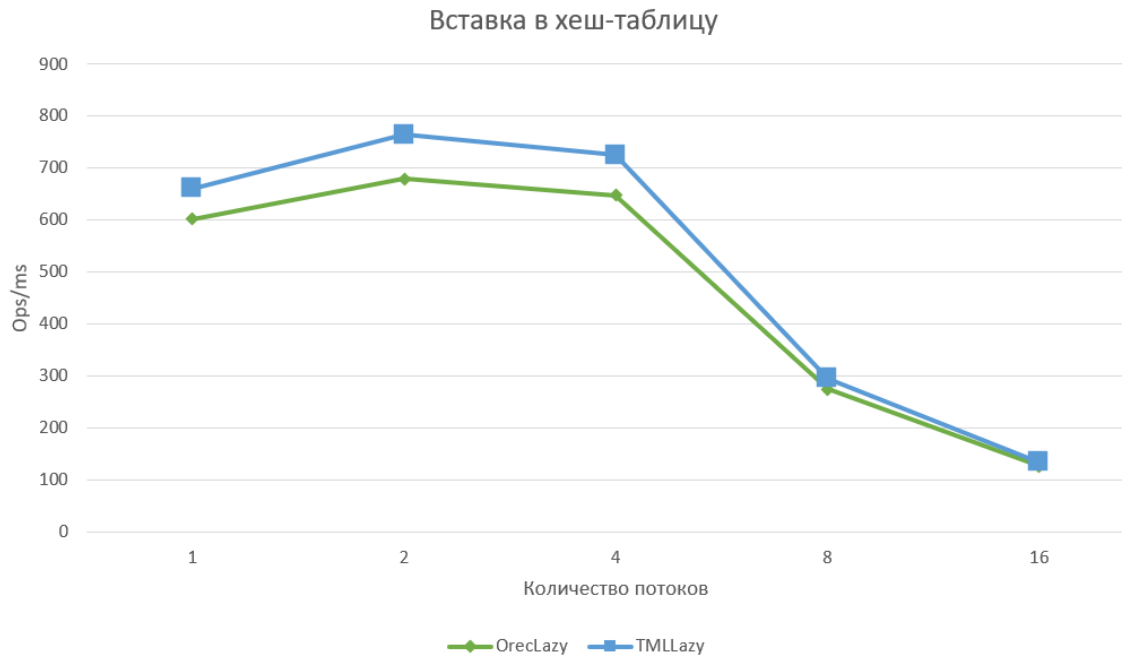

Рисунок 3.3 – график зависимости для вставки в хеш-таблицу

Анализируя представленный график, можно понять, что TMLLazy действует эффективнее на текущих тестах.

В целом OrecLazy показал себя хуже всех алгоритмов с ленивой политикой обновления объектов в памяти.

На рисунке 3.4 представлен график зависимости Ops/ms от количества потоков для удаления элемента дерева для двух алгоритмов с ранней и ленивой политикой обновления объектов в памяти

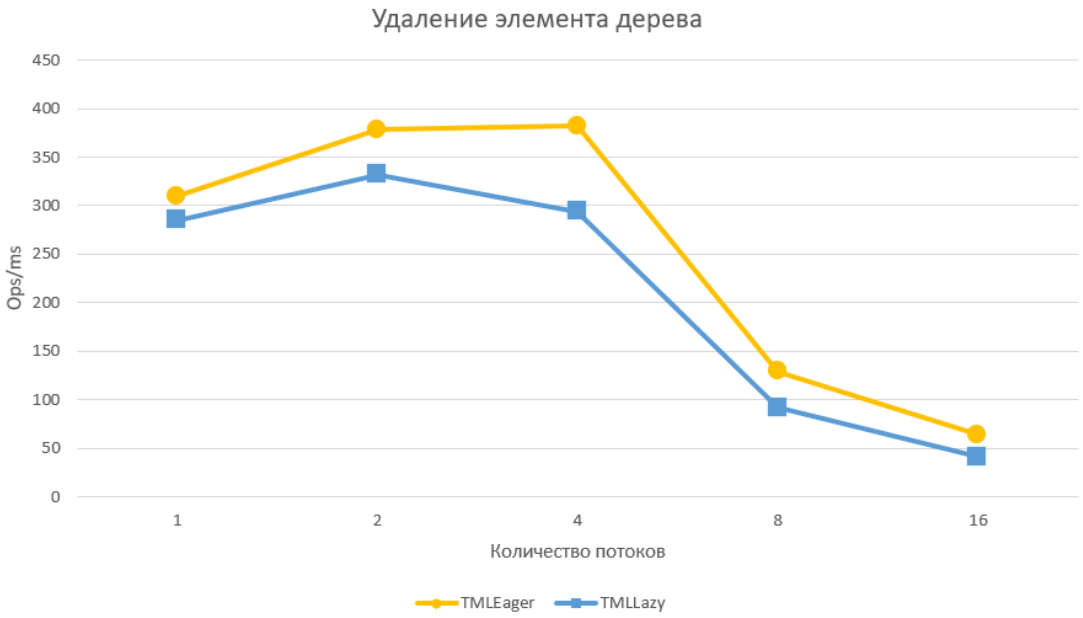

Рисунок 3.4 – график зависимости для удаления элемента дерева

Как видно их графика, если сравнивать два похожих алгоритма, отличающихся только политикой обновления объектов в памяти, быстрее работает тот, которому не приходится тратить дополнительные ресурсы во время операции фиксации транзакции. Однако, если будет такая задача, где часто встречаются конфликты, ситуация может сильно поменяться.

4 Разработка и стандартизация программных средств

В данном разделе приводится описание диаграммы Ганта, описывающей процесс разработки программ, определение теоретических затрат на выполнение проекта и определение классификационного кода.

4.1 Диаграмма Ганта

Результатом данной работы является сравнение производительности алгоритмов, для чего был написан программный код. Чтобы контролировать ход выполнения работы и при необходимости вносить изменения в организацию выполнения работы необходимо формализовать представление совокупности необходимых работы.

Так как ВКР является небольшим проектом, то для планирования и управления ходом проекта подойдет диаграмма Ганта.

Диаграмма Ганта [6] представляет собой отображение хода работ в виде отрезков времени. Каждый этап работы может характеризоваться датами начала и конца этапа работы, визуальным представлением в виде отрезка, и исполнителями. Но в данном графике нет явной корреляции между отдельной работой и объемами ресурсов для ее выполнения, что может создать препятствие при изменении графика работ.

Диаграмма Ганта для данного проекта представлена в таблице 4.1.

Таблица 4.1 – Диаграмма Ганта

Номер	Работы	Начало	Конец	Временные периоды											Исполнители
				15.03	25.03	04.04	14.04	24.04	04.05	14.05	24.05	03.06	13.06	23.06	
1	Изучение материала	05.03	15.03	=====											Ивашкин В. В.
2	Изучение библиотеки алгоритмов	16.03	01.04		=======										Ивашкин В. В.
3	Внедрение библиотеки в Clang/LLVM	02.04	25.04			==========	==========								Ивашкин В. В.
4	Разработка тестов производительности	26.04	15.05						==========						Ивашкин В. В.
5	Тестирование алгоритмов	16.05	24.05								=====				Ивашкин В. В.
6	Оформление пояснительной записки	25.05	16.06									==========	==========		Ивашкин В. В. Пазников А. А. Косухина М. А.
7	Подготовка диплома к сдаче	16.06	18.06											====	Ивашкин В. В.

4.2 Расчет затрат на выполнение проекта, расчет цены проекта

Для расчета затрат на выполнение ВКР принимаются условия:

- коэффициент загрузки исполнителя ВКР (студента) равен 1 ($К_{загр.испол.} = 1$).
- коэффициент загрузки соисполнителя (руководителя) равен 0.05 ($К_{загр.рук.} = 0.05$).
- коэффициент загрузки консультанта по дополнительному разделу равен 0.04 ($К_{загр.конс.} = 0.04$).

Таким образом, цена проекта может быть рассчитана по формуле 4.1

$$C_{\text{пр}} = \sum_{i=1}^{n} C_{\text{полн } i}\, T_i\, K_{\text{загр } i} \qquad (4.1),$$

где:

- $C_{\text{полн } i}$ - полная дневная стоимость работы i – го специалиста [руб./день].
- Ti – время участия i – го специалиста в работе над проектом [дней].
- $K\ загр\ i$ – коэффициент загрузки i – го специалиста работами в проекте.
- n – число специалистов, занятых в проекте.

При выполнении расчетов используются следующие обозначения и расчетные соотношения:

Zзп – средняя зарплата работника (специалиста) [руб./месяц]

Tср – среднемесячное число рабочих дней (в период выполнения проекта)

Zд – тарифная дневная ставка работника [руб./день]

$$Z_д = Z_{зп}/T_{ср}$$

Ф – процент (доля) страховых взносов, исчисляемых от фонда заработной платы

$С_{сд}$ – величина страховых взносов на работника в день [руб./день]

$$С_{сд} = Z_д \times Ф$$

$Z_{дс}$ – дневная оплата работника с учетом страховых взносов [руб./день]

$$Z_{дс} = Z_д + С_{сд} = Z_д \times (1 + Ф)$$

Н – процент (доля) накладных расходов

$С_{нр}$ – накладные расходы на одного работника в день [руб./день]

$$С_{нр} = Z_д \times Н$$

$С_{ч/д}$ – стоимость человеко/дня [руб./день]

$$С_{ч/д} = Z_{дс} + С_{нр} = Z_д \times (1 + Ф + Н)$$

П – доля (процент) прибыли (средняя прибыль)

$С_{прд}$ – дневная прибыль на одного работника [руб./день]

$$С_{прд} = С_{ч/д} \times П$$

$С_{дсс}$ – дневная ставка специалиста (без учета НДС) [руб./день]

$$С_{дсс} = С_{ч/д} + С_{прд} = С_{ч/д} \times (1 + П)$$

НДС – ставка налога на добавленную стоимость

$С_{ндс}$ – величина налога на добавленную стоимость в расчете на день [руб./день]

$$С_{ндс} = С_{дсс} \times НДС$$

$С_{полн}$ – полная дневная стоимость работы специалиста [руб./день]

$$С_{полн} = С_{дсс} + С_{ндс} = С_{дсс} \times (1 + НДС)$$

Обобщая, получаем:

$$С_{полн} = (Z_{зп}/Т_{ср}) \times (1 + Ф + Н) \times (1 + П) \times (1 + НДС) \text{ [руб./день]}$$

Сведенные данные отображены в таблицах 4.2 и 4.3

Таблица 4.2 – Полные затраты в день для студента

Наименование статей	ед. изм.	нормативы/затраты	Примечание
Величина среднемесячной начисленной заработной платы специалиста (Нс1) [7]	руб./месяц	40 000,00	Оклад сотрудника Zзп
Среднемесячное количество рабочих дней (Тср) [8]	дней/месяц	20,58	Среднее за 2023 год
Расчет ставки специалиста в день:			
Тарифная ставка дневная (Нс)	руб./день	1 943,63	Zд = Zзп/Тср
Страховые взносы 30,2% от суммы зарплаты работников	руб./день	586,97	Ссд = Zд х Ф
Оплата основных работников с со страховыми взносами (Zдс),	руб./день	2 530,60	Zдс = Zд + Сд = Zд х (1 + Ф)
Накладные расходы (Снр)	руб./день	816,32	Снр = Zд х Н
Себестоимость одного человек/дня (Сч/д),	руб./день	3 346,92	Сч/д = Zдс + Снр =Zд х (1 +Ф + Н)
Дневная прибыль (Спрд),	руб./день	502,03	Спрд = Сч/д х П
Ставка специалиста без учета НДС (Сдсс),	руб./день	3 848,95	Сдсс = Сч/д + Спрд
Дневная сумма НДС (Сндс),	руб./день	769,79	Сндс = Сдсс х НДС
Ставка специалиста в день с учётом НДС (Сполн),	руб./день	4 618,74	Сполн

Таблица 4.4 – Полные затраты в день для руководителя и консультанта по дополнительному разделу

Наименование статей	ед. изм.	нормативы/затраты	Примечание
Величина среднемесячной начисленной заработной платы специалиста (Нс1)	руб./месяц	160 000,00	Оклад сотрудника Zзп
Среднемесячное количество рабочих дней (Тср)	дней/месяц	20,58	Среднее за 2023 год
Расчет ставки специалиста в день:			
Тарифная ставка дневная (Нс)	руб./день	7 774,53	Zд = Zзп/Тср
Страховые взносы 30,2% от суммы зарплаты работников	руб./день	2 347,91	Ссд = Zд х Ф
Оплата основных работников с со страховыми взносами (Zдс),	руб./день	10 122,43	Zдс = Zд + Сд = Zд х (1 + Ф)
Накладные расходы (Снр)	руб./день	3 265,30	Снр = Zд х Н
Себестоимость одного человек/дня (Сч/д),	руб./день	13 387,73	Сч/д = Zдс + Снр =Zд х (1 +Ф + Н)
Дневная прибыль (Спрд),	руб./день	2 008,15	Спрд = Сч/д х П
Ставка специалиста без учета НДС (Сдсс),	руб./день	15 395,88	Сдсс = Сч/д + Спрд
Дневная сумма НДС (Сндс),	руб./день	3 079,17	Сндс = Сдсс х НДС
Ставка специалиста в день с учётом НДС (Сполн),	руб./день	18 475,06	Сполн

Исходя из полученных данных рассчитываем полную цену проекта по формуле 4.1.

$С_{пр}$ = (4 618,74 руб.· 30 · 1) + (18 475,06 руб.· 16 · 0,07) + (18 475,06 руб.· 16 · 0,05) = 307054,75 руб.

Данный проект не рассчитан на вывод на рынок, поэтому делать расчет цены не имеет смысла

4.3 Определение качества программного продукта

При разработке программного продукта должны быть предусмотрены мероприятия по обеспечению его качества, в данном случае таким мероприя-

тием является отладка программы. Отладка подразумевает нахождение ошибок в логике программы путем ее запуска и сверки результатов.

Работа по отладке является цикличной, поэтому рассмотрим цикл Демминга, изображенный на рисунке 4.1.

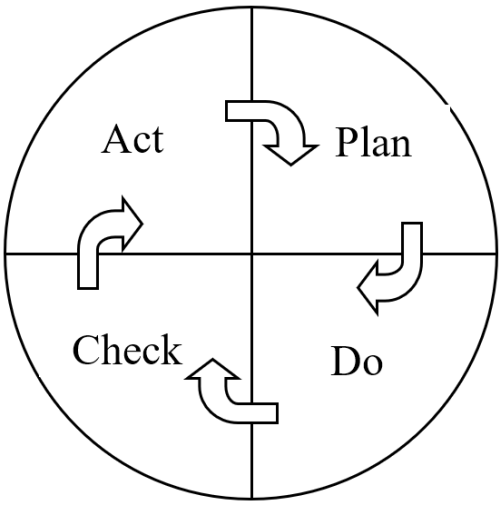

Рисунок 4.1 – цикл Демминга

В рамках цикла Демминга предусмотрен следующий порядок действий:

P – (Plan) - установить процессы и их последовательность, необходимые для получения результатов в соответствии с требованиями потребителя, определить ресурсы, необходимые для реализации процессов;

D – (Do) – внедрить процессы, запустить их в действие;

C – (Check) – контроль (мониторинг) продукции и процессов с целью оценки состояния и выявления отклонений;

A – (Act) – воздействовать с целью достижения результата (ликвидация отклонений) и совершенствования процессов.

4.4 Определение кода программного продукта

В данной работе был написан код для прикладных задач, т.е. по классификации ОКПД2 ВКР подпадает под 62.01.11 [6]— услуги по проектированию, разработке информационных технологий для прикладных задач и тестированию программного обеспечения.

По классификации ОКП ВКР подпадает под 5012309 [6] — Программные средства организации и обслуживания вычислительного процесса.

4.5 Определение списка стандартов

При написании данной работы не планировалось выпускать продукт на рынок, поэтому код программы не соответствует каким-либо стандартам, но при оформлении ВКР стоит обратить внимание на следующие ГОСТы: ГОСТ 19.701-90, ГОСТ 19.102-77, ГОСТ 19.201-78, ГОСТ 19.301-79, ГОСТ 19.401-78 [6].

ЗАКЛЮЧЕНИЕ

В работе были рассмотрены несколько алгоритмов реализации программной транзакционной памяти. Было проведено сравнение различных политик обновления объектов в памяти и стратегий обнаружения конфликтов. Для каждой политики и стратегии были рассмотрены достоинства и недостатки.

В результате выполнения выпускной квалификационной работы была разработана программа для анализа алгоритмов реализации STM.

Для каждого рассмотренного алгоритма были проведены тесты и замеры эффективности работы, которые показали, что программная транзакционная память, основанная на глобальной переменной, отслеживающей блокировки может быть лучше памяти, основанной на записях владения объектами.

В то же время надо учитывать, как часто могут случаться конфликты, чтобы правильно выбрать политику обновления объектов в памяти. Так, если конфликтов много, ленивая политика может быть выгоднее ранней.

Результаты работы в виде обзора ключевых моментов связанных с транзакционной памятью можно использовать для дальнейшего изучения транзакционной памяти и популяризации этой технологии среди разработчиков.

Включение дополнительного раздела, посвященного разработке и стандартизации программных средств, оказало положительное влияние на проект. Он позволил более точно оценить затраты, связанные с будущими исследованиями, а также сделал процесс разработки более простым и структурированным.

Дальнейшие исследования по данной теме могут затронуть алгоритмы с другими характеристиками, в том числе и те, которые используются в аппаратной транзакционной памяти.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Плагин с алгоритмами [Электронный ресурс] – URL: https://github.com/mfs409/llvm-transmem (дата обращения: 06.05.2023)

2. "Simplifying Transactional Memory Support in C++" / PanteA Zardoshti, Tingzhe Zhou, Pavithra Balaji [и др.]. ACM Transactions on Architecture and Code Optimization (TACO), 2019.

3. Maurice Herliny, Nir Shavit, Victor Luchangco The Art of Multiprocessor programming = Искусство мультипроцессорного программирования. М.: Newnes, 2020. 417 с.

4. В. А. Смирнов, А. Р. Омельниченко, А. А. Пазников. Алгоритмы реализации потокобезопасных ассоциативных массивов на основе транзакционной памяти // Информатика и компьютерные технологии. 2018, вып. (№) 1. С. 12–17.

5. Кулагин И.И. Средства архитектурно-ориентированной оптимизации выполнения параллельных программ для вычислительных систем с многоуровневым параллелизмом: дисс. канд.тех.наук / СибГУТИ, Новосибирск, 2017.

6. Фомин В. И. Разработка и стандартизация программных средств. Учебное пособие. СПб: Издательство СПбУУЭ, 2020. 35 с.

7. Зарплаты в ИТ. [Электронный ресурс] // Хабр-Карьера: [сайт]. [2023]. URL: https://career.habr.com/salaries (дата обращения: 14.06.2023)

8. Количество дней (календарных/рабочих/выходных и праздничных) и нормы рабочего времени в 2023 году. [Электронный ресурс] // КонсультантПлюс: [сайт]. [2023]. URL: https://www.consultant.ru/document/cons_doc_LAW_419959/c69042fcbbcf86817f8871212f4df28bf268c0b7/ (дата обращения: 14.06.2023)

9. Форум с описанием проблемы гиперпоточности в virtualbox // ForumVirtualBox [Электронный ресурс]. URL:

https://forums.virtualbox.org/viewtopic.php?f=6&t=101659 (дата обращения 14.05.2023)

10. Транзакционная память: история и развитие // Хабр [Электронный ресурс]. URL: https://habr.com/ru/articles/221667/ (дата обращения 10.05.2023)

11. Виртуальные функции и их переопределение [Электронный ресурс] – URL: https://metanit.com/cpp/tutorial/5.11.php (дата обращения 02.05.2023г.)

12. Pthreads: Потоки в русле POSIX // Хабр [Электронный ресурс]. URL: https://habr.com/ru/articles/326138/ (дата обращения 12.05.2023)

ПРИЛОЖЕНИЕ А

Программный код класса NumbersTest

```
  class NumbersTest: public AbstractTest {
  public:
      /*
      * Generates random numbers and marks data for threads
      * inputSize - numbers quantity
      * threadsCount - threads quantity
      */
      virtual void generate(size_t inputSize, size_t threadsCount)
      {
          AbstractTest::generate(inputSize, threadsCount);
          std::mt19937 rnd(m_rnd());
          m_input.resize(m_inputSize);
          for(size_t i = 0; i < m_inputSize; i++) {
              m_input[i] = rnd() % inputSize;
          }

          //marks out data for threads
          const size_t keysPerThread = m_inputSize / m_threadsCount;
          m_ranges.resize(m_threadsCount);
          size_t currentKey = 0;
          for(size_t threadId = 0; threadId < m_threadsCount; threadId++) {
              m_ranges[threadId].first = currentKey;
              currentKey += keysPerThread;
              m_ranges[threadId].second = currentKey;
          }

          m_ranges[m_threadsCount-1].second = m_inputSize;
      }
      /*
      * Create threads for executing "worker" function
      */
      virtual void run() {
          std::vector<std::thread> threads;
          threads.resize(m_threadsCount);

          for(size_t threadId = 0; threadId < m_threadsCount; threadId++) {
              threads[threadId] = std::thread(std::bind(
&NumbersTest::worker,
                      this, m_ranges[threadId].first,
m_ranges[threadId].second));
          }

          for(size_t threadId = 0; threadId < m_threadsCount; threadId++) {
              threads[threadId].join();
          }
      }
  protected:
      /*
      * Function to override in tests
      */
      virtual void worker(size_t start, size_t end) = 0;

      std::vector<int> m_input;
      std::vector< std::pair<size_t, size_t> > m_ranges;
  };
```

ПРИЛОЖЕНИЕ Б

Программный код тестировщика

```
#include <iostream>
#include <fstream>
#include <iomanip>
#include <vector>
#include <set>
#include <map>
#include <chrono>


#include "test/ArraySumTest.h"
#include "test/ArrayInsertTest.h"
#include "test/HashInsertTest.h"
#include "test/ListInsertTest.h"
#include "test/TreeInsertTest.h"
#include "test/TreeRemoveTest.h"


if defined(TML_EAGER)
#define ALGORITHM "tml_eager"
#elif defined(OREC_EAGER)
#define ALGORITHM "orec_eager"
#elif defined(TML_LAZY)
#define ALGORITHM "tml_lazy"
#elif defined(OREC_MIXED)
#define ALGORITHM "orec_mixed"
#elif defined(OREC_LAZY)
#define ALGORITHM "orec_lazy"
#else
#define ALGORITHM "NONE"
#endif


using namespace std;

typedef map<string, function<ITest *(void)>> TestsMap;

static const TestsMap TESTS = {
    {"ArraySumTest", [] { return new ArraySumTest(); }},
    {"ArrayInsertTest", [] { return new ArrayInsertTest(); }},
    { "ListInsertTest", [] { return new ListInsertTest(); } },
    { "TreeInsertTest", [] { return new TreeInsertTest(); } },
    { "TreeRemoveTest", [] { return new TreeRemoveTest(); } },
    { "HashInsertTest", [] { return new HashInsertTest(); } },
};

typedef std::chrono::high_resolution_clock Clock;

int main()
{
    cout << ALGORITHM+ "starts testing" << endl;

    size_t testsCount = 0;

    ifstream params("Data/params.txt"); //get testing params
    ofstream  results(string("Data/")+ALGORITHM+"Result.txt");  //output
file
```

```
        if (params.is_open() && results.is_open())
        {

            while (true)
            {
                string testName;
                size_t threadsCount;
                size_t inputSize;
                size_t repeatCount;

                const int sym = params.get();
                if (sym == -1) {
                    // end of file
                    break;
                } else if (!std::isalpha(sym)) {
                    // skip comment and white space
                    params.unget();
                    string line;
                    getline(params, line);
                    continue;
                } else {
                    params.unget();
                    params >> testName >> threadsCount >> inputSize >> re-
peatCount;
                    if (!params.good()) {
                        continue;
                    }
                }

                TestsMap::const_iterator it = TESTS.find(testName);
                if (it == TESTS.end())
                {
                    cerr << "Test not found: " << testName << endl;
                    continue;
                }

                results << "Test: " << testName << endl;
                results << "Threads count: " << threadsCount << endl;
                results << "Input size: " << inputSize << endl;
                results << "Repeat count: " << repeatCount << endl;

                ITest *test = it->second();
                double msThreadedAv = 0.0;

                bool isOk = true;

                for (size_t i = 0; i < repeatCount; i++)
                {
                    // generate data
                    test->generate(inputSize, threadsCount);
                    test->setup();

                    // Testing
                    results << setw(20) << left << "\tRun...";
                    Clock::time_point t0 = Clock::now();
                    test->run();
                    Clock::time_point t1 = Clock::now();

                    //Check if result is right
                    if (test->check))
                    {
                        const  double  ms  =  std::chrono::nanoseconds(t1  -
t0).count() * 1e-6;
```

```
                        const      size_t      opsPerSec      =      stat-
ic_cast<size_t>(ceil((double)inputSize / ms));

                        msThreadedAv += ms;
                        results << "OK " << fixed << setprecision(3) << ms
<< " ms, " << opsPerSec << " ops/s" << endl;
                    }
                    else
                    {
                        isOk = false;
                        results << "FAIL " << endl;
                    }

                    test->teardown();

                    results << flush;
                }

                if (isOk)
                {
                    msThreadedAv /= repeatCount;

                    size_t           opsPerSecAv           =           stat-
ic_cast<size_t>(ceil((double)inputSize / msThreadedAv));

                    results << endl
                         << "> " << testName << " " << ALGORITHM << " OK "
                         << inputSize << " " << threadsCount << " " << re-
peatCount << " " << msThreadedAv << " " << opsPerSecAv << endl;

                        cout << endl
                         << "> " << testName << " " << ALGORITHM << " OK "
                         << inputSize << " " << threadsCount << " " << re-
peatCount << " " << msThreadedAv << " " << opsPerSecAv << endl;
                }
                else
                {
                    cout << endl
                         << "> " << testName << " fail" << endl;
                    results << endl
                         << "> " << testName << " fail" << endl;
                }

                delete test;

                results << endl
                     << endl;
                testsCount++;
            }
        }
        params.close();
        results.close();

        if (testsCount > 0)
        {
            return 0;
        }
        else
        {
            return 1;
        }
    }
```