СОДЕРЖАНИЕ

ВВЕДЕНИЕ 6
1. Обзор предметной области 8
1.1. Используемые термины 8
1.1.1. Состояние гонки 8
1.1.2. Блокировка 8
1.1.3. Блокировка чтения-записи 8
1.1.4. Голодание писателя 8
1.1.5. Спинлок 9
1.1.6. Секлок 9
1.1.7. Бакет 11
1.1.8. Сегментированный массив 11
1.1.9. Кеш линия 12
1.1.10. Ложное разделение 12
1.1.11. Неопределенное поведение 12
1.1.12. Тривиально копируемый класс 12
1.1.13. Класс с тривиальным деструктором 12
1.2. Кандидаты на внедрение 12
1.2.1. Хеш таблица с кукушкиным хешированием 13
1.2.2. Хеш таблица с хешированием Робина Гуда 15
Выводы по главе 1 17
2. Секлок 18
2.1. Корректность работы 18
2.1.1. Ассемблерные вставки 18
2.1.2. Решение корректное с точти зрения стандарта 19
2.2. Оптимальный секлок 23
Выводы по главе 2 25
3. Внедрение секлока 26
3.1. Общие сущности 26
3.1.1. Секлок 26
3.1.2. Бакет 26
3.1.3. Сегментированный массив 26
3.1.4. Хранилище бакетов 26
3.1.5. Вспомогательные сущности 26

3.2. Внедрение секлока в хеш таблицу с кукушкиным хешированием. 26
3.2.1. Хранилище бакетов. 26
3.2.2. Хранилище блокировок 27
3.2.3. Рехеширование 27
3.3. Внедрение секлока в хеш таблицу с хешированием Робина Гуда . 27
Выводы по главе 3 27
4. Тесты, бенчмарки и их результаты 28
4.1. Тесты. 28
4.1.1. Unit тесты 28
4.1.2. Stress тесты. 28
4.2. Бенчмарки. 28
4.2.1. Find exists 29
4.2.2. Find modify 30
4.2.3. Random. 32
4.2.4. Inserts. 34
4.2.5. Insert or assigns 36
4.2.6. Erase exists. 38
ЗАКЛЮЧЕНИЕ 40
СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ 41

ВВЕДЕНИЕ

С ростом объема данных возникает необходимость в их быстрой обработке. В прошлом этого добивались увеличением производительности одного ядра, поскольку она росла экспоненциально, однако к началу 2000-х годов рост замедлился и пришлось искать другие методы увеличения производительности. Основным методом повышения производительности стало горизонтальное масштабирование, которое достигалось за счет увеличения количества ядер и даже процессоров. Стала остро стоять проблема оптимальной утилизации всех вычислительных мощностей системы для повышения производительности всей системы.

Один из самых базовых алгоритмов– хеш таблица, реализующая интерфейс ассоциативного массива. Она используется практически везде и от ее производительности зачастую очень сильно зависит производительность всей программы.

Открытая адресация – метод хранения данных, когда сразу выделяется кусок памяти, и в нем уже конструируются объекты, этот метод в отличии от хранения указателей на отдельные объекты позволяет добиться лучшей кеш локальности, и избежать частого выделения памяти. Все это положительным образом сказывается на производительности.

Оптимизация хеш таблиц для многопоточной нагрузки является острой проблемой. Существует множество алгоритмов конкурентных хеш таблиц, показывающих отличную масштабируемость, и если в случае пишущей нагрузки, сложно выжать из них больше, в силу семантических ограничений (возможность записи только в эксклюзивном режиме), для операции чтения такого ограничения нет, она может выполнятся параллельно с другими чтениями.

Для решения подобной проблемы подходят блокировки чтения-записи, позволяющие отлично масштабировать параллельные чтения, одним из таких алгоритмов является секлок – легковесная блокировка, имеющая такое важное свойство, как отсутствие голодания писателя. Использование секлока накладывает на устройство хеш таблицы некоторые ограничения, например невозможность деаллокации памяти к которой мог получить доступ читающий поток, это заставляет поменять само устройство хранения данных в хеш таблице и обращения к ним.

Внедрения секлока в лучшие конкурентные хеш таблицы, такие как хеш

таблица с кукушкиным хешированием[1, 9], реализованная в `libcuckoo`[14], и написание реализаций с нуля таких хеш таблиц, как хеш таблица с хешированием Робина Гуда[8], в силу отсутствия открытых реализаций и являются целью этой работы.

ГЛАВА 1. ОБЗОР ПРЕДМЕТНОЙ ОБЛАСТИ

В этой главе будут обсуждаться вводные для понимания работы данные: термины, базовое объяснения работы секлока, обзор исходных алгоритмов хеш таблиц.

1.1. Используемые термины

Здесь будут введены термины, которые будут активно использоваться по ходу всей работы.

1.1.1. Состояние гонки

Состояние гонки – состояние, при которой работа системы зависит от того, в каком порядке выполняются части кода. Типичными примерами состояния гонки являются параллельные записи неатомарных данных, в такой ситуации может получиться неконсистентное состояние записываемых данных, например первая часть структуры будет от первой записи, а вторая от второй. К такой же ошибке приводит параллельное чтение и запись неатомарных данных.

1.1.2. Блокировка

Блокировка – механизм синхронизации, позволяющий обеспечить исключительный доступ к разделяемому ресурсу между несколькими потоками, тоесть в критической секции кода, которую защищает блокировка может находится одновременно только один поток.

1.1.3. Блокировка чтения-записи

Вид блокировки, позволяющий параллельные между собой чтения, но так же гарантирующий что записи всегда будут выполнятся эксклюзивно, как по отношению к другим записям, так и к чтениям. Является широко используемой концепцией, дающей ощутимый прирост производительности в случае нагрузки с доминирующим количеством чтений.

1.1.4. Голодание писателя

Проблема, присущая некоторым блокировкам чтения-записи. Она выражается в том, что поток не может взять блокировку в силу постоянно приходящих операций чтения, которые не отпускают блокировку.

1.1.5. Спинлок

Алгоритм блокировки, использующий активное ожидание, им обычно защищают легковесные части кода. Суть работы спинлока заключается в том, что он хранит один атомарный флаг, если он выставлен в `true`, то блокировка взята, если в `false`, то нет.

Листинг 1 – Пример реализации спинлока

```
struct spinlock {
    void lock() {
        while (flag.test_and_set(std::memory_order_acq_rel));
    }
    void unlock() {
        flag.clear(std::memory_order_release);
    }
private:
    std::atomic_flag flag;
};
```

1.1.6. Секлок

Алгоритм блокировки, похожий на спинлок, он так же использует активное ожидание. Отличие состоит в том, что секлок уже хранит не флаг а эпоху, которая инкрементируется при взятии и отпускании блокировки. Указанная ниже реализация не является оптимальной, но наиболее наглядно демонстрирует смысл работы секлока.

Листинг 2 – Пример реализации секлока

```
struct seqlock {
    void lock() {
        while (true) {
            uint64_t prev = get_value();
            if (!is_locked(prev) && epoch.compare_exchange_strong(
               prev, prev + 1, std::memory_order_acq_rel)) {
                return
            }
        }
    }

    void unlock() {
        epoch.fetch_add(1, std::memory_order_acq_rel);
```

```
    }

    uint64_t get_value() const {
        epoch.load(std::memory_order_acquire);
    }

    static constexpr is_locked(uint64_t v) {
        return (v & 1) != 0;
    }
private;
    std::atomic<uint64_t> epoch;
};
```

Пишущие операции так же, как и в случае спинлока честно берут блокировку при работе с секлоком, читающие же операции, вместо этого избегают дорогую операцию записи в атомарную переменную `epoch`, это позволяет избежать вытеснения `epoch` из кеша других ядер и параллельно совершать множество чтения без потери производительности. Ниже представлен алгоритм чтения данных защищаемых секлоком без углубления в детали корректности порядка чтений и возможности их оптимизаций компилятором:

Листинг 3 – Чтение данных, защищаемых секлоком

```
Value read_value_under_seqlock(size_t index) {
    while (true) {
        size_t lock_index = get_lock_index(index);

        uint64_t lock_value_before = locks[lock_index].get_value()
           ;

        if (seqlock::is_locked(lock_value_before)) {
            continue;
        }

        Value result = read_value(index);

        uint64_t lock_value_after = locks[lock_index].get_value();

        if (lock_value_before == lock_value_after) {
            return result;
        }
    }
```

```
}
```

Если взглянуть на этот код становится понятно откуда берется ограничение на деаллокацию данных в хеш таблице с секлоком: читающая операция может прочитать индекс по которому лежат данные, и если в этот момент совершить перенос данных и деаллоцировать прошлое хранилище, мы попытаемся прочитать память, которая уже освобождена. Эта проблема вызвана тем, что мы читаем данные не под блокировкой. Эта проблема похожа на невозможность деаллокации блокировок в других хеш таблицах. Так же алгоритм накладывает еще одно важное ограничение – данные хранящиеся в хеш таблице должны быть тривиально копируемыми, чтобы было возможно читать неконсистентные данные, во время записи, просто побайтово.

1.1.7. Бакет

Хранилище нескольких пар ключ-значение с одинаковым индексом, порядковый номер пары в бакете обычно называют слотом. Бакет может так же хранить дополнительную информацию, например какие слоты бакета заняты, произведена ли миграция бакета.

1.1.8. Сегментированный массив

Для решения проблемы невозможности деаллокации памяти, нам необходим примитив для хранения блокировок и бакетов, который позволяет увеличивать количество хранимых сущностей без реаллокации памяти. Возможной реализацией решения такой проблемы является так называемый сегментированный массив[15] – массив размером 64, хранящий указатели на куски памяти, где размер i-того куска равен сумме размеров предыдущих(нулевой равен единице). Чтобы найти пару индексов, по которым можно достать j-тый элемент, нужно выполнить элементарные битовые операции:

Листинг 4 – Определения индексов необходимого элемента в сегментированном массиве

```
std::pair<size_t, size_t> find_value(size_t j) {
    static constexpr uint16_t type_size_ind = sizeof(size_t) << 3;

    if (j == 0) {
        return {0, 0};
    }
```

```
    const size_t left_bit_ind = type_size_ind - std::countl_zero(j
        );
    return {left_bit_ind, j ^ (size_t(1) << (left_bit_ind - 1))};
}
```

Из минусов данного подхода можно выделить дополнительный поход в память, но он практически нивелируется кешем трансляции.

1.1.9. Кеш линия

Блок памяти фиксированного размера, данные в кеш загружаются и хранятся именно такими блоками.

1.1.10. Ложное разделение

Ситуация, когда две атомарные переменные лежат рядом друг с другом и попадают в одну кеш линию. Это приводит к тому, что при изменении, например первой переменной потоком А, а второй потоком Б, каждый раз при записи любого из потоков, вся кеш линия будет сбрасываться и в другом потоке, такая ситуация ведет к сильной деградации производительности.

1.1.11. Неопределенное поведение

Для языка C++[6] – ситуация, когда стандарт языка никак не гарантирует поведение кода, а зависит оно от текущей платформы и компилятора, возникает в различных маргинальных ситуациях, например допущение состояния гонки, чтение по нулевому указателю, и так далее.

1.1.12. Тривиально копируемый класс

В языке C++ – такой класс, который можно копировать побайтово, тоесть не вызывая конструктор.

1.1.13. Класс с тривиальным деструктором.

В языке C++ – такой класс, у которого можно не вызывать деструктор, его память можно переиспользовать сразу.

1.2. Кандидаты на внедрение

Для внедрения и реализации были выбраны одни из лучших алгоритмов конкурентных хеш таблиц.

1.2.1. Хеш таблица с кукушкиным хешированием

Данная хеш таблица является одной из лучших по производительности конкурентной хеш таблицей. И имеет отличную открытую реализацию – `libcuckoo`, это позволяет не писать хеш таблицу с нуля, а сосредоточиться на внедрении самого секлока. Перечислим ключевые моменты внутреннего устройства хеш таблицы:

1.2.1.1. Бакет

Бакет в этой хеш таблице представляет собой 3 массива

— values – память для пар ключ-значение.
— partials – частичные хеши.
— occupied – индикаторы, занят ли i-тый слот в бакете.

1.2.1.2. Блокировка

Роль блокировки в этом алгоритме исполняет обычный спинлок, выровненный по 64 байта (размер кеш линии на большинстве платформ) и хранящий два дополнительных поля:

— Мигрированы ли бакеты, которые защищает блокировка.
— Количество элементов в бакетах, которые защищает блокировка.

1.2.1.3. Хранилище блокировок

Как и в любой конкурентной хеш таблице с блокировками, здесь тоже существует проблема невозможности деаллокации блокировок. Эта проблема в этом алгоритме решена таким образом: в хеш таблице хранится лист массивов блокировок, актуальными блокировками считаются те, что лежат в хвосте листа, остальные хранятся для корректности возможных операций запаздывающих потоков.

1.2.1.4. Хранилище бакетов

Хранилище бакетов здесь представлено классом, оперирующим сырой выделенной памятью с помощью аллокатора переданного в хеш таблицу. При ленивом рехешировании в хеш таблице одномоментно хранятся две версии хранилища бакетов – старая и новая.

1.2.1.5. Решение проблемы коллизий

Для решения проблемы коллизий данный алгоритм, помимо хранения элементов в бакетах, использует технику кукушкиного хеширования, основная идея которого заключается в том, что для каждого ключа есть два индекса: основной и альтернативный.

1.2.1.6. Поиск по ключу

Поиск по ключу является просто взятием двух блокировок и проверкой двух бакетов – по основному и альтернативному индексам.

1.2.1.7. Вставка по ключу

Для вставки данный алгоритм использует следующую технику:

— Если в основном или альтернативном бакете есть свободное место – кладем нашу пару туда.
— Если же мест нет, то мы выталкиваем любую пару из одного из бакетов, и пытаемся положить ее в парный бакет, и так далее, пока либо не найдем свободное место, либо не случится одно из следующего:
  – Попадем в бесконечный цикл.
  – Превысим некоторый лимит перекладываний, зависящий от размера хеш таблицы.

  В обоих случаях мы начинаем операцию рехеширования, тем самым расширяя хеш таблицу.

1.2.1.8. Рехеширование

Рассмотрим важное свойство рехеширование, индекс ключа при рехешировании может либо остаться прежним, либо стать больше на старый размер хеш таблицы, происходит это потому, что увеличением на два мы буквально начинаем рассматривать следующий бит индекса и он может быть равен 0 или 1. Учитывая свойство блокировок, которые при достижении лимита `L` на свое количество начинают защищать каждый `L`-тый бакет после себя, получаем, что при ленивом рехешировании блокировка защищает как раз оба возможных бакета для переноса – и старый и новый.

Поток, который занимается рехешированием всегда берет все блокировки, далее существуют два вида рехеширования:

— Для не броюсающих исключения типов.

  - Доделываем предыдущее рехеширование, если оно было ленивым, и деаллоцируем старую версию хранилища бакетов.
  - Добавляем новый массив блокировок равный по размеру новому количеству бакетов, если мы не достигли лимита `L`.
  - Аллоцируем новое хранилище блокировок, которое вдвое больше предыдущего, далее возможны два варианта:
    * Если старое количество бакетов меньше того же лимита `L`, то просто переносим данные из старых бакетов в новые и деаллоцируем старое хранилище.
    * Если количество бакетов больше, оставляем старые бакеты и помечаем все блокировки, как не мигрированные, теперь любой поток, который возьмет эту блокировку, должен будеть сначала перенести все пары ключ-значение из старых бакетов, которые защищает эта блокировка в новые.

— Для броюсающих исключения типов.

  В данном случае, чтобы гарантировать необходимые гарантии исключений, алгоритм использует `swap-trick`: создает пустую хеш таблицу, вставляет все текущие пары ключ-значение в новую хеш таблицу и производит swap необходимых полей с хеш таблицей, хранилища блокировок, бакетов, и так далее.

1.2.2. Хеш таблица с хешированием Робина Гуда

Как и хеш таблица с кукушкиным хешированием, считается одной из лучших конкурентых хеш таблиц, однако не имеет адекватных открытых реализаций. Её было решено реализовать с нуля, с гораздо более скромным интерфейсом, чем например `libcuckoo`, но тем не менее достаточным для проведения тестов на корректность и запуска бенчмарков.

1.2.2.1. Блокировка

Роль блокировки для алгоритма без секлока будет так же выполнять обычный спинлок.

1.2.2.2. Хранилище блокировок и данных

Для этой хеш таблицы было решено применить технику хранения блокировок и данных вместе. Тоесть подряд будут лежать блокировка и некоторое количество пар ключ-значение, которые она защищает. Высокоуровневым

хранилищем этих последовательностей так же будет выступать сегментированный массив, в силу своего удобства.

1.2.2.3. Решение проблемы коллизий

Принцип решения проблемы коллизий в этом алгоритме похож на принцин самого Робина Гуда, в честь которого он и назван: ”Крадет у богатых и отдает бедным”. Для каждого ключа есть его оригинальный индекс в массиве, далее будем обозначать его как `orig_index`. В алгоритме элемент может располагаться на позициях от `orig_index` до `orig_index` + `max_diff`, где `max_diff`, есть некоторое число, логарифмически зависящее от размера хеш таблицы, более подробно об этом подходе будет рассказано при обозревании вставки по ключу.

1.2.2.4. Поиск по ключу

Для поиска по ключу просто проходим от `orig_index` до `orig_index` + `max_diff` и ищем нужный ключ.

1.2.2.5. Вставка по ключу

Обозначим как `init_pos` оригинальный индекс ключа. Вставка по ключу представляет собой рекурсивный алгоритм, рассмотрим один его шаг, на вход этому шагу подается ключ для вставки, а так же текущая позици, обозначим ее как `pos`, алгоритм заканчивается в двух случаях:

— Если позиция `pos` не занята, тогда мы кладем нашу пару ключ-значение сюда.

— Если `pos` − `init_pos` > `max_diff`, необходимо расширить хеш таблицу и мы начинаем алгоритм рехеширования.

Теперь подробнее рассмотрим сам шаг рекурсии, на текущей позиции `pos` лежит некий ключ, если его дальность от оригинального индекса меньше, чем у вставляемого ключа, то он лежит к своей оригинальной позиции ближе чем мы, тоесть в терминах алгоритма он ”Богатый”. В таком случае мы вставляем текущую пару ключ-значение на его место и продолжаем алгоритм вставки уже для этого ключа сдвигаясь в следующую ячейку.

1.2.2.6. Рехеширование

Был реализован самый простой, неленивый метод рехеширования – расширяем сегментированный массив и проходимся по всем существующим ключам, удаляя и вставляя их на новые позиции.

Выводы по главе 1

В этой главе был произведен обзор предметной области: введены основные термины, которые будут использоваться в дальнейшем описании работы, были базово описаны алгоритмы, в которые будет внедрен секлок.

ГЛАВА 2. СЕКЛОК

В этой главе будет более подробно рассмотрен алгоритм секлока и даны ответы на следующие вопросы:

— Как сделать чтение и запись защищаемых им данных корректным?
— Как добиться максимальной производительности секлока?

2.1. Корректность работы

Семантика работы секлока такова, что у нас могут случаться состояния гонки, когда мы одновременно пишем и читает неатомарные данные, что для языка C++ ведет к неопределенному поведению, мы можем положиться на то, что компилятор на определенной платформе генерируют код, который продолжает работать корректно, а можем написать переносимое решение, которое будет исключать гонку данных и соответствовать стандарту.
Основной проблемой для нас является то, что в данном коде:

Листинг 5 – Итерация чтения данных защищенных секлоком

```
lock.load(std::memory_order_acquire);

Value value = read_value(); // non atomic read

lock.load(std::memory_order_acquire);
```

который является частью чтения значения под секлоком, возникает следующая проблема:
В силу того, что `load with acquire` запрещает операциям переупорядочиваться только до себя, неатомарное чтение может переупорядочиться после второго чтения блокировки.
Для обеспечения корректности можно воспользоваться двумя методами:

2.1.1. Ассемблерные вставки

Этот метод является наиболее оптимальным с точки зрения производительности, но для каждой архитектуры приходится вставлять свой барьер. Также, программа написанная таким образом, с точки зрения стандарта некорректна, так как допускает гонку данных, здесь мы полагаемся на то, что все существующие компиляторы в этом случае все равно генерируют корректный код.
Рассмотрим решения для двух самых популярных архитектур:

2.1.1.1. X86-64

Для `X86-64`[2, 13] не нужны никакие барьеры, так как `load` и `store` операции в этой архитектуре не генерируют барьеры, необходимо только указать компилятору на невозможность переупорядочивания этих операций в процессе оптимизаций компилятора, таким образом код будет выглядет так:

Листинг 6 – Итерация чтения данных защищенных секлоком корректная для X86-64

```
lock.load(std::memory_order_acquire);

Value value = read_value(); // non atomic read

asm volatile ("":::"memory");

lock.load(std::memory_order_acquire);
```

2.1.1.2. ARM64

Для архитектуры `ARM64`[3] необходимо не только запретить компилятору переупорядочивать операции, но еще и вставить барьер, который генериует компилятор при `acquire` операции – `DMB ISHLD`, он ждет выполнения всех читающих операций до себя, таким образом код будет выглядет так:

Листинг 7 – Итерация чтения данных защищенных секлоком корректная для ARM64

```
lock.load(std::memory_order_acquire);

Value value = read_value(); // non atomic read

asm volatile ("DMB ISHLD":::"memory");

lock.load(std::memory_order_acquire);
```

2.1.2. Решение корректное с точти зрения стандарта

Обеспечим корректность программы в условиях модели памяти языка[7]. Описанное здесь решение было вдохновлено статьей лаборатории HP[4]. Начнем с того, что чтобы обеспечить корректность работы алгоритма с точки зрения стандарта, нам необходимо избавиться от гонки данных. Мы можем добиться этого сделав данные, защищаемые секлоком атомарными, так же

вспомним, что у нас есть требование для данных, которые защищаются секлоком – они должны быть тривиально копируемыми, это значит, что мы можем читать их просто побайтово. Введем две вспомогательные функции для чтения такого буффера байтов `atomic_load_memcpy` и `atomic_store_memcpy`.

Листинг 8 – Функции для побайтового чтения и записи данных

```
template <typename T>
constexpr size_t uintmax_count = sizeof(T) / sizeof(uintmax_t);
template <typename T>
constexpr size_t uintmax_bytes = uintmax_count<T> * sizeof(
    uintmax_t);

template<typename T>
std::atomic<T>& as_atomic(T& t) {
  return reinterpret_cast<std::atomic<T>&>(t);
}

template<typename T>
const std::atomic<T>& as_atomic(const T& t) {
  return reinterpret_cast<const std::atomic<T>&>(t);
}

template <typename T>
void atomic_load_memcpy(T* dest, const T* source) {
  for (size_t i = 0; i < uintmax_count<T>; ++i) {
    auto& dst = reinterpret_cast<uintmax_t*>(dest)[i];
    auto& src = reinterpret_cast<const uintmax_t*>(source)[i];

    dst = as_atomic(src).load(std::memory_order_relaxed);
  }
  for (size_t i = uintmax_bytes<T>; i < sizeof(T); ++i) {
    auto& dst = reinterpret_cast<char*>(dest)[i];
    auto& src = reinterpret_cast<const char*>(source)[i];

    dst = as_atomic(src).load(std::memory_order_relaxed);
  }
}

template <typename T>
void atomic_store_memcpy(T* dest, const T* source) {
  for (size_t i = 0; i < uintmax_count<T>; ++i) {
```

```
    auto& dst = reinterpret_cast<uintmax_t*>(dest)[i];
    auto& src = reinterpret_cast<const uintmax_t*>(source)[i];

    as_atomic(dst).store(src, std::memory_order_relaxed);
  }
  for (size_t i = uintmax_bytes<T>; i < sizeof(T); ++i) {
    auto& dst = reinterpret_cast<char*>(dest)[i];
    auto& src = reinterpret_cast<const char*>(source)[i];

    as_atomic(dst).store(src, std::memory_order_relaxed);
  }
}
```

Теперь производя все операции с данными, которые могут параллельно писаться и читаться, с помощью операций `atomic_load_memcpy` и `atomic_store_memcpy`, мы добились того, что у нас нет гонок данных и программа не ведет к неопределенному поведению и корректна с точки зрения стандарта.

Остается решить проблему переупорядочивания чтения данных и второго чтения эпохи, для этого используем `atomic_thread_fence`[5], итоговые функции чтения и записи в таком случае будут выглядеть примерно так:

Листинг 9 – Чтения и запись корректные с точки зрения стандарта

```
1  template <typename T>
2  T read(size_t i) {
3      uint64_t e_before;
4      uint64_t e_after;
5      seqlock& lock = locks[lock_ind(i)];
6      const T& value = get_value_ref(i);
7      T res;
8      do {
9          e_before = lock.load(std::memory_order_acquire);
10
11         atomic_load_memcpy(&res, &value);
12
13         std::atomic_thread_fence(std::memory_order_acquire);
14
15         e_after = lock.load(std::memory_order_relaxed);
16     } while (e_before == e_after);
17 }
18
```

```
19 template <typename T>
20 T write(size_t i, const T& value) {
21     seqlock& lock = locks[lock_ind(i)];
22     lock.lock(); // with acq_rel
23
24     std::atomic_thread_fence(std::memory_order_release);
25
26     atomic_store_memcpy(get_value_ref(i), &value);
27
28     lock.unlock(); // with release
29 }
```

Рассмотрим подробнее почему такой подход обеспечивает корректность порядка чтений. Будем пользоваться такими понятиями из стандарта языка C++, как:

- `Sequenced before` – порядок, относящийся к операциям в одном потоке. Операции `A` по отношении к операции `Б`, является Sequenced before, если гарантируется, что операция `A` выполниться раньше операции `Б`.
- `Synchronize with` – если атомарная операция записи в потоке `A` с `memory_order_release`, атомарная операция чтения в потоке `Б` с `memory_order_acquire` в ту же самую переменную, и операция чтения в потоке `Б` читает значение, которое записал поток `A`, тогда запись в потоке `A` находится в отношении `Synchronize with` с чтением в потоке `B`.
- `Happens before`[12] – в первом приближении, это комбинация `Sequenced before` и `Synchronize with`.

Будем оперировать номерами строк из представленных выше функций `read` и `write`. Рассмотрим в каких отношения состоят некоторые из них:

- `Sequenced before`
  - 22 $\rightarrow$ 24 – в силу `release` порядка операции `lock()`, делающей все следующие операции `Sequenced before`.
  - 24 $\rightarrow$ 26 – по правилу для барьеров
  - 11 $\rightarrow$ 13 – по правилу для барьеров.
- `Synchronize with`
  - 24 $\rightarrow$ 13 – исходит из правила для барьеров, по атомарной переменной, которой являются части защищаемых данных.

— Happens before

– 22 → 15 – по правилу для синхронизации двух барьеров, так как между ними есть Synchronize with, а так же принимая во внимание порядки Sequenced before указанные выше.

Таким образом мы добились следующего – если изменены защищаемые данные, то мы не можем не увидеть изменения счётчика эпохи, это именно то, что нам было нужно для корректности алгоритма.

2.2. Оптимальный секлок

Указанная ранее версия секлока является простой для понимания, но не слишком оптимальной, операции fetch_add и compare_exchange_strong являются достаточно тяжелыми по сравнению с load и store, на архитектуре X86-64 они на десятки процентов замедляют работу алгоритма по сравнению с более оптимальной версией.

Развязывает руки в оптимизациях еще и то обстоятельство, что структуры блокировок обычно выравнивают по 64 байта, для избежания ложного разделения кеш линий. Это позволяет нам хранить некоторые вспомогательные поля. Ниже представлен код наиболее оптимальной реализации секлока, которой удалось достичь.

Листинг 10 – Наиболее оптимальная реализация секлока средствами языка, которой удалось достичь

```
class alignas(64) seqlock {
    static constexpr uint64_t epoch_type_size = sizeof(uint64_t)
      << 3;
    static constexpr uint64_t lock_bit = uint64_t(1) << (
      epoch_type_size - 1);

public:
    uint64_t lock() {
        while (lock.test_and_set(std::memory_order_acq_rel));

        const uint64_t res = ++cur_epoch_ | lock_bit;
        epoch.store(res, std::memory_order_release);

        return res;
    }
```

```
    void unlock() {
        ++cur_epoch_;
        std::atomic_thread_fence(std::memory_order_release);
        epoch.store(cur_epoch_, std::memory_order_relaxed);
        lock.clear(std::memory_order_relaxed);
    }

    uint64_t get_epoch() const {
        return epoch.load(std::memory_order_acquire);
    }

    constexpr static bool is_locked(uint64_t value) {
        return (value & lock_bit) != 0;
    }

private:
    uint64_t cur_epoch;

    std::atomic<uint64_t> epoch;
    std::atomic_flag lock;
};
```

Обсудим, почему данная реализация оптимальние предыдущей:

— **Зачем нужны два атомика?**

Рассмотрим ситуацию, если бы переменная `epoch` исполняла бы еще и роль синхронизирующей для взятия блокировки, в таком случае нам пришлось бы использовать довольно дорогие операции `compare_exchange_strong` `compare_exchange_weak`. Учитывая, что с одной переменной для синхронизации, других вариантов нет, выходом является заведение дополнительного `atomic_flag`, операция `test_and_set` которого, является гораздо более дешевой. Теперь `epoch` используется только по своему прямому назначению – эпоха для читающих операций. Так же для более оптимального изменения эпохи, добавлена неатомарная переменная `cur_epoch` хранящая текущую эпоху и позволяющая инкрементировать ее с помощью дешевой операции `store`, вместо дорогой `fetch_add`.

— **Почему два атомика не ухудшают производительность?**

Заметим, что оба атомика всегда меняются вместе из одного потока, это

приводит к тому, что эти изменения не сбрасывают кеш линию и второй атомик изменяется очень быстро.

Возможна так же реализация секлока с помощью одного атомика, но уже не средствами библиотеки <atomic>. Для взятия блокировки можем так же использовать флаг, пусть это будет первый бит нашей новой эпохи, тогда на нем можно так же сделать операцию `test_and_set` с помощью асемблерной инструкции `bts`. Это решение не было реализовано в силу непортируемости и неясного выигрыша по сравнению с представленным решением.

Выводы по главе 2

В этой главе были изложены тонкости реализации секлока. Были подняты вопросы как корректности, так и производительности для разных архитектур.

ГЛАВА 3. ВНЕДРЕНИЕ СЕКЛОКА

В этой главе будут описаны процессы внедрения секлока в различные хеш таблицы. Подробнее будут показаны изменения которые приходилось вносить в архитектуру хеш таблиц.

3.1. Общие сущности

Для решения поставленной задачи для различных хеш таблиц, полезно вынести общие примитивы, которые можно использовать в любой хеш таблице.

3.1.1. Секлок

Код секлока описанный выше является общим для всех хеш таблиц.

3.1.2. Бакет

Бакет, хранящий минимальный набор данных: хранилище пар ключ-значение, индикаторы, есть ли в i-том слоте значение. Реализации бакетов для конкретных алгоритмов наследуются от него.

3.1.3. Сегментированный массив

Был реализован сегментированный массив для абстрактного типа и аллокатора, его можно использовать для хранения как бакетов, так и блокировок.

3.1.4. Хранилище бакетов

Абстрактное хранилище бакетов использующее сегментированный массив указанный выше, работает с абстрактным бакетом, требует переопределения функций копирования и удаление пары ключ-значение.

3.1.5. Вспомогательные сущности

Сюда входят такие функции и классы как: вычисление количества необходимых бакетов по количеству ключей и слотов, копируемая обертка для атомарной переменной, RAII обертка для блокировки и так далее.

3.2. Внедрение секлока в хеш таблицу с кукушкиным хешированием

3.2.1. Хранилище бакетов

Теперь, для удовлетворения ограничений секлока, мы храним бакеты в указанном выше хранилище на базе сегментированного массива. Таким образом мы можем расширяться без деаллокации старых бакетов. Старые бакеты для ленивого рехеширования мы больше не храним, рехеширование будет осуществяться `in-place`.

3.2.2. Хранилище блокировок

Учитывая что наше новое сегментированное хранилище удовлетворяет требованию отсутсвия деаллокаций, оно было использовано и для хранения блокировок.

3.2.3. Рехеширование

Основная модификация логики алгоритма происходит в рехешировании, если раньше перенос был организован из одного хранилища в другое, то теперь нам необходимо делать перенос `in-place`, так же необходимо сохранить важнейшее свойство ленивости рехеширования. Новый алгоритм теперь выглядит так:

- Завершаем прошлое ленивое рехеширование, если такое имело место.
- Расширяем хранилище блокировок, если не достигнут лимит.
- Расширяем хранилище бакетов.
- Далее существует два варианта переноса ключей-значение:
  - Если размер хеш таблицы меньше некоторого лимита, производим рехеширование однопоточно, перенося необходимые ключи-значение в новые бакеты.
  - Если же размер хеш таблицы больше, помечаем все блокировки, как не мигрированные, теперь потоку взявшему блокировку придется сначала выполнить рехеширование для всех бакетов, которые она защищает.

3.3. Внедрение секлока в хеш таблицу с хешированием Робина Гуда

Для описанной в первой главе реализации единственным изменением стала замена спинлока на секлок и соответствующая замена обычного чтения на чтение под секлоком.

Выводы по главе 3

В этой главе были рассмотрены детали внедрения секлока в конкурентные хеш таблицы, рассмотрены общие примитивы, которые можно применять в подобного рода задачах.

ГЛАВА 4. ТЕСТЫ, БЕНЧМАРКИ И ИХ РЕЗУЛЬТАТЫ

4.1. Тесты

Основой для тестов послужили тесты из библиотеки `libcuckoo`, было реализовано два вида тестов:

4.1.1. Unit тесты

Тесты были переписаны с фреймворка `catch` на более удобный – `google test`[11]. Были адаптированы некоторые тесты проверяющие количество аллокаций, оно уменьшилось в силу использования сегментированного массива. Тест проверяющий возможность использования хеш таблицы с значениями без конструктора копирования был удален в силу семантических ограничений новой хеш таблицы, при операции чтения мы должны уметь копировать значение.

4.1.2. Stress тесты

Был исправлен следующий недочет существующих тестов – тесты не проверяли корректность рехеширования, сразу выделяя нужное количество памяти для заданного количества ключей-значений. Это обстоятельство являлось сильным недочетом, посколько операция рехеширования является самой сложной с точки зрения корректности и отсутсвие ее проверки в стресс тестах – большой недочет.

4.2. Бенчмарки

Для проверки скорости работы хеш таблиц было реализовано удобное окружение позволяющее выбирать сценации бенчмарков и задавать их параметры. В качестве фреймфорка был использован `google-benchmark`[10].Бенчмарки поддерживают 4 типа операций:

- `find` – поиск по ключу.
- `insert` – вставка в хеш таблицу, если не существует другого элемента с таким ключом.
- `insert_or_assign` – вставка в хеш таблицу, если элемента с таким ключом не существует, иначе присваивание переданного значения.
- `erase` – удаление по ключу.

Все бенчмарки производились на двух системах:

- `Kunpeng-920` на архитектуре `ARM64`, 4 `NUMA` узла по 24 ядра, 2 сокета.

— `Intel Xeon Gold 6338` на архитектуре `X86-64` 2 `NUMA` узла по 24 ядра + гипертрединг.

Описания бенчмарков и их результаты:

4.2.1. Find exists

Самый оптимистичный сценарий нагрузки для хеш таблиц с секлоками – чтение существующих ключей без записей, представлены две версии бенчмарка, с малым количеством ключей и следующей отсюда большей конкуренцией потоков (представлен сверху), с большим количеством ключей, и как следствие меньшей конкуренцией потоков (представлен снизу):

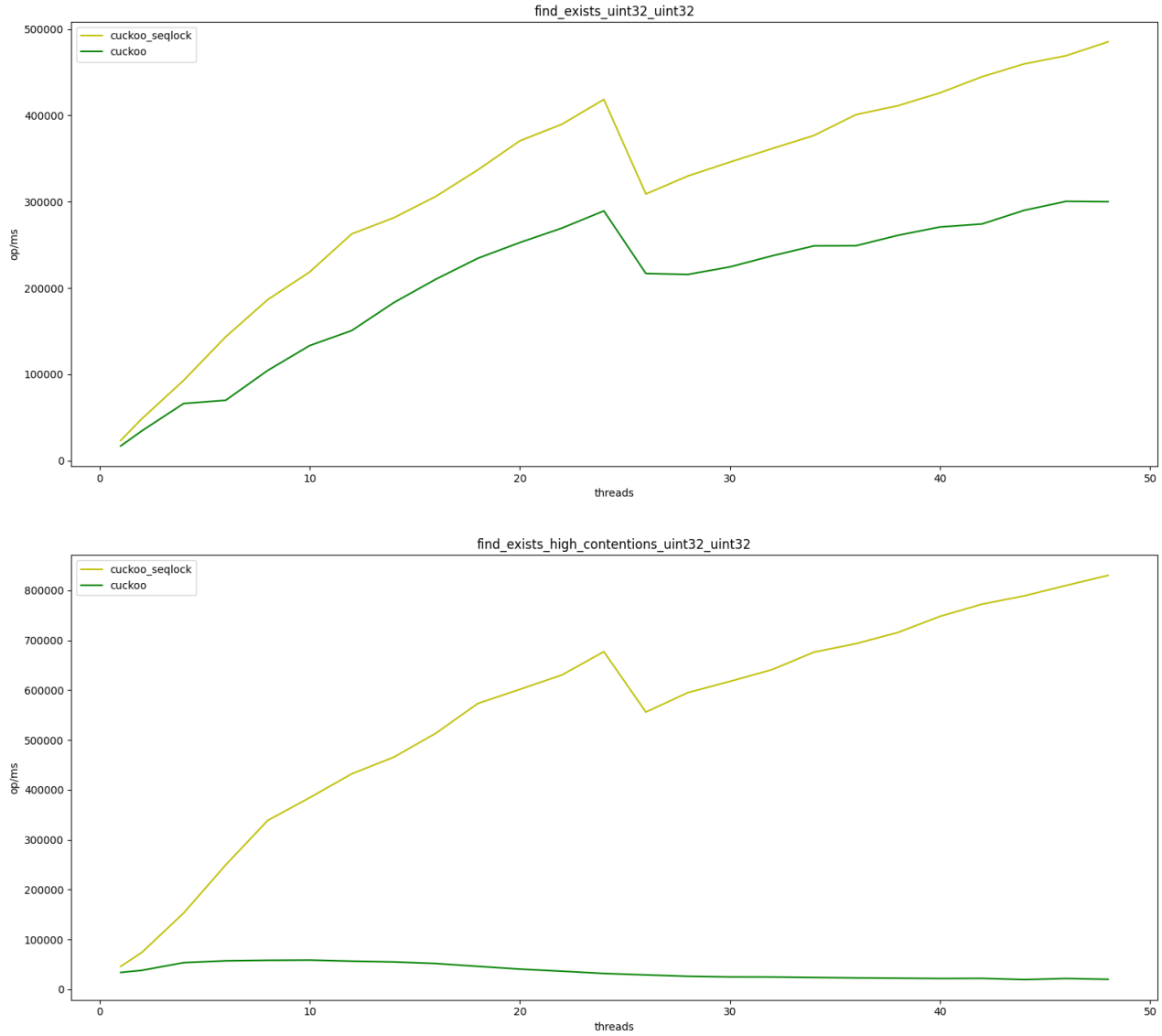

Рисунок 1 – Find exists X86-64

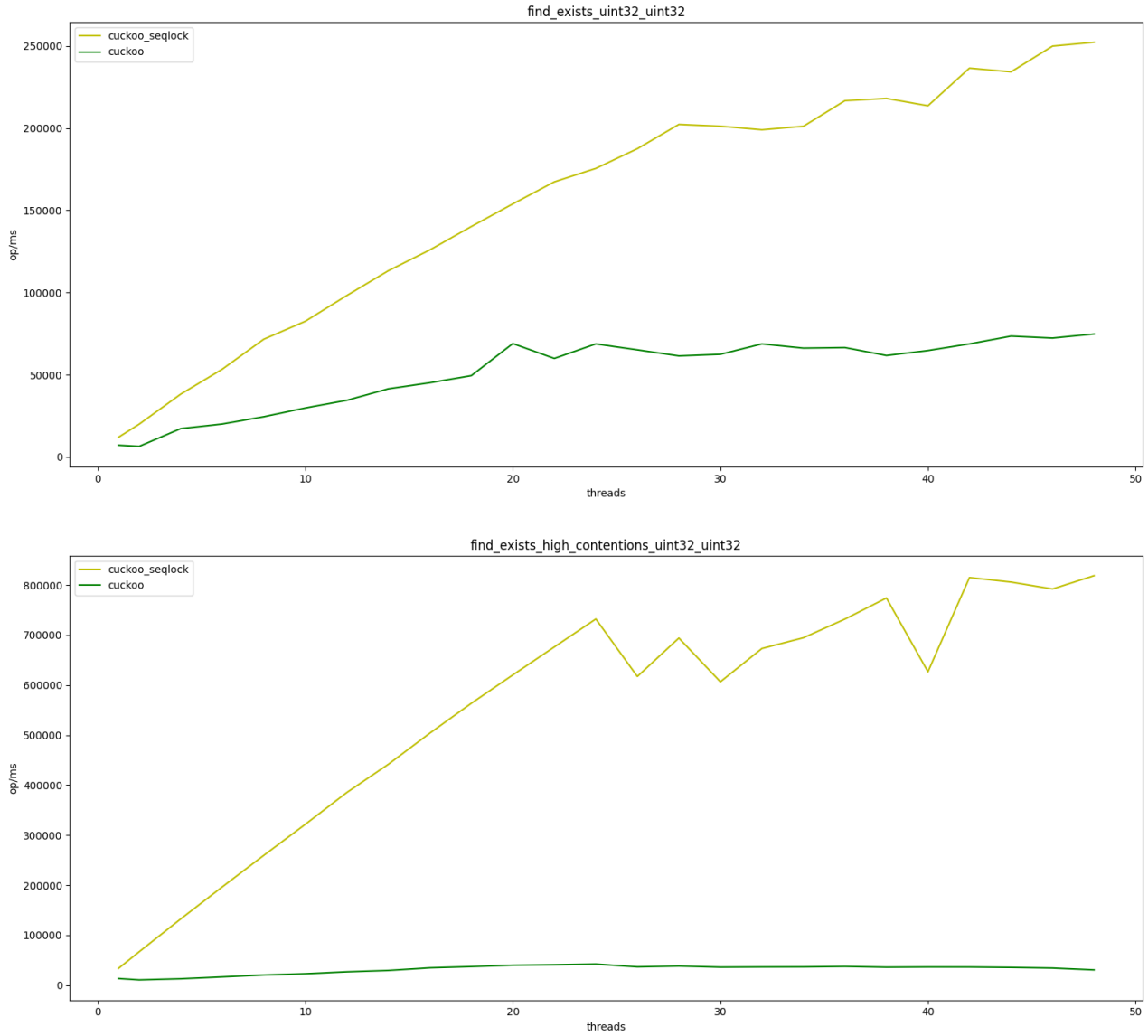

Рисунок 2 – Find exists ARM64

4.2.2. Find modify

Бенчмарк в котором читающие и пишущие операции распределены поровну. Бенчмарк так же представлен в двух версиях аналогично Find exists.

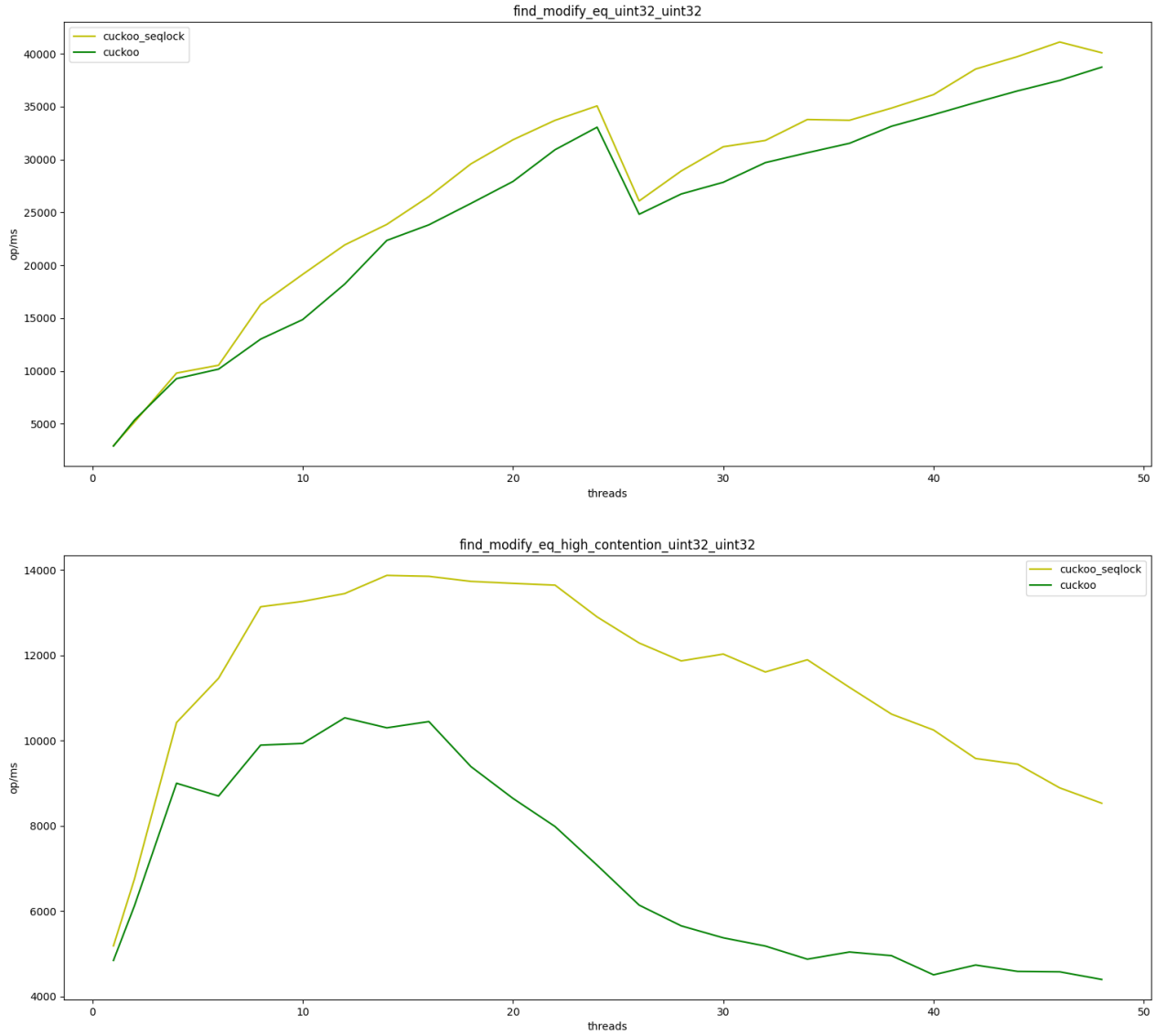

Рисунок 3 – Find modify X86-64

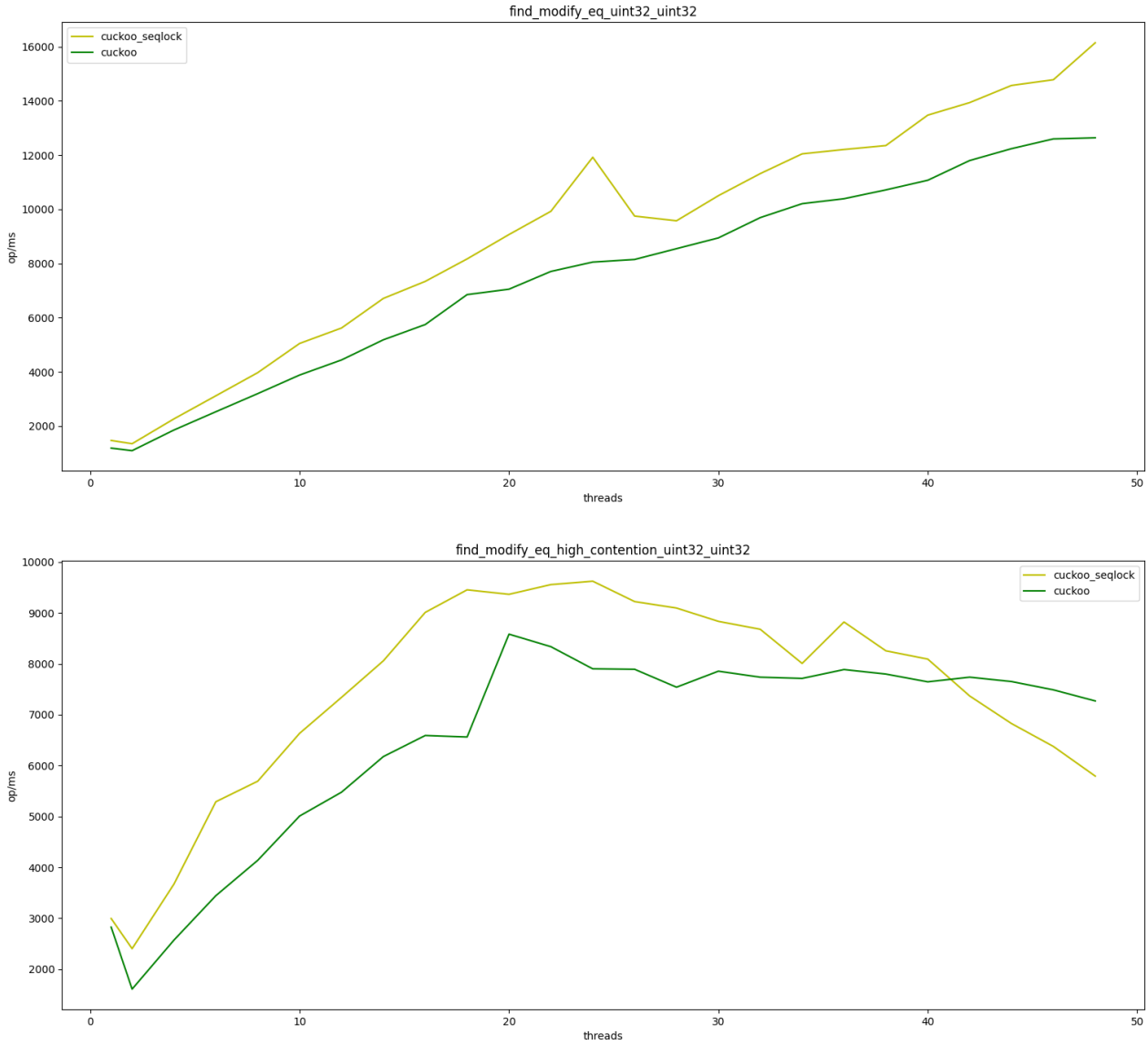

Рисунок 4 – Find modify ARM64

4.2.3. Random

Бенчмарк в котором все операции распределеные поровну, читающие операции к пишущим относятся как 1 к 3. Бенчмарк так же представлен в двух версиях аналогично Find exists.

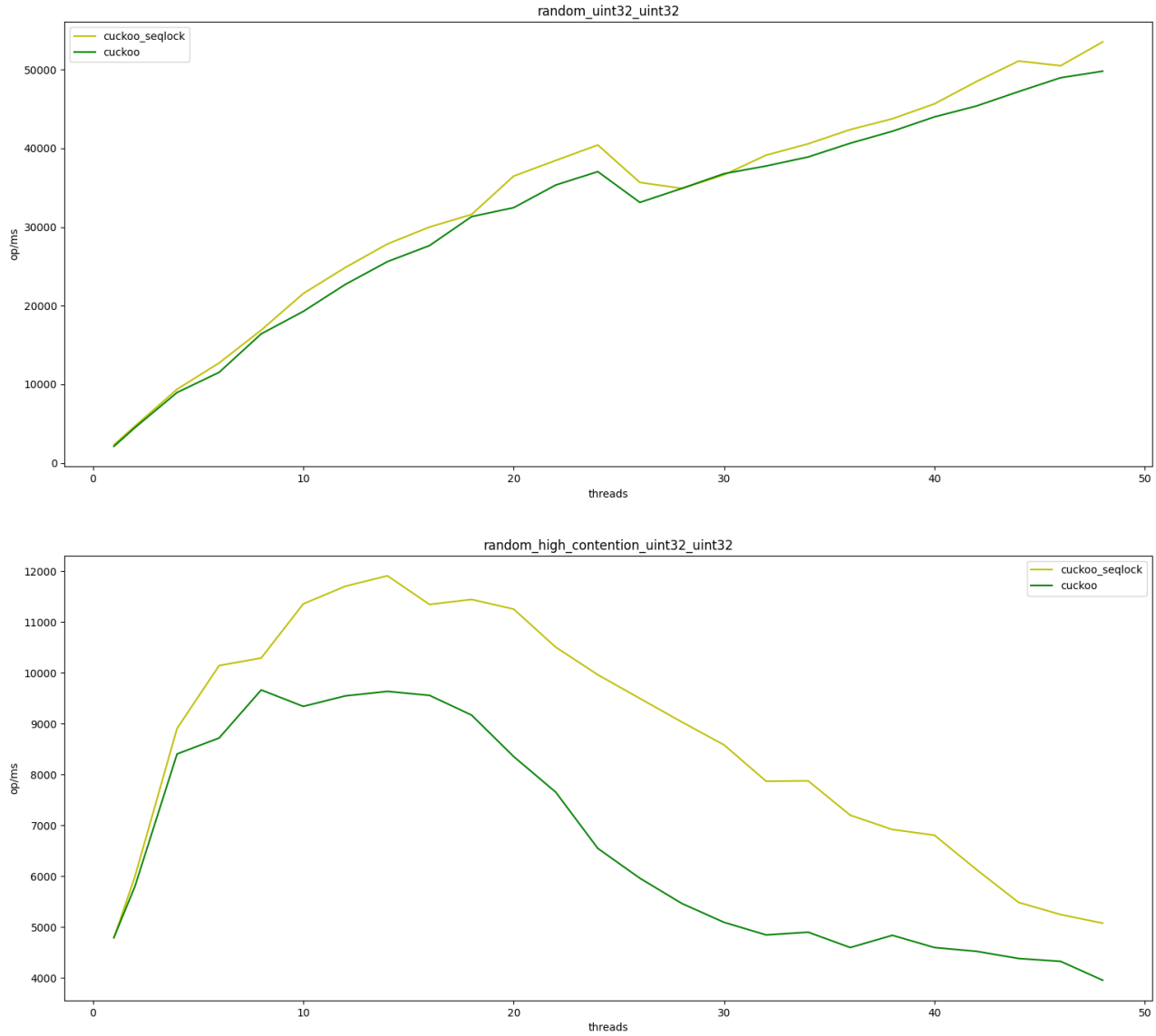

Рисунок 5 – Random X86-64

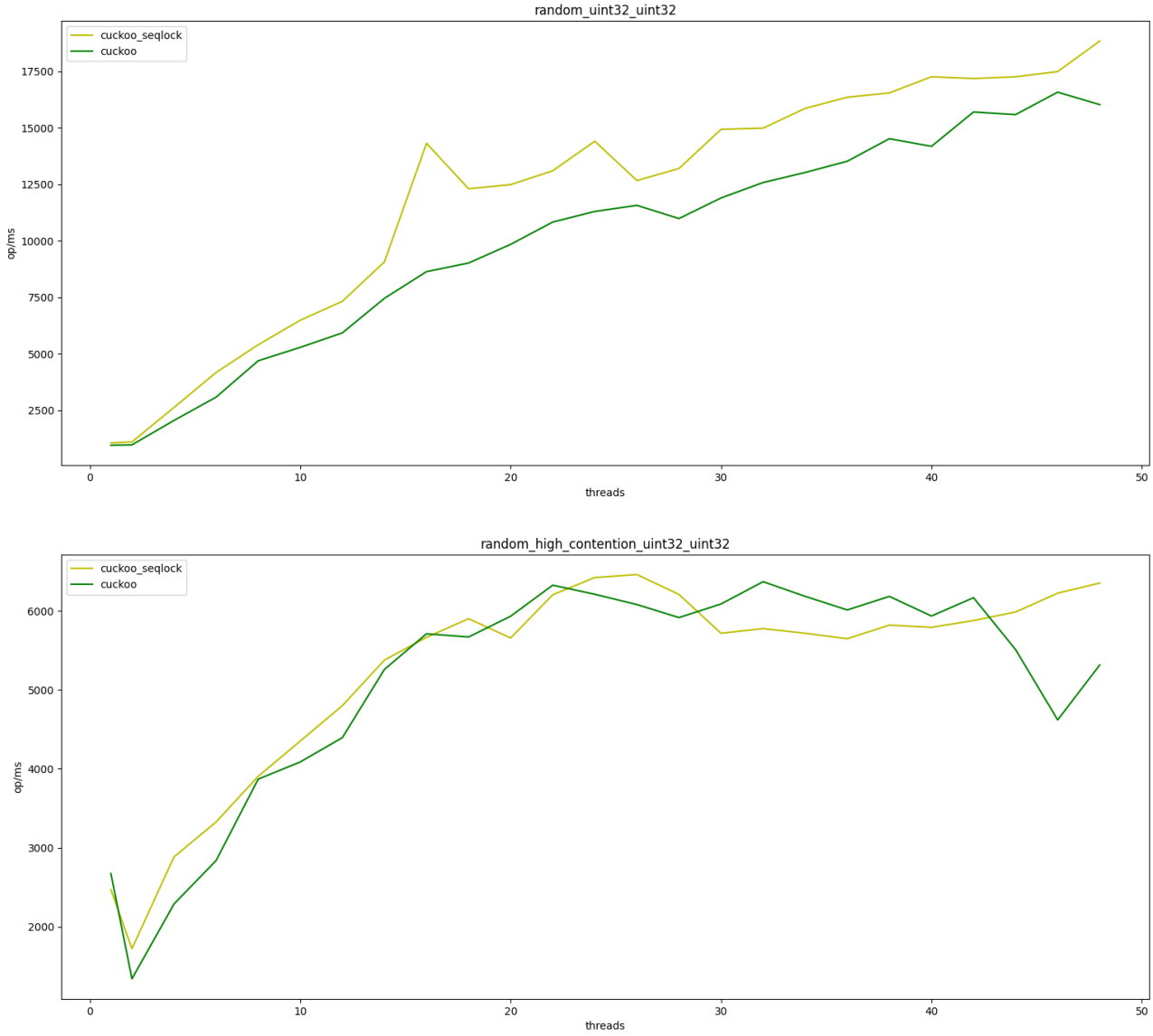

Рисунок 6 – Random ARM64

4.2.4. Inserts

Бенчмак представляющий собой вставку большого количества уникальный ключей. Представлен в двух версиях – с не выделенным заранее хранилищем, то есть совершается еще и рехеширование, и с заранее выделенной памятью.

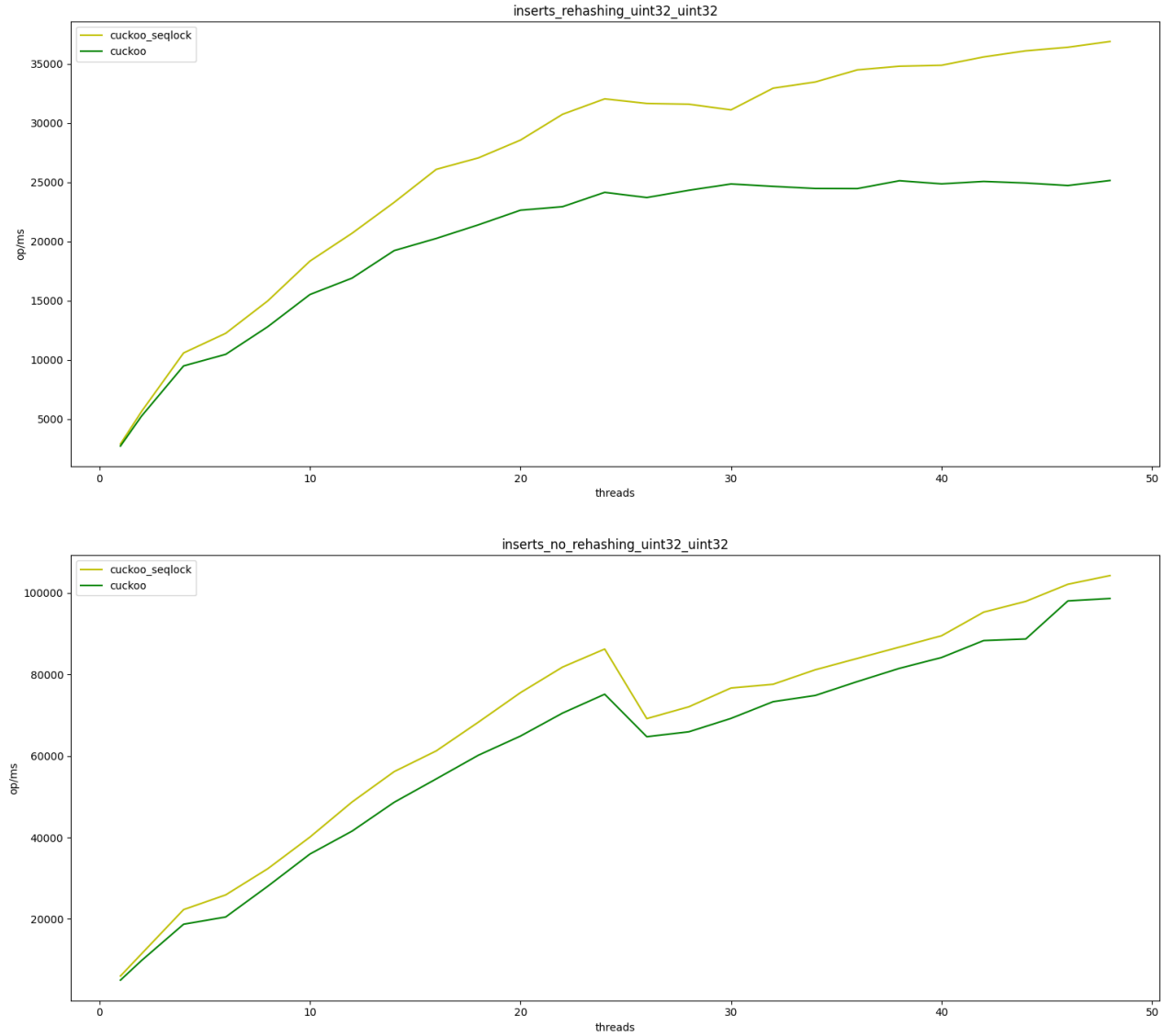

Рисунок 7 – Inserts X86-64

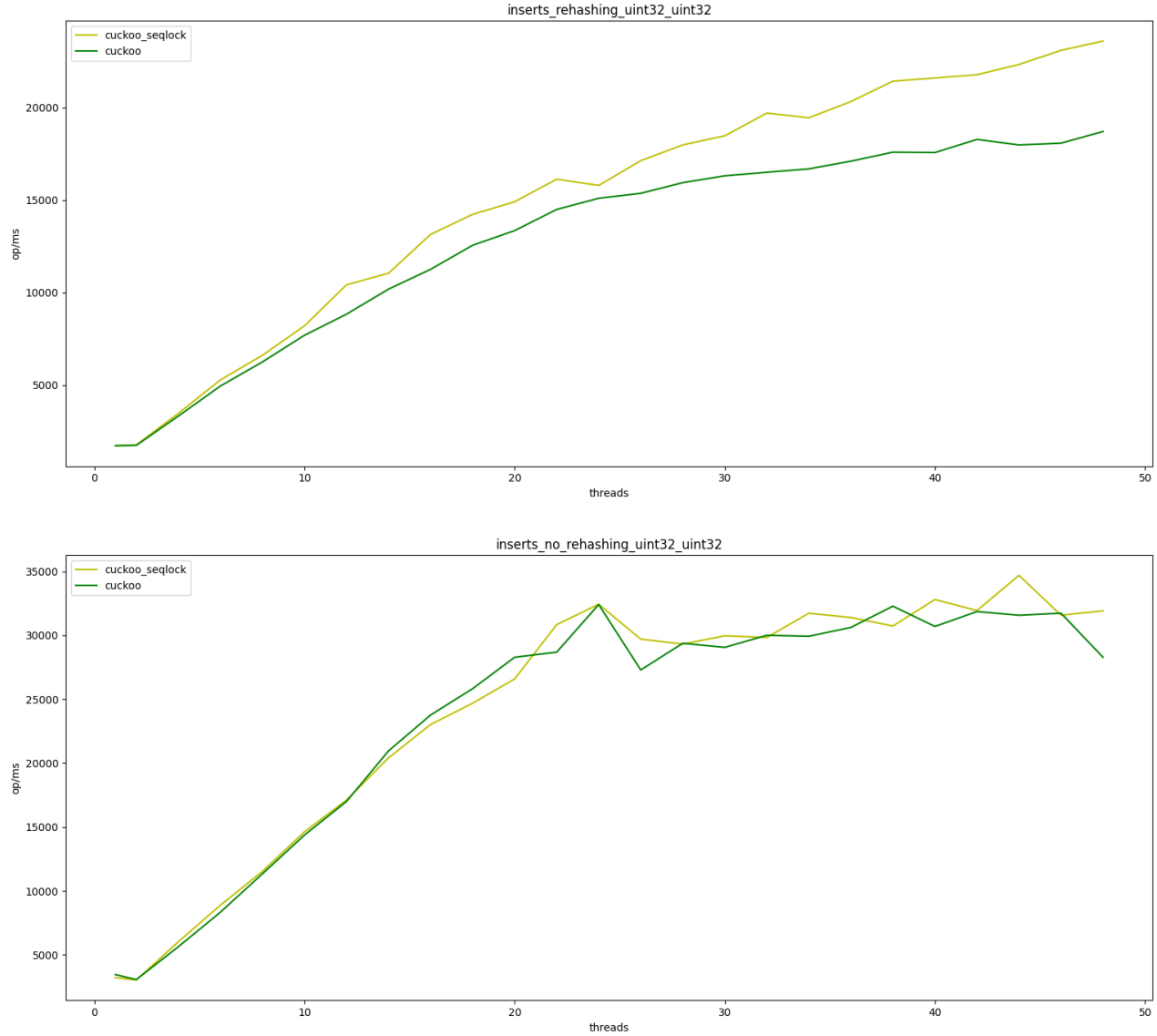

Рисунок 8 – Inserts ARM64

4.2.5. Insert or assigns

Бенчмарк представляющий собой операции вставки или присваивания. Так же реализован в двух вариантах как Find exists.

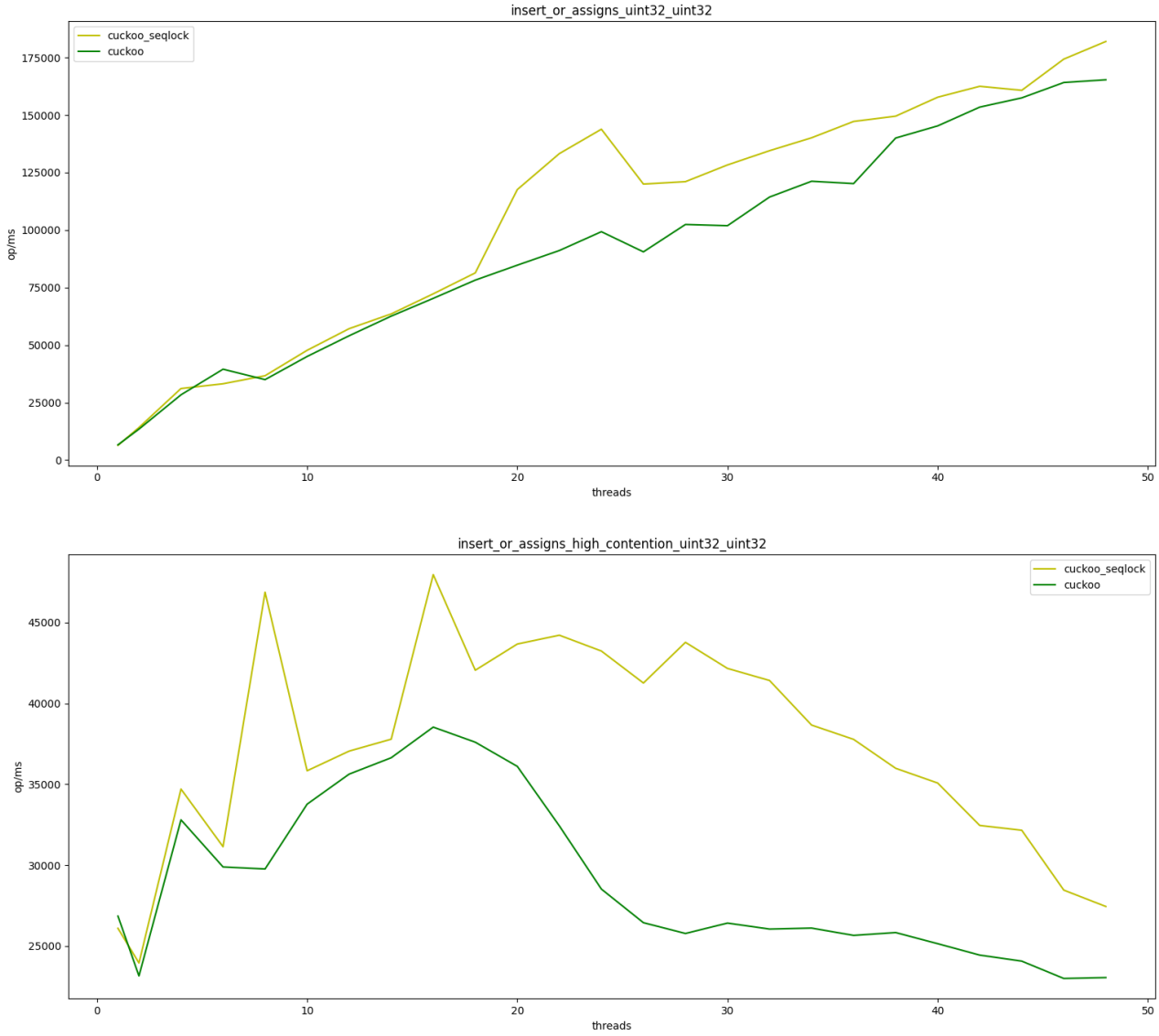

Рисунок 9 – Insert or assigns X86-64

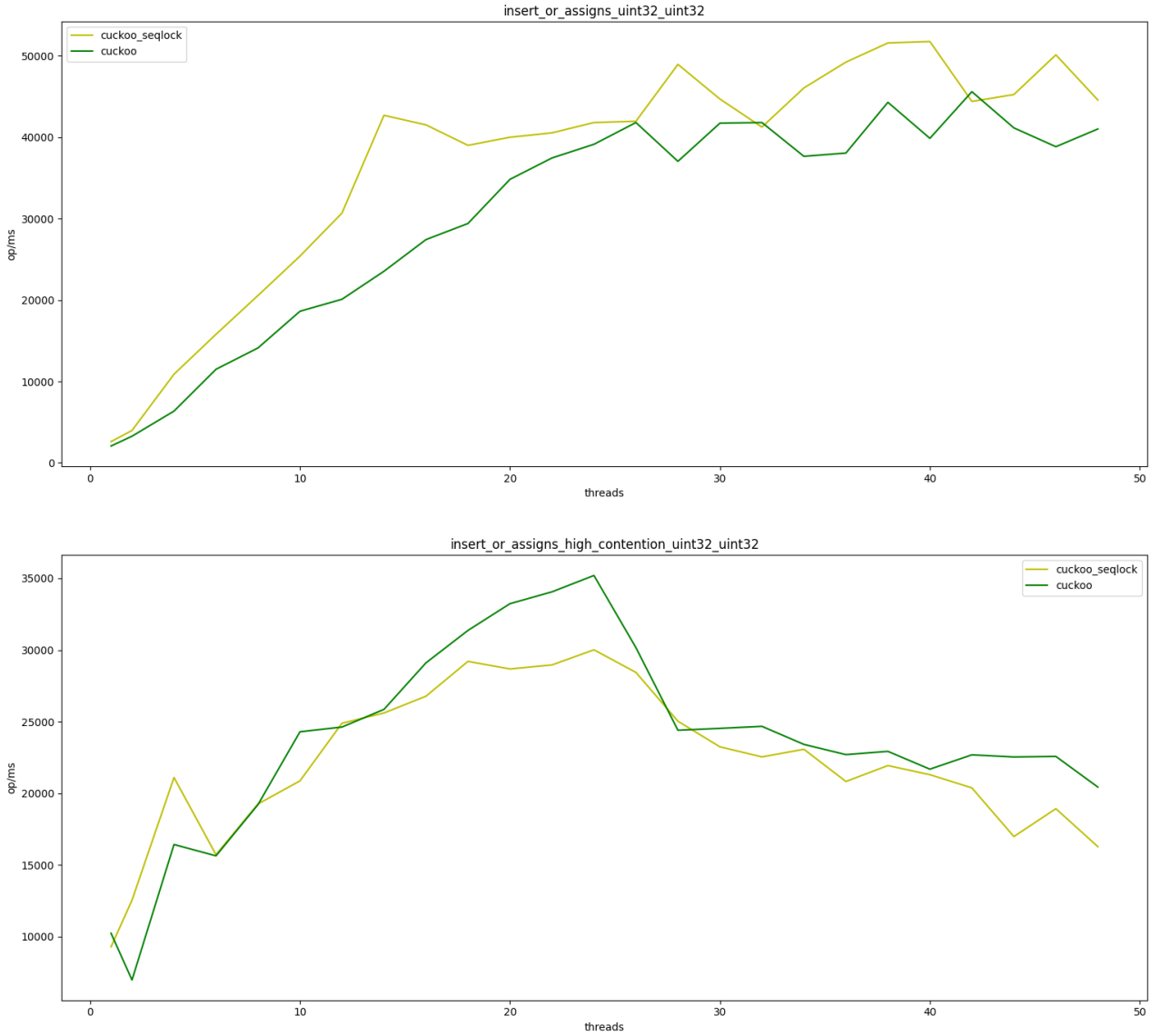

Рисунок 10 – Insert or assigns ARM64

4.2.6. Erase exists

Бенчмарк представляющий удаление большого количества ключей.

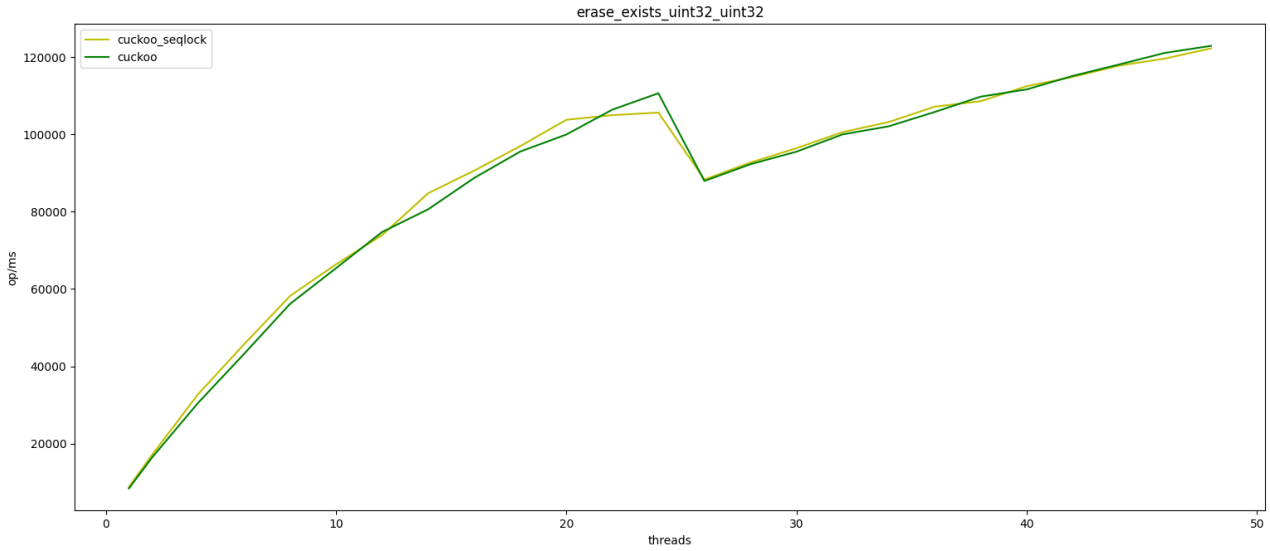

Рисунок 11 – Erase exists X86-64

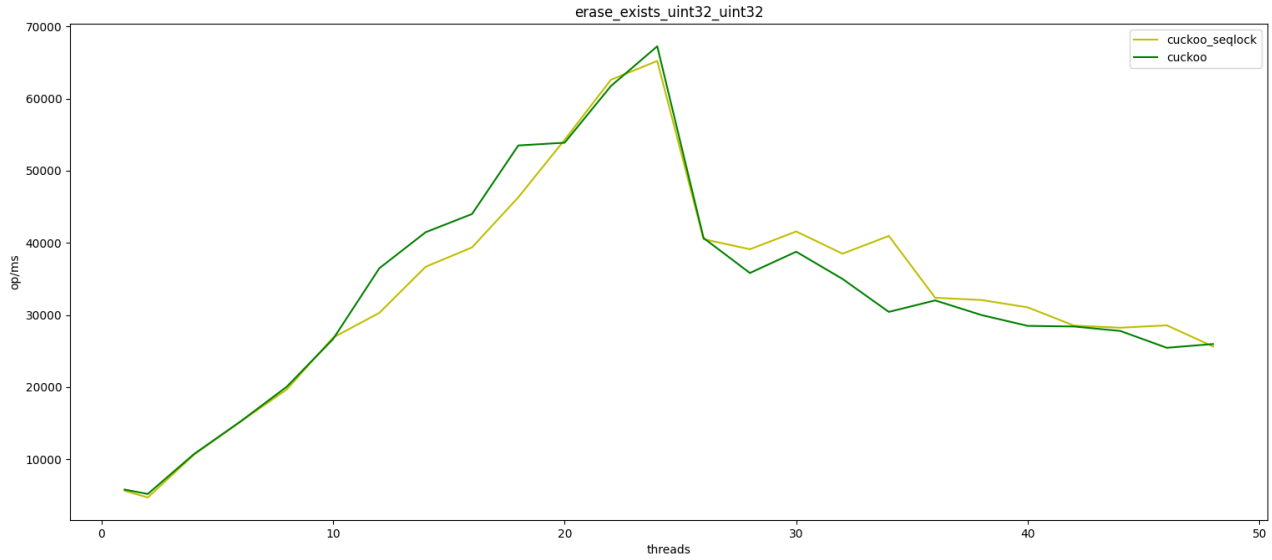

Рисунок 12 – Erase exists ARM64

ЗАКЛЮЧЕНИЕ

Удалось внедрить синхронизацию с секлоком в одни из лучших конкурентных хеш таблиц с открытой адресацией и добиться значительной оптимизации читающей нагрузки без какого либо оверхеда в пишущей, а рехеширование было даже оптимизировано в силу использования сегментированного массива.

Были исследованы особенности работы секлока на двух основных архитектурах, использующихся в современном мире, эксперементально и логически были выделены самые оптимальные реализации для секлока для каждой архитектуры.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1 Algorithmic Improvements for Fast Concurrent Cuckoo Hashing / X. Li [et al.] // EuroSys. — 2014.

2 AMD64 architecture documentation [Электронный ресурс]. — URL: https://www.amd.com/system/files/TechDocs/40332.pdf.

3 ARM64 architecture documentation [Электронный ресурс]. — URL: https://developer.arm.com/documentation/.

4 *Boehm H.-J.* Can Seqlocks Get Along With Programming Language Memory Models? — 2012.

5 C++ atomic_thread_fence documentation [Электронный ресурс]. — URL: https://en.cppreference.com/w/cpp/atomic/atomic_thread_fence.

6 C++ language documentation [Электронный ресурс]. — URL: https://en.cppreference.com.

7 C++ memory model documentation [Электронный ресурс]. — URL: https://en.cppreference.com/w/cpp/language/memory_model.

8 *Celis P.* Robin Hood Hashing. — 1985.

9 *Fan B.*, *Andersen D. G.*, *Kaminsky M.* MemC3: Compact and Concurrent MemCache with Dumber Caching and Smarter Hashing // NSDI. — 2013.

10 Google Benchmark framework documentation [Электронный ресурс]. — URL: https://github.com/google/benchmark/blob/main/docs/user_guide.md.

11 Google Test framework documentation [Электронный ресурс]. — URL: https://google.github.io/googletest.

12 Happens before description [Электронный ресурс]. — URL: https://en.wikipedia.org/wiki/Happened-before.

13 Intel64 architecture documentation [Электронный ресурс]. — URL: https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html.

14 libcuckoo cuckoo hash map algorithm [Электронный ресурс]. — URL: https://github.com/efficient/libcuckoo.

15 *Malakhov A. A.* Per-bucket concurrent rehashing algorithms. — 2015.