

Санкт-Петербургский государственный университет

Калашников Матвей Михайлович

Выпускная квалификационная работа

Оптимизация системы управления очередью задач в Tokio runtime

Уровень образования: бакалавриат

Направление *09.03.04 «Программная инженерия»*

Основная образовательная программа *СВ.5080.2021 «Программная инженерия»*

Научный руководитель:
доцент кафедры системного программирования, к. ф.-м. н., Гориховский В. И.

Рецензент:
Ведущий инженер по разработке ПО, ООО "Ядро Центр Программных Разработок",
Ли С. В.

Санкт-Петербург
2025

Saint Petersburg State University

Kalashnikov Matvey

Bachelor's Thesis

Optimisation of the task queue management system in the Tokio runtime

Education level: bachelor

Speciality *09.03.04 "Software Engineering"*

Programme *CB.5080.2021 "Software Engineering"*

Scientific supervisor:
C.Sc, docent. V. I. Gorikhovskii

Reviewer:
Lead Software Engineer, OOO "YADRO Software Development Center" S.V. Li

Saint Petersburg
2025

Оглавление

Введение	4
1. Постановка задачи	6
2. Обзор механизма планирования задач tokio	7
2.1. Организация многопоточного выполнения в tokio	7
2.2. Метрики для оценки производительности	13
2.3. Обзор аналогов tokio	14
3. Сбор метрик и сценарии тестирования	15
3.1. Используемые инструменты	15
3.2. Сценарий повышенной нагрузки на глобальную очередь	16
3.3. Анализ метрик	17
4. Оптимизация алгоритма управления очередями	21
4.1. Варианты оптимизации	21
4.2. Дополнительная очередь	22
4.3. Используемые очереди	23
5. Особенности реализации	24
5.1. Реализация очередей	24
6. Экспериментальное исследование	28
6.1. Исследовательские вопросы	28
6.2. Измерение производительности очередей	28
6.3. Сравнение производительности модификаций tokio . . .	30
6.4. Выводы	32
Заключение	33
Список литературы	35

Введение

Многопоточное программирование стало неотъемлемой частью разработки высокопроизводительных систем. Такой подход позволяет выполнять множество операций одновременно, эффективно используя вычислительные ресурсы процессора. Это особенно важно для серверных приложений и систем хранения данных, которые подвергаются высокой нагрузке. Однако такой подход ставит перед разработчиком задачу правильной организации доступа к общим данным.

При работе с разделяемыми ресурсами, такими как память, файлы или сетевые соединения, возникает необходимость синхронизации. Существует два вида синхронизации: блокирующая и неблокирующая. Использование неблокирующей синхронизации позволяет снизить задержки доступа к ресурсам, но усложняет реализацию и отладку. Использование блокирующей синхронизации упрощает взаимодействие между потоками, но может привести к простоему процессора из-за ожидания общего ресурса. Эти ограничения требуют более эффективных подходов к выполнению многопоточного кода.

Один из таких подходов — это асинхронные системы поддержки периода исполнения (runtime), которые позволяют эффективно управлять выполнением задач в многопоточных приложениях, минимизируя задержки, связанные с блокирующей синхронизацией. Они распределяют задачи между потоками и обеспечивают кооперативное переключение между ними, уменьшая накладные расходы на создание новых потоков и синхронизацию доступа к общим ресурсам. Такие системы реализованы во многих популярных языках программирования, таких как Python ¹, C# ², Go ³ и многих других, что говорит об их востребованности. В языке Rust одной из самых часто используемых систем поддержки периода исполнения является tokio ^{4 5}.

¹Python asyncio documentation — <https://docs.python.org/3/library/asyncio.html> (22.12.2024).

²C# asynchronous programming documentation — <https://learn.microsoft.com/en-us/dotnet/csharp/asynchronous-programming/> (22.12.2024).

³Effective Go — https://go.dev/doc/effective_go#goroutines (22.12.2024).

⁴Lib.rs — index of Rust libraries and applications — <https://lib.rs/std> (22.12.2024).

⁵Tokio — <https://tokio.rs/tokio> (22.12.2024).

Основой эффективной работы асинхронной системы поддержки периода исполнения является механизм распределения задач между потоками-работниками. Планировщик в `tokio` реализует множество оптимизаций, одной из которых является использование двух типов очередей для хранения задач. Помимо общей глобальной очереди, каждый поток-работник имеет свою локальную очередь для хранения задач ⁶. Такая архитектура позволяет снизить конкуренцию за доступ к глобальной очереди и повысить степень параллелизма.

Не смотря на это, в определенных сценариях исполнения многопоточных программ под управлением `tokio` наблюдается снижение производительности при обработке большого количества коротких задач. Эта проблема была выявлена сотрудниками команды проекта `TATLIN.BACKUP` компании ООО «Ядро Центр Программных Разработок». При этом конкуренция за доступ к глобальной очереди задач становится узким местом, снижая производительность многопоточного приложения.

Таким образом, существующая реализация `tokio` может быть улучшена для определенных сценариев работы. Для этого необходимо провести измерение производительности, а также проанализировать уже существующие решения. Эта работа направлена на проектирование и внедрение улучшенного алгоритма управления очередями задач в `tokio`.

⁶Making the `tokio` scheduler 10x faster — <https://tokio.rs/blog/2019-10-scheduler> (22.12.2024).

1. Постановка задачи

Целью работы является проектирование и внедрение улучшенного алгоритма управления очередями задач в tokio на основании анализа существующих решений и метрик. Для её выполнения были поставлены следующие задачи:

1. Провести анализ существующего алгоритма управления очередями задач в tokio, выделить ключевые метрики для оценки его производительности;
2. Разработать тестовый сценарий для сбора выбранных метрик в условиях повышенной конкуренции за глобальную очередь;
3. Спроектировать модификацию алгоритма управления очередями на основе обзора современных систем поддержки периода исполнения и собранных метрик;
4. Реализовать и интегрировать модификацию алгоритма управления очередями задач;
5. Провести экспериментальное исследование производительности реализованной модификации алгоритма;

2. Обзор механизма планирования задач tokio

Tokio – это библиотека для языка Rust, предоставляющая инструменты для написания масштабируемых многопоточных приложений. Она состоит из нескольких высокоуровневых компонентов.

- Инструменты для работы с асинхронными задачами, такие как примитивы синхронизации, каналы, таймауты, функции ожидания (sleep) и интервалы.
- API для выполнения асинхронного ввода-вывода, включая работу с TCP и UDP сокетами, файловыми операциями, а также управление процессами и сигналами.
- Система поддержки периода исполнения для выполнения асинхронного кода: включает планировщик задач, драйвер ввода-вывода и таймер.

Планировщик задач — это один из ключевых компонентов системы поддержки периода исполнения tokio, который отвечает за эффективное выполнение асинхронного кода. Tokio предоставляет как многопоточный, так и однопоточный планировщик задач. Наиболее часто используется многопоточный планировщик, так как он позволяет выполнять задачи параллельно на нескольких потоках, что особенно важно для высоконагруженных приложений.

2.1. Организация многопоточного выполнения в tokio

Как показано на рисунке 1, многопоточная версия tokio представляет собой систему на основе фиксированного пула потоков. Количество потоков в пуле обычно определяется количеством ядер процессора, но может быть настроено пользователем. Каждый поток-работник имеет

свою локальную очередь задач, в которую помещаются задачи, ожидающие выполнения. Кроме локальных очередей, существует глобальная очередь задач, к которой каждый поток-работник может получить блокирующий доступ.

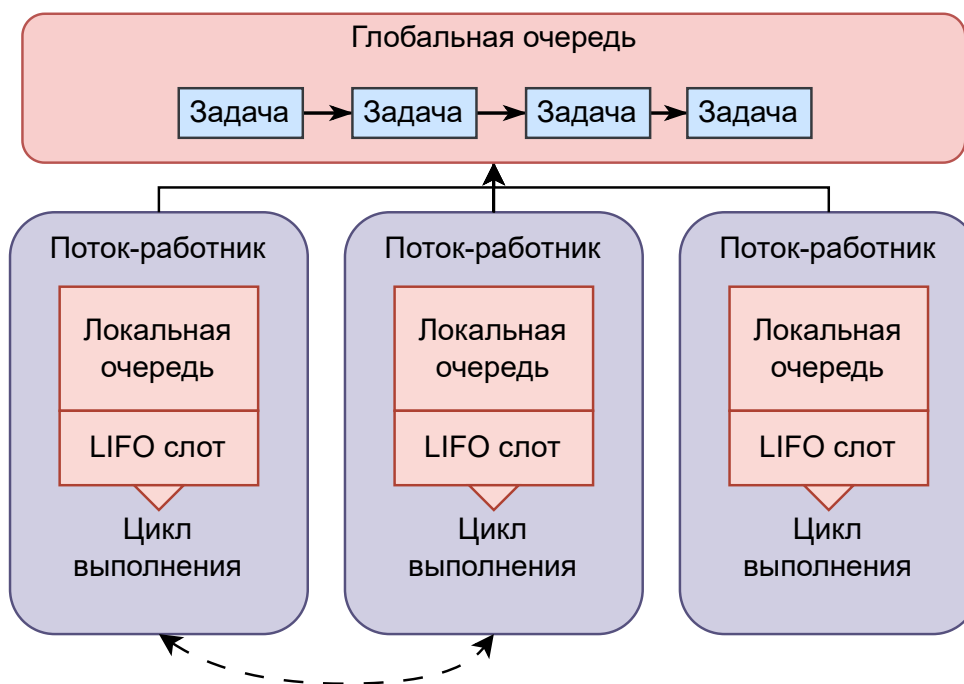


Рис. 1: Архитектура многопоточной системы поддержки периода исполнения

В tokio единицей планирования является задача. Запуск задачи отправляет ее в планировщик. Когда в коде задачи вызывается конструкция `.await`, выполнение этой задачи приостанавливается, и она возвращается в планировщик. В следующий раз, когда задача будет запущена, она продолжит выполнение с того же места, где была остановлена. В это время задача может быть перемещена между потоками.

2.1.1. Очереди задач

Tokio использует две разновидности очередей задач: локальные очереди для каждого потока и общую глобальную очередь.

Локальные очереди представляют собой кольцевой буфер фиксиро-

ванного размера, что позволяет избежать лишних операций по выделению памяти при добавлении новых задач в очередь. Очередь реализована как структура SPMC (Single Producer, Multiple Consumer). В этом случае единственным производителем является поток, который владеет очередью. Это позволяет упростить механизмы синхронизации.

В дополнение к локальным используется единая для всех потоков глобальная очередь, которая представляет собой связанный список, доступ к которому синхронизируется с помощью блокировки. Задачи попадают в глобальную очередь, когда планирование задачи происходит вне контекста потока-работника, а также в случае переполнения локальной очереди. В тех случаях, когда локальная очередь переполняется или пуста, задачи могут быть вытеснены в глобальную очередь или взяты из нее.

2.1.2. Алгоритм работы потока-работника

```
1 def run(core):
2     while not core.is_shutdown:
3         core.tick()
4         # Пробуем получить следующую задачу
5         task = core.next_task(worker)
6         if task:
7             core = run_task(task, core)
8             continue
9         # Пробуем украсть задачу
10        task = core.steal_work(worker)
11        if task:
12            core = run_task(task, core)
13        else:
14            # Проверяем задачи ожидающие ресурсов
15            if not defer_is_empty():
16                core = park_timeout(core, timeout=0)
17            else:
18                core = park(core) # Ждем появления задач
19    return
```

Листинг 1: Псевдокод обработки задач потоком-работником

Алгоритм работы потока-работника⁷, представленный в листинге 1, представляет собой цикл, последовательно обрабатывающий задачи. В

⁷https://docs.rs/tokio/1.42.1/src/tokio/runtime/scheduler/multi_thread/worker.rs.html#522

начале цикла инкрементируется счетчик, который служит для выполнения периодических задач. После этого поток пробует получить следующую задачу из очередей. Если это не удастся, поток пробует украсть задачи из локальных очередей других потоков-работников. Если не удастся получить задачу из очередей или украсть ее у других потоков-работников, поток проверяет наличие задач, ожидающих наступления событий, таких как завершение ввода-вывода или других сигналов для пробуждения. При их отсутствии поток переходит в состояние ожидания.

2.1.3. Алгоритм получения следующей задачи

```
1 def next_task(self, worker):
2     if self.tick % 61 == 0:
3         task = worker.handle.next_remote_task()
4         if task is not None:
5             return task
6         else:
7             return self.next_local_task()
8     else:
9         task = self.next_local_task()
10        if task is not None:
11            return task
12        if worker.global_queue().is_empty():
13            return None
14        # Количество задач для получения из глобальной очереди
15        capacity = min(
16            self.local_queue.remaining_slots(),
17            self.local_queue.max_capacity() // 2,
18            worker.global_queue().len() //
19            len(worker.handle.shared.remotes) + 1
20        )
21        n = max(1, n)
22        with worker.handle.shared.synced.lock():
23            tasks = worker.global_queue().pop_n(n)
24            # Берем первую задачу для выполнения
25            ret = tasks.pop(0)
26            # Остальные добавляем в локальную очередь
27            self.local_queue.push_back(tasks)
28            return ret
29 def next_local_task(self):
30     task = self.lifo_slot.take()
31     if task is None:
32         task = self.run_queue.pop()
33     return task
```

Листинг 2: Псевдокод получения следующей задачи

Функция, показанная в листинге 2, управляет получением следующей задачи потоком-работником⁸. Алгоритм может извлекать задачи как из глобальной, так и из локальной очереди.

Каждую 61 итерацию поток пробует получить задачу из глобальной очереди. В остальных случаях поток обращается сначала к локальной очереди. Если в локальной очереди нет задач, то поток пробует получить сразу группу задач из глобальной очереди и помещает их в локальную очередь. Это позволяет снизить конкуренцию за глобальную

⁸https://docs.rs/tokio/1.42.1/src/tokio/runtime/scheduler/multi_thread/worker.rs.html#783

очередь. Размер группы задач ограничивается вместимостью локальной очереди и количеством задач в глобальной очереди, делённого на число потоков. Это позволяет равномерно распределять задачи между потоками. Несмотря на пакетное получение задач, при большом числе потоков проблема высокой конкуренции за глобальную очередь сохраняется и приводит к снижению производительности.

Также `tokio` реализует оптимизацию, направленную на более эффективное использование кэшей процессора. Вместо того, чтобы помещать задачу, созданную потоком, в конец очереди, задача помещается в специальный слот. Это позволяет повысить производительность в тех случаях, когда задачи связаны по данным.

2.1.4. Механизм `work stealing`

В `tokio` для эффективного распределения нагрузки между потоками-работниками используется механизм `work stealing`. Этот метод позволяет потокам, завершившим обработку своих собственных задач, забирать задачи из других локальных очередей.

```
1 def steal_work(self, worker):
2     # Переход в состояние поиска работы
3     if not self.transition_to_searching(worker):
4         return None
5     num_workers = len(worker.handle.shared.remotes)
6     # Выбор случайного начального потока-работного
7     start = self.rand.fastrand(num_workers)
8     for i in range(num_workers):
9         index = (start + i) % num_workers
10        # Не красть у самого себя
11        if index == worker.index:
12            continue
13        target = worker.handle.shared.remotes[index]
14        # Попытка украсть задачу из другой локальной очереди
15        task = target.steal.steal_into(self.run_queue, self.stats)
16        if task:
17            return task
18    # Проверить глобальную очередь
19    return worker.handle.next_remote_task()
```

Листинг 3: Псевдокод кражи задач

В листинге 3 представлен псевдокод кражи задач⁹. Когда локальная очередь потока пуста, поток входит в состояние поиска задач. Случайно выбирается один из соседних потоков, и производится попытка украсть у него задачи. Если локальная очередь выбранного потока пуста, переходим к следующему потоку-работнику. Если в локальных очередях других потоков нет задач, поток единожды пробует получить задачу из глобальной очереди.

На практике часто возникает ситуация, когда у множества потоков одновременно заканчиваются задачи в локальных очередях. Для того чтобы избежать высокой конкуренции при попытках кражи задач, количество потоков, которые могут находиться в этом состоянии, ограничивается. Для этого используется атомарный счетчик, который увеличивается перед началом поиска задач и уменьшается после завершения операции. Количество потоков-работников, которые находятся в состоянии поиска, ограничено половиной от числа всех потоков-работников.

2.2. Метрики для оценки производительности

Необходимо выбрать набор первичных метрик, которые позволят оценить эффективность работы tokio в сценариях повышенной конкуренции за глобальную очередь. Были выделены две целевые метрики, которые характеризуют производительность системы поддержки периода исполнения: время выполнения и пропускная способность. Пропускная способность – определяет количество выполненных задач за единицу времени.

Также для анализа поведения системы поддержки периода исполнения были выбраны вспомогательные метрики: количество задач в глобальной очереди, количество переполнений локальных очередей и суммарная длина локальных очередей потоков-работников. Они позволяют получить представление о поведении tokio в различных сценариях тестирования.

⁹https://docs.rs/tokio/1.42.1/src/tokio/runtime/scheduler/multi_thread/worker.rs.html#847

2.3. Обзор аналогов tokio

Для поиска возможных оптимизаций алгоритма управления очередями задач в рамках tokio были рассмотрены архитектуры аналогичных систем. В качестве примеров были выбраны реализации систем поддержки периода исполнения в Go, Kotlin и Erlang.

- **Go.** Tokio во многом вдохновлен архитектурой системы поддержки периода исполнения Go и наследует множество его идей, в том числе наличие глобальной очереди и локальных очередей потоков-работников. Таким образом, Go не может быть использован для оптимизации алгоритма управления очередями.
- **Kotlin Coroutines.** Kotlin Coroutines также имеет архитектурное сходство с системой поддержки периода исполнения Go. Каждый поток-работник имеет собственную локальную очередь. Отличительной особенностью является использование двух глобальных очередей: одной для процессорозависимых задач, второй для блокирующих операций, таких как I/O. Для обработки блокирующих задач планировщик может создавать дополнительные потоки. Несмотря на разделение очередей по типам задач доступ к ним также может являться узким местом при высокой конкуренции.
- **Erlang.** В отличие от перечисленных систем, Erlang использует модель акторов, в которой каждый процесс обладает собственной очередью сообщений, а глобальная очередь задач отсутствует. В такой архитектуре отсутствует проблема конкуренции за общую структуру данных. Тем не менее, интеграция такой модели в tokio потребовала бы его полной переработки.

Существующие системы поддержки периода исполнения либо имеют схожую с tokio архитектуру, либо значительно отличаются. Это не позволяет применить их решения для оптимизации алгоритма управления очередями.

3. Сбор метрик и сценарии тестирования

Работа, описанная в этом разделе, выполнялась совместно с Игорем Ериным. Разработка велась в следующем репозитории ¹⁰. Я работал над сбором метрик и построением графиков.¹¹

Для оценки производительности и сравнения подходов к оптимизации работы с очередями был разработан тестовый сценарий, а также код для сбора дополнительных метрик. Для измерения времени и пропускной способности использовалась библиотека `criterion`, а для сбора метрик — библиотека `tokio-metrics`.

3.1. Используемые инструменты

Criterion ¹² — библиотека, предназначенная для измерения производительности кода. Она предоставляет инструменты для измерения времени выполнения кода с высокой точностью ¹³, а также для анализа полученных результатов.

Процесс замера состоит из фазы разогрева и фазы измерения времени. Фаза измерения состоит из множества групп запусков. Каждая группа запусков состоит из возрастающего числа итераций, например, 10, 20, 30 и более. Общее время выполнения каждой группы затем используется для оценки времени выполнения тестируемого кода. Результаты используются для вычисления среднего и медианного времени, доверительных интервалов и других статистических показателей.

Tokio metrics ¹⁴ — библиотека для сбора метрик в асинхронных приложениях, написанных с использованием `tokio`. Она позволяет отслеживать как состояние системы поддержки периода исполнения, так и состояние отдельных задач. Основные метрики, доступ к которым

¹⁰Репозиторий проекта `tokiobench` — <https://github.com/IgorErin/tokiobench>

¹¹<https://github.com/IgorErin/tokiobench/pulls?q=is%3Apr+is%3Amerged+author%3Aforeverjun>

¹²`Criterion` — statistics-driven benchmarking library for Rust — <https://github.com/bheisler/criterion.rs> (23.12.2024).

¹³`Criterion` documentation — <https://bheisler.github.io/criterion.rs/book/analysis.html> (23.12.2024).

¹⁴`Tokio metrics` — utilities for collecting metrics from a `tokio` application — <https://github.com/tokio-rs/tokio-metrics> (23.12.2024).

предоставляет библиотека, включают число рабочих потоков, количество задач в глобальной и локальных очередях, количество задач, вытесненных из локальных очередей, и другие метрики.

Serde¹⁵ — фреймворк для сериализации и десериализации данных в различных форматах.

3.2. Сценарий повышенной нагрузки на глобальную очередь

Сценарий повышенной нагрузки на глобальную очередь разработан на основе существующих сценариев тестирования tokio, а также с учетом требований сотрудников компании «Ядро» из команды проекта TATLIN.BACKUP. Его цель состоит в том, чтобы имитировать ситуации, когда глобальная очередь заполняется большим количеством коротких процессорозависимых задач. С помощью этого сценария можно оценить, как изменяется производительность в зависимости от количества потоков-работников, взаимодействующих с глобальной очередью.

Нагрузка создается путем добавления задач в локальные очереди потоков-работников. С увеличением количества задач локальные очереди переполняются, что приводит к вытеснению задач в глобальную очередь. Для этого используются задачи-производители, которые создают пустые задачи, выполняющие роль нагрузки как для глобальной, так и для локальных очередей.

3.2.1. Варьируемые параметры

Для оценки поведения системы поддержки периода исполнения и сбора метрик в различных условиях варьируются следующие параметры:

1. **Число потоков-работников**, обрабатывающих задачи из локальных и глобальной очередей.

¹⁵Serde – serialization framework for Rust — <https://github.com/serde-rs/serde> (23.12.2024).

2. **Количество пустых задач**, добавляемых каждой задачей-производителем.
3. **Количество задач-производителей** определяет, сколько задач создается для генерации пустых задач.

3.3. Анализ метрик

Для измерения времени и пропускной способности используется библиотека Criterion. Она выполняет все рутинные вычисления и сохраняет результаты в формате JSON для последующего анализа.

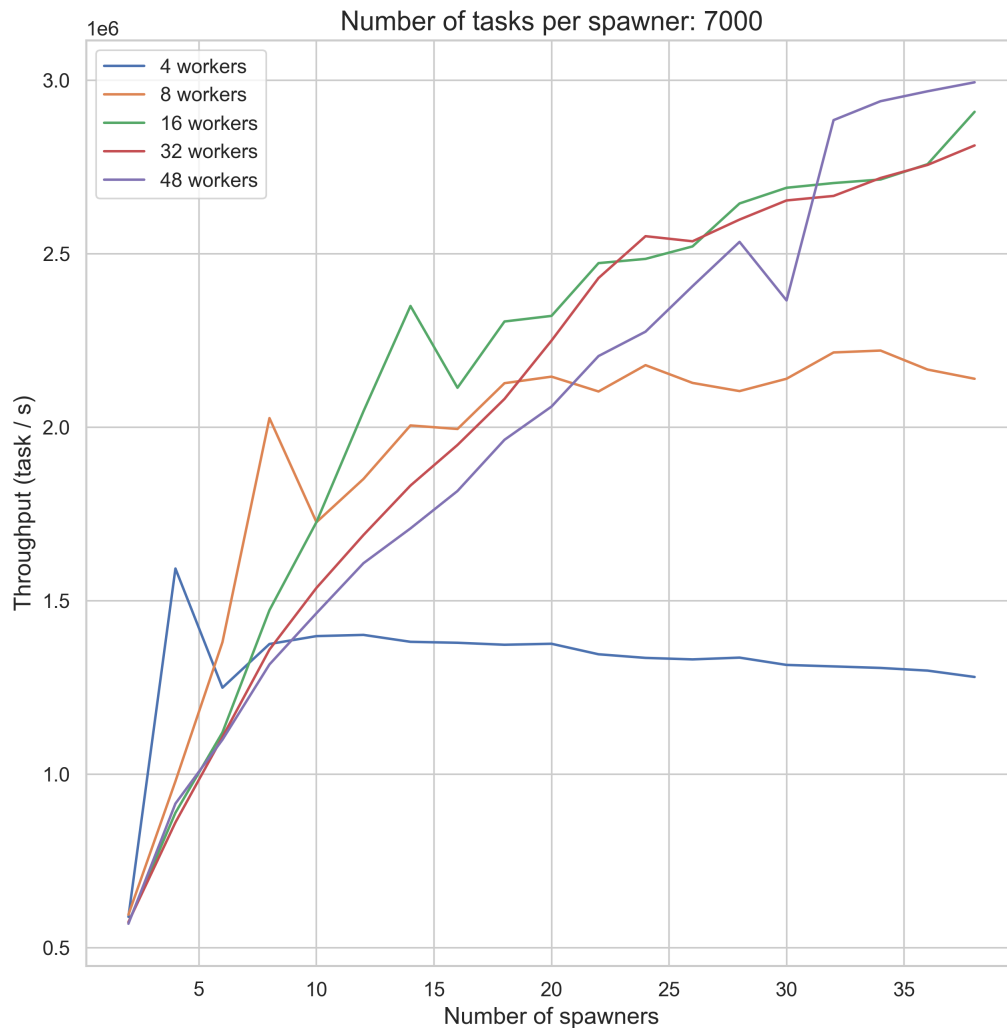


Рис. 2: График пропускной способности tokio. Больше — лучше

График на рисунке 2 отражает зависимость пропускной способности системы поддержки периода исполнения от количества задач-производителей и количества потоков-работников. Можно заметить, что с увеличением количества задач-производителей пропускная способность растет, пока их число не достигает определенного порога. Это происходит из-за того, что есть предел производительности системы, определяемый числом доступных потоков-работников.

Увеличение количества потоков-работников позволяет системе достигать более высокой пропускной способности, однако этот рост непропорционален и наблюдается только после достижения определенного количества задач-производителей. В случаях, когда количество задач-производителей недостаточно для полной загрузки системы, прирост производительности от увеличения количества потоков отсутствует. Это связано с конкуренцией за разделяемые ресурсы системы поддержки периода исполнения, в частности, за глобальную очередь задач.

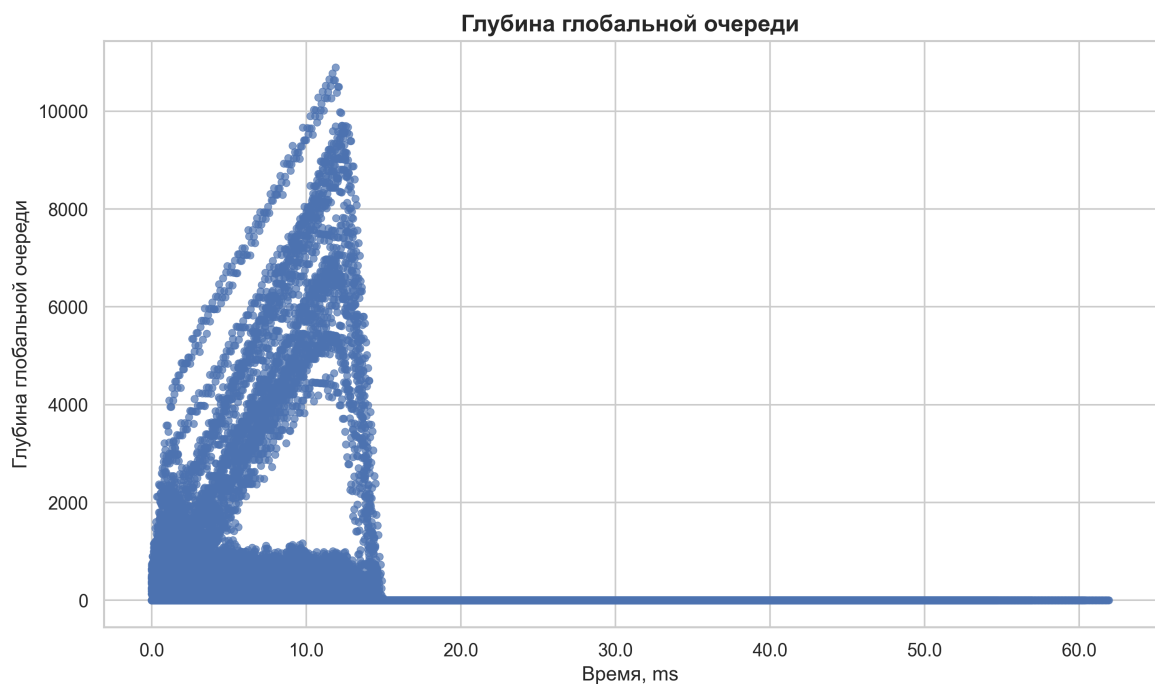


Рис. 3: Глубина глобальной очереди при 8 задач-производителях и 16 потоках-работниках

Для того чтобы дополнительно подтвердить факт вытеснения задач в глобальную очередь, рассмотрим график количества задач в глобаль-

ной очереди для 8 задач-производителей, 16 потоков-работников и 7000 задач для каждой задачи производителя. На рисунке 3 каждая точка представляет собой количество задач в очереди в данный момент времени. На графике собраны данные от 100 запусков. Видно, что количество задач в глобальной очереди начинает расти с самого начала измерений, достигая значения порядка 10 000. Это еще раз подтверждает то, что потоки-работники активно взаимодействуют с глобальной очередью в данном сценарии тестирования.

Решение этой проблемы можно условно разделить на два независимых подхода. В этой работе основное внимание уделяется оптимизации взаимодействия потоков-работников с глобальной очередью — в частности, снижению накладных расходов на получение задач. Альтернативный подход к улучшению производительности рассматривается в ВКР Игоря Ерина «Оптимизация алгоритмов *injected queue* в рантайме *Tokio*». Он сосредоточен на шардировании глобальной очереди. Оба подхода направлены на решение одной и той же проблемы с разных сторон, поэтому их совместное применение возможно сможет дать лучший результат.

Технические характеристики

Тестирование производительности выполнялось на виртуальной машине "Yandex Cloud". Для обеспечения воспроизводимости и стабильности все тесты были выполнены на одном NUMA-узле (логические процессоры 0-47), что соответствует одному физическому сокету. Закрепление на ядрах производилось с использованием утилиты `taskset` 0-47. Приоритет выполнения тестовых потоков был повышен до максимального значения (`nice -20`).

- **Модель CPU:** Intel Xeon Processor (Icelake)
- **Используемые ресурсы CPU для бенчмарка (NUMA-узел 0):**

— Физические ядра: 24

- Логические процессоры (потoki): 48
- **Кэш-память (для используемых 24 ядер / 48 потоков):**
 - **L1d (кэш данных):** 1.5 MiB
 - **L1i (кэш инструкций):** 1.5 MiB
 - **L2:** 96 MiB
 - **L3:** 16 MiB

Виртуальная машина была сконфигурирована с 96 ГБ оперативной памяти и использовала гипервизор KVM. Для обеспечения максимальной производительности был запрошен полный доступ к ядрам процессора.

4. Оптимизация алгоритма управления очередями

В ходе анализа результатов работы тестовых сценариев и архитектуры tokio было выявлено, что он сталкивается с ограничением масштабируемости. С увеличением количества потоков-работников растет конкуренция за глобальную очередь. В рамках поиска решения были рассмотрены различные варианты модификации tokio. В результате было принято решение использовать дополнительную очередь, работающую совместно с глобальной. Такой подход позволит снизить количество обращений к глобальной очереди.

4.1. Варианты оптимизации

Существуют различные подходы к оптимизации алгоритмов управления очередями в системах поддержки периода исполнения. Можно оптимизировать как взаимодействие потоков с глобальной очередью, так и взаимодействие потоков между собой.

В одной из рассмотренных статей предлагается разбить локальные очереди потоков-работников на блоки [2]. Утверждается, что механизм кражи задач значительно влияет на производительность владельца локальной очереди, и даже при небольшом количестве попыток кражи пропускная способность сокращается на 17.8%. Разбиение на блоки позволило добиться увеличения пропускной способности на 12%. Для нашего случая такой подход неприменим, поскольку глобальная очередь быстро заполняется задачами, и механизм кражи задач перестает работать.

Также мне показался интересным подход с использованием N^2 очередей, где каждому потоку-работнику соответствует отдельная очередь для взаимодействия с каждым другим потоком [10]. Из-за необходимости полной переработки tokio данный подход неприменим в нашем случае.

На первый взгляд, замена глобальной очереди на более производи-

тельную неблокирующую реализацию (например, очередь с ослабленным порядком из статьи [3]) решила бы проблему снижения производительности. Однако в tokio глобальная очередь и состояние потоков-работников находятся под общей блокировкой, при этом переходы между состояниями потоков зависят от наличия задач в глобальной очереди. Разделить этот примитив синхронизации и удостовериться в корректности получившейся модификации не удалось из-за высокой сложности tokio. Было решено сохранить глобальную очередь, но уменьшить число обращений к ней за счет добавления дополнительной очереди.

4.2. Дополнительная очередь

Дополнительная очередь выполняет роль буфера между потоками-работниками и глобальной очередью. Работа с дополнительной очередью состоит из следующих этапов.

1. **Вытеснение задач из локальной очереди.** Когда локальная очередь потока-работника переполняется, половина задач из нее переносится в глобальную очередь.
2. **Перенос задач в дополнительную очередь.** При достижении определенного количества задач в глобальной очереди группа задач из начала очереди переносится в дополнительную очередь.
3. **Приоритет дополнительной очереди.** При поиске следующей задачи для выполнения поток-работник сначала обращается к дополнительной очереди. Это позволяет снизить нагрузку на глобальную очередь.
4. **Обращение к глобальной очереди.** Если дополнительная очередь пуста, поток-работник обращается к глобальной очереди для получения задач.

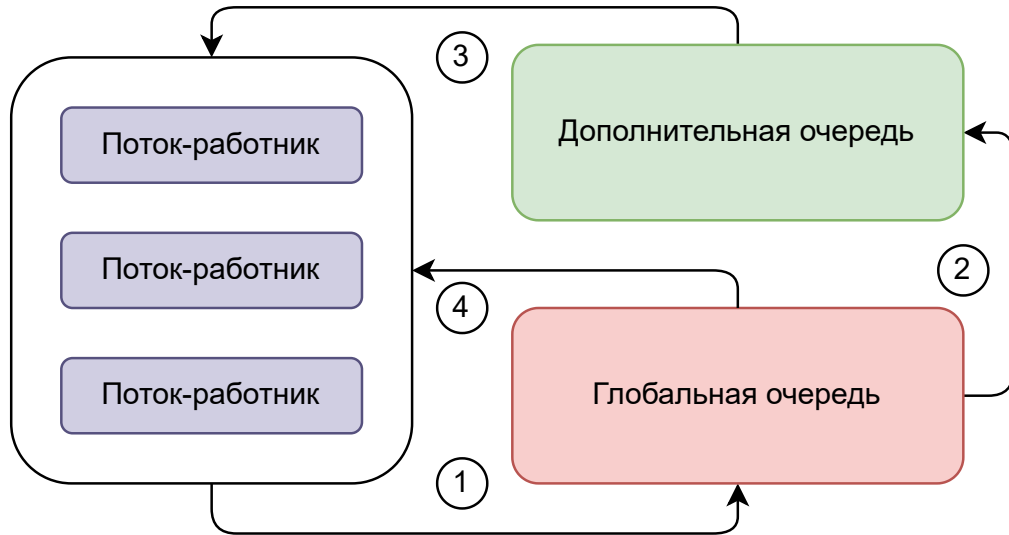


Рис. 4: Схема работы системы с дополнительной очередью

4.3. Используемые очереди

В качестве дополнительной очереди использовались структуры данных, основанные на атомарных операциях. Такой подход эффективнее при высокой конкуренции и снижает расходы на синхронизацию.

Для использования в качестве дополнительной очереди были выбраны следующие реализации:

- **SegQueue** — очередь, состоящая из блоков, каждый из которых представляет собой массив фиксированного размера ¹⁶.
- **FAAArrayQueue** — неблокирующая очередь, состоящая из блоков фиксированного размера. Использует операции FAA для синхронизации чтения и записи[5].
- **BatchQueue** — Упрощенная реализация неблокирующей очереди с пакетными операциями. Ускорение достигается за счет операций с группами элементов[1].

¹⁶SegQueue. An unbounded multi-producer multi-consumer queue — <https://docs.rs/crossbeam/latest/crossbeam/queue/struct.SegQueue.html> (05.05.2025).

5. Особенности реализации

Интеграция дополнительной очереди в `tokio` потребовала модификации логики получения следующей задачи потоком-работником и логики переполнения локальных очередей. Дополнительная очередь размещена в разделяемой структуре и доступна всем потокам-работникам.

Для каждой очереди была реализована обертка, соответствующая трейту `FastQueue` для унификации операций. Она обеспечивает преобразование задач в элементы очереди и обратно, а также проверку на пустоту.

Исходный код реализации опубликован в GitHub-репозитории ¹⁷¹⁸.

Алгоритм работы потоков-работников изменен согласно построенной архитектуре. Поток-работник выполняет следующие шаги:

1. Проверка локальной очереди и выполнение задач из нее.
2. Если локальная очередь пуста, обращение к дополнительной очереди и извлечение пакета из 64 задач.
3. Если дополнительная очередь пуста, обращение к глобальной очереди, получение из нее пакета задач.

При переполнении локальной очереди потока-работника половина задач переносится в глобальную очередь. При достижении порога в глобальной очереди группа размером в 256 задач переносится в дополнительную очередь.

5.1. Реализация очередей

Для дополнительной очереди используются структуры на основе атомарных операций: `SegQueue`, `FAAArrayQueue` и `BatchQueue`. Ниже

¹⁷Интеграция дополнительной очереди `FAAArrayQueue` в `tokio` — <https://github.com/foreverjun/tokio/pull/2>.

¹⁸Интеграция дополнительной очереди `SegQueue` в `tokio` — <https://github.com/foreverjun/tokio/pull/1>.

описаны их алгоритмы и особенности управления памятью. Подробности реализации и исходный код доступны в репозитории ¹⁹.

5.1.1. SegQueue

SegQueue — это частично неблокирующая очередь, из библиотеки **crossbeam**. Она организована как связный список узлов, где каждый узел представляет собой массив фиксированного размера из 31 элемента. Узел включает в себя ячейки для хранения элементов с атомарными флагами состояния и атомарный указатель на следующий узел.

Алгоритм работы **SegQueue** основан на модифицированной версии **Michael-Scott Queue** [9], которая используется для организации связного списка узлов. Операции вставки и удаления элементов внутри узла выполняются с помощью атомарных операций **CAS**, устанавливающих флаги состояния ячеек. При переполнении текущего узла создается и присоединяется новый. Память управляется с помощью ручного освобождения неиспользуемых блоков.

5.1.2. FAAArrayQueue

FAAArrayQueue — это неблокирующая очередь, основанная на реализации из блога **ConcurrencyFreaks** [5]. Как показано на рисунке 5, очередь организована как связный список узлов, где каждый узел — массив фиксированного размера из 1024 элементов. Структура узла включает массив атомарных указателей на элементы, атомарные индексы головы и хвоста, а также указатель на следующий узел.

Алгоритм **FAAArrayQueue** использует операции **FAA** и **CAS**. Операции вставки и удаления используют **FAA** для получения индекса элемента в массиве из атомарного индекса головы или хвоста. Для операций с указателем на элемент используется **CAS**. Аналогично **SegQueue** использует для связывания узлов алгоритм **Michael-Scott Queue**.

Очередь реализована с двумя вариантами управления памятью:

¹⁹Реализации неблокирующих очередей для **tokio**. Репозиторий на GitHub — <https://github.com/foreverjun/queue-bench>.

Hazard Pointers [8], используя библиотеку `haphazard`²⁰, и Epoch-Based Reclamation [7], используя библиотеку `crossbeam-epoch`²¹. Оба механизма обеспечивают безопасное освобождение узлов, предотвращая доступ к освобожденной памяти.

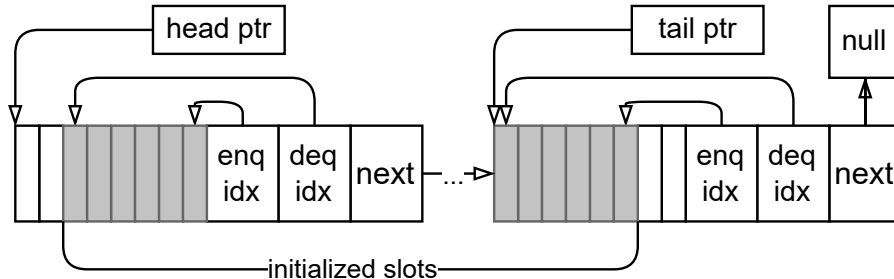


Рис. 5: Структура неблокирующей очереди `FAAArrayQueue`

5.1.3. BatchQueue

`BatchQueue` — неблокирующая очередь, основанная на алгоритме, представленном в статье «BQ: A Lock-Free Queue with Batching» [1]. Этот алгоритм расширяет очередь Michael-Scott поддержкой отложенных и пакетных операций. В реализованной на языке Rust версии предусмотрены одиночные операции, а также однородные пакетные операции, состоящие только из последовательностей вставок или удалений.

Очередь организована как односвязный список узлов, где каждый узел содержит элемент и указатель на следующий узел. Указатели на голову и хвост обновляются с помощью атомарных операций. Одиночные операции вставки и удаления реализованы аналогично Michael-Scott Queue. Однородные пакетные операции позволяют выполнять группу вставок или удалений с оптимизированной синхронизацией. Пакет удалений извлекает группу элементов, обновляя голову очереди с помощью CAS. Для выполнения пакета вставок используется структура `Announcement`, которая помещается в голову очереди, уведомляя другие потоки о пакетной вставке. После выполнения вставки

²⁰Haphazard: Hazard Pointers for Rust — <https://docs.rs/haphazard/0.1.8/haphazard/index.html> (05.05.2025).

²¹Crossbeam: Tools for Concurrent Programming in Rust — https://docs.rs/crossbeam-epoch/0.9.18/crossbeam_epoch/index.html (05.05.2025).

Announcement удаляется, а указатель на голову восстанавливается в прежнее положение.

Смешанные пакеты и отложенные операции, описанные в статье, не реализованы, так как не требовались для текущей архитектуры tokio. **Hazard Pointers** обеспечивают безопасную работу с памятью, но увеличивают накладные расходы по сравнению с оптимистической моделью памяти оригинальной BQ.

6. Экспериментальное исследование

В рамках экспериментов сравнивалась производительность реализованных очередей. По их результатам наиболее производительные решения были выбраны для интеграции в tokio. Далее производительность модифицированных версий tokio сравнивалась с производительностью оригинальной системы. Для сбора данных использовался сервер, характеристики которого описаны в разделе 3.

6.1. Исследовательские вопросы

- RQ1:** Какие из реализованных алгоритмов очередей показывают наилучший результат в изолированных тестах «производитель–потребитель»?
- RQ2:** Правда ли что модифицированный tokio с использованием дополнительной очереди лучше существующего решения для заданного сценария тестирования?

6.2. Измерение производительности очередей

Измерение производительности очередей проводилось в сценарии «производитель-потребитель», разработанном по аналогии со сценарием тестирования из статьи [11]. В этом сценарии потоки делятся на две группы: производители только добавляют элементы в очередь, а потребители только извлекают. Потоки оперируют пакетами размером 64 элемента. Основной метрикой производительности была выбрана пропускная способность, которая измеряется как количество элементов, извлеченных из очереди за одну секунду.

Пропускная способность измерялась для различных соотношений производителей и потребителей: 1:1 (равное количество производителей и потребителей), 1:2 (потребителей вдвое больше), 2:1 (производителей вдвое больше). Чтобы сделать нагрузку более реалистичной, между операциями имитировалась псевдослучайная работа. Работа сбалан-

сированно распределялась между производителями и потребителями: при большем числе производителей потребители выполняли меньше работы, и наоборот. Тест для каждой конфигурации выполнялся 20 раз. Длительность одного запуска — 3 секунды. По полученным данным рассчитывалась средняя пропускная способность для каждой конфигурации. Исходный код сценария измерения производительности доступен в репозитории ²².

Полученные результаты представлены в виде графиков, отражающих зависимость пропускной способности от количества потоков.

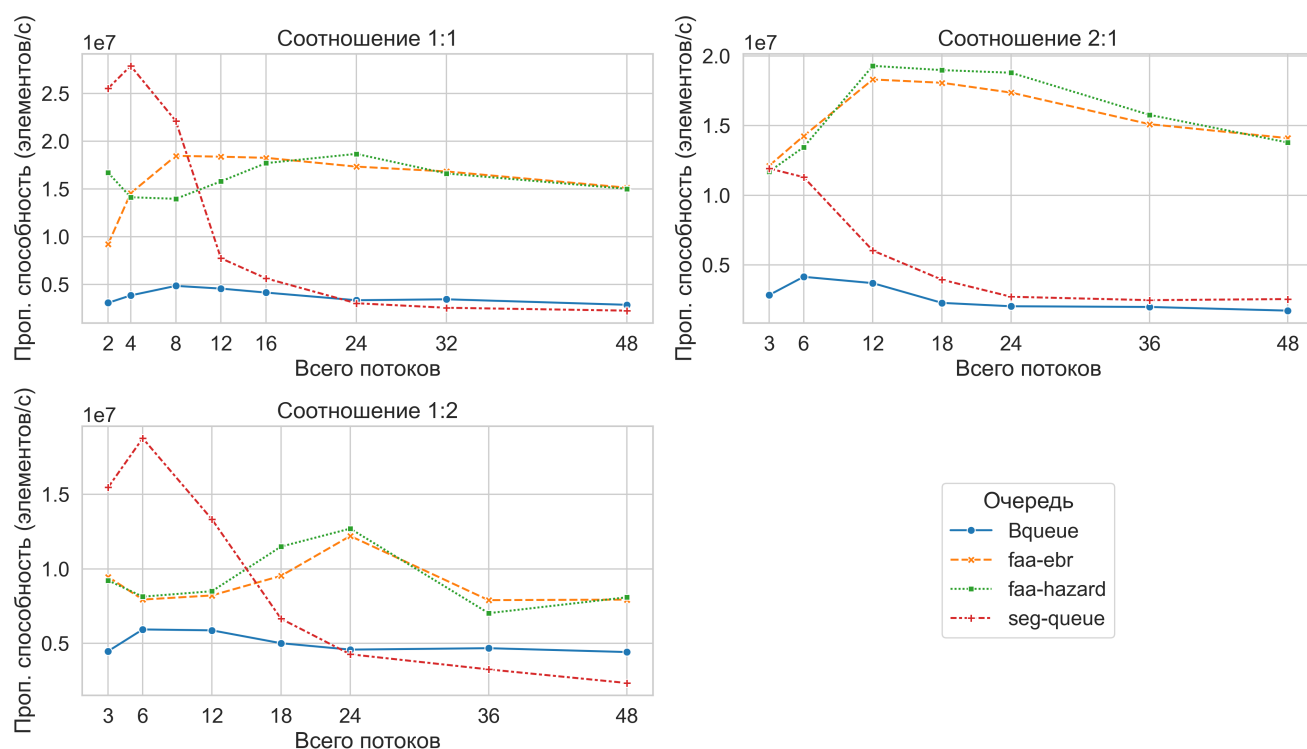


Рис. 6: Сравнение пропускной способности реализованных очередей и SeqQueue. Больше — лучше

На графике 6 видно, что по итогам сравнения наилучшую производительность при большом количестве потоков показала `FAAArrayQueue`. Заметим, что использование различных способов освобождения памяти (EBR и Hazard Pointers) практически не влияет на пропускную способность. Несмотря на то, что производительность `FAAArrayQueue` падает при большем числе потребителей [6], она все

²²Реализации неблокирующих очередей для tokio — <https://github.com/foreverjun/queue-bench>.

равно превосходит остальные очереди по пропускной способности. В свою очередь, `SegQueue` показывает высокую производительность при малом количестве потоков. Несмотря на то, что `BatchQueue` реализует пакетные операции, она все равно проигрывает в производительности `FAAArrayQueue`, возможно, из-за неполной реализации алгоритма, описанного в статье. Косвенное подтверждение тому, что `BatchQueue` может быть менее эффективной, чем `FAAArrayQueue`, можно увидеть в работе [4]. В результате анализа производительности для интеграции в `tokio` были выбраны `SegQueue` и `FAAArrayQueue` с реализацией `Epoch-Based Reclamation` из библиотеки `crossbeam`. Эта библиотека широко используется в сообществе Rust, благодаря своей активной поддержке, длительной истории разработки и высокой надежности.

6.3. Сравнение производительности модификаций `tokio`

Измерение производительности проводилось в условиях повышенной нагрузки на глобальную очередь, смоделированных по сценарию из раздела 3.2.

Собранные данные представлены в виде тепловых карт, показывающих отношение пропускной способности модифицированной системы к пропускной способности оригинального `tokio`. Ось X соответствует количеству задач-производителей, а ось Y — числу потоков-работников. Каждая ячейка отражает относительную производительность системы.

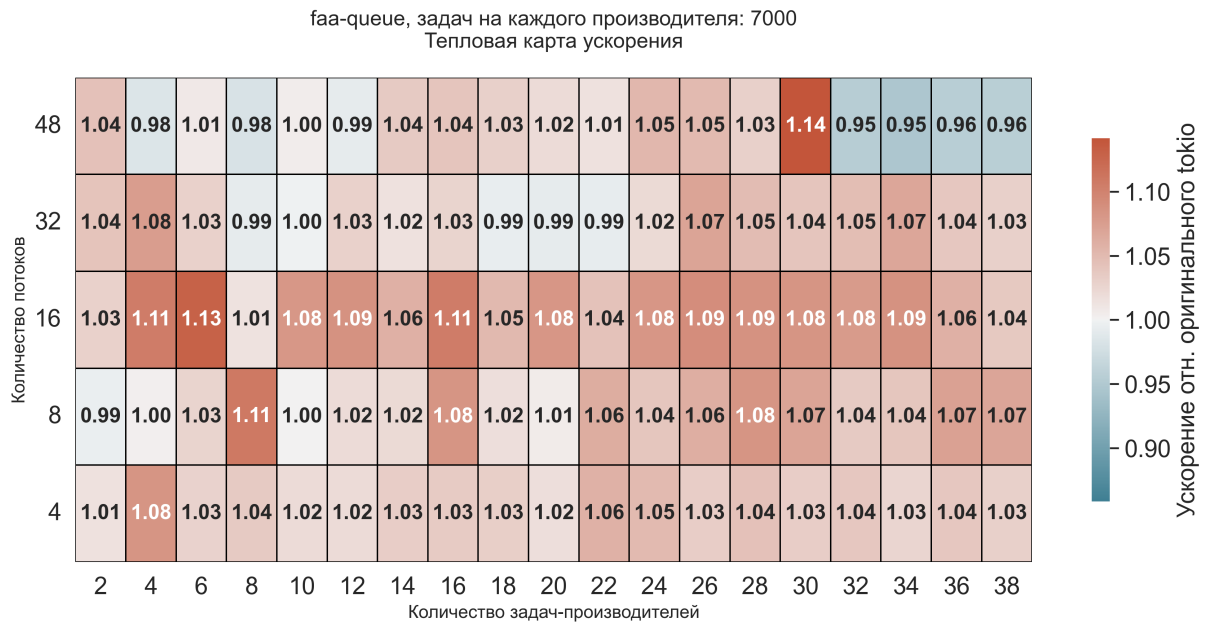


Рис. 7: Пропускная способность tokio с `FAAArrayQueue`, относительно оригинала

На рисунке 7 отображено улучшение производительности tokio с использованием `FAAArrayQueue` в качестве дополнительной очереди. В рамках теста каждая задача-производитель создает по 7000 пустых задач. Для большинства конфигураций модифицированная версия показывает лучшую производительность, что особенно заметно при 16 потоках-работниках.

На рисунке 8 видно, что использование `SegQueue` в качестве дополнительной очереди также приводит к ускорению, однако прирост производительности уступает модификации на основе `FAAArrayQueue`. Разница между двумя подходами сравнительно небольшая, несмотря на превосходство `FAAArrayQueue` в изолированных тестах очередей.

Это можно объяснить тем, что интеграция с tokio сглаживает преимущества `FAAArrayQueue`, выявленные в изолированных тестах очередей. На производительность системы поддержки периода исполнения влияет не только скорость работы дополнительной очереди, но и издержки на синхронизацию и управление доступом к другим разделяемым ресурсам.

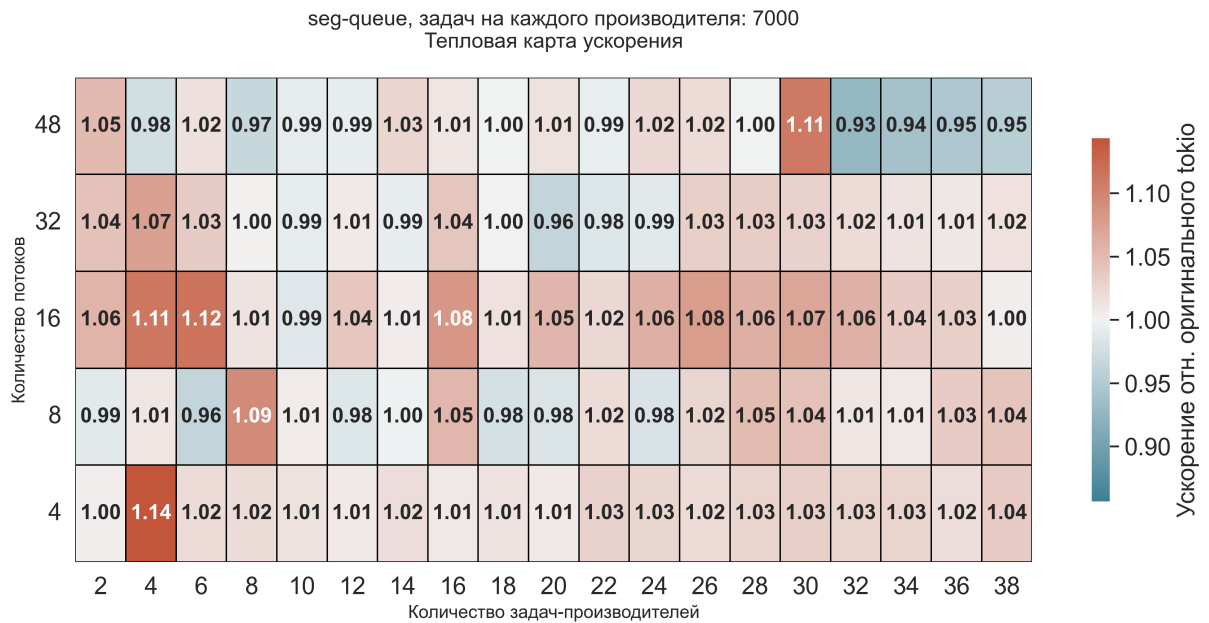


Рис. 8: Пропускная способность tokio с SegQueue, относительно оригинала

6.4. Выводы

RQ1. На основании изолированных тестов формата «производитель-потребитель» (рисунок 6) наиболее высокая пропускная способность достигается у `FAAArrayQueue`. При малом числе потоков в сценариях с соотношением производителей и потребителей 1:1 и 2:1 лучшую производительность показывает `SegQueue`.

RQ2. По сравнению с оригиналом модифицированный tokio показывает лучшие результаты пропускной способности. При использовании `FAAArrayQueue` достигается прирост пропускной способности до 13% (рисунок 7).

Заметим, что реализованная модификация tokio показывает наибольший прирост производительности при использовании 16 потоков-работников. При дальнейшем увеличении количества потоков-работников эффективность подхода снижается. В таких случаях целесообразнее использовать метод шардирования глобальной очереди, рассмотренный в работе Игоря Ерина «Оптимизация алгоритмов injected queue в рантайме Tokio».

Заключение

В рамках дипломной работы были достигнуты следующие результаты.

- Проведен анализ существующего алгоритма управления очередями задач и выделены ключевые метрики: пропускная способность и время выполнения.
- Разработан сценарий тестирования²³, соответствующий требованиям проекта TATLIN.BACKUP.
- Спроектирована модификация tokio с использованием дополнительной очереди задач, направленная на снижение количества обращений к глобальной очереди.
- Реализована модификация алгоритма управления очередями задач с использованием `FAAArrayQueue`²⁴²⁵, в качестве дополнительной очереди.
- Проведено сравнение модифицированной и оригинальной версии tokio в сценарии повышенной нагрузки на глобальную очередь. Наибольшего увеличения пропускной способности (до 13%) удалось добиться при использовании 16 потоков-работников.

Результаты этой работы были представлены на конференции «МАТ-МЕХ. НАУКА — 2025».

Асинхронные системы поддержки периода исполнения, такие как tokio, Kotlin coroutines и Goroutines, разрабатываются как универсальные решения. Однако на практике, как в нашем случае, встречаются

²³<https://github.com/IgorErin/tokiobench/pulls?q=is%3Apr+is%3Amerged+author%3Aforeverjun>

²⁴Реализации неблокирующих очередей для tokio. Бенчмарк очередей — <https://github.com/foreverjun/queue-bench>

²⁵Интеграция дополнительной очереди `FAAArrayQueue` в tokio — <https://github.com/foreverjun/tokio/pull/2>.

сценарии с нетипичной нагрузкой, в которых универсальные механизмы могут работать неэффективно. Это создает возможности оптимизаций, которые могут значительно улучшить производительность для определенных сценариев нагрузки.

Более того, упомянутые системы следуют единой модели. Это сходство делает их потенциально уязвимыми к схожим проблемам с производительностью. В случае возникновения подобных проблем предложенные в данной работе подходы могут быть адаптированы и использованы для повышения производительности в этих системах поддержки периода исполнения.

²⁵В процессе подготовки настоящей работы для редактуры и обеспечения единообразного стиля текста использовалась большая языковая модель OpenAI o4-mini.

Список литературы

- [1] BQ: A Lock-Free Queue with Batching / Gal Milman-Sela, Alex Kogan, Yossi Lev et al. // [ACM Trans. Parallel Comput.](#) — 2022. — . — Vol. 9, no. 1. — 49 p. — URL: <https://doi.org/10.1145/3512757>.
- [2] BWoS: Formally Verified Block-based Work Stealing for Parallel Processing / Jiawei Wang, Bohdan Trach, Ming Fu et al. // USENIX Symposium on Operating Systems Design and Implementation. — 2023. — URL: <https://api.semanticscholar.org/CorpusID:259858796>.
- [3] [Balanced Allocations over Efficient Queues: A Fast Relaxed FIFO Queue](#) / Kåre von Geijer, Philippas Tsigas, Elias Johansson, Sebastian Hermansson // Proceedings of the 30th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming. — PPOPP '25. — New York, NY, USA : Association for Computing Machinery, 2025. — P. 382–395. — URL: <https://doi.org/10.1145/3710848.3710892>.
- [4] [Boost Lock-free Queue and Stack with Batching](#) / Ao Li, Wenhai Li, Yuan Chen, Lingfeng Deng // Proceedings of the 30th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming. — PPOPP '25. — New York, NY, USA : Association for Computing Machinery, 2025. — P. 548–550. — URL: <https://doi.org/10.1145/3710848.3710880>.
- [5] FAAArrayQueue - MPMC lock-free queue. Pedro Ramalhete, Andreia Correia. — URL: <https://concurrencyfreaks.blogspot.com/2016/11/faaarrayqueue-mpmc-lock-free-queue-part.html> (дата обращения: 5 мая 2025 г.).
- [6] FAAArrayQueue not good in in consumer-dominated scenarios. — URL: <https://concurrencyfreaks.blogspot.com/2016/12/lcrq-and-faaarrayqueue-are-not-good-for.html> (дата обращения: 20 мая 2025 г.).

- [7] Practical lock-freedom : Rep. / University of Cambridge, Computer Laboratory ; Executor: Keir Fraser : 2004.
- [8] Michael Maged M. Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects // [IEEE Trans. Parallel Distrib. Syst.](#) — 2004. — . — Vol. 15, no. 6. — P. 491–504. — URL: <https://doi.org/10.1109/TPDS.2004.8>.
- [9] Michael Maged M., Scott Michael L. [Simple, fast, and practical non-blocking and blocking concurrent queue algorithms](#) // Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing. — PODC '96. — New York, NY, USA : Association for Computing Machinery, 1996. — P. 267–275. — URL: <https://doi.org/10.1145/248052.248106>.
- [10] Nookala Poornima, Chard Kyle, Raicu Ioan. X-OpenMP—eXtreme fine-grained tasking using lock-less work stealing // Future Generation Computer Systems. — 2024. — Vol. 159. — P. 444–458.
- [11] Romanov Raed, Koval Nikita. [The State-of-the-Art LCRQ Concurrent Queue Algorithm Does NOT Require CAS2](#) // Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming. — PPOPP '23. — New York, NY, USA : Association for Computing Machinery, 2023. — P. 14–26. — URL: <https://doi.org/10.1145/3572848.3577485>.