https://doi.org/10.25559/SITITO.020.202403.687-698
УДК 519.683
Оригинальная статья

Исследование эффективности потокобезопасных очередей на устройствах с асимметричным набором ядер

М. Д. Молотков
ФГАОУ ВО «Московский физико-технический институт (национальный исследовательский университет)», г. Долгопрудный, Российская Федерация
Адрес: 141701, Российская Федерация, Московская область, г. Долгопрудный, Институтский переулок, д. 9
molotkov.md@phystech.edu

Аннотация

В рамках статьи было проведено исследование зависимости пропускной способности потокобезопасных очередей от количества push- и pop-потоков при различных сопровождающих нагрузках для каждого пропускаемого элемента. В результате поиска подходов к созданию потокобезопасных очередей для исследования были выбраны *single lock queue*, *two lock queue* и *lock free queue*, для каждого из них была проведена предварительная оценка преимуществ и недостатков, которые могут повлиять на итоговую производительность. Очереди были реализованы с учетом специфики работы с устройствами с асимметричным набором ядер и с минимально необходимыми накладными расходами по времени CPU. Для исследования пропускной способности были реализованы бенчмарки, которые отражают сценарии использования очередей одинаковым количеством push- и pop-потоков при различной полезной нагрузке. Параметризация нагрузки реализована при помощи множителя вычислений. Этот подход позволяет учесть различие ядер одного устройства в мощности. Измерения проводились на двух устройствах: P50pro и Mate60pro. Такой выбор обусловлен тем, что P50pro имеет меньше ядер, чем Mate60pro, однако частота ядер у P50pro выше. В результате была получена эмпирическая зависимость эффективности работы очередей, из которой можно сделать выводы об области применимости той или иной очереди на разных устройствах. Для устройства с большим числом ядер *lock free queue* очередь либо выигрывает, либо эквивалентна *two lock queue* очереди при любой нагрузке, кроме нулевой. На устройстве с небольшим числом ядер *two lock queue* очередь выигрывает при низких нагрузках, *lock free queue* выигрывает при высоких. Эта зависимость позволяет оценить наиболее эффективное количество ядер для работы с очередью при известных нагрузках на каждый пропускаемый элемент, а значит, позволяет получить максимальную производительность на одно ядро.

Ключевые слова: многопоточность, алгоритм

Конфликт интересов: автор заявляет об отсутствии конфликта интересов.

Для цитирования: Молотков М. Д. Исследование эффективности потокобезопасных очередей на устройствах с асимметричным набором ядер // Современные информационные технологии и ИТ-образование. 2024. Т. 20, № 3. С. 687-698. https://doi.org/10.25559/SITITO.020.202403.687-698

Original article

Effectiveness of Thread-Safe Queues on Devices with an Asymmetric Set of Cores

M. D. Molotkov
Moscow Institute of Physics and Technology (National Research University), Dolgoprudny, Russian Federation
Address: 9 Institutskiy per., Dolgoprudny 141701, Moscow Region, Russian Federation
molotkov.md@phystech.edu

Abstract

In the article we studied the dependence of the throughput of thread-safe queues on the number of `push` and `pop` threads at various loads for each element. We have chosen and implemented *single lock queue, two lock queue* and *lock free queue* for the research. To study the throughput, we have implemented benchmarks that reflect the scenarios of using queues with the same number of `push` and `pop` threads with different loads. Load parameterization is calculation multiplier, which allows you to take into account the difference in power between the cores of one device. The measurements were carried out on two devices: P50pro and Mate60pro. They were chosen because the P50pro has fewer cores than the Mate60pro, but they are more powerful. Finally, we have obtained an empirical dependence of the efficiency of queues. For a device with a large number of cores, the *lock free queue* either wins or is equivalent to the *two lock queue* at any load other than zero. On the device with a small number of cores, *two lock queue* wins with low loads, *lock free queue* wins with high loads. This dependency allows us to estimate the optimal number of cores to use when working with a queue at known load levels for each element. This in turn allows for maximum performance per core.

Keywords: multithreading, algorithm

Conflict of interests: The author declares no conflict of interests.

For citation: Molotkov M.D. Effectiveness of Thread-Safe Queues on Devices with an Asymmetric Set of Cores. *Modern Information Technologies and IT-Education.* 2024;20(3):687-698. https://doi.org/10.25559/SITITO.020.202403.687-698

1. Введение

Каждый из нас в повседневной жизни пользуется мобильными устройствами. За последнее десятилетие, они превратились в компактные и при этом мощные вычислительные машины. Одновременно с этим и ожидания от работы таких устройств возросли. Теперь мы ожидаем от них эффективной работы с тяжелыми приложениями. Ярким примером таких приложений являются игры. Одновременно с этим мы хотим качественной работы фоновых сервисов и служб самого телефона. Все это означает, что разработчикам необходимо еще более эффективно использовать имеющиеся ресурсы телефона для того, чтобы оправдать ожидания пользователей.

Зачастую современные мобильные устройства имеют большое количество физических ядер, но они имеют разную рабочую частоту. Вдобавок, у каждого ядра есть множество возможных рабочих частот. ОС может менять рабочую частоту на свое усмотрение. Это позволяет более эффективно расходовать заряд батареи без критического уменьшения производительности телефона в целом [1]. Однако, для разработчика это может стать проблемой. Так, время работы программы может сильно варьироваться в зависимости от ядра, на которое попал поток выполнения, и от частоты этого ядра. Для того чтобы более эффективно использовать железо устройства, разработчики прибегают к многопоточности и распараллеливанию своих программ [2, 3]. Однако при такой асимметричной конфигурации ядер сложно оценить, какой подход к использованию многопоточности будет наиболее эффективным.

В данной статье мы рассмотрим несколько подходов к реализации потокозащищенной очереди. Это стандартный пример компоненты, при помощи которой происходит распараллеливание программ. Мы опишем и реализуем *single lock queue, two lock queue* и *lock free queue* и попробуем оценить их пропускную способность для определенных сценариев. Далее мы составим и реализуем бенчмарки, которые будут отражать эффективность работы очередей при разном уровне конкуренции. Конкуренция будет варьироваться объемом нагрузки после каждого изъятия элемента из очереди. В итоге, мы получим картину эффективности различных подходов в зависимости от уровня конкуренции.

2. Цель исследования

Цель данного исследования: изучение эффективности работы потокобезопасных очередей на устройствах с асимметричным набором ядер. Задачи данного исследования: поиск и изучение различных видов потокобезопасных очередей; реализация этих очередей; исследование и получение эмпирической зависимости эффективности работы очереди от числа `push`- и `pop`-потоков при различном уровне конкуренции.

3. Теоретический анализ

3.1 Потокобезопасные алгоритмы

В данном исследовании мы будем рассматривать ситуацию работы нескольких потоков исполнения в одной общей области памяти. В таком случае при реализации потокобезопасных алгоритмов и структур данных (в том числе и очередей) используются следующие подходы: *lock-based, lock-free* и *wait-free* алгоритмы.

Lock-based алгоритмы предполагают обеспечение доступа к памяти со стороны только одного потока одновременно. Так, один поток выполняет алгоритм, а остальные потоки ожидают, пока первый покинет «критическую секцию». Основными инструментами в таких алгоритмах являются мьютексы (их еще называют локи) и условные переменные. К несомненным плюсам таких алгоритмов можно отнести простоту реализации, так как в «наивном» случае достаточно использовать уже готовый однопоточный алгоритм и только защищать все его методы одним единственным мьютексом. А если для выполнения операции необходимо определенное состояние структуры данных (например, для `pop`-операции необходимо, чтобы очередь содержала хотя бы один элемент), можно воспользоваться условными переменными. Они позволяют ожидать одному или нескольким потокам сигнал о пробуждении от другого потока. Главным же минусом такого подхода является низкая эффективность при росте конкуренции. Внутри мьютексы используют следующий механизм: сначала поток в цикле ожидает освобождения критической секции; по прошествии некоторого числа циклов поток уходит в системное ожидание при помощи специального системного вызова. Такая процедура очень долгая по сравнению с ожиданием в цикле. Так, высокая конкуренция приводит к тому, что потоки уходят в системное ожидание, тем самым дополнительно замедляют систему. В дополнение ко всему, что было сказано ранее, необходимо рассмотреть rw-мьютексы. Они имеют две пары операций: Lock() и Unlock(). WLock()/WUnlock() гарантируют единственность доступа, как и в простых мьютексах, но RLock()/RUnlock позволяют нескольким потокам одновременно обращаться к внутренним структурам, пока WLock() свободен. RLock()/RUnlock предназначены только для чтения данных из внутренней структуры, и потоки, использующие этот «лок», не должны менять состояние системы. Этот инструмент позволяет более эффективно работать с данными, для которых свойственно частое чтение и редкое изменение.

Lock-free подход формально означает следующее: для *lock-free* системы, в которой выполняются N потоков, гарантируется системный прогресс хотя бы одного потока вне зависимость от состояния других потоков. Такому определению не соответствуют алгоритмы, использующие мьютекс. В случае если поток, владеющий мьютексом, остановится, остальные потоки будут вынуждены дожидаться, пока тот не проснется. *Lock-free* алгоритмы используют атомарные операции для синхронизации [4, 5]. Главным инструментов можно назвать операцию `compare_and_swap` (далее будем называть ее `cas`), которая сравнивает значение переменной с экземпляром и, в случае совпадения, меняет значение переменной на другое. Такая операция позволяет синхронизировать сколько угодно потоков. При помощи атомарных операций мы можем синхронизировать доступ к критической секции. Однако для этого зачастую необходимо больше, чем один `cas`. Поток, успешно исполнивший первую подмену, может остановиться перед следующей. Поток, который провалил операцию `cas`, должен «помочь» другому потоку завершить его алгоритм. Это необходимо, так как в противном случае промежуточное состояние системы может блокировать работу других потоков. Слож-

ность реализации и отладки *lock-free* алгоритмов заключается как раз во «вспомогательных» путях исполнения. Главным плюсом таких алгоритмов является хорошая масштабируемость при росте количества потоков. Так, потоки будут конкурировать друг с другом, но гарантия *lock-free* говорит нам о том, что они же будут помогать друг другу, чтобы система изменяла свое состояние. Главным минусом же является низкая эффективность работы при слабой конкуренции. Она связана с реализацией атомарных операций в архитектуре процессора [6, 7]. Так, дополнительные расходы (например, на алгоритм когерентности кешей) могут привести к тому, что мьютекс, как инструмент синхронизации, будет более эффективным при определенных конфигурациях потоков[1].

Стоит отметить, что важно указывать, для какой конфигурации потоков создан алгоритм. Например, *lock-free* алгоритм может быть рассчитан только на 2 потока: один push-поток и один pop-поток. Такой алгоритм будет намного лучше работать в заданных границах, чем аналоги, однако непригоден для работы с более чем двумя потоками. В рамках нашего исследования мы увидим только *single producer single consumer lock-free queue* [8] и *multi producer multi consumer lock-free queue* [9].

Wait-free подход требует более жестких гарантий, чем *lock-free*: для *wait-free* системы, в которой выполняется N потоков, гарантируется, что алгоритм выполняется за определенное число шагов, независимо от других потоков. Такой подход применяется для небольшого круга алгоритмов для работы с очень большим числом потоков [10-12]. Реализация очереди с данным подходом в этой работе рассмотрена не будет.

Отметим, что допускаются смешения подходов. Например, пусть у нас есть *lock-free* структура данных, рассчитанная на 2 потока, которые выполняют разные методы этой структуры. Можно обернуть методы структуры в два разных мьютекса, которые будут синхронизировать доступ множества потоков независимо друг от друга [13]. Строго говоря, этот подход все еще является *lock-based* алгоритмом, однако он может быть намного эффективнее наивных реализаций и при этом оставаться относительно простым для реализации и отладки.

3.2 Устройства с асимметричным набором ядер

Асимметричным набором ядер мы назовем конфигурацию процессора, которая подразумевает наличие ядер разной мощности, предназначенных для вычислений одного рода. Так, системы с сопроцессорами, предназначенными только для вычислений с плавающей точкой, не входят в данное определение. Зачастую асимметричный набор ядер встречается в мобильных устройствах. Такое устройство при распределении вычислений на разные ядра более эффективно потребляет заряд аккумулятора. Для телефонов свойственно наличие от 1 до 2 быстрых ядер, от 3 до 4 средних ядер и от 4 до 8 слабых ядер. Еще одной особенностью процессоров мобильных устройств является большой набор возможных частот для каждого ядра. Так, при перегреве устройства или малом заряде батареи ОС принудительно снизит частоту ядер, чтобы снизить энергопотребление и теплогенерацию. С другой стороны, в случае длительной нагрузки ОС может увеличить частоту процессора для повышения производительности. Подход с использованием ядер с разными частотами называется *“big.LITTLE Processing Technology”*[2] [1], [14].

Таким образом, можно сказать, что заранее предсказать, какой из подходов будет более эффективен, не представляется возможным. Если рассматривать подход, подразумевающий блокировку, то «слабые ядра» / «ядра с пониженными частотами» могут проводить много времени в критической секции, тем самым замедляя работу других потоков. Для *lock-free* подхода существует проблема того, что потоки на медленных ядрах будут часто «проигрывать» конкуренцию у других потоков, и выполнять бесполезную работу, что может привести к снижению частот из-за сильного энергопотребления.

3.3 Объекты исследования

В данном исследовании мы рассмотрим 3 потокобезопасные очереди: *single lock queue*, *two lock queue* и *lock free queue*. Каждая из них имеет свою особенность, но при этом они все отвечают следующим критериям:

- размер очереди не является фиксированным, и память для хранения элементов выделяется из кучи. Один такой фрагмент мы назовем узлом. Один узел должен позволять хранить некоторое заранее известное количество элементов;
- очередь должна быть потокобезопасной относительно любого заранее известного количества push- и pop-потоков;
- требуется реализовать следующие методы: Push, Pop, Size, IsEmpty.

Все вышеперечисленные очереди были написаны на языке C++, а значит, здесь мы также рассмотрим то, как происходит управление памятью. В случае с *lock free queue* это наиболее важно [15]. Теперь мы можем перейти к рассмотрению очередей и их реализаций.

Single lock queue – наивная реализация потокобезопасной очереди с использованием одного мьютекса. В такой реализации используется простая очередь (`std::queue`), методы которой обрамлены во взятие/отпускание мьютекса. Также для реализации корректного ожидания новых элементов в методе Pop будет использоваться условная переменная. Далее представлен псевдокод с определением полей и методов *single lock queue*.

```
class SingleLockQueue {
    lock lock_;
    std::queue queue_;
    std::conditional_variable cond_var;
public:
    void Push(uintptr_t val) {
        push_.Lock();
        queue_.push(val);
        cond_var.signal_one();
        push_.Unlock();
```

[1] McKenney P. E. Memory Barriers: a Hardware View for Software Hackers [Электронный ресурс] // RainDrop Laboratories, 2024. URL: http://www.rdrop.com/users/paulmck/scalability/paper/whymb.2010.06.07c.pdf (дата обращения: 03.06.2024).

[2] Stevens A. Introduction to AMBA 4 ACE and big. LITTLE Processing Technology [Электронный ресурс] // ARM White Paper, CoreLink Intelligent System IP by ARM, 2011. 15 p. URL: https://picture.iczhiku.com/resource/paper/WyIderGepoDsTXXb.pdf (дата обращения: 03.06.2024).

```
    }
    uintptr_t Pop() {
        lock_.Lock();
        while (queue.is_empty()) {
            cond_var.wait(&lock_);
        }
        auto val = queue_.pop(val);
        lock_.Unlock();
        return val;
    }
    bool IsEmplty() {
        lock_.Lock();
        bool res = queue_.IsEmpty();
        lock_.Unlock();
        return res;
    }
}
```

Видно, что в методе Pop происходит проверка на наличие элементов в очереди, и если их нет, то поток уйдет в ожидание и процессор переключится на выполнение других задач, пока из метода Push не придет сигнал, который пробудит спящий поток. Методы Size и IsEmpty тоже обрамлены мьютексами.

Имея алгоритм, мы можем сказать, что он достаточно легок в реализации и не сильно усложняет отладку программы при использовании. Использование мьютекса позволяет явно остановить все потоки и проверить полное состояние очереди, если это окажется необходимым. Однако алгоритм заставляет конкурировать за один мьютекс все потоки, что сильно замедляет программу, особенно в случае разного количества push и pop потоков. Вдобавок алгоритм зависим от реализации мьютекса и планировщика в операционной системе. Такая очередь чаще всего используется для редкой передачи элементов, которые необходимо ждать.

Контроль памяти в такой очереди наиболее простой. Удаление происходит под мьютексом в случае, если это необходимо и эта часть алгоритма уже реализована в используемой очереди из стандартной библиотеки.

Two lock queue – очередь, представляющая собой надстройку над *single producer single consumer lock-free queue* (далее будем называть ее «внутренней» очередью) [8]. Так, для синхронизации доступа к «внутренней» очереди, для push-потоков и pop-потоков используются два разных мьютекса. Ниже приведен элемент реализации *double lock queue*.

```
class TwoLockQueue {
    lock push_lock_, pop_lock_;
    InternalQueue queue_;

public:
    void Push(uintptr_t val) {
        push_lock_.Lock();
        queue_.Push(val);
        push_lock_.Unlock();
    }
    uintptr_t Pop() {
        pop_lock_.Lock();
        auto val = queue_.Pop(val);
        pop_lock_.Unlock();
        return val;
    }
    bool IsEmpty() {
        return queue_.IsEmpty();
    }
}
```

Теперь рассмотрим «внутреннюю очередь». Она представляет собой односвязный список из «узлов», в каждый из которых может поместиться некоторое количество элементов. В структуре данных хранятся: указатели на первый и последний узлы, номер следующего элемента на следующую pop-операцию и номер нового элемента после push-операции. К каждому из этих полей может быть только атомарный доступ. Размер такого «узла» следует выбирать исходя из размера кэш-блока [16]. Такая структура является общей для всех *lock-free* очередей на узлах, однако она может дополняться в зависимости от требований к очереди. В нашем случае необходимо синхронизировать один push-поток *(producer)* и один pop-поток *(consumer)*. Для этого будет достаточно пары операций load/store, что хорошо, так как операции cas достаточно времязатратны [17].

Ниже приведен элемент реализации внутренней очереди. Мы опустили метод Pop, так как он является симметричным методом относительно Push-а.

```
class InternalQueue {
    struct Node {
       uintptr_t buffer_[NODE_SIZE];
       Node *next_ = nullptr;
    }

    std::atomic<Node*> head_ = nullptr, tail_ = nullptr;
    std::atomic<size_t> push_index_ = 1U, pop_index_ = 1U;
public:
    InternalQueue {head_ = tail_ = new Node()}
    ~InternalQueue { delete head_ }
    void Push(uintptr_t val) {
        auto push_index = push_index_.load(std::memory_
order_acquire);
        auto tail = tail_.load(std::memory_order_relaxed);
        auto node_push_index = push_index % NODE_SIZE;
        if (node_push_index == 0) {
            auto new_node = new Node();
            new_node->buffer[0] = val;
            tail->next = new_node;
            tail_.store(new_node, std::memory_order_relaxed);
        } else {
            tail->buffer[node_push_index] = val;
        }
        push_index_.store(push_index + 1, std::memory_
order_release);
    }
    bool IsEmpty() {
        auto push_index = push_index_.load(std::memory_
order_acquire);
        auto pop_index = pop_index_.load(std::memory_
order_acquire);
        return push_index == pop_index;
    }
}
```

Из плюсов такого алгоритма можно отметить следующее:
• методы Size и IsEmpty вообще не требуют использования мьютексов, ведь использование этих методов не изменяют состояние «внутренней» очереди;
• алгоритм достаточно простой для реализации и за счет составной структуры может быть хорошо протестирован;
• такой алгоритм лишен недостатка *single lock queue* в случае разного числа push- и pop-потоков, так как потоки одной природы синхронизируются сначала между собой, а значит, нет ситуаций, когда большое количество pop-потоков не дают push-потоку произвести свою операцию;
• контроль памяти достаточно прост, узел должен быть создан push-потоком при необходимости нового узла, и удален pop-потоком, если тот достает последний элемент из узла.
В то же время этому алгоритму свойственны следующие проблемы:
• в таком алгоритме реализация системного ожидания элементов в методе Pop может сильно замедлить работу метода Push, так как для сигнала pop-потока необходимо брать pop-мьютекс;
• алгоритмы зависят от реализации мьютекса и планировщика в операционной системе;
• необходимо учитывать проблемы, такие как *false sharing*, которые связаны с устройством аппаратуры [18].
Lock free queue – очередь, отвечающая требованиям к *lock-free* структурам данных. Эта структура данных состоит из тех же элементов, что и *single producer single consumer lock-free queue*. Однако для реализации *lock-free* для большого числа push- и pop-потоков необходима синхронизация с использованием операции cas. Алгоритм схож с реализацией каналов в Golang, но имеет свои особенности [9].
Основным методом реализации *lock-free* структур данных является реализация «основного» и «вспомогательных» путей исполнения алгоритма. Так, изначально поток А идет по «основному» пути, который в случае успешного выполнения всех подмен полей приведет к исполнению метода. Однако, если хоть одна подмена не проходит, то поток А точно знает, что какой-то другой поток Б уже смог успешно подменить то же поле. Следовательно, поток А может помочь в выполнении алгоритма другому потоку, так как поток Б мог прерваться и остановить таким образом изменение системы. Для этого существуют «вспомогательные» пути исполнения. Это является главной сложностью реализации алгоритма. Отладка всегда затруднительна, так как необходимо продумывать больше количество потенциальных состояний системы. При наличии ошибок в коде их поиск будет занимать долгое время.
Отдельное внимание следует уделить управлению памятью. Так как много потоков могут владеть одним узлом, невозможно удалить его сразу, как только он становится ненужным. Необходимо реализовывать дополнительную структуру, которая контролировала бы память. Существует множество подходов к контролю памяти в таких случаях[3] [19-22]. Здесь мы прибегнем к *hazard pointers*. Этот подход позволяет синхронизировать работу с памятью, при этом из атомарных операций потребуются только load и store. С другой стороны, такой подход требует выделение памяти под хранение указателей для каждого потока, но в данном случае нам необходима скорость исполнения при приемлемом потреблении памяти.
Ниже приведен элемент реализации *lock-free* очереди. Из методов мы опустили метод Pop, так как он является симметричным методом относительно Push-а.

```
class LockFreeQueue {
    enum class Status: size_t {EMPTY, SUCCESS, FAIL };
    struct Node {
       Node(size_t id): id_(id);

       size_t id_ = 0;
       std::atomic<STATUS> statuses_[NODE_SIZE];
       std::atomic<uintptr_t> buffer_[NODE_SIZE];
       std::atomic<Node *> next_ = nullptr;
    }

    std::atomic<Node*> head_ = nullptr, tail_ = nullptr;
    std::atomic<size_t> push_index_ = 1U, pop_index_ = 1U;
    hazard_ptr_storage hp_;
public:
    LockFreeQueue {head_ = tail_ = new Node()}
    ~LockFreeQueue {delete head_}
    void Push(uintptr_t val) {
        while (true) {
            auto push_index = push_index_.
load(std::memory_order_acquire);
            auto pop_index = pop_index_.load(std::memory_
order_acquire);
            if (pop_index > push_index) {
                continue;
            }
            if (push_index == pop_index) {
                if (ConcurrentAddValueToQueue(push_index,
val)) {
                    return;
                }
                continue;
            }
            QueueNode *head = hp_.Load(&head_);
            auto buffer_pop_index = pop_index % NODE_SIZE;
            auto pop_node_id = pop_index / NODE_SIZE;
            if (head->id > pop_node_id) {
                continue;
            }
            if (head->id < pop_node_id) {
                QueueNode *node = hp_.Load(&head->next);
                if (head_.cas(head, node, std::memory_
order_acq_rel)) {
                    hp_.Retire(head);
                }
                continue;
            }
            if (ConcurrentGetPopIndexStatus(head, buffer_
pop_index) == Status::FAIL) {
```

[3] Fraser K. Practical lock-freedom. Technical Report No. UCAM-CL-TR-579. University of Cambridge, Computer Laboratory, 2004. 116 p. [Электронный ресурс]. URL: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-579.pdf (дата обращения: 03.06.2024).

```
                pop_index_.cas(pop_index, pop_index + 1,
std::memory_order_acq_rel);
                continue;
            }
            if (ConcurrentAddValueToQueue(push_index,
val)) {
                return;
            }
        }
    }
    bool ConcurrentAddValueToQueue(size_t push_index,
uintptr_t val);
    bool ConcurrentMoveTail(Node *tail, size_t push_index,
uintptr_t val);
    bool ConcurrentWriteValueInNode(QueueNode *tail,
size_t push_index, T &val);

    bool IsEmpty() {
            while (true) {
            auto push_index = push_index_.
load(std::memory_order_acquire);
            auto pop_index = pop_index_.load(std::memory_
order_acquire);
            if (push_index < pop_index) {
                continue;
            }
            return push_index == pop_index;
        }
    }
}
```

Как видно, реализация даже одного метода крайне громоздка и требует введения новых вспомогательных функций. В самом методе Push мы уже можем видеть реализацию концепции «основного» и «вспомогательных» путей исполнения. Основным мы считаем выполнение метода ConcurrentAddValueToQueue, так как он подразумевает непосредственное добавление элемента в очередь. Остальные if секции предназначены для помощи Pop методу. Так, может происходить подмена узла головы или сдвиг pop-индекса. Метод ConcurrentAddValueToQueue так же будет содержать «вспомогательные» пути по вставке элемента другого потока. В коде метод compare_exchange_strong был заменен методом cas в угоду чистоты кода.

```
    bool ConcurrentAddValueToQueue(size_t push_index,
uintptr_t &val)
    {
        QueueNode *tail = HazardPtrScope::Load(&tail_);
        auto buffer_push_index = push_index % NODE_SIZE;
        auto push_node_id = push_index / NODE_SIZE;
        if (tail->id > push_node_id) {
            return false;
        }
        if (tail->id == push_node_id && buffer_push_index
== 0) {
            push_index_.cas(push_index, push_index + 1,
std::memory_order_acq_rel);
            push_index += 1;
            buffer_push_index = 1;
        }
        if (buffer_push_index != 0) {
            return ConcurrentWriteValueInNode(tail, push_
index, val);
        }
        return ConcurrentMoveTail(tail, push_index, val);
    }

    bool ConcurrentWriteValueInNode(Node *tail, size_t
push_index, uintptr_t val)
    {
        if (!push_index_.cas(push_index, push_index + 1,
std::memory_order_acq_rel)) {
            return false;
        }
        auto buffer_push_index = push_index % NODE_SIZE;
        tail->buffer[buffer_push_index] = val;
        if (!TryMarkAsGreen(tail, buffer_push_index)) {
            return false;
        }
        return true;
    }

    bool TryMarkAsGreen(Node *node, size_t index)
    {
        auto current_status = Status::NONE;
        auto &st = node->statuses[index]
        return st.cas(current_status, Status::SUCCESS,
std::memory_order_acq_rel);
    }

    bool ConcurrentMoveTail(Node *tail, size_t push_index,
uintptr_t val)
    {
        while (true) {
            Node *node = hp_.Load(&tail->next);
            if (node != nullptr) {
                tail_.cas(tail, node, std::memory_order_
acq_rel);
                push_index_.cas(push_index, push_index +
1, std::memory_order_acq_rel);
                return false;
            }
            node = new Node(tail->id + 1);
            node->buffer[0U] = val;
            node->statuses[0U] = Status::SUCCESS;
            hp_.Save(node);
            Node *null_node = nullptr;
            if (tail->next.cas(null_node, node,
std::memory_order_acq_rel)) {
                tail_.cas(tail, node, std::memory_order_
acq_rel);
                push_index_.cas(push_index, push_index +
1, std::memory_order_acq_rel);
                return true;
            }
            delete node;
        }
    }
```

Выше мы видим, что «основной» путь исполнения рассматривает два случая: когда элемент можно вставить в уже существующий узел и когда нужно создать новый узел. Каждый из этих случаев стоит рассмотреть по отдельности.

Начнем с метода ConcurrentWriteValueInNode. Здесь синхронизация между потоками происходит поэтапно: сначала синхронизируются push потоки, пытаясь сдвинуть push index, а затем тот поток, который смог сдвинуть индекс, синхронизируется с pop-потоком, пытаясь изменить статут ячейки с NONE на SUCCESS. Если у него это получается, элемент считается вставленным. Обратный случай означает, что pop-поток пометил ячейку как FAIL и что весь процесс нужно начинать сначала. Такой подход, когда pop-поток может забраковать ячейку, является частью реализации гарантии *lock-free* и позволяет избежать ожидания вставки элемента от одного push-потока, который выиграл всю конкуренцию, но остановился на моменте подмены. Но такое решение имеет проблему. Возможен сценарий, когда push-поток пытается вставить элемент в очередную ячейку, но pop-поток каждый раз успевает забраковать эту ячейку. Это *livelock*, и он усложняет оценку времени работы программы. Решение этой проблемы заключается в правильной вставке нового узла.

Перейдем к методу ConcurrentMoveTail. В нем поток создает узел, сразу в него записывает свой элемент и только после этого пытается вставить этот узел в систему. Так, pop-поток не сможет забраковать первую ячейку. Сценарий выше может развиваться только пока свободные ячейки в узле не закончатся. Сам метод внутри себя содержит «вспомогательные» пути, которые позволяют помочь другому потоку вставить узел с элементом.

Проблема алгоритма в целом заключается в создании нового узла. Внутри аллокатора используется глобальный мьютекс, поэтому такой алгоритм не может считаться *lock-free* в полном смысле слова. Решением может стать реализация *lock-free* аллокатора, однако он все равно будет содержать *lock-based* функции для выделения памяти [23]. Некоторые разработчики обходят такое ограничение создавая *lock-free* очереди с фиксированным размером, но в таком алгоритме неизбежно ожидание *producer*-ов, пока место не освободится [24]. В нашем случаем мы примем такую не *lock-free* составляющую, так как количество потоков будет много меньше, чем количество ячеек в одном узле (что, вообще говоря, является наиболее частым случаем в практике), а значит, создание новой ячейки должно быть достаточно редким событием и не сильно влиять на производительность.

Теперь мы можем поговорить о преимуществах и недостатках такого алгоритма. Несомненными плюсами *lock-free* очередей являются:

- высокая эффективность при высокой конкуренции. Каждый поток стремится подвинуть систему вперед, а не ждет;
- алгоритм наименее зависим от планировщика задач в операционной системе. Если планировщик остановит поток, который производит операцию, другие потоки смогут либо самостоятельно произвести ее, либо помочь потоку выполнить операцию, пока он спит и продолжить работу.

Из минусов следует отметить:

- сложность реализации и отладки, в особенности сложность работы с памятью;
- так как CAS реализуется по-разному в различных архитектурах, а значит, эффективность очереди может сильно зависеть от нее [25];
- необходимо учитывать проблемы, такие как *false sharing*, которые связаны с аппаратурой [4].

Во всех очередях выше видно, что их эффективность так или иначе зависит от конкуренции. Однако нет выражения, которое бы позволяло оценить, с каким «значением» конкуренции та или иная очередь будет эффективнее.

4. Методика эксперимента

Теперь нам необходимо определить то, каким образом мы будем изучать эти очереди. Любое использование потокобезопасной очереди необходимо для того, чтобы распределять вычисления. Тогда эффективность очередей мы определяем через их пропускную способность, то есть количество элементов, которое может пропустить через себя очередь за единицу времени. Нам интересно получить графики изменения пропускной способности в зависимости от внешних параметров. Далее мы определим эти параметры, и относительно них выберем устройства и построим бенчмарки для замеров.

Пропускная способность может зависеть как от количества потоков, так и от частоты работы ядер. Для того чтобы рассмотреть разные случаи, возьмем 2 устройства. В одном из них физических ядер будет меньше, но частота работы всех «видов» ядер будет выше, а в другом, напротив, физических ядер будет больше, но частота их работы будет ниже. Пропускная способность очереди напрямую зависит от того, какая нагрузка приходится на каждый использующий ее поток. В редких случаях нагрузка нулевая, но она может позволить оценить максимально возможную пропускную способность очереди. Более частый случай нагрузки это большое количество задач одинаковой ненулевой сложности. В нашем случае время на выполнение одной задачи сильно зависит от ядра, на котором эта задача выполняется. Потому задачи следует параметризовать не временем выполнения, а некоторой константой, определяющей количество вычислений.

Для изучения эффективности очередей мы протестируем их на Mate60pro и P50pro. Mate60pro имеет следующую конфигурацией ядер: 2 ядра с максимальной частотой 2,62 ГГц, 4 ядра с максимальной частотой 2,15 ГГц, 2 ядра с максимальной частотой 1,53 ГГц. P50pro имеет следующую конфигурацией ядер: 1 ядро с максимальной частотой 2,84 ГГц, 3 ядра с максимальной частотой 2,42 ГГц и 4 ядра с максимальной частотой 1,80 ГГц. Такие конфигурации соответствуют определению асимметричного набора ядер и подходят нам для эксперимента. Для обоих устройств свойственно большое количество рабочих частот. То, на какой частоте будет работать телефон, определяет операционная система. Так, может произойти ситуация, когда процессор с меньшей максимальной частотой показывает большую производительность, чем более мощное ядро. Для избежания таких ситуаций и чистоты эксперимента мы зафиксируем частоту работы процессоров на около максимальной.

Две из трех исследуемых очередей используют в своей реализации мьютексы, а значит, они зависимы от планировщика операционной системы. В отдельно взятых случаях исполне-

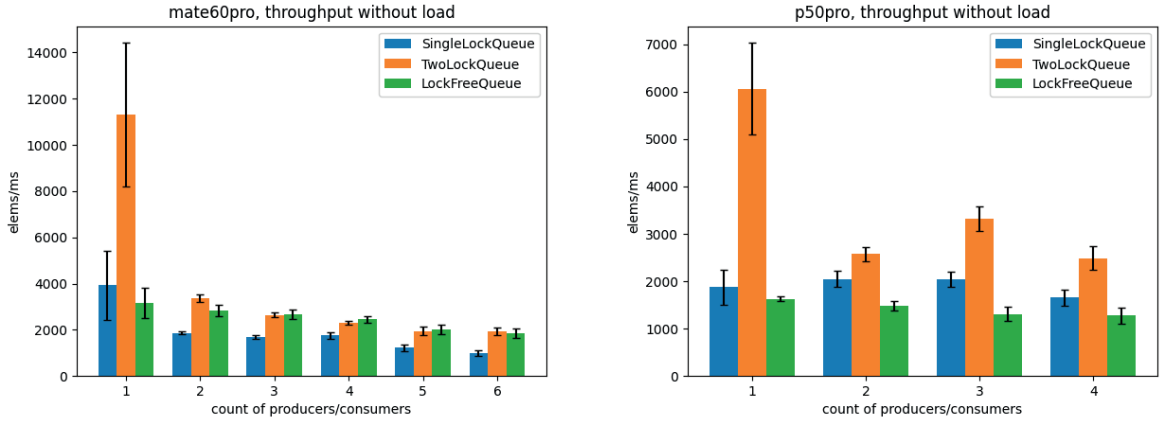
ние теста может замедляться, если планировщик не отдаст приоритет хотя бы одному потоку из тех, что используют очереди. С другой стороны, операционный планировщик способен в любой момент приостановить исполнение программы потоком, даже если мьютексы не используются. Так, каждое измерение необходимо производить большое количество раз для получения постоянной ошибки, связанной с работой планировщика.

Первым бенчмарком мы проверим, насколько эффективно очереди работают с разным числом push- и pop-потоков без нагрузки. Бенчмарк формулируется следующим образом: «Пусть к очереди одновременно обращаются `N` push- и `N` pop-потоков. Каждый push-поток пытается положить M элементов в очередь. Каждый pop-поток пытается достать как можно больше элементов. Какова пропускная способность очереди при таком сценарии использования?» Такой бенчмарк является типичным сценарием использования потокобезопасных очередей, предназначенных для работы с большим количеством как push-, так и pop-потоков. Пропускная способность будет определяться как отношение общего числа элементов к измеренному времени.

Второй бенчмарк призвана изучить эффективность очередей при варьирующейся полезной нагрузке на pop-потоки. «Пусть к очереди обращаются `N` push- и `N` pop-потоков. Каждый push-поток пытается положить M элементов в очередь. Каждый pop-поток пытается достать как можно больше элементов и производит некоторые вычисления с результатом pop-операции. Вычисления параметризуются некоторым значением P. Какова пропускная способность очереди при таком сценарии использования?» Главное отличие от предыдущего теста заключается в параметризуемой нагрузке.

5. Результаты исследования

Для начала посмотрим на результаты первой бенчмарки. Она даст нам максимально возможную пропускную способность каждой из очередей. В результате изменения пропускной способности первой бенчмарки были получены следующие диаграммы (рис. 1).

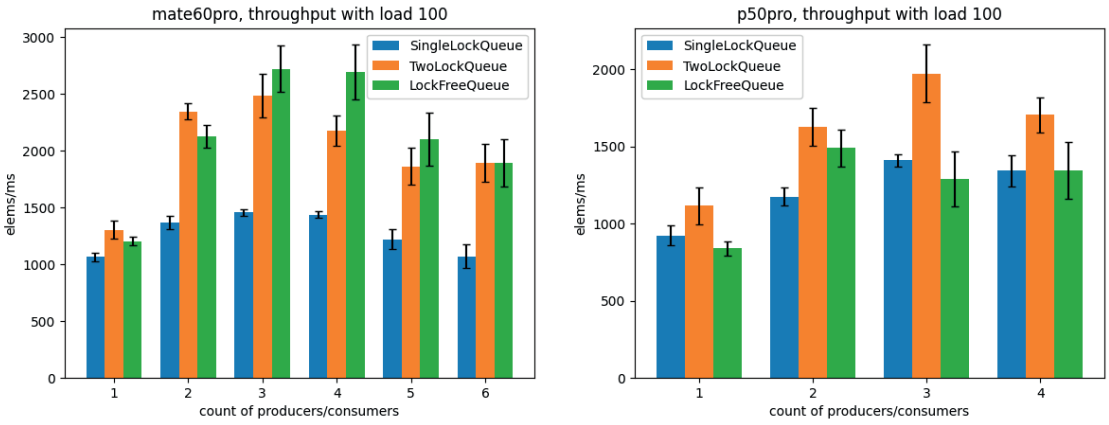


Р и с. 1. Результаты первого бенчмарка на Mate60pro и P50pro

F i g. 1. First benchmark results on Mate60pro and P50pro

Источник: здесь и далее в статье все рисунки составлены автором.

Source: Hereinafter in this article all figures were drawn up by the author.



Р и с. 2. Результаты второго бенчмарка при множителе нагрузки 100 на Mate60pro и P50pro

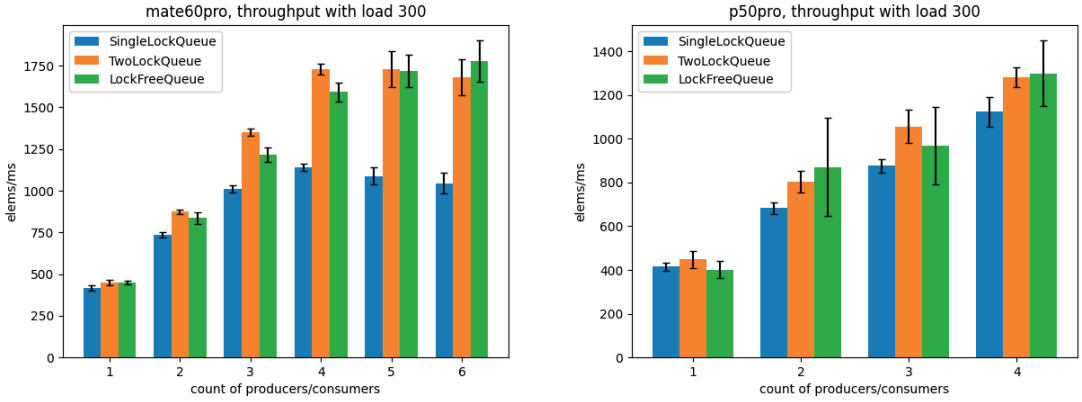
F i g. 2. Results of the second benchmark at a load multiplier of 100 on Mate60pro and P50pro

Для начала рассмотрим состояние `1 push-` и `1 pop-`поток. Тут Mate60pro показывает намного большую пропускную способность нежели P50pro, хотя частоты в среднем выше у P50pro. Такой отрыв связан с тем, что Mate60pro может использовать по одному сильному ядру на поток, а значит, нет ситуации, когда сильное ядро ждет, пока более слабое положит элемент в очередь, или когда слабое ядро обрабатывает взятие медленнее, чем сильный поток кладет элементы в очередь, а значит, происходит накапливание. Если посмотреть на очереди, то наилучший результат показывает *two lock queue* на обоих устройствах с большим отрывом. Это не удивительно, так как в основе этой очереди лежит *single producer single consumer lock free* очередь, то есть потоки вообще не конкурируют друг с другом. При этом *single lock queue* показывает большую эффективность, чем *lock free queue*. При росте числа потоков пропускная способность падает, так как конкуренция увеличивается, но для обоих устройств *two lock queue* показывает наибольшую эффективность. Интересно, что теперь на Mate60pro *lock free queue* работает лучше, чем *single lock queue*, а на P50pro – наоборот.

Первые результаты можно считать работой с очередью с около нулевой нагрузкой. Теперь посмотрим на результаты второй бенчмарки, которая покажет работу с нагрузкой. Для начала возьмем «низкую» нагрузку. В результате изменения пропускной способности второй бенчмарки с множителем нагрузки 100 были получены следующие диаграммы (рис. 2).

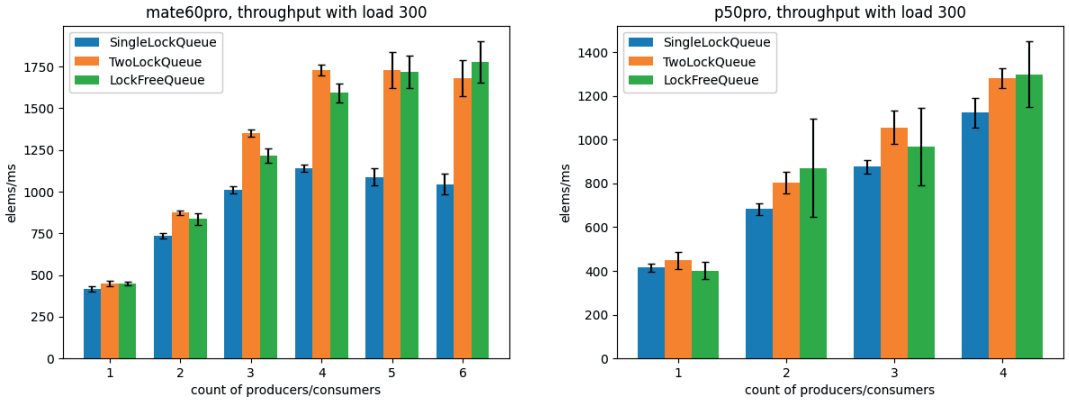
При появлении нагрузки мы можем наблюдать, что наиболее эффективным состоянием становится `4 push-`/`pop-`потока для Mate60pro и `3 push-`/`pop-`потока для P50pro. Так, при низких нагрузках использование всех ядер может замедлить работу с очередью. Для Mate60pro наиболее эффективной оказывается *lock free queue*. При этом эффективность *lock free queue* и *two lock queue* на 3 потоках примерно одинаковая, но на 4 потоках *two lock queue* сильно проседает, в том время как *lock free queue* сохраняет эффективность. Для P50pro наиболее эффективен *two lock queue* на почти всех количествах потоков, лишь при 2 потоках пропускные способности *lock free queue* и *two lock queue* примерно одинаковы. Из увиденного можно сделать вывод, что в ситуации с низкой нагрузкой для систем с большим количеством ядер следует использовать *lock free queue* и для системы с более сильными ядрами *two lock queue*.

Теперь стоит рассмотреть более высокую нагрузку для понимания тенденции изменения эффективности работы очередей. Повысим нагрузку до 500. В результате изменения пропускной способности второй бенчмарки с множителем нагрузки 500 были получены следующие диаграммы (рис. 3).



Р и с. 3. Результаты второго бенчмарка при множителе нагрузки 500 на Mate60pro и P50pro

F i g. 3. Results of the second benchmark at a load multiplier of 500 on Mate60pro and P50pro



Р и с. 4. Результаты второго бенчмарка при множителе нагрузки 300 на Mate60pro и P50pro

F i g. 4. Results of the second benchmark at a load multiplier of 300 on Mate60pro and P50pro

Для высокой нагрузки мы видим, что чем больше ядер используется, тем лучше (везде, за исключением *single lock queue* на Mate60pro). Пропускная способность повышается с ростом числа потоков. При этом *lock free queue* становится наиболее эффективным для обоих устройств.

Для того чтобы проследить тенденцию изменения наиболее эффективной конфигурации потоков, дополнительно возьмем множитель 300, что-то среднее между двумя предыдущими. В результате изменения пропускной способности второй бенчмарки с множителем нагрузки 300 были получены следующие диаграммы (рис. 4).

Мы видим, что для Mate60pro пропускная способность примерно одинакова для 4, 5 и 6 `push-/pop-` потоков. Для P50pro общая зависимость не изменилась по сравнению с множителем 500, однако сама пропускная способность увеличилась. Таким образом, мы получаем, что для P50pro это все еще является сильной нагрузкой, но для Mate60pro такая нагрузка может быть эффективно распределена на меньшее число потоков.

Так, мы получили эмпирическую зависимость пропускной способности от числа `push-/pop-`потоков и количества времени работы с элементами очереди.

6. Обсуждение и заключение

В данной статье мы изучили эффективность потокобезопасных очередей на асимметричных конфигурациях ядер. Мы реализовали *single lock queue*, *two lock queue* и *lock free queue* и получили эмпирические зависимости их пропускной способности от количества времени работы с элементами очереди и числа `push-/pop-`потоков. Ее можно использовать в эвристической реализации добавления новых и удаления ненужных потоков для получения максимальной производительности. В данной работе не был рассмотрен случай, когда нагрузка находится не только на стороне pop-потока, но и на стороне `push-`потока, так как для некоторых использованных очередей реализация системного ожидания сложна и может приводить к сильному замедлению.

References

[1] Zhu Y., Reddi V.J. High-performance and energy-efficient mobile web browsing on big/little systems. In: 2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA). Shenzhen, China: IEEE Press; 2013. p. 13-24. https://doi.org/10.1109/HPCA.2013.6522303

[2] Sallow A.B. Android Multi-threading Program Execution on single and multi-core CPUs with Matrix multiplication. *International Journal of Engineering & Technology*. 2018;7(4):6603-6608. https://doi.org/10.14419/ijet.v7i4.29340

[3] Marcu M., et al. Power Efficiency Study of Multi-threading Applications for Multi-core Mobile Systems. *WSEAS Transactions on Computers*. 2008;7(12):1875-1885. Available at: http://www.wseas.us/e-library/transactions/computers/2008/27-687.pdf (accessed 03.06.2024).

[4] Herlihy M., Moss J.E.B. Transactional memory: architectural support for lock-free data structures. *ACM SIGARCH Computer Architecture News.* 1993;21(2):289-300. https://doi.org/10.1145/173682.165164

[5] Barnes G. A method for implementing lock-free shared-data structures. In: Proceedings of the fifth annual ACM symposium on Parallel Algorithms and Architectures (SPAA '93). New York, NY, USA: Association for Computing Machinery; 1993. p. 261-270. https://doi.org/10.1145/165231.165265

[6] Goncharenko E.A., Paznikov A.A., Tabakov A.V. Evaluating the performance of atomic operations on modern multicore systems. *Journal of Physics: Conference Series*. 2019;1399(3):033107. https://doi.org/10.1088/1742-6596/1399/3/033107

[7] Hoseini F., Atalar A., Tsigas P. Modeling the Performance of Atomic Primitives on Modern Architectures. In: Proceedings of the 48th International Conference on Parallel Processing (ICPP '19). New York, NY, USA: Association for Computing Machinery; 2019. Article number: 28. https://doi.org/10.1145/3337821.3337901

[8] Maffione V., Lettieri G., Rizzo L. Cache-aware design of general-purpose Single-Producer-Single-Consumer queues. *Software: Practice and Experience*. 2019;49(5):748-779. https://doi.org/10.1002/spe.2675

[9] Koval N., Alistarh D., Elizarov R. Scalable FIFO Channels for Programming via Communicating Sequential Processes. In: Yahyapour R. (Eds.) Euro-Par 2019: Parallel Processing. Euro-Par 2019. *Lecture Notes in Computer Science.* Vol. 11725. Cham: Springer; 2019. p. 317-333. https://doi.org/10.1007/978-3-030-29400-7_23

[10] Nikolaev R., Ravindran B. WCQ: A Fast Wait-Free Queue with Bounded Memory Usage. In: Proceedings of the 34th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '22). New York, NY, USA: Association for Computing Machinery; 2022. p. 307-319. https://doi.org/10.1145/3490148.3538572

[11] Kogan A., Petrank E. Wait-free queues with multiple enqueuers and dequeuers. In: Proceedings of the 16th ACM symposium on Principles and practice of parallel programming (PPoPP '11). New York, NY, USA: Association for Computing Machinery; 2011. p. 223-234. https://doi.org/10.1145/1941553.1941585

[12] Alistarh D., Censor-Hillel K., Shavit N. Are lock-free concurrent algorithms practically wait-free? *Journal of the ACM.* 2016;63(4):31. https://doi.org/10.1145/2903136

[13] Michael M.M., Scott M.L. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In: 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. https://doi.org/10.1145/248052.248106

[14] Rodrigues F.V., Guinde N.B. Performance Analysis of Big.LITTLE System with Various Branch Prediction Schemes. In: Verma G.K., Soni B., Bourennane S., Ramos A.C.B. (eds.) Data Science. Transactions on Computer Systems and Networks. Singapore: Springer; 2021. p. 59-72. https://doi.org/10.1007/978-981-16-1681-5_5
[15] Pöter M., Träff J.L. Memory Models for C/C++ Programmers. *arXiv:*1803.04432. 2018. https://doi.org/10.48550/arXiv.1803.04432
[16] Kamil S., et al. Impact of modern memory subsystems on cache optimizations for stencil computations. In: Proceedings of the 2005 workshop on Memory system performance (MSP '05). New York, NY, USA: Association for Computing Machinery; 2005. p. 36-43. https://doi.org/10.1145/1111583.1111589
[17] Asgharzadeh A., et al. Free atomics: hardware atomic operations without fences. In: Proceedings of the 49th Annual International Symposium on Computer Architecture (ISCA '22). New York, NY, USA: Association for Computing Machinery; 2022. p. 14-26. https://doi.org/10.1145/3470496.3527385
[18] Bolosky W.J., Scott M.L. False sharing and its effect on shared memory performance. In: 4th Symp. on Experiences with Distributed and Multiprocessor Systems (SEDMS'93). San Diego, CA, USA: USENIX Association; 1993. p. 1-15. Available at: https://www.cs.rochester.edu/u/scott/papers/1993_SEDMS_false_sharing.pdf (accessed 03.06.2024).
[19] Michael M.M. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Transactions on Parallel and Distributed Systems.* 2004;15(6):491-504. https://doi.org/10.1109/TPDS.2004.8
[20] Herlihy M., Luchangco V., Martin P., Moir M. Nonblocking memory management support for dynamic-sized data structures. *ACM Transactions on Computer Systems.* 2005;23(2):146-196. https://doi.org/10.1145/1062247.1062249
[21] Cohen N., Petrank E. Efficient Memory Management for Lock-Free Data Structures with Optimistic Access. In: Proceedings of the 27th ACM symposium on Parallelism in Algorithms and Architectures (SPAA '15). New York, NY, USA: Association for Computing Machinery; 2015. p. 254-263. https://doi.org/10.1145/2755573.2755579
[22] Gidenstam A., Papatriantafilou M., Sundell H., Tsigas P. Efficient and Reliable Lock-Free Memory Reclamation Based on Reference Counting. *IEEE Transactions on Parallel and Distributed Systems*. 2009;20(8):1173-1187. https://doi.org/10.1109/TPDS.2008.167
[23] Michael M.M. Scalable lock-free dynamic memory allocation. *ACM SIGPLAN Notices*. 2004;39(6):35-46. https://doi.org/10.1145/996893.996848
[24] Miniskar N.R., Liu F., Vetter J.S. A Memory Efficient Lock-Free Circular Queue. In: 2021 IEEE International Symposium on Circuits and Systems (ISCAS). Daegu, Korea: IEEE Press; 2021. p. 1-5. https://doi.org/10.1109/ISCAS51556.2021.9401239
[25] Schweizer H., Besta M., Hoefler T. Evaluating the Cost of Atomic Operations on Modern Architectures. In: Proceedings of the 2015 International Conference on Parallel Architecture and Compilation (PACT) (PACT '15). USA: IEEE Computer Society; 2015. p. 445-456. https://doi.org/10.1109/PACT.2015.24

Поступила 03.06.2024; одобрена после рецензирования 12.08.2024; принята к публикации 23.09.2024.
Submitted 03.06.2024; approved after reviewing 12.08.2024; accepted for publication 23.09.2024.

Об авторе:

Молотков Михаил Дмитриевич, аспирант кафедры микропроцессорных технологий в интеллектуальных системах факультета радиотехники и кибернетики, ФГАОУ ВО «Московский физико-технический институт (национальный исследовательский университет)» (141701, Российская Федерация, Московская область, г. Долгопрудный, Институтский переулок, д. 9), **ORCID: https://orcid.org/0009-0003-3305-8559,** molotkov.md@phystech.edu

Автор прочитал и одобрил окончательный вариант рукописи.

About the author:

Mikhail D. Molotkov, Postgraduate Student of the Chair of Microprocessor Technologies in Intelligent Systems, Department of Radio Engineering and Cybernetics, Moscow Institute of Physics and Technology (National Research University) (9 Institutskiy per., Dolgoprudny 141701, Moscow Region, Russian Federation), **ORCID: https://orcid.org/0009-0003-3305-8559,** molotkov.md@phystech.edu

The author has read and approved the final manuscript.