Исследование оптимизаций блокирующей очереди на двух мьютексах

М.Д. Молотков

***Аннотация* — На большинстве современных устройств, в том числе и на телефонах, программы могут работать в режиме реального параллелизма, однако эффективность распараллеливания все еще остается серьезным вопросов. При этом существует некоторая специфика для телефонов, призванная уменьшить расход заряда батареи устройства, которая тоже вносит неопределенность в вопрос эффективности распараллеливания программы.**

Инструмент распараллеливания - это одна из самых важных частей любой многопоточной программы. Наиболее популярным инструмент является пул потоков, основным элементом которого является потокозащищенная очередь. Это означает, что время между созданием задачи и началом ее исполнением напрямую зависит от эффективности внутренней очереди пула потока.

В данной статье рассматривается блокирующая очередь на двух мьютексах как основной элемент для распределения задач в пуле потоков. Представлена его общая реализация, а также список оптимизаций. Среди них есть как оптимизации отдельных методов, так и способы уменьшения зависимости методов друг от друга. Все оптимизации были проверены по отдельности на современном мобильном устройстве с асимметричным набором ядер. В результате были получены наиболее эффективные конфигурации оптимизаций, а также было проведено сравнение их с наиболее распространённой реализацией.

***Ключевые слова* — Многопоточность, Конкуренция, Алгоритмы**

I. Введение

Активное развитие процессоров в последние десятилетия идёт в сторону увеличения возможности к параллельному исполнению программ. Это прослеживается как для стационарных компьютеров, так и для мобильных устройств, в том числе и для телефонов. Спецификой телефонов является использование асимметричного набора ядер [1][2]. Это уменьшает общее энергопотребление телефона, но и общую эффективность распараллеливания снижается. Использование lock-free алгоритмов в таком случае может быть неоправданным [3], а наивные реализации многопоточных алгоритмов будут все ещё слишком медленными. В таких случаях многопоточные алгоритмы зачастую подбираются и настраиваются под конкретные устройства.

Один из самых распространённых способов реализации распараллеливания программ является использование пула потоков [4][5][6]. Он позволяет разработчике перейти от парадигмы потоков к парадигме задач, что зачастую упрощает разработку. Основным компонентом такой структуры является потокозащищенная очередь, у которой существует множество подходов к реализации [7][8][9]. Все реализации варьируются по сложности и эффективности. Одно из компромиссных решений — это блокирующая очередь на двух мьютексах [7]. Она имеет простую реализацию, не требует сложного управления памятью для решения таких проблем как проблема ABA [10], и намного эффективнее наивных реализаций.

Наиболее важным параметром пула потоков является его пропускная способность, которая напрямую зависит от пропускной способности используемой очереди. Таким образом, мы приходим к проблеме эффективности работы потокозащищенного алгоритма в задаче распараллеливания программы.

II. Цель и задачи исследования

В данной статье мы рассмотрим оптимизации для блокирующей очереди на двух мьютексах и постоим наиболее эффективную ее реализацию. Задачи, которые мы поставили перед собой в этой статье: поиск и реализация оптимизаций для потокозащищенной очереди; исследования оптимизаций на целевом устройстве; комбинация наиболее эффективных оптимизаций для получения наилучшей пропускной способности.

III. Про очередь и бенчмарки

Блокирующая очередь на двух мьютексах — это примитив синхронизации, который использует два разных мьютекса для защиты операций Push и Pop по отдельности. Синхронизация между операциями Push и Pop в свою очередь работает схожим образом с single producer single consumer lock-free queue [11]. Общий вид реализации можно увидеть в аннотации (Аннотация 1).

В представленной реализации следует отметить некоторые детали.

- Первый элемент будет вложен в / взят из не нулевой, а первой ячейки первого узла. Это сделано с целью уменьшения необходимого количества проверок в if секции. В строках 4Pu и 9Po происходит проверка на то, необходимо ли производить работу с новым узлом. В обоих случаях происходит проверка на то, выходит ли индекс за границы узла. Если индекс внутреннего массива узла (остаток от деления индекса очереди) стал равен 0, тогда следует произвести особые действия по смене узла. Но нулевая ячейка тоже подпадает пот это правило, и нужно будет добавить проверку на то, что значение соответствующего индекса очереди не

Статья получена 17.08.2025
Молотков Михаил Дмитриевич, НИУ МФТИ (e-mail: molotkov.md@phystech.edu).

```
class Node {
  next_: Node ptr; buffer_: elem type[N];
}

class TwoLockQueue {
 hLock_: lock type; tLock_: lock type;
 hIndex_: size type; tIndex_: size type;
 head_: Node; tail_: Node
}

TwoLockQueueInit(TwoLockQueue* queue) {
 queue → hIndex_ = 1; queue → tIndex_ = 1;
 queue → head_ = queue → tail_ = new Node;
}

TwoLockQueuePush(TwoLockQueue* queue, elem type v) {
 1Pu lock_guard(queue → tLock_);
 2Pu tIndex = AtomicLoad(queue → tIndex_);
 3Pu tail = AtomicLoad(queue → tail_);
 4Pu if (tIndex % N == 0) {
 5Pu  newTail = new Node;
 6Pu  tail → next_ = newTail;
 7Pu  newTail → buffer_[0] = v;
 8Pu  AtomicsStore(queue → tail_, newTail);
 9Pu } else {
10Pu  tail → buffer_[tIndex % N] = v;
11Pu }
12Pu AtomicStore(queue → tIndex_, tIndex + 1);
}

TwoLockQueuePop(TwoLockQueue* queue) elem type {
 1Po lock_guard(queue → hLock_);
 2Po hIndex = AtomicLoad(queue → hIndex_);
 3Po tIndex = AtomicLoad(queue → tIndex_);
 4Po if(hIndex == tIndex) {
 5Po  return nil;
 6Po }
 7Po head = AtomicsLoad(queue → head_);
 8Po elem type v = nil;
 9Po if (hIndex % N == 0) {
10Po  newHead = head → next_;
11Po  delete newHead;
12Po  AtomicsStore(queue → head_, newHead);
13Po  v = newHead → buffer_[0]
14Po } else {
15Po  v = head → buffer_[hIndex % N];
16Po }
17Po AtomicStore(queue → hIndex_, hIndex + 1);
18Po return v;
}

TwoLockQueueIsEmpty(TwoLockQueue* queue) boolean {
1I hIndex = AtomicLoad(queue → hIndex_);
2I tIndex = AtomicLoad(queue → tIndex_);
3I return hIndex == tIndex;
}
```

Аннотация 1. Псевдокод реализации блокирующей очереди на двух мьютексах

равно 0. А это дополнительные вычисления, которые можно легко избежать.

- Для демонстрации работы с атомарными переменными мы используем функции AtomicLoad и AtomicStore (2Pu, 12Pu, 2Po и т.д.). В этих функциях находятся атомарные операции с заданными моделями памяти [12]. Позже мы обсудим, какие модели будут использоваться, но отметим сразу, что в различных местах будут использовать разные модели даже если в аннотации они обозначены одной функцией.
- Операция TwoLockQueuePop, которая достает элемент из очереди, имеет особенность. Она возвращает nil (т. е. условное обозначение, что элемента нет), если очередь пуста, что скорее свойственно TryPop операцию. Чтобы реализовать Pop операцию с ожиданием, следует обернуть нынешнюю реализацию в цикл.
- Мы предполагаем, что размер памяти, выделяемый под поля структуры, не важен. Мы допускаем возможность расположения полей так, чтобы они находились в различных кэш линиях. Это дает гарантию отсутствия false sharing-a [13]. Отметим, что это не самый эффективный способ борьбы с false sharing-ом. Лучше использовать правильно расположение полей. Необходимо, чтобы часто изменяемые многими потоками поля находились отдельно от тех, которые принадлежат только одному потоку или тех, что не изменяются. Некоторые компиляторы поддерживают перестановку полей для уменьшения false sharing-a [14].

В этой статье мы рассмотрим работу с очередью, как с инструментом передачи и распределения «задач», которые переходят от потоков «производителей» к потокам «потребителям». Пусть мы имеем N задач, P потоков «производителей» и K потоков «потребителям». Потоки-производители параллельно заполняют очередью задачами, в то время как потоки-потребители достают эти задачи и выполняют их. Мы будем рассматривать максимальную нагрузку, а значит время выполнения задачи мы примем за 0. Таким образом нам интересно, за какое время N задач пройдут через очередь при P потоках «производителях» и K потоках «потребителях». Отсюда легко найти пропускную способность очереди, как отношение количества пропускаемых элементов ко времени выполнения бенчмарка.

IV. Оптимизации

Существует множество путей оптимизации многопоточных алгоритмов. Если алгоритм основан на использовании мьютексов, то оптимизации могут заключаться в: уменьшении критической секции; разбиении этой критической секции на части; оптимизации мьютексов под критическую секцию. Эти оптимизации могут быть полезны в нашем случае, так как потоки внутри одной операции синхронизируются мьютексом. Для алгоритмов с реальным параллельным исполнением (в нашем случае потоки в Push и Pop операциях выполняются реально параллельно) оптимизации заключаются в уменьшении зависимости между методами структуры данных.

Так, мы получаем, что для нашего алгоритма существует большое количество путей оптимизации. Однако, следует понимать, что одна оптимизация может негативно сказаться на другой и, наоборот, определенные оптимизации могут не работать без другой. Это значит, что комбинировать оптимизации следует с умом.

A. Корректные барьеры и кэширование индекса

Рассмотрим уменьшение зависимости между Push и Pop операциями. Это подразумевает уменьшение числа выполнений алгоритма когерентности кэшей. Лишние барьеры памяти или гарантия глобальной последовательности может сильно замедлить работу многопоточных алгоритмов [15].

Если рассматривать алгоритм Push операции, то можно заметить, что успешность операции не зависит от состояния очереди. Поток будет либо заполняет следующую свободную ячейку, либо выделяет новый узел и класть- новый элемент уже туда. При этом, все поля очереди, которые необходимы для работы алгоритма, изменяются только этим потоком. Это означает, что атомарное чтение этих полей не требует использования acquire memory order-a. Pop операция в свою очередь использует результат Push операции, а значит нам необходимо гарантировать видность вставленного элемента. Это можно сделать с помощью release записи индекса в Push операции и acquire чтения в Pop операции. Все остальные поля, которые использует Pop операция, не изменяются и не читаются другими потоми. Это значит, что нам не нужно использовать барьеры памяти. В итоге мы получаем использование всего одной пары release записи и acquire чтения.

Однако количество acquire чтений можно сократить еще больше и ещё сильнее уменьшить зависимость Pop операции от Push-a. Рассмотрим ситуацию, когда в очереди уже хранится некоторое число элементов, и существует 2 потока, один из которых выполняет Push операцию, а другой Pop. Пусть эти потоки работают одновременно, и разница между Push и Pop индексом сохраняется. Pop операции будет использовать acquire чтение Push индекса, чтобы гарантировать видность следующего элемента. Но тогда, помимо следующего элемента, видными станут все элементы с индексами вплоть до прочитанного. Этим можно воспользоваться, и кэшировать значение Push индекса. При этой кэшированный индекс будет изменяться только тогда, когда все элементы до него прочитаны, что уменьшаем количество вызовов acquire барьеров.

В итоге мы уменьшили зависимость между Pop и Push операциями, что должно позитивно сказаться на пропускной способности очереди.

B. Хранение узлов вместо их освобождения

Для работы очереди необходимы узлы, память для которых программа получаем при помощи аллокатора (5Pu в Аннотации 1). Позже эту память необходимо освободить (11Po Аннотация 1). В идеальной системе выполнение выделения и освобождения памяти вообще не должны занимать дополнительное время. Однако сложная внутренняя структура и необходимость использования системных вызовов приводит к обратном [16]. Это означает, что критическая секция Push операции становится очень большой при попытке выделить нового узла. Аналогично, Pop операция становится дольше при освобождении узла. Это может привести к уменьшению пропускной способности. Для того, чтобы уменьшить частоту использования аллокатора, можно сохранять уже созданные узлы в кэше и использовать их позже.

При очередном освобождении узла он будет добавляться в кэш, а следующий запрос на выделение узла сначала проверит кэш. Механизм хранения представляет собой lock-free стек Майкла-Скотта с внешними узлами [17]. Для таких узлов существует проблема ABA [10], но в нашем случае она не появляется, так как со стеком будут взаимодействовать только один Push и один Pop потоки. Сам кэш может быть ограниченным и неограниченным. Для неограниченного кэша отсутствует удаление во время исполнения и очистка проходит только при уничтожении очереди. Это позволит вообще не создавать новые узлы после достижения пиковой заполненности. С другой стороны, отсутствие удаления во время исполнения не позволяет освобождать лишнюю память. Ограниченный же кэш позволяет удалять узлы, но при этом большие колебания в нагрузке на очередь могут привести к дополнительным выделениям памяти и эффект от использования кэша будет ограниченным.

Отметим, что аллокатор это синглтон на уровне процесса. Множество потоков обращаются к одному аллоткатору, и их необходимо синхронизировать. Как ранее уже отмечалось, в аллокаторах используется сложная внутренняя структура, и ее синхронизация зачастую требует использования мьютексов. Так, даже редкое использование аллокатора может приводить к нежелательному замедлению. Частично это можно решить при помощи lock-free аллокаторов [18][19], однако системные вызовы для дополнительного выделения памяти уже из операционной системы все еще влияют на производительность [20].

C. Оптимальное количество элементов в узле

Количество элементов в одном узле также является крайне важным элементом оптимизации. Это связано с особенностями работы кэша и аллокатора.

Использование маленького размера узла может приводить к неэффективному использованию кэша, так как загрузка очередной порции данных из памяти будет проходить слишком часто, что значительно замедляет программу. Это значит, что размер узла должен быть больше и равен размеру кэш линии.

С другой стороны, использование большого размера узла приводит к гарантированному использования системных вызовов в аллокаторе, так как необходимо запросить новый набор адресов. Это также может замедлить работу программы. Получаем, что верхний предел для размера также существует, но определить его можно только экспериментально.

Предлагается воспользоваться небольшими размерами узлов кратными размеру кэш линии. Это позволит уменьшить расходы на выделение памяти за счет внутренней структуры аллокатора и при этом воспользоваться преимуществом кэша процессора помаксимому.

D. Использование различных мьютексов

При работе с критической секцией мьютекс часто воспринимается как инструмент, который мгновенно «забирается» одним потоком, другие потоки мгновенно уходят в ожидание и при «освобождении» мьютекса все мгновенно пробуждаются. В реальности мьютексы оптимизируют под определенные процессоры и определенные сценарии.

Мьютесы бывают с активным и системным ожиданием.

Активное ожидание подразумевает постоянную проверку состояния какой-то переменной в цикле [21]. Взятие, освобождение и пробуждение в таких мьютексах достаточно быстрое, особенно при правильном подходе к написанию алгоритма. Однако, долгое ожидание занимает ресурсы CPU и увеличивает энергопотребление батареи устройства. Такие мьютексы называют спинлоками и зачастую используются для коротких критических секций [22].

Системное ожидание это передача управления другим потокам через системный планировщик. Процесс пробуждения и засыпания занимает время [23], но такая процедура позволяет не занимать ресурсы CPU.

В мьютексе могут применяться оба подхода, чтобы получить наиболее быструю реализацию. Такие мьютексы часто называют гибридными.

Для нашей очереди существует два сценария для обеих операций. Для начала рассмотрим Push операцию:

1. В хвосте очереди есть место для записи нового элемента. Тогда — количество команд на операцию будет ограничено;
2. В хвосте очереди нет места для записи нового элемента. Тогда — операция должна воспользоваться аллокатором (даже если мы пользуемся кэшем для узлов отсутствие запросов к аллокатору не гарантируется). Это сильно увеличивает время выполнения операции.

Для Push операции наиболее подходящим будет использование гибридного мьютекса, чтобы наиболее эффективно работать в обоих сценариях.

В случае Pop операции есть небольшие отличия от Push:

1. Pop индекс указывает на головной узел очереди. Тогда — вывод очередного элемента будет происходить за ограниченное количество команд.
2. Pop индекс указывает на первый элемент следующего узла после головного. В таком случае появляется два варианта. Если используется кэш узлов без ограничения по размеру, то количество команд все еще небольшое. В обратном же случае есть вероятность, что операция будет освобождать узел, и это будет долгой операцией.

Для Pop операции возможны варианты. Наиболее подходящим будет также гибридный мьютекс, однако мьютекс на активном ожидании также может быть применим и выигрывать за счёт простоты внутреннего алгоритма.

Наиболее часто используемый мьютекс, а именно стандартный POSIX мьютекс [24][25], использует в себе только системной ожидание. При этому его используют в самых разных случаях, в том числе, когда наличие спинлока может сильно ускорить выполнение операций.

Предлагается воспользоваться спинлоком вместо стандартного мьютекста в реализации Pop операции. Это позволит уменьшить накладные расходы на синхронизацию, а значит ускорить работу алгоритма в целом. При этом отсутствие системного ожидания не столь существенно, так как сценарий использования подразумевает взятие задачи и ее исполнение, а значит между Pop операциями обязательно есть «полезная» прослойка, дающая другим потокам время на взятие элемента из очереди.

V. Результаты

Первые результаты затрагивают каждую серию оптимизаций по отдельности, а затем, на основе полученных данных, мы построим наиболее оптимальные конфигурации, которые сравним. Все замеры были проведены на Mate60Pro со следующей конфигурацией: 2 ядра с максимальной частотой 2,62 ГГц, 6 ядра с максимальной частотой 2,15 ГГц, 4 ядра с максимальной частотой 1,53 ГГц.

Так как мы рассматриваем случай использования очереди в механизме пула потоков, мы зафиксируем количество потоков-потребителей на 4. В лучшем случае эти потоки будут работать на 2 сильных и 2 средних ядрах, что позволит получить более реалистичную пропускную способность. Через очередь будут проходить элементы размером в 8 байт, что минимизирует время, потраченное на копирования элементов

A. Корректные барьеры и кэширование индекса

Для начала рассмотрим оптимизацию, связанную с барьерами памяти. Она влияет на все операции, так как на прямую обуславливает частоту использование алгоритма когерентности кэшей. Рассмотрим следующий график (Рис. 1), где:

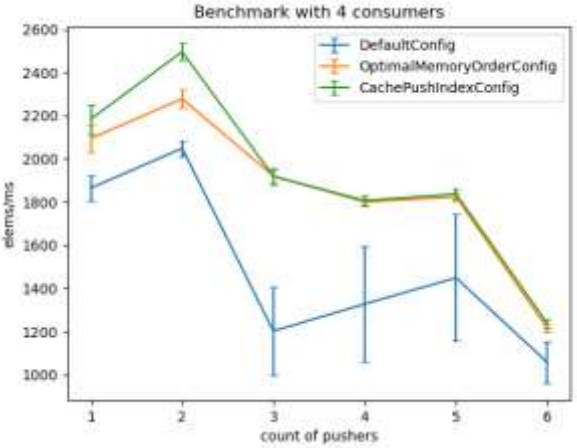

Рис. 1. Пропускная способность очереди от числа push-потоков в зависимости от используемых барьеров при 4 потоках «потребителях»

DefaultConfig — использование release-acquere модель памяти для каждой операции записи/чтения атомарных

переменных; OptimalMemoryOrderConfig — использование release-acquere модель памяти только для не локальных переменных, в нашем случае это индекс push операции; CachePushIndexConfig — использование OptimalMemoryOrderConfig и применение кэширования push индекса для уменьшение использования release-acquere модель памяти.

Из графика видно, что постоянное использование алгоритма когерентности кэшей (DefaultConfig) приводит к явному замедлению. Использование записи индекса (CachePushIndexConfig) дает выигрыш по сравнению с оптимальными барьерами (OptimalMemoryOrderConfig) только на небольшом количестве push потоков. В остальных случаях две последние конфигурации дают примерно одинаковый прирост пропускной способности. Такое поведение связано с тем, что при росте числа потоков, шанс того, что слабое ядро будет выполнять push операцию увеличивается, а значит и время нахождения в критической секции увеличивается. Это приводит к тому, что «потребители» могут опустошить очередь и каждое чтение будет сопровождаться acquire барьером.

Важно, что использование этой оптимизации может благоприятно повлиять на другие.

B. Оптимальное количество элементов в узле

Далее рассмотрим оптимизацию размера узлов. Важно использовать наиболее оптимальные барьеры памяти, чтобы минимизировать сброс кэшей. В качестве стандартного параметра мы взяли размер 128 байт (CachePushIndexConfig), для толерантного к L2 кэшу узла был взят размер в 64 байта (размер кэш линии) (L2CacheTolerantAndCachePushIndexConfig), в качестве большого размеры был взят размер в 1024 байт (BigNodeAndCachePushIndexConfig). График с результатами представлен ниже (Рис. 2).

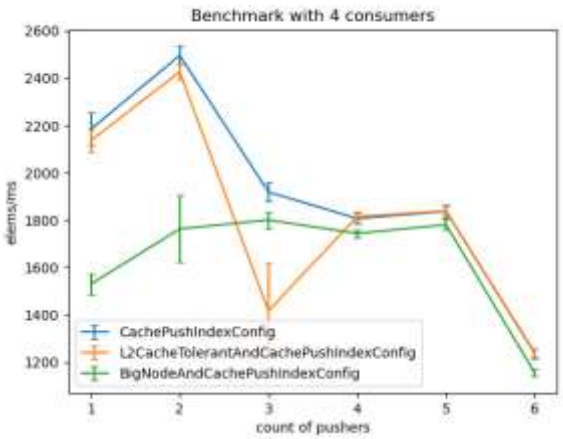

Рис. 2. Пропускная способность очереди от числа push-потоков в зависимости от количества элементов в одном узле при 4 потоках «потребителях»

Результаты показывают, что использование большого размера узлов не оправданно, малый размер выигрывает, при этом разница производительности очереди с 64 байтовыми и 128 байтовыми узлами не велика. В отельных случаях 128 байт дают выигрыш. Далее будем рассматривать именно 128 байтовые узлы в очереди.

C. Хранение узлов вместо их освобождения

Рассмотрим использование кэширования улов. Данная оптимизация направлена на уменьшение использования аллокатора. Мы рассматриваем ограниченный (LimitedNodeCacheConfig) и неограниченный (UnlimitedNodeCacheConfig) кэши. В обоих случаях кэши имели заранее выделенные узлы, которыми могли пользоваться push-потоки. В качестве размера ограниченного кэша было взято 16 узлов. Такое же количество узлов было заранее выделено в каждом из кэшей (Таким образом заранее было выделено 2Кб памяти). Результаты представлены ниже (Рис. 3).

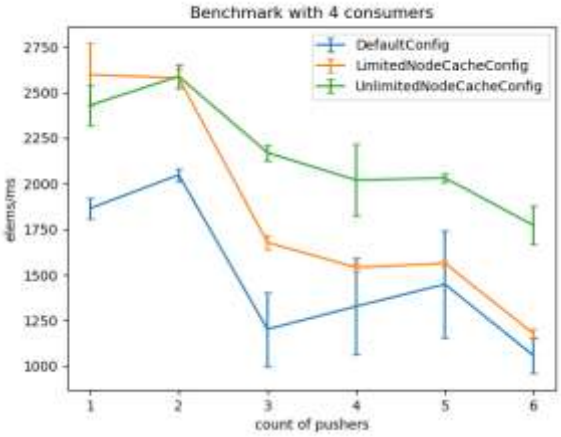

Рис. 3. Пропускная способность очереди от числа push-потоков в зависимости от конфигурации кэша узлов очереди при 4 потоках «потребителях»

Из графика (Рис. 3) видно, что использование кэша узлов хорошо сказывается на эффективности работы очереди. При этом видно, что неограниченный кэш намного эффективнее ограниченного при росте числа push потоков. Пусть неограниченный кэш и дает высокую эффективность, его будет сложно применить на практике, так как такая очередь будет «держать» память вплоть до окончания работы программы, хотя она может быть ненужна самой очереди. Так, следует использовать ограниченный кэш, но его размер следует определять исходя из нагрузок на очередь, чтобы получить эффект примерно равный эффекту от неограниченного кэша.

D. Использование различных мьютексов

Последней оптимизацией, которую мы рассмотрим, будет использование спинлоков в pop операции. Здесь мы рассмотрим простой, но эффективный TestAndSetLock (TestAndSetLockConfig) и более сложный TicketLock, который дает гарантию исполнения для всех ждущих на нем потоков (TicketLockConfig).

В качестве стандартного решения мы использовали мьютекс, который реализован в стандартной библиотеке C++. Результаты представлены ниже (Рис. 4).

По графику (Рис. 4) видно, что при использовании спинлока эффективность работы очереди

увеличивается. Разница в эффективности TestAndSetLock и TicketLock присутствует, но она не выходит за пределы погрешности

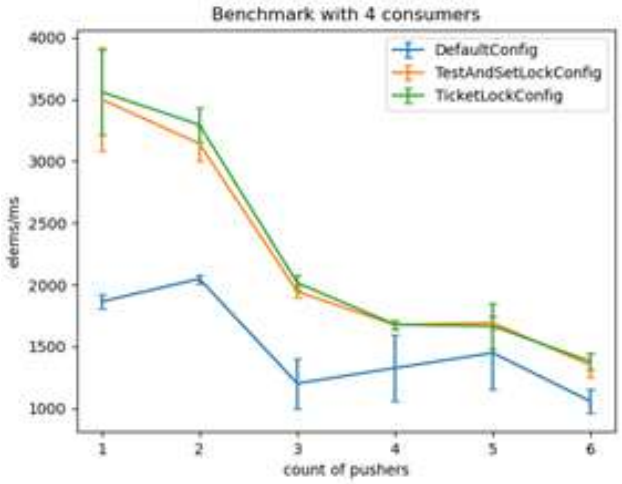

Рис. 4. Пропускная способность очереди от числа push-потоков в зависимости от используемого мьютекса при 4 потоках «потребителях»

E. Финальные конфигурации очереди

В итоге мы получаем следующий набор оптимизаций, который позволит максимально увеличить пропускную способность очереди:

- Размер узлов в 128 байт
- Кэширование push индекса при эффективных барьерах памяти
- Кэширования памяти под узлы (с большим размеров или вообще без ограничения)
- Pop операция использует спинлок (в зависимости от цели можно пользоваться как TestAndSetLock-ом, таки TicketLock-ом)

Осталось воспользоваться всеми этими оптимизациями. В качестве базы для сравнения мы возьмем реализацию, имеющую оптимальные барьеры памяти, так как такая реализация будет наиболее часто встречаться. В итоге мы получаем следующий график (Рис. 5).

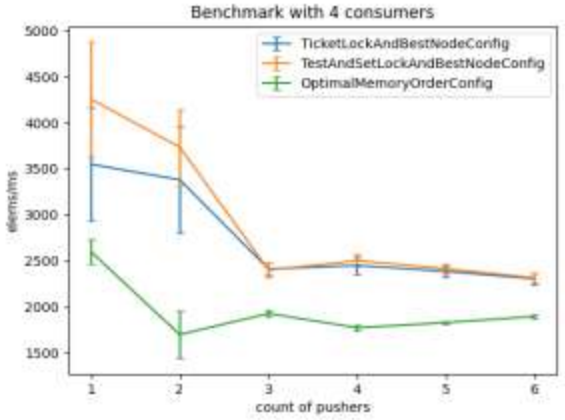

Рис. 5. Пропускная способность очереди от числа push-потоков в зависимости от финальной конфигурации при 4 потоках «потребителях»

Из него становиться видно, что все оптимизации начинают работать при увеличении числа потоков. При этом использование TestAndSetLock при малом числе push потоков работает более эффективно, чем TicketLock. Также отметим, что обе конфигурации со всеми оптимизациями дают большую пропускную способность, чем любая из примененных в них оптимизация по отдельности.

VI. Заключение

В итоге мы исследовали блокирующую очередь на двух мьютексах на предмет ее оптимизаций под задачу реализации пула потоков. Мы представили список оптимизаций, а именно: оптимизация барьеров памяти, кэширование узлов очереди, оптимизация количества элементов в очереди и использования мьютексов с активным ожиданием. Все эти оптимизации мы протестировали на современном мобильном устройстве и предложили несколько итоговых конфигураций, которые показали хороший прирост пропускной способности по сравнению с самым часто используемым вариантом реализации потокозащищенных алгоритмов.

Библиография

[1] Zhu Y., Reddi V. J. High-performance and energy-efficient mobile web browsing on big/little systems //2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA). – IEEE, 2013. – С. 13-24. URL: https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=cf64cdc889a4edaf641a307aa2b11d89d4d10a09

[2] Stevens A. Introduction to AMBA® 4 ACE™ and big. LITTLE™ Processing Technology //ARM White Paper, CoreLink Intelligent System IP by ARM. – 2011. URL: https://picture.iczhiku.com/resource/paper/WyIderGepoDsTXXb.pdf

[3] Hunt N., Sandhu P. S., Ceze L. Characterizing the performance and energy efficiency of lock-free data structures //2011 15th Workshop on Interaction between Compilers and Computer Architectures. – IEEE, 2011. – С. 63-70. URL: https://www.cs.fsu.edu/~xyuan/INTERACT-15/papers/paper06.pdf

[4] Xu D. Performance study and dynamic optimization design for thread pool systems. – Ames Lab., Ames, IA (United States), 2004. – №. IS-T 2359. URL: https://www.osti.gov/servlets/purl/835380

[5] Shiina S. et al. Lightweight preemptive user-level threads //Proceedings of the 26th ACM SIGPLAN symposium on principles and practice of parallel programming. – 2021. – С. 374-388. URL: https://dl.acm.org/doi/pdf/10.1145/3437801.3441610

[6] Sai A. M. A. et al. Producer-Consumer problem using Thread pool //2022 3rd international conference for emerging technology (incet). – IEEE, 2022. – С. 1-5. URL: https://iopscience.iop.org/article/10.1088/1742-6596/1616/1/012073/pdf

[7] Michael M. M., Scott M. L. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms //Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing. – 1996. – С. 267-275. URL: https://dl.acm.org/doi/pdf/10.1145/248052.248106

[8] Nikolaev R., Ravindran B. Wcq: A fast wait-free queue with bounded memory usage //Proceedings of the 34th ACM Symposium on Parallelism in Algorithms and Architectures. – 2022. – С. 307-319. URL: https://dl.acm.org/doi/pdf/10.1145/3490148.3538572

[9] Koval N., Alistarh D., Elizarov R. Fast and scalable channels in Kotlin coroutines //Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming. – 2023. – С. 107-118. URL: https://arxiv.org/pdf/2211.04986

[10] Dechev D. The ABA problem in multicore data structures with collaborating operations //7th International Conference on Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom). – IEEE, 2011. – С. 158-167. URL: https://www.osti.gov/servlets/purl/1118172

[11] Maffione V., Lettieri G., Rizzo L. Cache-aware design of general-purpose Single-Producer–Single-Consumer queues //Software: Practice and Experience. – 2019. – Т. 49. – №. 5. – С. 748-779. URL: https://arpi.unipi.it/bitstream/11568/942147/7/master.pdf

[12] Pöter M., Träff J. L. Memory models for C/C++ programmers //arXiv preprint arXiv:1803.04432. – 2018. URL: https://arxiv.org/pdf/1803.04432

[13] Bolosky W. J., Scott M. L. False sharing and its e ect on shared memory performance //4th symposium on experimental distributed and multiprocessor systems. – 1993. – С. 57-71. URL: https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=8090a0702dae2a90bb614e6ef8de4f049e596233

[14] Jeremiassen T. E., Eggers S. J. Reducing false sharing on shared memory multiprocessors through compile time data transformations //ACM SIGPLAN Notices. – 1995. – Т. 30. – №. 8. – С. 179-188. URL: https://dl.acm.org/doi/pdf/10.1145/209937.209955

[15] Liu N., Zang B., Chen H. No barrier in the road: a comprehensive study and optimization of ARM barriers //Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. – 2020. – С. 348-361. URL: https://ipads.se.sjtu.edu.cn/zh/publications/LiuPPoPP20.pdf

[16] Zhou J. et al. MemPerf: Profiling Allocator-Induced Performance Slowdowns //Proceedings of the ACM on Programming Languages. – 2023. – Т. 7. – №. OOPSLA2. – С. 1418-1441. URL: https://dl.acm.org/doi/pdf/10.1145/3622848

[17] Michael M. M., Scott M. L. Nonblocking algorithms and preemption-safe locking on multiprogrammed shared memory multiprocessors //journal of parallel and distributed computing. – 1998. – Т. 51. – №. 1. – С. 1-26. ULR: https://www.sciencedirect.com/science/article/pii/S0743731598914460/pdf?md5=799575cda60e46da338868a2c9635da2&pid=1-s2.0-S0743731598914460-main.pdf&_valck=1

[18] Michael M. M. Scalable lock-free dynamic memory allocation //Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation. – 2004. – С. 35-46. URL: https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=068d0b393db03678ea1d346ee01871e91e88c560

[19] Leite R., Rocha R. LRMalloc: A modern and competitive lock-free dynamic memory allocator //International Conference on Vector and Parallel Processing. – Cham: Springer International Publishing, 2018. – С. 230-243. URL: https://www.dcc.fc.up.pt/~ricroc/homepage/publications/2018-VECPAR.pdf

[20] Purohit A. et al. Improving application performance through system call composition. – Technical Report FSL-02-01, Computer Science Department, Stony Brook University, 2003. URL: https://www.am-utils.org/docs/cosy-perf/cosy.pdf

[21] Scott M. L., Scherer W. N. Scalable queue-based spin locks with timeout //ACM SIGPLAN Notices. – 2001. – Т. 36. – №. 7. – С. 44-52. URL: https://urresearch.rochester.edu/fileDownloadForInstitutionalItem.action?itemId=1325&itemFileId=1613

[22] Anderson T. E. The performance of spin lock alternatives for shared-memory multiprocessors //IEEE Transactions on Parallel and Distributed Systems. – 1990. – Т. 1. – №. 1. – С. 6-16. URL: https://pdos.csail.mit.edu/6.828/2010/readings/anderson-locks.pdf

[23] Baier C. et al. Waiting for locks: How long does it usually take? //International Workshop on Formal Methods for Industrial Critical Systems. – Berlin, Heidelberg: Springer Berlin Heidelberg, 2012. – С. 47-62. URL: https://wwwtcs.inf.tu-dresden.de/ALGI/spinlock-FMICS2012.pdf

[24] Mueller F. et al. A Library Implementation of POSIX Threads under UNIX //Usenix winter. – 1993. – С. 29-42. URL: https://arcb.csc.ncsu.edu/~mueller/ftp/pub/PART/pthreads_usenix93.pdf

[25] Bellassai D. et al. Supporting logical execution time in multi-core POSIX systems //Journal of Systems Architecture. – 2023. – Т. 144. – С. 102987. URL: http://retis.sssup.it/~a.biondi/papers/LET_JSS23.pdf

Optimizations of two lock queue

Mikhail Molotkov

***Abstract* — Real-time parallelism has become commonplace for modern devices. But the effectiveness of parallelization still remains a problem. Architecture on mobile devices also adds uncertainty to this problem.**

The parallelization tool is one of the most important parts of any multithreaded program. The most popular tool is a thread pool. Its main element is a thread-safe queue. This means that the time between the creation of a task and start of its execution directly depends on the efficiency of the internal queue of the thread pool.

This article discusses the optimization of two lock queue in a thread pool scenario. We have provided a number of optimizations based on different parameters of thread-safe containers such as the duration of the critical section and the dependence of different methods on each other. We have tested all optimizations individually on device and presented the most effective combinations of these optimizations. In the end, we compared these combinations with the most common implementations.

***Keywords* — Multithreading programming, Concurrency, Algorithms**

REFERENCES

[1] Zhu Y., Reddi V. J. High-performance and energy-efficient mobile web browsing on big/little systems //2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA). – IEEE, 2013. – С. 13-24. URL: https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=cf64cdc889a4edaf641a307aa2b11d89d4d10a09

[2] Stevens A. Introduction to AMBA® 4 ACE™ and big. LITTLE™ Processing Technology //ARM White Paper, CoreLink Intelligent System IP by ARM. – 2011. URL: https://picture.iczhiku.com/resource/paper/WyIderGepoDsTXXb.pdf

[3] Hunt N., Sandhu P. S., Ceze L. Characterizing the performance and energy efficiency of lock-free data structures //2011 15th Workshop on Interaction between Compilers and Computer Architectures. – IEEE, 2011. – С. 63-70. URL: https://www.cs.fsu.edu/~xyuan/INTERACT-15/papers/paper06.pdf

[4] Xu D. Performance study and dynamic optimization design for thread pool systems. – Ames Lab., Ames, IA (United States), 2004. – №. IS-T 2359. URL: https://www.osti.gov/servlets/purl/835380

[5] Shiina S. et al. Lightweight preemptive user-level threads //Proceedings of the 26th ACM SIGPLAN symposium on principles and practice of parallel programming. – 2021. – С. 374-388. URL: https://dl.acm.org/doi/pdf/10.1145/3437801.3441610

[6] Sai A. M. A. et al. Producer-Consumer problem using Thread pool //2022 3rd international conference for emerging technology (incet). – IEEE, 2022. – С. 1-5. URL: https://iopscience.iop.org/article/10.1088/1742-6596/1616/1/012073/pdf

[7] Michael M. M., Scott M. L. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms //Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing. – 1996. – С. 267-275. URL: https://dl.acm.org/doi/pdf/10.1145/248052.248106

[8] Nikolaev R., Ravindran B. Wcq: A fast wait-free queue with bounded memory usage //Proceedings of the 34th ACM Symposium on Parallelism in Algorithms and Architectures. – 2022. – С. 307-319. URL: https://dl.acm.org/doi/pdf/10.1145/3490148.3538572

[9] Koval N., Alistarh D., Elizarov R. Fast and scalable channels in Kotlin coroutines //Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming. – 2023. – С. 107-118. URL: https://arxiv.org/pdf/2211.04986

[10] Dechev D. The ABA problem in multicore data structures with collaborating operations //7th International Conference on Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom). – IEEE, 2011. – С. 158-167. URL: https://www.osti.gov/servlets/purl/1118172

[11] Maffione V., Lettieri G., Rizzo L. Cache-aware design of general-purpose Single-Producer–Single-Consumer queues //Software: Practice and Experience. – 2019. – Т. 49. – №. 5. – С. 748-779. URL: https://arpi.unipi.it/bitstream/11568/942147/7/master.pdf

[12] Pöter M., Träff J. L. Memory models for C/C++ programmers //arXiv preprint arXiv:1803.04432. – 2018. URL: https://arxiv.org/pdf/1803.04432

[13] Bolosky W. J., Scott M. L. False sharing and its e ect on shared memory performance //4th symposium on experimental distributed and multiprocessor systems. – 1993. – С. 57-71. URL: https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=8090a0702dae2a90bb614e6ef8de4f049e596233

[14] Jeremiassen T. E., Eggers S. J. Reducing false sharing on shared memory multiprocessors through compile time data transformations //ACM SIGPLAN Notices. – 1995. – Т. 30. – №. 8. – С. 179-188. URL: https://dl.acm.org/doi/pdf/10.1145/209937.209955

[15] Liu N., Zang B., Chen H. No barrier in the road: a comprehensive study and optimization of ARM barriers //Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. – 2020. – С. 348-361. URL: https://ipads.se.sjtu.edu.cn/zh/publications/LiuPPoPP20.pdf

[16] Zhou J. et al. MemPerf: Profiling Allocator-Induced Performance Slowdowns //Proceedings of the ACM on Programming Languages. – 2023. – Т. 7. – №. OOPSLA2. – С. 1418-1441. URL: https://dl.acm.org/doi/pdf/10.1145/3622848

[17] Michael M. M., Scott M. L. Nonblocking algorithms and preemption-safe locking on multiprogrammed shared memory multiprocessors //journal of parallel and distributed computing. – 1998. – Т. 51. – №. 1. – С. 1-26. ULR: https://www.sciencedirect.com/science/article/pii/S0743731598914460/pdf?md5=799575cda60e46da338868a2c9635da2&pid=1-s2.0-S0743731598914460-main.pdf&_valck=1

[18] Michael M. M. Scalable lock-free dynamic memory allocation //Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation. – 2004. – С. 35-46. URL: https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=068d0b393db03678ea1d346ee01871e91e88c560

[19] Leite R., Rocha R. LRMalloc: A modern and competitive lock-free dynamic memory allocator //International Conference on Vector and Parallel Processing. – Cham: Springer International Publishing, 2018. – С. 230-243. URL: https://www.dcc.fc.up.pt/~ricroc/homepage/publications/2018-VECPAR.pdf

[20] Purohit A. et al. Improving application performance through system call composition. – Technical Report FSL-02-01, Computer Science Department, Stony Brook University, 2003. URL: https://www.am-utils.org/docs/cosy-perf/cosy.pdf

[21] Scott M. L., Scherer W. N. Scalable queue-based spin locks with timeout //ACM SIGPLAN Notices. – 2001. – Т. 36. – №. 7. – С. 44-52. URL: https://urresearch.rochester.edu/fileDownloadForInstitutionalItem.action?itemId=1325&itemFileId=1613

[22] Anderson T. E. The performance of spin lock alternatives for shared-memory multiprocessors //IEEE Transactions on Parallel and Distributed Systems. – 1990. – Т. 1. – №. 1. – С. 6-16. URL: https://pdos.csail.mit.edu/6.828/2010/readings/anderson-locks.pdf

[23] Baier C. et al. Waiting for locks: How long does it usually take? //International Workshop on Formal Methods for Industrial Critical Systems. – Berlin, Heidelberg: Springer Berlin Heidelberg, 2012. – С. 47-62. URL: https://wwwtcs.inf.tu-dresden.de/ALGI/spinlock-FMICS2012.pdf

[24] Mueller F. et al. A Library Implementation of POSIX Threads under UNIX //Usenix winter. – 1993. – С. 29-42. URL: https://arcb.csc.ncsu.edu/~mueller/ftp/pub/PART/pthreads_usenix93.pdf

[25] Bellassai D. et al. Supporting logical execution time in multi-core POSIX systems //Journal of Systems Architecture. – 2023. – Т. 144. – С. 102987. URL: http://retis.sssup.it/~a.biondi/papers/LET_JSS23.pdf