Аннотация

Целью данный работы является исследование различных реализаций параллельных структур данных на примере стека и декартова дерева (`treap`). В ходе исследования построены структура данных стек с использованием блокировок, `CAS`, `hazard pointers` и транзакционной памяти и структура данных `treap` с использованием блокировок и транзакционной памяти. После сравнения времени работы данных реализаций были сделаны выводы о том, что транзакционная память (а конкретно - `Intel TSX`) позволяет построить наиболее эффективные параллельные структуры данных.

Содержание

Глава 1

Введение

Современные процессоры уже весьма сложно ускорить так же, как это делалось десятки лет назад. Закон Мура, столь успешно описывавший увеличение числа транзисторов в процессорах с 1975 года, перестал соответствовать действительности уже к концу нулевых годов в силу атомарной природы вещества и ограничения скорости света. Возникает вопрос — неужели заставить работать вычислительную технику быстрее уже невозможно?

Производители процессоров подготовили свой ответ на этот вопрос, представив в начале нулевых первые многоядерные процессоры: IBM представила Power4 в 2001, Sun Microsystems представила UltraSparc IV в 2004 году, Intel и AMD представили свои двухъядерные процессоры в 2005 году. Параллелизм на уровне инструкций позволял даже старым однопоточным программам исполняться на новых процессорах быстрее без изменений в коде.

Таким образом, для получения всей выгоды от возросшей производительности ЦП алгоритмы должны разрабатываться соответствующим образом. Межпроцессное взаимодействие (`IPC`) в таких программах организуется двумя типами: через разделяемую память (`Shared Memory`) и с помощью сообщений (`Message passing`). Программы с разделяемой памятью имеют наибольший потенциал увеличения производительности, поскольку основной метод реализации межпроцессного взаимодействия с помощью сообщений — переключение контекста, которое занимает много времени.

Алгоритмы, работающие с разделяемой памятью, могут быть бло-

кирующими (с использованием таких примитивов синхронизации, как мьютекс, семафор и т.п.) и неблокирующими. Потенциально, последние могут работать быстрее, но для этого требуется тщательная работа с памятью, поэтому важно умение правильным образом разрабатывать структуры данных для таких алгоритмов.

Глава 2

Постановка задачи

Цель данной работы — изучить влияние различных параллельных реализаций некоторых структур данных на эффективность.

Этапами работы являются:

- изучение структур данных Стек (`Stack`) и Декартово дерево (`Cartesian tree` или `Treap`);
- построение оптимистичной неблокирующей реализации Стека и Декартового дерева и анализ возможных ошибок в работе;
- использование `Hazard pointer` для построения корректно работающего неблокирующего стека и анализ применимости данного подхода для построения декартового дерева;
- анализ транзакционной памяти (`Intel TSX`) и использование `RTM (Restricted Transactional Memory)` для построения неблокирующих структур данных;
- тестирование производительности построенных структур данных и сравнение полученных результатов.

Глава 3

Обзор источников

В книге Э.Таненбаума «Современные операционные системы» [8] описана история компьютеров и операционных систем. Среди прочего, Таненбаум рассказывает о том, как разработчики повышали производительность процессоров, уменьшая размеры транзисторов и увеличивая их количество. Автор подчеркивает, что со временем становилось все более очевидно, что уменьшению размеров транзистора препятствуют законы физики, точнее говоря — законы квантовой механики. Таким образом, Эндрю Таненбаум приходит к выводу о необходимости нового шага в развитии. Если раньше увеличивалось число функциональных блоков (собственно транзисторов), то теперь этого уже недостаточно; нужно дублировать и части управляющей логики. Это и есть неотъемлемая часть современных процессоров — многопоточность (`Multithreading` по версии `Intel`). В книге описаны различные проблемы, связанные с взаимодействием потоков (например, `race conditions` — состояние гонок потоков), в том числе рассмотрены сложности совместной работы с данными нескольких потоков, отсюда была осознана вся важность успешного построения структур данных в параллельных программах.

Ранее были упомянуты проблемы, возникающие при взаимодействии потоков. Для того, чтобы подробнее разобраться в этой теме, была изучена книга Э.Уильямса «Параллельное программирование на `C++` в действии. Практика разработки многопоточных программ» [9]. Одна из тем, затронутых в книге — это разработка блокирующих и неблокирующих структур данных.

Для построения потокобезопасных структур данных необходимо ис-

пользование атомарных типов данных и операций. Атомарные операции — неделимые, то есть операция либо выполнена окончательно, либо не выполнена совсем. В C++11 атомарные операции возможно совершать только с данными атомарных типов (`std::atomic`). Кроме них, содержимое структур данных может быть защищено критическими секциями, для чего требуется использовать мьютексы и блокировки. Некоторые структуры данных требуют применения крупной гранулярности, для некоторых возможно использование мелкой. Крупно-гранулярные структуры данных (например, список с блокировкой его головы) устроены так, что блокируются большие участки памяти. В мелко-гранулярных структурах данных (например, список с отдельной блокировкой каждого элемента списка) блокируются более мелкие участки памяти, что может обеспечить более эффективный параллельный доступ к объекту. Однако, фундаментальные ограничения на совместный доступ приводят к тому, что параллельный код дает слабый выигрыш в производительности.

В структуре данных без блокировок критические секции в узком смысле этого термина, то есть, мьютексы и блокировки, не используются. Это позволяет избежать ряда проблем, присущих структурам данных с блокировками, однако усложняет разработку структуры данных. Неблокирующая структура данных становится открытой для одновременного доступа со стороны сразу нескольких потоков (при этом необязательно эти потоки будут совершать одинаковые операции). Построить такую структуру данных, допускающую параллельное использование разными потоками, довольно сложно, на каждом шаге требуется, чтобы хоть какой-то поток продвигался вперед. Среди неблокирующих структур данных выделяют структуры данных, свободные от ожидания (wait-free). На них наложено еще более серьезное ограничение: каждый поток должен завершить свою работу со структурой данных за ограниченное число шагов, независимо от работы других потоков. То есть, на каждом шаге должны продвигаться все потоки, ни один из них не ждет других.

Однако, разработка таких неблокирующих структур данных — процесс достаточно сложный, к тому же у таких структур данных имеются свои недостатки. Несмотря на то, что они позволяют лучше распаралле-

лить операции и сократить время ожидания, ничто не гарантирует нам повышение производительности программы. Это связано с тем, что атомарные операции, которые будут часто использоваться в неблокирующей структуре данных, исполняются достаточно долго, к тому же, возможно, взаимодействие между потоками может быть реализовано неоптимально. Вышеизложенные выводы послужили основанием для принятия решения рассмотреть две структуры данных — стек и декартово дерево (`Treap`), и построить их неблокирующие реализации.

В книге Херлихая «Искусство многопроцессорного программирования» [3] исследованы параллельные реализации таких структур данных, как очередь и стек. Наиболее наивная неблокирующая реализация этих структур данных основана на оптимистическом подходе. Поток, который будет работать с данными, делает снимок текущего состояния структуры данных, модифицирует данные, проверяет перед записью, не изменилась ли за это время структура, и, если не поменялась (в надежде на это и состоит оптимизм), то записывает модифицированные данные. В случае, если сторонний поток за это время уже внес какие-либо изменения, поток начинает работу заново, то есть снова делает снимок текущего состояния и т.д. Чтобы проверить, изменилось ли что-либо в структуре, используется операция `CAS (Compare And Swap)`. В качестве аргументов операции передаются сохраненные копии изначальных данных: указатель на структуру данных (то есть на текущее ее состояние) и модифицированные текущим потоком данные, которые надо вставить в структуру данных.

К сожалению, такой подход несет в себе опасность — структура данных может быть подвержена ошибке `ABA`. Это ошибка, которая возникает, если поток ориентируется на постулат, что если ячейка памяти, например, указатель, не изменилась за время работы, то не изменилась и сама структура данных. Однако другой поток мог между двумя считываниями изменить значение данной ячейки, исполнить какие-либо действия и восстановить значение ячейки.

Рассмотрим проблему `ABA` применительно к неблокирующему стеку. Изначально стек содержит $head \longrightarrow A \longrightarrow B \longrightarrow C$. Первый поток начинает выполнять операцию `pop`. В это время структуру данных пе-

рехватывает второй поток, успевая закончить выполнение операции pop раньше первого потока. Затем второй поток выполняет операцию `push (D)`, таким образом в голове стека находится элемент с указателем D. Второй поток изменяет данные по указателю A и выполняет функцию `push (A)`. Только после этого операция `CAS` успешно выполнится, поскольку успешным будет сравнение двух одинаковых элементов A. Поэтому первый поток будет считать, что ничего не было изменено и выполнит функцию pop, то есть вместо указателя на элемент A поставит указатель на элемент B. Тогда в стеке пропадает элемент с указателем D. Структура данных становится несогласованной.

Более подробно проблема `ABA` описана в книге А.Г. Тормасова «Параллельное программирование многопоточных систем с разделяемой памятью» [7] помимо основных аспектов работы с разделяемой памятью, таких как причины условий гонок и необходимость использования атомарных операций, более подробно разобраны проблемы, возникающие при работе с разделяемой памятью, в том числе проблема `ABA`. Кроме описания проблемы, в книге приведен один из способов ее решения — `RCU`. `RCU (Read, Copy, Update)` — алгоритм чтения, копирования и обновления. Для изменения данных поток-писатель делает себе полную копию, проводит операции над ней и атомарно переставляет указатель, который теперь показывает на свою копию. Поток-писатель сначала находится в фазе `Removal phase`, в рамках которой происходит изменение структуры данных без непосредственного удаления элемента; затем переходит в `Grace Period start phase`, которая объявляет о начале `Grace Period`, промежутка, в рамках которого все потоки не находятся в критической секции; следующая фаза — `Grace Period end waiting phase`, в которой происходит ожидание окончания `Grace Period`; все заканчивается `Reclamation Phase` — фазой, в которой писатель фиксирует проведенные изменения. Однако, к сожалению, замена указателя должна происходить не «в удобное для потока-писателя момент времени», а «в удобное для всех потоков время», то есть нужно дождаться выполнения кода на каждом потоке и быть уверенным, что нигде нет неправильной ссылки на старую область памяти. При большом числе потоков замена указателя может откладываться очень надолго, что приводит в дальнейшем к

долгой потере отзывчивости системы. На уровне ядра операционной системы проблема решается — управлением занимается планировщик потоков, а поток-писатель находится в ожидании команд от процессора. Если структуры данных, требующие использование RCU, находятся в пользовательском адресном пространстве (user-space), в котором явный вызов примитивов планировщика недоступен, то приходится прибегать к использованию так называемых user-space RCU, которые представляют собой совершенно нетривиальные структуры данных с большим объемом кода. Использование такого механизма приводит к большой потере времени исполнения, делая борьбу с блокировками непрактичной.

Среди других способов решения проблемы ABA можно отметить такие методы, как Garbage Collector, Reference Counting, Double-length CAS, LL/SC. Garbage Collector — весьма простой подход к решению проблемы ABA, заключается в хранении указателей, по которым «в удобный момент времени», то есть только тогда, когда никто не будет использовать данные по указателям, будет высвобождаться память. Без блокировок такой механизм не реализуется, следовательно для написания неблокирующих структур данных он не подходит. Часто в Garbage Collector используются Reference Counter — счетчики ссылок на удаляемый объект. Такие счетчики можно использовать и без Garbage Collector, но без блокировок операции с ними трудно реализуются и эффективность lock-free структуры данных резко снижается. Замена операции CAS на Double-length CAS достаточно элегантно решает проблему ABA, храня во второй части счетчик. Сравнение тогда производится по паре из предыдущего значения и счетчика с текущим значением и счетчиком. При совпадении — происходит обмен (swap). В случае возникновения проблемы ABA значения совпадут, а счетчики — нет, то есть в таком случае CAS не выполнится. Если для 32-битных машин Double-length CAS — это 64-битный CAS и современные устройства способны обеспечить их поддержку, то большинство 64-битных вычислительных систем не поддерживают 128-битный CAS, что не позволяет считать Double-length CAS эффективным решением проблемы ABA.

На ряде аппаратных архитектур существует другой подход — примитив аппаратной синхронизации LL/SC. Это пара инструкций, первая из

которых, LL (load linked), фиксирует адрес ячейки памяти, вторая, SC (store conditional), записывает по этому адресу свое значение тогда и только тогда, когда между выполнением второй и первой инструкций не было других записей в эту ячейку памяти. При всех плюсах такого механизма (отсутствие блокировок, защищенность от проблемы ABA и т.д.), эти инструкции не реализованы в архитектуре Intel.

В данной работе в качестве оптимального решения проблемы ABA был выбран механизм, который называется «Опасные указатели» (Hazard pointers). Он был предложен в статье Maged M. Michael «Safe memory reclamation for lock-free objects» [4]. Основная идея метода состоит в следующем: каждый поток сохраняет фиксированное число опасных указателей (как правило один или два), указывающий на данные, которые, возможно, будут изменены. Каждый такой опасный указатель может быть записан только владеющим им потоком, однако читать данные по этому указателю могут все потоки (режим «один писатель — много читателей»). Если какой-то поток хочет удалить указатель, то этот указатель помещается в особый список, из которого указатели удаляются в тот момент, когда они перестанут быть опасными для всех потоков. Эффективность данного подхода заключается в том, что он занимает дополнительно всего лишь константный фиксированный промежуток времени для каждого изменяемого объекта. Hazard pointers не только работают без блокировок (lock-free), но и без ожидания (wait-free), то есть гарантируется прогресс каждого потока.

Применение опасных указателей в построении неблокирующих структур данных довольно подробно разобрано в статье А. Александреску «Неблокирующие структуры данных с опасными указателями» [1]. В качестве примера структуры данных в статье взята WRRMMap (класс, в котором хранится указатель на однопоточный std::map и обеспечен многопоточный неблокирующий доступ к нему). Автор описывает саму структуру опасных указателей, устройство списка указателей «на удаление» и алгоритм по сканированию опасных указателей. Затем, с помощью операций (Update) и (Lookup), Александреску встраивает опасные указатели в структуру данных. Автор доказывает, что данные операции неблокирующие (lock-free); кроме того, в статье приведены указания,

как надо изменить операции, чтобы они стали свободными от ожидания (`wait-free`). Пользуясь материалами из данной статьи, можно смело приступать к построению произвольных неблокирующих структур данных.

Так как непосредственно реализация опасных указателей не входила в планы данной работы, было принято решение использовать хорошо зарекомендовавшие себя готовые реализации данного механизма. В библиотеке `libcds` [2] представлены методы безопасного освобождения памяти, в том числе опасные указатели (`Hazard pointers`). Они реализованы в виде синглтона, к которому нужно подключать все создаваемые потоки. При помощи методов этой библиотеки была создана неблокирующая реализация стека, не подверженная проблеме `ABA`.

Стек — относительно простая структура данных, имеющая всего один указатель. При построении сложных структур данных, состоящих из нескольких указателей, атомарное обновление каждого указателя по отдельности не имеет смысла (при отсутствии блокировок), а совместное обновление затруднительно. В связи с этим встает вопрос о разработке неблокирующей структуры данных, реализующей парадигму `CRUD` — create, read, update, delete. Бинарное дерево поиска хорошо подходит для реализации абстрактных типов данных `set`, `map`. В случае необходимости хранить упорядоченные по ключам данные эти структуры подходят больше, чем, например, хэш-таблица. В качестве примера подобной структуры данных было взято декартово дерево (`cartesian tree` или `Treap`). Эта структура данных впервые была описана в статье Seidel, Aragon «Randomised search trees» [5]. Она объединяет в себе бинарное дерево поиска и бинарную кучу (пирамиду). Говоря более конкретно, эта структура данных хранит пары (x, y) таким образом, что является бинарным деревом по элементам x и бинарной пирамидой по элементам y. Если некоторый элемент дерева содержит (x_0, y_0), то у всех элементов в левом поддереве $x \leq x_0$, а у всех элементов в правом поддереве $x \geq x_0$, а также и в левом, и в правом поддереве $y \leq y_0$. Если выбирать приоритеты случайно, то декартово дерево станет рандомизированным бинарным деревом поиска, которое обладает сложностью $\log(N)$ во всех методах `CRUD`.

Основные операции, через которые реализуется работа с декартовым деревом — это `split` и `merge`. Операция `split` разбивает дерево на два поддерева по ключу так, чтобы в первом поддереве были элементы с меньшим ключом, а во втором — элементы с большим ключом. Реализовать такую функцию будет удобно рекурсивно: если ключ совпадает с ключом корня, то результатом будут левое и правое поддеревья. Если ключ больше ключа корня, то корнем первого поддерева будет корень исходного дерева, а корень второго поддерева появится в результате применения операции `split` к правому поддереву. Все выполняется симметрично, если ключ меньше ключа корня.

Операция `merge` сливает два дерева в одно, сохраняя при этом структуру (по ключам и приоритетам). Предполагается, что оба дерева, которые передаются функции, обладают соответствующим порядком, то есть все ключи в первом дереве меньше, чем ключи во втором. Тогда сравниваются приоритеты корней первого и второго деревьев, если приоритет первого выше, то он и будет являться корнем. Тогда выполняется операция `merge`, получая в качестве аргументов правое поддерево первого дерева и второе дерево. Если приоритет корня второго дерева выше, то выполняется все симметрично.

Поскольку каждый элемент декартова дерева содержит два указателя, которые, при изменении структуры данных, необходимо одновременно атомарно обновлять, то опасные указатели (`Hazard Pointers`) не могут обеспечить корректную работу. В поиске подходящих механизмов снова пришлось обратиться к книге Тормасова [7]. В ней упоминается подход, называемый транзакционной памятью, суть которого в следующем: любые операции с памятью, выделенные в специальный блок, выполняются транзакционно, то есть или выполняются все, или не выполняется ни одна. Существенное ограничение в работе с транзакционной памятью — все операции внутри транзакции должны быть откатываемы. Выделение памяти, обмен данными со внешними устройствами или с другими процессами, вывод на экран — все эти операции нельзя откатить, а потому подобные этим операции не могут выполняться внутри транзакции. Кроме того, данные, над которыми выполняются операции в рамках транзакции, должны быть внутри одного ядра процессора. Это

в свою очередь еще сильнее ограничивает набор допустимых операций, которые можно исполнять в рамках одной транзакции.

G++ поддерживает транзакционную память (STM), начиная с версии 4.7. Идея программного подхода к транзакционной памяти состоит в том, чтобы выделить участок кода, который будет внутри транзакции (для g++ код нужно писать внутри фигурных скобок __transaction_atomic{}). Однако, максимальную эффективность дает аппаратная поддержка транзакционной памяти, которую Intel встроила в свои процессоры в 2013 году (Intel TSX). Документация Intel [6] предлагает два набора команд: HLE и RTM. HLE (Hardware Lock Elision) предоставляет две инструкции - XACQUIRE и XRELEASE. В случае, если начать транзакцию не получается, система автоматически пробует сделать это снова (выполнить XACQUIRE), таким образом, нельзя задать алгоритм действий на случай, если транзакция не завершилась успешно. В RTM такой проблемы нет, поэтому в данной работе будет использоваться именно этот набор команд. RTM включает в себя три инструкции: XBEGIN, XEND, XABORT. XBEGIN и XEND отмечают начало и конец транзакции, а XABORT прерывает транзакцию, при этом в регистре eax содержится статус транзакции. Проверить этот статус можно с помощью инструкции XTEST.

Глава 4

Методология исследования

В данной работе разрабатываются и исследуются разные реализации структур данных стек и декартово дерево (`Treap`). Реализованы эти структуры на языке `C++`, который является стандартом для системного программирования и для которого есть большое количество эффективных компиляторов и библиотек. В версию `C++11` была добавлена поддержка многопоточного программирования (`std::thread`).

В качестве компилятора в данной работе был выбран `g++ 8.3`. Данный компилятор используется большинством разработчиков и является стабильным, поскольку поддерживается `Free Software Foundation` с 1987 года. Используется компьютер с процессором `Intel Core i7 8700B, Coffee Lake`, который поддерживает аппаратную транзакционную память (`Intel TSX`). При проведении эксперимента `Turbo Boost` был отключен.

Для наиболее точных замеров времени работы используется ассемблерная инструкция `rdtsc`, которая возвращает число тактов с момента последнего сброса процессора. Разница двух величин (замеров до начала тестирования структуры данных и после начала) дает время (в тактах процессора) исполнения операций над структурой данных.

Как известно, параллельные структуры данных принято делить на типы: интересующиеся значением объекта или ключа (`set`, `map` и т.п.) и не интересующиеся значением объекта («пулы» — стеки, очереди и

т.п.). Исследованию подверглись оба типа структур данных; стек является контейнером типа «пул» и для него вставлялись и удалялись случайные значения. Контейнеры, использующие ключ для адресации представлены `treap` — структурой данных, реализующей парадигму `CRUD`. Тестирование структуры данных `treap` заключается во вставке и удалении большого количества ключей. Для чистоты эксперимента, данные генерируются случайным образом и операции вставки и удаления тоже вызываются случайно; чтобы минимизировать шанс удаления элемента из уже пустой структуры данных, изначально вставляется относительно большое число данных (в замерах времени эта порция вставок не участвует).

Глава 5

Результаты

5.1 Стек

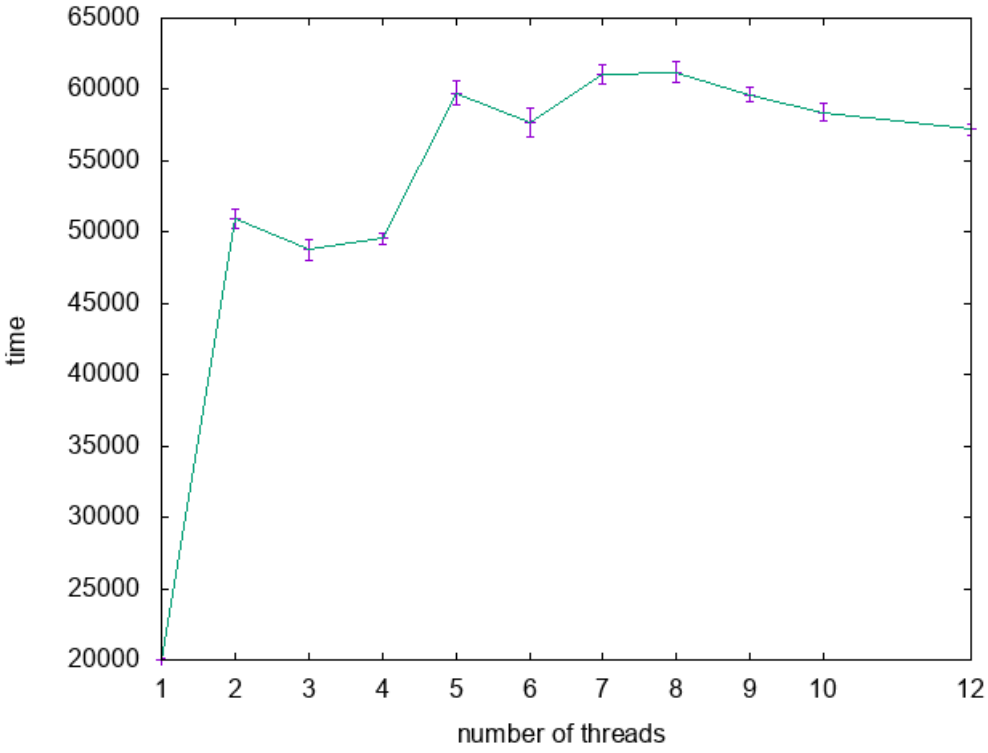
Стек был реализован с помощью опасных указателей (`hazard pointers`) и транзакционной памяти (`RTM`). Однако, полезно для сравнения иметь реализации с использованием блокировок (`mutex`) и оптимистические, то есть только с использованием `CAS`.

5.1.1 Блокировки

Стек с параллельным доступом удобно строить на основе списка — это решает проблему выделения памяти при переполнении. Блокировки реализуются с помощью `mutex`, который блокирует структуру данных как только один из потоков вносит изменения в структуру данных. `lock` и `unlock` вызываются в модифицирующих структуру данных функциях `push` и `pop` (см. приложение 1)

Замеры времени работы стека с блокировками:

Потоки	Время исполнения	Погрешность
1	20046	44
2	50958	683
3	48748	727
4	49560	370
5	59722	838
6	57649	1025
7	61055	688
8	61206	711
9	59642	515
10	58397	650
12	57183	443



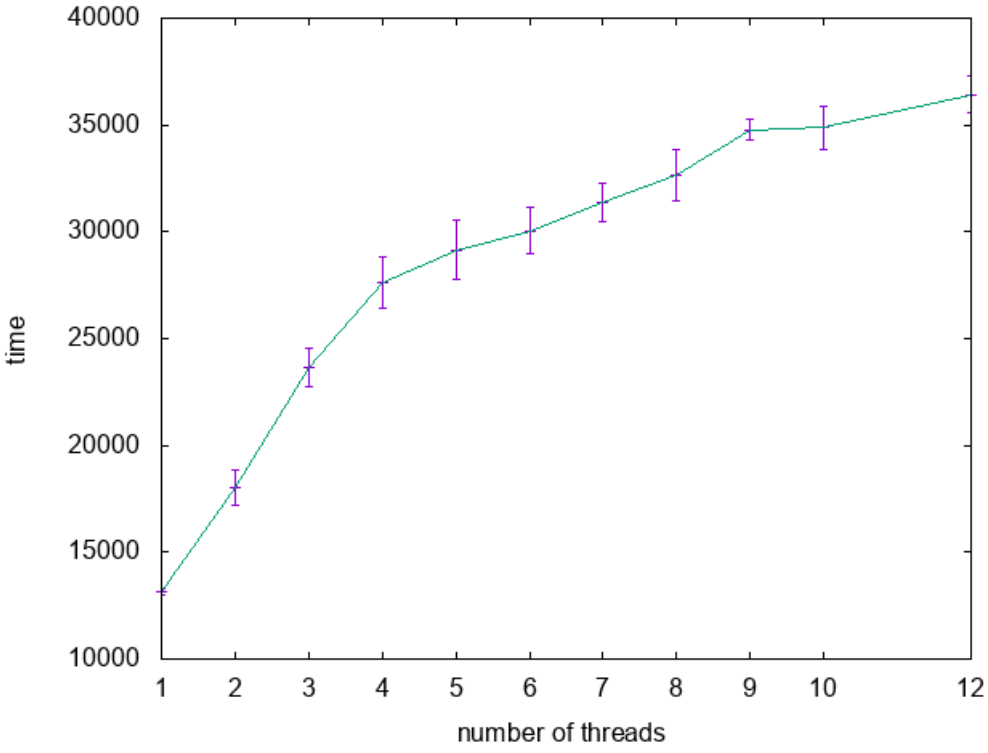
На графике видно, что время работы структуры данных резко увеличивается при увеличении числа взаимодействующих с ней потоков. Очевидно, при увеличении числа потоков вероятность обращения разных потоков к одним и тем же элементам структуры данных становится все выше, увеличивая время ожидания освобождения элементов от блокировок.

5.1.2 `CAS`

В оптимистической реализации стека блокировок нет, модификация данных выполняется с помощью операции `CAS` в функциях `push` и `pop`. Операция `CAS` заключена в цикл, из которого программа выходит тогда, когда сравнение в рамках операции `CAS` выполняется успешно (см. приложение 2)

Замеры времени работы оптимистической реализации стека:

Потоки	Время исполнения	Погрешность
1	13144	181
2	18024	812
3	23641	911
4	27650	1196
5	29161	1374
6	30062	1104
7	31368	927
8	32661	1210
9	34772	493
10	34866	980
12	36433	891



Данная реализация исполняет операции быстрее, чем блокирующий стек, однако так же подвержена замедлению при увеличении числа потоков. Очевидно, некоторое время тратится на успешный выход из цикла с операцией `CAS`, что замедляет каждое изменение структуры данных тем сильнее, чем больше потоков будет взаимодействовать со структурой данных.

5.1.3 `Hazard Pointers`

В данной работе не приводится непосредственно реализация опасных указателей `hazard pointers`, было принято решение воспользоваться готовой библиотекой `libcds`, одним из инструментов которой и являются `hazard pointers`. Указатель, данные по которому будут изменяться, помечается «опасным» и перестает быть таковым тогда, когда все операции над ним завершены (см. приложение 3).

Замеры времени работы стека с использованием опасных указателей (`hazard pointers`):

Потоки	Время исполнения	Погрешность
1	18609	291
2	17815	265
3	17202	158
4	18769	301
5	19204	449
6	20277	655
7	21652	564
8	22494	400
9	23160	741
10	24418	907
12	26705	1122

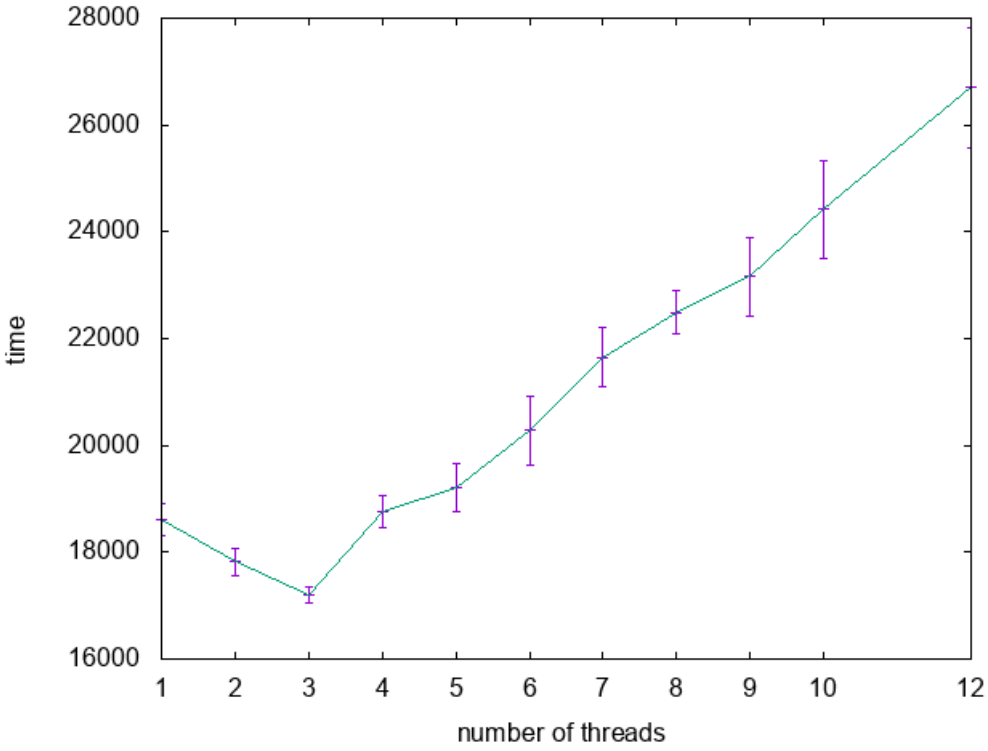


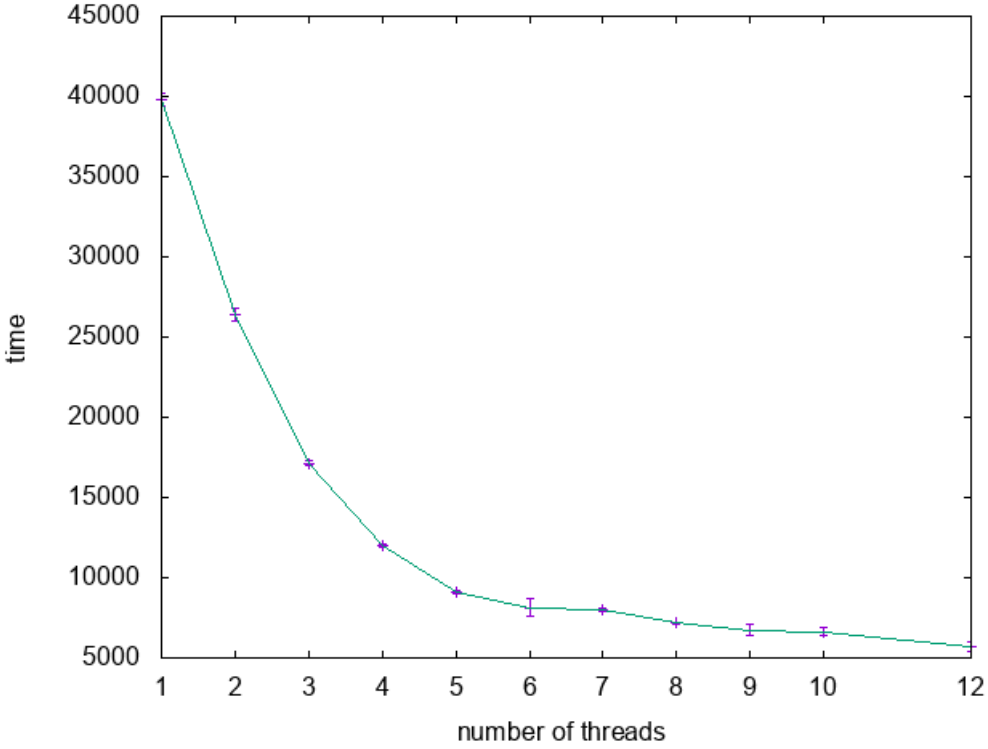
График показывает, что после небольшого участка увеличения производительности наступает участок роста времени работы. Подобное поведение связано с особенностями устройства сторонней библиотеки `libcds`, которая использовалась непосредственно для работы с опасными указателями. Выделение памяти для бóльшего числа указателей, их сканирование и удаление затрачивают ресурсы процессора и замедляют структуру данных.

5.1.4 Транзакционная память

В релизации стека с транзакционной памятью внутри функций `push` и `pop` ведется подготовка данных к транзакции и ее запуск. Используемые в транзакции данные должны быть помещены компилятором в регистр или в кэш-память перед запуском. Затем производится запуск транзакции, в случае, если она запущена успешно (идет проверка переменной `status`), выполняется модификация данных (см. приложение 4).

Замеры времени работы стека с использованием транзакционной памяти (`RTM`):

Потоки	Время исполнения	Погрешность
1	39775	391
2	26393	418
3	17113	153
4	11990	81
5	9077	104
6	8144	529
7	7966	104
8	7160	74
9	6705	350
10	6608	256
12	5718	298



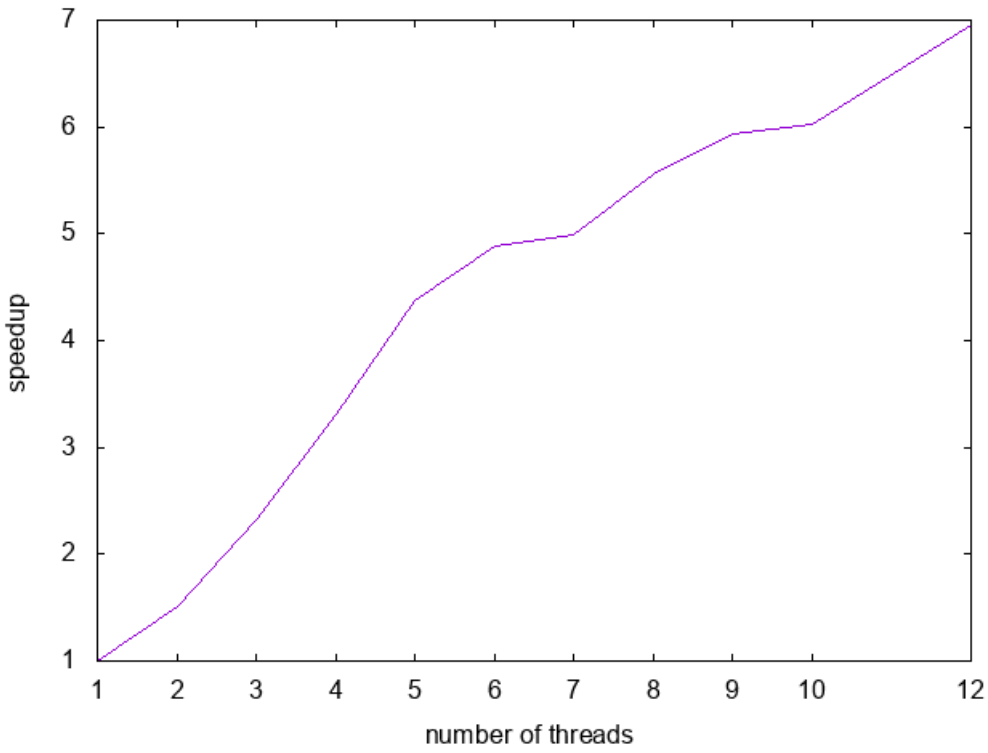


График показывает увеличение производительности структуры данных при возрастании числа потоков. Как видно из графиков, ускорение доходит до 7 раз при 12 потоках.

5.2 Декартово дерево

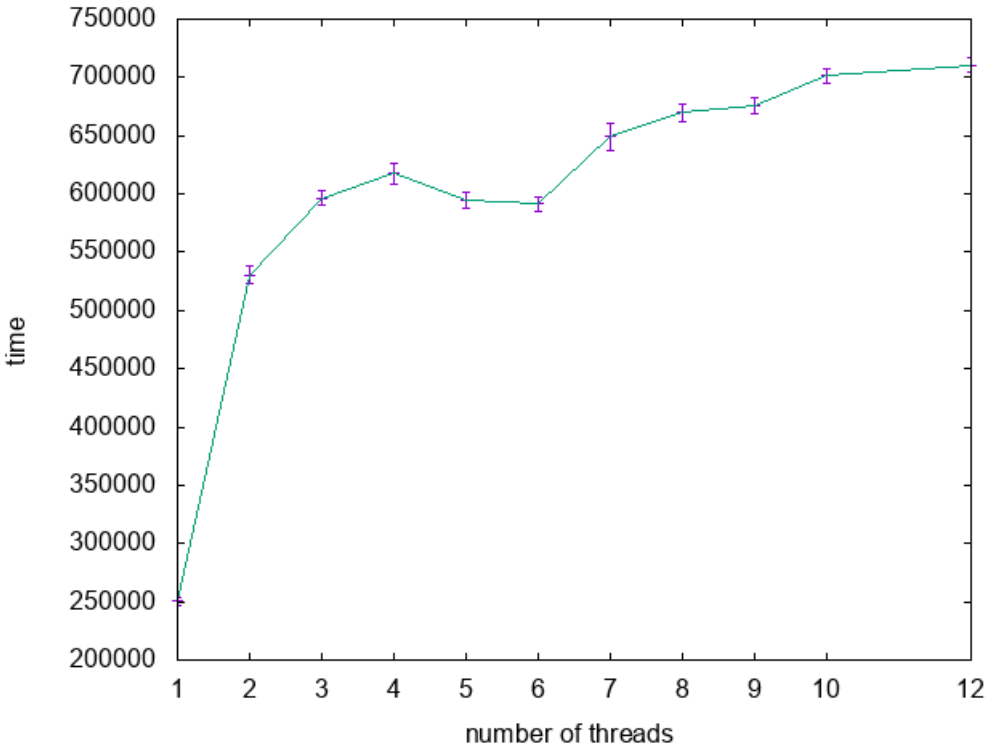
`Treap` был реализован с помощью `RTM`, для сравнения была разработана реализация `Treap` с использованием блокировок.

5.2.1 Блокировки

`Treap` — дерево, в котором операции `split` и `merge` могут вызывать друг друга рекурсивно, что усложняет применение `mutex`. `Recursive mutex` позволяет потоку, который установил блокировку, снова вызвать `lock`; при этом объект будет доступен другим потокам только тогда, когда текущий поток вызовет столько же unlock, сколько ранее было `lock`. Пользуясь подобным механизмом, внутри функций `insert` и `delete` структура данных блокировалась до тех пор, пока с ней не закончит работу поток (см. приложение 5)

Замеры времени работы `Treap` с блокировками:

Потоки	Время исполнения	Погрешность
1	250450	3838
2	530526	7586
3	596581	6757
4	617337	8370
5	594668	7217
6	591309	5974
7	649317	11785
8	669746	7717
9	675389	6807
10	701582	5929
12	710324	6381



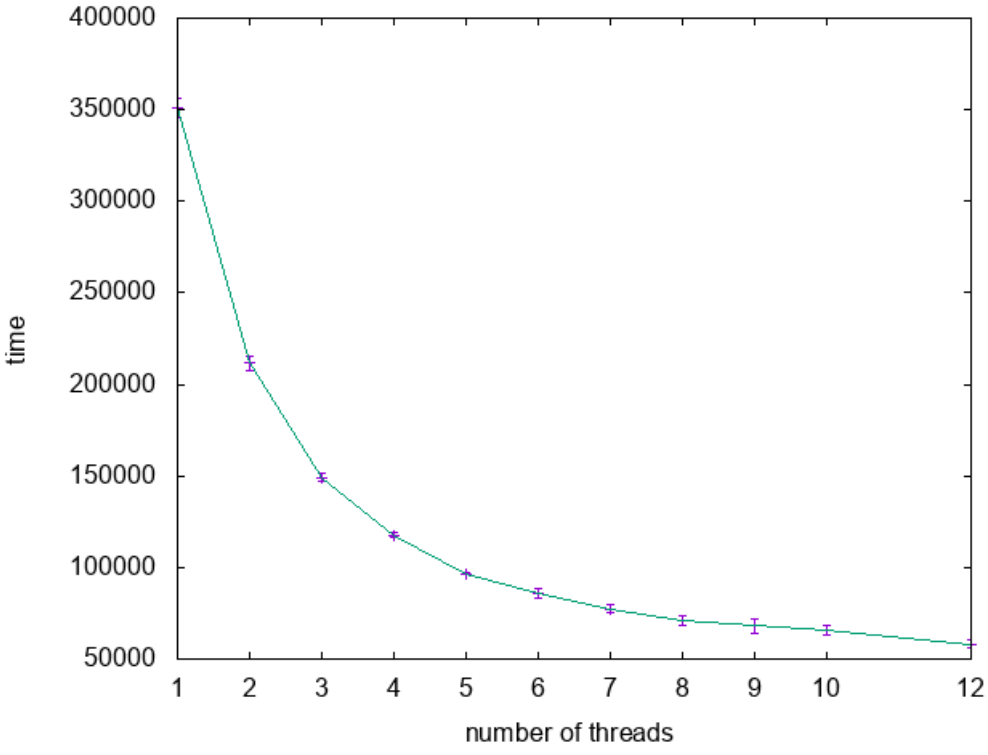
Аналогично стеку, `Treap` с блокировками замедляется при увеличении числа потоков. Чем больше потоков - тем чаще будут обращения разных потоков к одним и тем же элементам структуры и тем больше они будут ждать друг друга, что и является причиной увеличения времени работы.

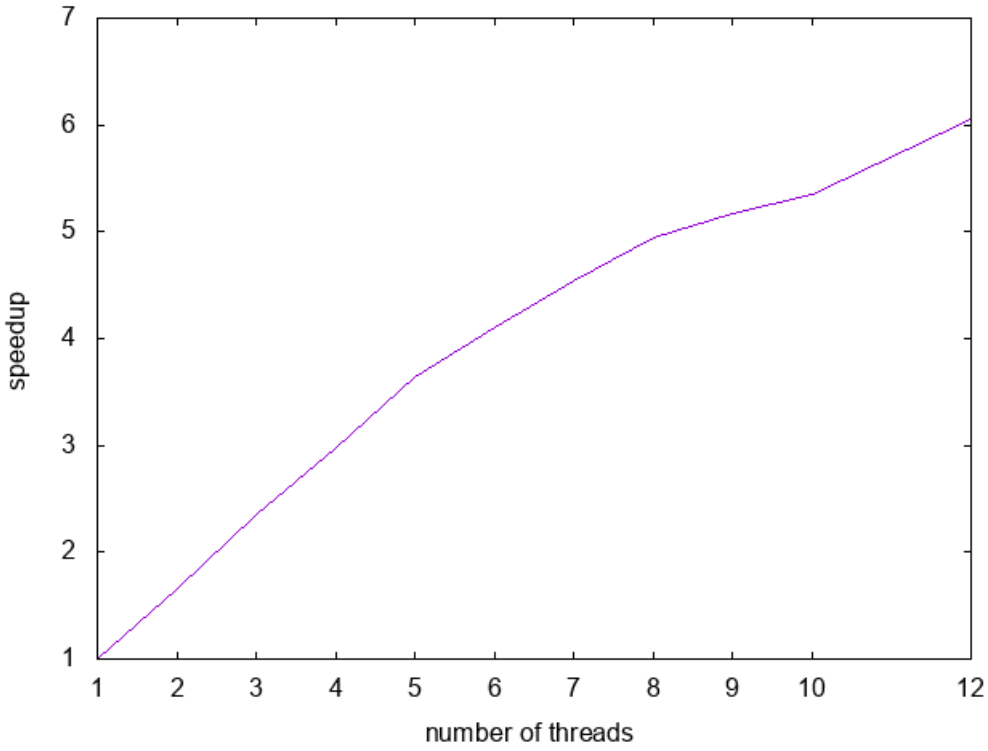
5.2.2 Транзакционная память

Функции `split` и `merge` - основные в `Treap`. Через них реализуются другие функции над структурой данных. В этих функциях и происходит непосредственное изменение структуры дерева, поэтому транзакции имеет смысл запускать именно в них (см. приложение 6).

Замеры времени работы `Treap` с `RTM`:

Потоки	Время исполнения	Погрешность
1	351210	5228
2	211438	4146
3	149176	2267
4	117722	1529
5	96612	575
6	85634	2742
7	77460	1998
8	71120	2908
9	67973	3952
10	65602	2340
12	57972	2092





Здесь наблюдается ускорение работы структуры данных с ростом числа потоков. Как себя ведет ускорение видно из графиков, при 12 потоков ускорение достигает 6.

Глава 6

Анализ результатов

Измерив время работы различных реализаций стека, можно сделать следующие выводы: блокирующая реализация стека работает медленнее неблокирующих. С ростом числа потоков время работы оптимистической реализации стека, стека с блокировками и с `hazard pointers` увеличивается. Однако реализация стека с `RTM` не только не замедляется, но и ускоряется при увеличении числа потоков. Таким образом, можно определить применимость реализаций — стек с `hazard pointers` логично использовать при небольшом числе потоков, а стек с `RTM` будет выгодно смотреться даже при большом числе потоков. Начиная с одного потока с увеличением их числа стек с `RTM` ускоряется, что делает данную реализацию наиболее оптимальной для использования.

Для структуры данных `Treap` было разработано две реализации — с блокировкой и с `RTM`. Блокирующая структура данных аналогично стеку с блокировками резко замедляется с ростом числа потоков, что делает ее невыгодной для использования. Крайне выгодно по сравнению с ней выглядит реализация `Treap`, основанная на `RTM`. Она не только не замедляется, но и работает быстрее.

На основе анализа работы различных реализаций двух структур данных (стека и `treap`) можно сделать выводы о преимуществе транзакционной памяти над остальными примитивами синхронизации. С помощью относительно простого инструмента — `Intel TSX` были разработаны неблокирующие структуры данных с параллельным доступом, которые не только работают быстрее других реализаций, но и ускоряются с увеличением числа потоков, в то время как для остальных реализаций

задача максимум — не замедляться с увеличением числа потоков.

Глава 7

Заключение

В данной работе была исследована возможность параллельных реализаций двух структур данных (стек и `treap`), построены и проанализированы их параллельные реализации. Среди построенных реализаций были как блокирующие, так и свободные от блокировок. После разработки и анализа стека и декартового дерева (`Treap`) с блокировками были получены результаты, свидетельствующие о невыгодности использования блокировок в параллельных структурах данных. В качестве основы для неблокирующих реализаций структур данных был предложен оптимистический подход на основе операций `CAS`. Данный подход реализуем только для стека, `Treap` при операциях вставки/удаления должен атомарно заменять сразу два указателя, что невозможно осуществить в архитектуре `Intel` при использовании `CAS`. Оптимистическая реализация стека, время работы которой слабо растет при увеличении числа потоков, обладает рядом слабостей, в частности она подвержена проблеме `ABA`. Для ее решения были использованы опасные указатели (`hazard pointers`). Стек с использованием опасных указателей работал стабильно, однако замедлялся с ростом числа потоков, что делает эту реализацию невыгодной.

Был проведен анализ транзакционной памяти для построения неблокирующих структур данных, в результате которого сделан вывод о том, что `Intel TSX`, а конкретно набор команд `RTM`, позволяют реализовать параллельные неблокирующие структуры данных как с одним, так и с несколькими указателями; при этом структуры данных не будут подвержены проблеме `ABA`. Было замерено время работы стека и `Treap` с ис-

пользованием RTM, результаты показали ускорение структур данных при увеличении числа потоков. Таким образом, главным результатом данной работы является разработка и реализации структур данных, предназначенных для параллельной работы без блокировок и замедления работы.

В дальнейшем, полезно замерить построенные структуры данных на более мощных компьютерах с большим количеством ядер, чтобы иметь более точные параметры ускорения работы структур данных при увеличении числа взаимодействующих потоков.

Кроме того, стоит обратить внимание на другие процессоры, например на архитектуре ARM. Архитектура ARM изначально проектировалась как RISC-архитектура, принципы построения машинных команд и инструментальные средства у неё значительно отличаются от x86/x64 архитектуры. О её жизнеспособности говорит тот факт, что 98% мобильных устройств основаны именно на ней. Впрочем, это тема уже для другого исследования.

Список литературы

[1] Andrei Alexandrescu, Maged Michael. “Lock-free data structures with hazard pointers”. *C++ User Journal* (2004), с. 17—20.

[2] *C++ concurrent data structure library*. URL: `http://libcds.sourceforge.net`.

[3] Maurice Herlihy, Nir Shavit. *Искусство многопроцессорного программирования*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2008. ISBN: 0123705916, 9780123705914.

[4] Maged M. Michael. “Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects”. *IEEE Trans. Parallel Distrib. Syst.* **15** 6 (июнь 2004), с. 491—504. ISSN: 1045-9219. DOI: `10.1109/TPDS.2004.8`. URL: `http://dx.doi.org/10.1109/TPDS.2004.8`.

[5] R. Seidel, C. R. Aragon. “Randomized search trees”. *Algorithmica* **16** 4 (окт. 1996), с. 464—497. ISSN: 1432-0541. DOI: `10.1007/BF01940876`. URL: `https://doi.org/10.1007/BF01940876`.

[6] Richard M. Yoo и др. “Performance Evaluation of Intel&Reg; Transactional Synchronization Extensions for High-performance Computing”. *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. SC ’13. Denver, Colorado: ACM, 2013, 19:1—19:11. ISBN: 978-1-4503-2378-9. DOI: `10.1145/2503210.2503232`. URL: `http://doi.acm.org/10.1145/2503210.2503232`.

[7] Тормасов А.Г. *Параллельное программирование многопоточных систем с разделяемой памятью*. Физматкнига, 2014. ISBN: 9785891552357.

[8] Э.С. Таненбаум. *Современные операционные системы: [перевод]*. Классика computer science: КС. Питер, 2010. ISBN: 9785498073064. URL: `https://books.google.ru/books?id=60p9kgEACAAJ`.

[9] Э. Уильямс. *Параллельное программирование на C++ в действии. Практика разработки многопоточных программ.* ЛитРес, 2017. ISBN: 9785457427020. URL: https://books.google.ru/books?id=1UXRAAAAQBAJ.

Стек с блокировками

```
1 void push(int d)
2 {
3     Node *pv = new Node;
4     m.lock ();
5     pv->d = d;
6     pv->p = head;
7     head = pv;
8     m.unlock ();
9 }
10
11 int pop ()
12 {
13     if (head == NULL) return NULL;
14     m.lock ();
15     int temp = head->d;
16     Node *pv = head;
17     head = head->p;
18     m.unlock ();
19     delete pv;
20     return temp;
21 }
```

Приложение 1. Стек с блокировками

Стек с CAS

```
1 void push (const int& data)
2 {
3     LockFreeStack* newStack = new LockFreeStack (data);
4     newStack->next = head.load ();
5
6     while (!head.compare_exchange_strong (newStack->next, newStack))
7         ;
8 }
9
10 LockFreeStack* pop ()
11 {
12     if (head == nullptr)
13     {
14         printf ("Stack if EMPTY!!! You can not pop from it! \n");
15         return 0;
16     }
17     LockFreeStack* newStack = head.load ();
18     LockFreeStack* poppedNode = new LockFreeStack (newStack->data);
19     while (!head.compare_exchange_strong (newStack, newStack->next))
20     {
21         poppedNode->data = newStack->data;
22     }
23     return poppedNode;
24 }
```

Приложение 2. Стек с CAS

Стек с hazard pointers

```
1 LockFreeStack* pop ()
2 {
3     if (head == nullptr)
4     {
5         printf ("Stack if EMPTY!!! You can not pop from it! \n");
6         return 0;
7     }
8     try
9     {
10        cds::gc::HP::Guard guard  = cds::gc::HP::Guard() ;
11        LockFreeStack* newStack = guard.protect (head);
12        LockFreeStack* poppedNode = new LockFreeStack (newStack->data);
13        while (!head.compare_exchange_strong (newStack, newStack->next))
14        {
15            poppedNode->data = newStack->data;
16        }
17        guard.release ();
18        return poppedNode;
19    }
20    catch (cds::gc::HP::not_enought_hazard_ptr_exception)
21    {
22        printf ("...!!!\n");
23    }
24 }
```

Приложение 3. Стек с hazard pointers

Стек с RTM

```
1 void push(int d)
2 {
3     Node *pv = new Node;
4     pv->d = 0;
5     unsigned status = _xbegin ();
6
7     if (status == _XBEGIN_STARTED)
8     {
9         pv->d = d;
10        pv->p = head;
11        head = pv;
12        _xend ();
13    }
14 }
15
16 int pop ()
17 {
18    if (head == NULL) return NULL;
19    int temp = -1;
20    Node *pv = NULL;
21    unsigned status = _xbegin ();
22    if (status == _XBEGIN_STARTED)
23    {
24        temp = head->d;
25        pv = head;
26        head = head->p;
27        _xend();
28    }
29    if (pv != NULL) delete pv;
30    return temp;
31 }
```

Приложение 4. Стек с RTM

Treap с блокировками

```
1 void erase (treap& t, int key)
2 {
3     std::lock_guard<std::recursive_mutex> lg(m);
4     if (t != NULL)
5     {
6         if (t->key == key)
7         {
8             t = merge (t->left, t->right);
9         }
10        else
11        {
12            if (key < t->key)
13            {
14                erase (t->left, key);
15            }
16            else
17            {
18                erase (t->right, key);
19            }
20        }
21    }
22 }
23
24 void insert (treap& t, treap toInsert)
25 {
26    std::lock_guard<std::recursive_mutex> lg(m);
27    if (t == nullptr) t = toInsert;
28    else if (toInsert->priority > t->priority)
29    {
30        treap dupl;
31        auto tmp = split (t, toInsert->key, &dupl);
32        toInsert->left = tmp.first;
33        toInsert->right = tmp.second;
34        t = toInsert;
35    }
36    else
37    {
```

```
38          if (toInsert->key < t->key)
39          {
40              insert (t->left, toInsert);
41          }
42          else
43          {
44              insert (t->right, toInsert);
45          }
46      }
47  }
```

Приложение 5. Treap с блокировками

Treap с RTM

```
1 void split (treap root, treap& left, treap& right, int key, treap* dupl)
2 {
3     if (root == nullptr)
4     {
5         left  = nullptr;
6         right = nullptr;
7         return;
8     }
9     if (root->key < key)
10    {
11        (*dupl) = nullptr;
12        split (root->right, root, root->right, key, dupl);
13    }
14    else if (root->key > key)
15    {
16        (*dupl) = nullptr;
17        split (root->left, root->left, root, key, dupl);
18    }
19    else
20    {
21        auto volatile v = *root;
22        auto volatile dv = *dupl;
23        auto volatile vl = left;
24        auto volatile vr = right;
25        auto volatile vll = root->left;
26        auto volatile vlr = root->right;
27        unsigned status = _xbegin ();
28        if (status == _XBEGIN_STARTED)
29        {
30            (*dupl) = root;
31            left    = root->left;
32            right   = root->right;
33            _xend ();
34        }
35    }
36 }
37
```

```
38 void merge (treap left, treap right, treap& result)
39 {
40     if (left == nullptr || right == nullptr)
41     {
42         if (right == nullptr) result = left;
43         else result = right;
44         return;
45     }
46     if (left->key > right->key)
47     {
48         auto volatile v = *result;
49         auto volatile vl = *left;
50         auto volatile vr = *right;
51         unsigned status = _xbegin ();
52         if (status == _XBEGIN_STARTED)
53         {
54             std::swap (left, right);
55             _xend ();
56         }
57         return;
58     }
59     if (left->priority > right->priority)
60     {
61         merge (left->right, right, left->right);
62         result = left;
63         return;
64     }
65     else
66     {
67         merge (left, right->left, right->left);
68         result = right;
69         return;
70     }
71 }
```

Приложение 6. Treap с RTM