Аннотация

Здесь идет аннотация

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 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 блоки-

рующими (с использованием мьютексов, семафоров и т.п.) и неблокирующими. Последние работают быстрее, но требуют корректной работы с памятью, поэтому важно умение правильным образом разрабатывать структуры данных для таких алгоритмов.

Данная работа является исследованием различных способов построения неблокирующих структур данных в параллельном программировании.

Глава 2

Постановка задачи

В этой работе изучаются разные реализации некоторых параллельных структур данных, сранивается их эффективность и отказоустойчивость:

- 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 память (`Intel TSX`) и использование `RTM (Restricted Transactional Memory)` для построения неблокирующих структур данных
- Тестирование 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 уменьшению размеров транзистора препятствуют законы физики — в силу вступают 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` или `Hyperthreading` по версии `Intel`). В книге описаны различные проблемы, связанные с взаимодействием потоков (например, `race conditions` — состояние гонок потоков), в том числе сложности с совместной работой с данными, отсюда была осознана всю 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 структур данных (с блокировками и без).

Для построения потокобезопасной структуры данных необходимо ис-

пользование атомарных типов данных и операций. Атомарные операции — неделимые, то есть операция либо выполнена окончательно, либо не выполнена совсем. В C++ атомарные операции возможно совершать только с данными атомарных типов (`std::atomic`). Кроме них, содержимое структуры данных защищено мьютексами и блокировками. В зависимости от строения структуры данных, может иметь место крупная и мелкая гранулярность (`coarse-grained` и `fine-grained`). Крупногранулярные структуры данных (например, список с блокировкой его головы) устроены так, что блокируются большие участки памяти. В мелкогранулярных структурах данных (например, список с отдельной блокировкой каждого элемента списка) блокируются более мелкие участки памяти, что несколько ускоряет параллельный доступ к объекту. Однако, и такие структуры данных работают медленно и параллельный код дает слабый выигрыш в производительности.

В структуре данных без блокировок не используются мьютексы или семафоры. Это позволяет избежать ряда проблем, присущих структурам данных с блокировками, однако усложняет написание структуры данных. Неблокирующая структура данных становится открытой для одновременного доступа со стороны сразу нескольких потоков (при этом необязательно эти потоки будут совершать одинаковые операции). Корректно построить такую структуру данных довольно сложно, на каждом шаге требуется, чтобы хоть какой-то поток продвигался вперед. Среди неблокирующих структур данных выделяют структуры данных, свободные от ожидания. 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, написание неблокирующих структур данных — процесс достаточно тяжелый, к тому же у таких структур данных имеются свои недостатки. Несмотря на то, что они позволяют лучше распараллелить 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 описано неоптимально. Опираясь на эти выводы, было принято решение рассмотреть две структуры данных — стек и декартово дерево (`Treap`), и построить их неблокирующие реализации.

В книге Херлихая «Искусство многопроцессорного программирования» [3] исследованы параллельные структуры данных очередь и стек. Наиболее наивная неблокирующая реализация 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 записываем модифицированные данные. В случае, если сторонний поток за это время уже внес какие-либо изменения, начинаем работу заново, то есть снова делаем снимок текущего состояния и т.д. Чтобы проверить, изменилась ли структура, 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.

Но такой подход несет в себе опасность — структура данных будет подвержена ошибке `ABA`. Это ошибка, которая возникает при работе с памятью в структуре данных, когда ячейка памяти читается дважды и дважды считывается одно и то же значение, что интерпретируется будто изменения не были внесены. Однако второй поток мог между двумя считываниями изменить значение, работать и восстановить предыдущее значение в ячейке памяти.

Рассмотрим проблему `ABA` применительно к неблокирующему стеку. Изначально стек содержит $head \longrightarrow A \longrightarrow B \longrightarrow C$. Первый поток выполняет операцию `pop`, которая возвращает A, head должен указывать на B. Одновременно с этим операцию `pop` дважды выполняет второй поток, при этом он успевает сделать `CAS` раньше первого потока, поэтому в стеке сначала будет $head \longrightarrow B \longrightarrow C$, затем $head \longrightarrow C$. Затем второй поток добавляет A в стек, превращая его в $head \longrightarrow A \longrightarrow C$.

Затем уже первый поток продолжает работу (стоит напомнить, что первому потоку осталось выполнить `CAS`, заменив указатель head с A на B. Поскольку в текущем состоянии стек указывает на A, то операция `CAS` успешно исполнится, хотя сам стек уже изменился и следует откатиться к предыдущему состоянию и повторить выполнение операции `pop` первым потоком. Вместо этого элемент A будет заменен на B, стек будет иметь вид $head \longrightarrow B \longrightarrow C$, что не соответствует действительности.

Более подробно проблема `ABA` описана в книге . 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 писатель делает себе полную копию, меняет эту копию и атомарно меняет указатель на свою копию. 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 коммитит изменения. Однако, замена указателя должна происходить «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 работоспособности системы. На уровне ядра операционной системы проблема решается — управлением занимается планировщик потоков, а поток-писатель находится в ожидании команд от процессора. Но структуры данных, требующие использование `RCU`, находятся в пользовательском адресном пространстве (`user-space`), в ко-

тором явный вызов примитивов планировщика недоступен. Приходится прибегать к использованию так называемых user-space RCU, которые представляют собой совершенно нетривиальные структуры с большим объемом кода и использование такого механизма приводит к большой потере времени исполнения, делая борьбу с блокировками непрактичной.

Среди других способов решения проблемы ABA — 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 когда только 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 CAS на Double-length CAS достаточно элегантно решает проблему ABA, храня во второй части счетчик. Сравнение тогда ведется по предыдущему значению и счетчику 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 машин не поддерживают 128-битный CAS, что не позволяет считать Double-length CAS эффективным решением проблемы ABA. В отличие от другого примитива аппаратно синхронизации — LL/SC (Load linked/Store conditional). Это пара инструкций, первая из которых возвращает значение ячейки памяти, второе записывает туда свое значение тогда и только тогда, когда между выполнением второй и первой инструкций не было других записей в ячейку памяти. При всех плюсах такого механизма (в отличие от CAS, LL/SC не подвержен проблеме ABA), эти инструкции не реализованы в архитектуре Intel.

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.

Однако непосредственно реализация опасных указателей не входила в планы данной работы, поэтому было принято решение искать 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 реализации сложной структуры данных. В качестве примера подобной структуры данных было взято декартово дерево (`cartesian tree` или `Treap`). Эта структура данных была описана в статье Seidel, Aragon «Randomised search trees» [5]. Эта структура данных объединяет в себе бинарное дерево поиска и бинарную кучу. Говоря более строго, эта структура данных хранит пары (x, y) так, 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.

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. Реализация такой функции будет, очевидно, основана на рекурсии: 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 струк-

туру (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 выполняем операцию `merge`, передав в качестве аргументов правое поддерево первого дерева и второе дерево. Если приоритет корня второго дерева выше, то выполняем 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 подлходящих 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 — подобные этим операции не могут выполняться внутри 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 имплементации 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 внедрена поддержка многопоточного программирования (`std::thread`), что является ключевым для данного исследования.

Компилятором для исходников в данной работе был выбран `g++`. Данный компилятор используется большинством разработчиков и является стабильным, поскольку поддерживается `Free Software Foundation` с 1987 года. Используется компьютер с процессором `Intel Core i5 7267U, Kaby 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) работы структуры данных.

Тестирование структуры данных заключается во вставке и удалении большого количества данных. 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

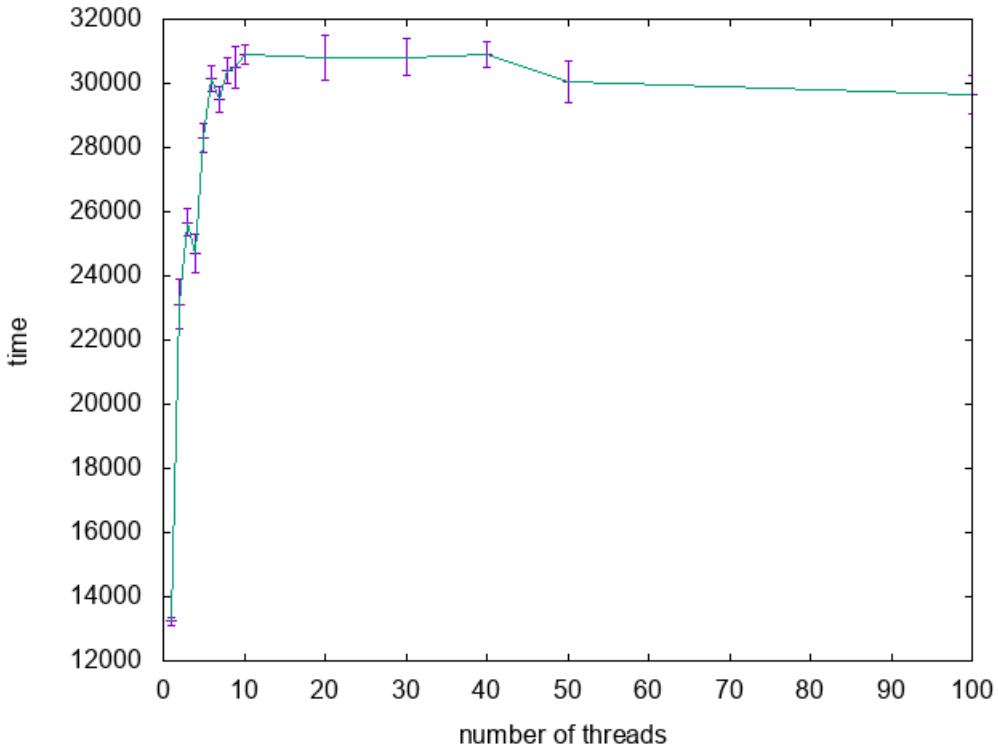
Стек был реализован с помощью опасных указателей (`hazard pointers`) и транзакционной памяти (`RTM`). Однако, полезно для сравнения иметь реализации с использованием блокировок (`mutex`) и оптимистические, то есть только с использованием `CAS`.

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	13227	137
2	23106	776
3	25672	413
4	24689	587
5	28303	456
6	30145	394
7	29513	398
8	30378	398
9	30490	651
10	30900	281
20	30802	703
30	30819	558
40	30880	403
50	30056	647
100	29664	598



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 становит-

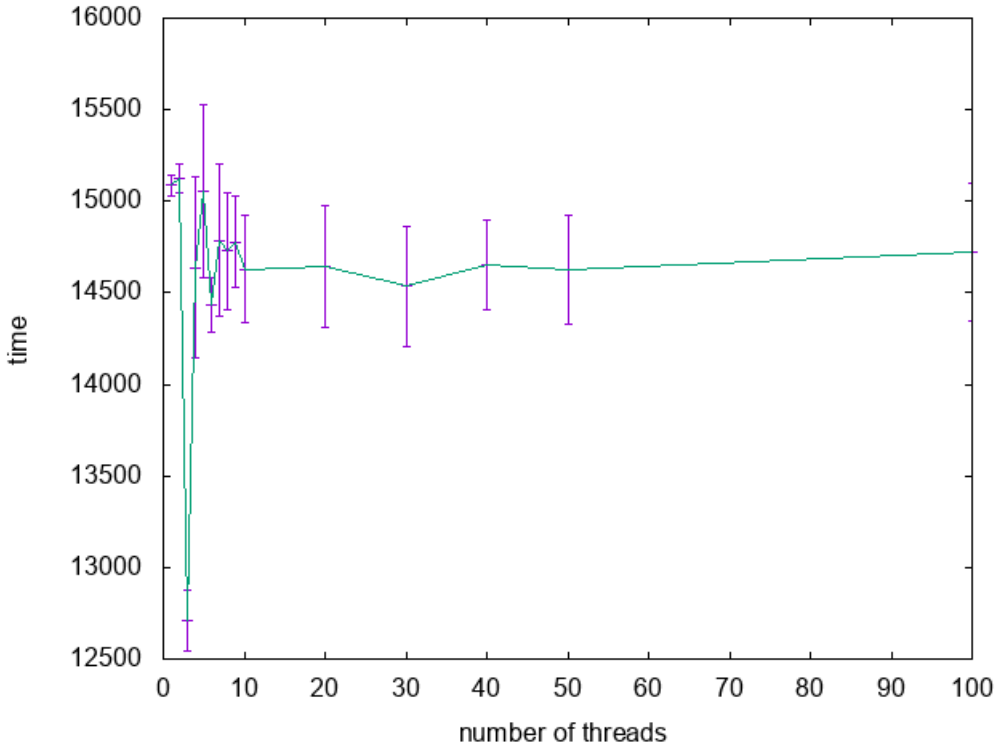
ся все выше, увеличивая время ожиждания освобождения элементов от блокировок.

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	15086	59
2	15121	79
3	12707	166
4	14638	495
5	15054	471
6	14433	149
7	14788	418
8	14727	323
9	14778	249
10	14630	296
20	14643	329
30	14538	328
40	14651	243
50	14628	298
100	14720	377



Поведение данной реализации структуры данных стоит взять за образец, поскольку с увеличением числа потоков время работы практически не увеличивается. Исключение составляют первые четыре величины, на которых наблюдается не очень значительное, но все же замедление работы. Это стоит связать с машиной, на которой проводилось тестирование; в рамках четырех потоков структура данных будет работать быстрее, ведь непосредственное исполнение операций может быть назначено ядрам процессора, тогда как при числе потоков больше четырех процессору необходимо организовывать взаимодействие между ними.

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	18258	300
2	18013	230
3	17266	322
4	18711	220
5	18933	477
6	19765	616
7	21307	1098
8	22189	881
9	23053	591
10	24213	472
20	38671	20383
30	84356	35415
40	207202	15507
50	332438	30644
100	781517	42787

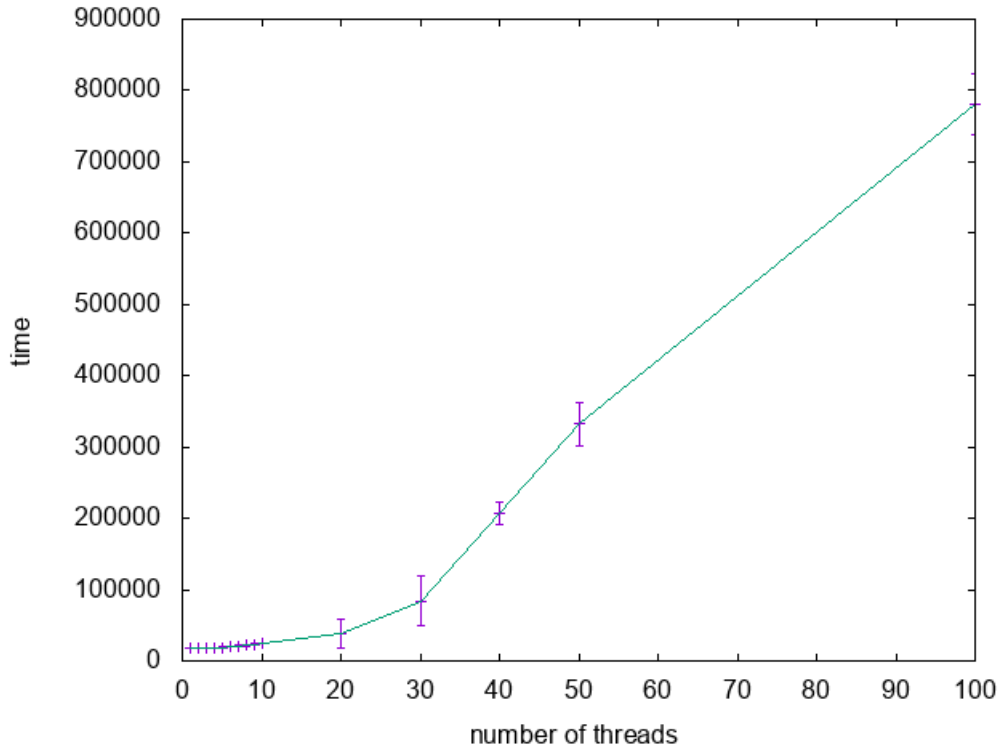


График показывает драматический рост времени работы при увеличении числа потоков. Подобное поведение связано с особенностями устройства сторонней библиотеки `libcds`, которая использовалась непосредственно для работы с опасными указателями. Выделение памяти для

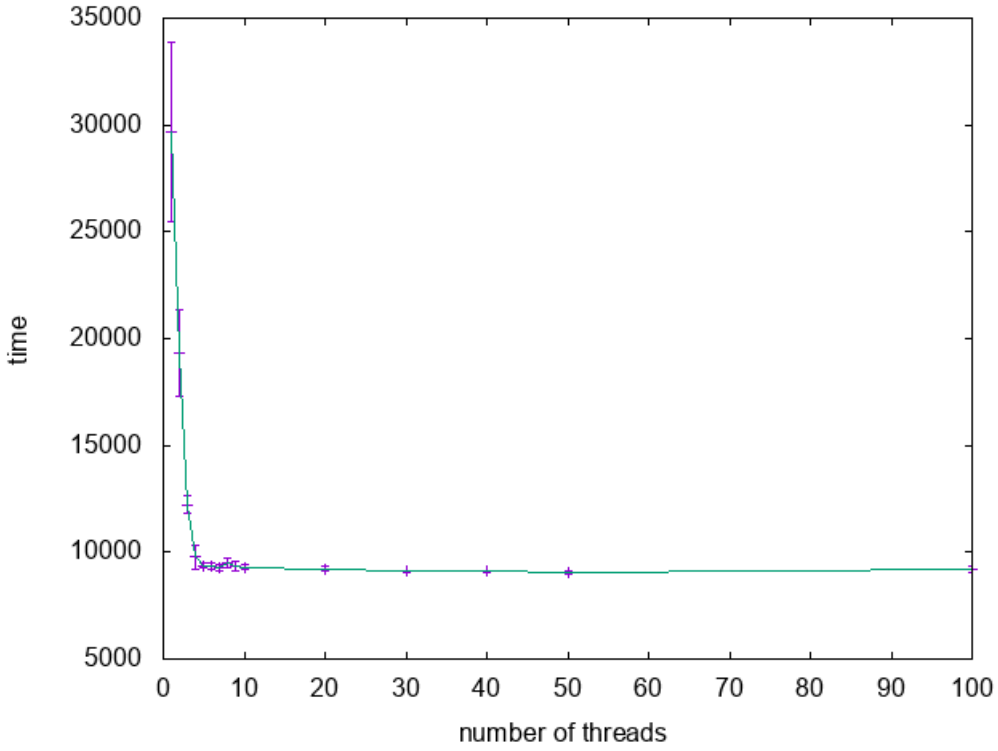
бóльшего числа указателей, их сканирование и удаление затрачивают ресурсы процессора и существенно замедляют структуру данных.

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 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	29652	4208
2	19336	2005
3	12221	395
4	9764	551
5	9364	111
6	9350	125
7	9252	139
8	9513	204
9	9354	208
10	9309	101
20	9236	141
30	9097	49
40	9088	68
50	9040	62
100	9190	132



Стек с `RTM` работает быстрее с увеличением числа потоков, особенно это заметно при числе потоков 1 — 4, что объясняется особенностями компьютера (а именно — числом ядер процессора), на котором проводилось тестирование.

5.2 Декартово дерево

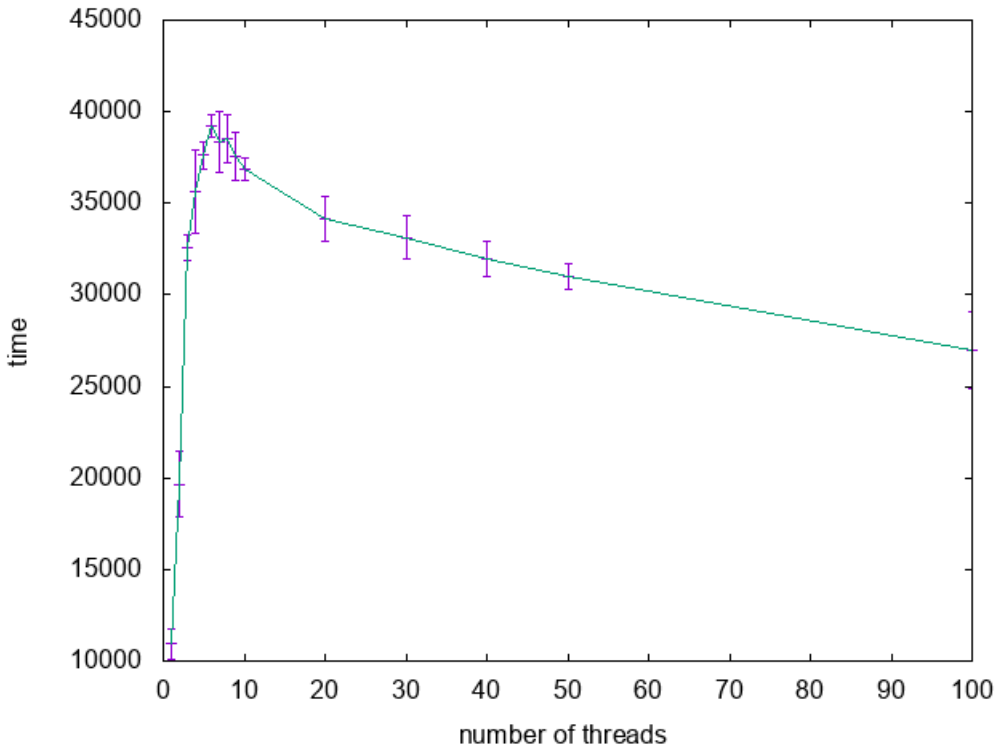
`Treap` был реализован с помощью `RTM`, для сравнения была разработана реализация `Treap` с использованием блокировок.

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	10943	818
2	19664	1755
3	32572	689
4	35625	2296
5	37623	730
6	39199	614
7	38343	1666
8	38534	1300
9	37549	1335
10	36848	620
20	34122	1212
30	33130	1163
40	31969	957
50	30962	701
100	26992	2080



Аналогично стеку, `Treap` с блокировками замедляется при увеличении числа потоков. Чем больше потоков - тем чаще будут обращения

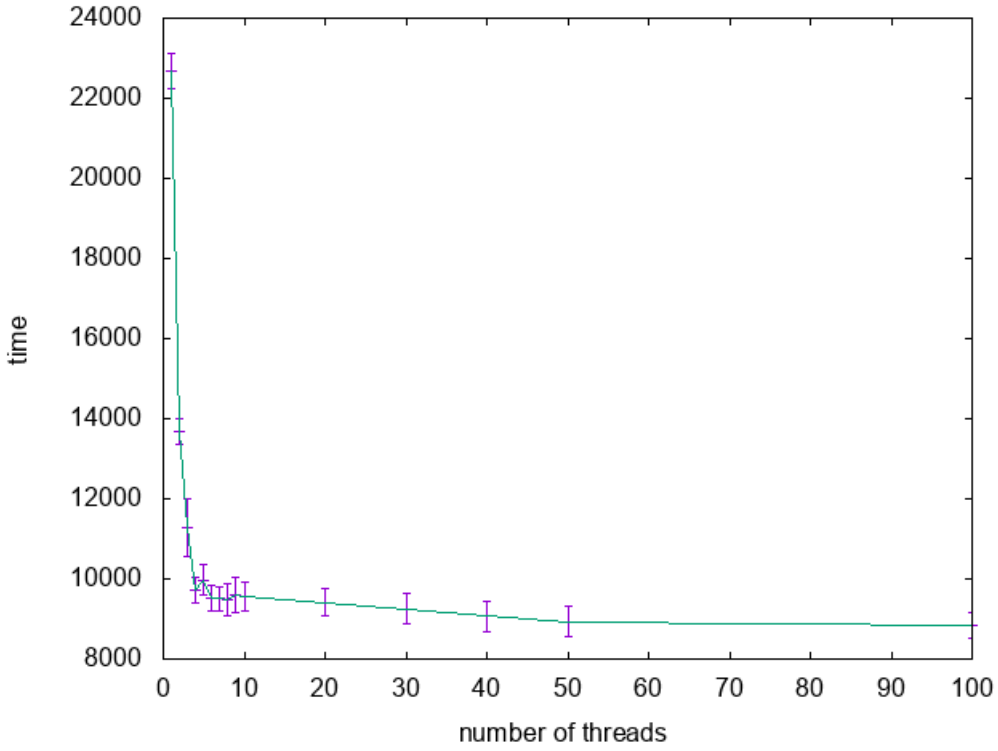
разных потоков к одним и тем же элементам структуры и тем больше они будут ждать друг друга, что и является причиной увеличения времени работы.

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 непосредственная работа с данными, поэтому транзакции имеет смысл запускать именно в них (см. приложение 6).

Замеры времени работы `Treap` с `RTM`:

Потоки	Время исполнения	Погрешность
1	22676	439
2	13679	304
3	11267	719
4	9708	327
5	9969	386
6	9535	323
7	9500	317
8	9482	394
9	9594	432
10	9572	356
20	9413	348
30	9258	396
40	9072	378
50	8937	373
100	8843	328



Здесь наблюдается ускорение работы структуры данных с ростом числа потоков, аналогично стеку это ускорение особенно сильно при числе потоков 1 — 4.

Глава 6

Анализ результатов

Измерив время работы различных реализаций стека, можно сделать следующие выводы: при малом числе потоков стек с блокировками будет работать на одном уровне с оптимистической реализацией стека, реализацией с использованием `hazard pointers` и `RTM`. С ростом числа потоков время работы оптимистической реализации практически не меняется, в то время как стек с блокировками и с `hazard pointers` замедляются (касательно стека с `hazard pointers` — замедление наблюдается только при числе потоков больше 10). 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, что особенно заметно на числе потоков от 1 до 4.

На основе анализа работы различных реализаций двух структур данных (стека и `treap`) можно сделать выводы о преимуществе транзак-

ционной 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 были исследованы две структуры 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. Оптимистическая реализация стека, время работы которой практически не меняется 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, но при большом числе процессов резко замедлялся, 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 и 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 использованием RTM, результаты показали ускорение структур данных при

увеличении числа потоков. Таким образом, главным результатом данной работы являются структуры данных, предназначенные 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`. Эти процессоры отличаются от `Intel` командами и инструментами. Однако процессоры на технологии `ARM` используются в 98% мобильных устройств, таким образом, актуальность подобного исследования очевидна.

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