Федеральное агентство связи
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Сибирский государственный университет телекоммуникаций и информатики»
(СибГУТИ)

Кафедра вычислительных систем
Допустить к защите

Зав.каф. _____*Мамойленко С.Н.*

# ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА БАКАЛАВРА

Разработка тестовых многопоточных программ на основе транзакционной памяти

Пояснительная записка

Студент _____ /Токарь Т.М./

Факультет ИВТ Группа ИУ-323

Руководитель _____ /Пазников А.А./

Новосибирск 2016 г.

Федеральное агентство связи
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Сибирский государственный университет телекоммуникаций и информатики»
(СибГУТИ)

**КАФЕДРА**

*вычислительных систем*

# ЗАДАНИЕ
# НА ВЫПУСКНУЮ КВАЛИФИКАЦИОННУЮ РАБОТУ
# БАКАЛАВРА

СТУДЕНТА Токарь Т.М. ГРУППЫ ИУ-323

**УТВЕРЖДАЮ**

« ___ » _____ 20__ г.

Зав. кафедрой
_____ / *С.Н. Мамойленко* /

Новосибирск 2016 г.

**1. Тема выпускной квалификационной работы бакалавра**
Разработка тестовых многопоточных программ на основе транзакционной памяти

утверждена приказом СибГУТИ от «20» февраля 2016 г. № **4/159о-16**

**2.Срок сдачи студентом законченной работы** «10» июня 2016 г.

**3.Исходные данные к работе**

- Riegel T., Fetzer C., Felber P. Time-based transactional memory with scalable time bases. ACM. 2007. 221-228 с.
- T. Riegel. Software Transactional Memory Building Blocks. PhD thesis, Technischen Universitat Dresden, geborenam 1.3.1979 in Dresden, 2013.
- Herlihy M., Moss J. E. B. Transactional memory: Architectural support for lock-free data structures. ACM, 1993, № 2, 289-300 с.
- Dave Boutcher. University of Minnesota. Software Transactional Memory in GCC. URL: http://www-users.cs.umn.edu/~boutcher/stm/ (дата обращения: 10.02.2016)
- Hans Boehm, Justin Gottschlich, Victor Luchangco, Maged Michael, Mark Moir, Clark Nelson, Torvald Riegel, Tatiana Shpeisman, Michael Wong. Transactional Language Constructs for C++. URL: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3341.pdf (дата обращения 05.02.2016).

| **4.Содержание пояснительной записки (перечень подлежащих разработке вопросов)** | **Сроки выполнения по разделам** |
|---|---|
| Введение | 03.02.2016 |
| Анализ предметной области и постановка задачи | 05.02.2016 |
| Выбор средств решения поставленной задачи | 02.03.2016 |
| Разработка потокобезопасной программы | 20.03.2016 |
| Проведение экспериментов | 20.04.2015 |
| Заключение | 30.05.2016 |

Дата выдачи задания «__» _____ 20__ г.
Руководитель _____

Задание принял к исполнению «__» _____ 20__ г.
Студент _____

# АННОТАЦИЯ

Выпускная квалификационная работа Токарь Т.М. по теме «разработка тестовых многопоточных программ на основе транзакционной памяти»

Объём работы - *52* страница, на которых размещены *11* рисунков. При написании работы использовалось *35* источника.

Ключевые слова: транзакционная память, многопоточное программирование, синхронизация многопоточных программ, потокобезопасные структуры данных.

Работа выполнена на кафедре вычислительных систем СибГУТИ

Реализован потокобезопасный красно-черное дерево на основе транзакционной памяти. Проведено моделирование, в ходе которого было установлено, что использование программной транзакционной памяти - эффективно, так как производительность программы, использующей транзакционную память сопоставима с реализации на мьютексах. Так же было выяснено что программная транзакционная память является эффективней чем мьютексы до 8 потоков.

**Федеральное государственное бюджетное образовательное учреждение**
**высшего образования**
**«Сибирский государственный университет телекоммуникаций и информатики»**
**(СибГУТИ)**

**Форма утверждена научно-методическим**
**советом университета**
**протокол № 3 от 19.02.2015 г.**

---

# ОТЗЫВ

на выпускную квалификационную работу студента Токарь Т.М. по теме «Разработка тестовых многопоточных программ на основе транзакционной памяти»

Одна из ключевых проблем, которые возникают при разработке многопоточных программ для вычислительных систем с общей памятью, – это гонки данных (data race), возникающие при совместном доступе параллельных потоков к общими ресурсами. На сегодняшний день основными методами решения данной проблемы являются: применение блокировок (mutex, spinlock и т.д.), алгоритмы и структуры данных, свободные от блокировок (lockless, lock-free) и транзакционная память (transactional memory).

Транзакционная память относится к перспективным средствам синхронизации потоков и предоставляет механизм, который позволяет организовывать транзакции в рамках которой выполняется доступ к разделенным областям памяти. Транзакции выполняются без блокирования потоков.

Токарь Т.М. разработал потокобезопасное красно-черное дерево на основе транзакционной памяти. Проведено моделирование потокобезопасных деревьев на основе различных примитивов синхронизаций потоков, представлены рекомендации по использованию программной транзакционной памяти. Результат эксперимента показал, что применение программной транзакционной памяти - эффективно. Разработанная потокобезопасная структура реализована в отдельной статической библиотеке.

Существенных замечаний по работе – нет.

Считаю, что выпускная квалификационная работа заслуживает оценки «отлично», а Токарь Т.М. – присвоения квалификации бакалавр по направлению 09.03.01 «Информатика и вычислительная техника».

Оценка уровней сформированности общекультурных и профессиональных компетенций обучающегося:

| Компетенции | | Уровень сформированности компетенции | | |
|---|---|---|---|---|
| | | Высокий | Средний | Низкий |
| Общекультурные | ОК-7 | X | | |

| | | | |
|---|---|---|---|
| Работа имеет практическую ценность | X | Тема предложена предприятием | |
| Работа внедрена | | Тема предложена студентом | X |
| Рекомендую работу к внедрению | X | Тема является фундаментальной | X |
| Рекомендую работу к опубликованию | X | Рекомендую студента в магистратуру | X |
| Работа выполнена с применением ЭВМ | X | Рекомендую студента в аспирантуру | |

Доцент кафедры _____ Пазников А.А.

Оглавление

1 ВВЕДЕНИЕ ..... 7
2 ПРЕДМЕТНАЯ ОБЛАСТЬ ..... 8
2.1 Вычислительные системы ..... 8
2.2 Организация управления доступом к разделяемой памяти .. 13
2.3 Параллельное программирование ..... 29
3 ПРОЕКТИРОВАНИЕ И РЕАЛИЗАЦИЯ ..... 31
3.1 Постановка задачи ..... 31
3.2 Инструментарий ..... 31
3.3 Библиотрека libitm ..... 31
3.4 Красно-черное дерево ..... 32
3.5 Реализация ..... 32
3.6 Описание библиотеки librbt-tm ..... 33
3.7 Сборка библиотеки librbt-tm ..... 33
3.8 Алгоритм выбора случайных операций потоками ..... 34
3.9 Скрипт для автомаизации тестирования ..... 35
3.10 Описание тестов ..... 37
3.11 Описание системы на которой проводились эксперименты 37
3.12 Результаты моделирования ..... 37
3.13 Обработка результатов измерений ..... 40
4 ЗАКЛЮЧЕНИЕ ..... 42
ПРИЛОЖЕНИЕ А ..... 43
ПРИЛОЖЕНИЕ Б ..... 46
ПРИЛОЖЕНИЕ В ..... 47

# 1 ВВЕДЕНИЕ

Идея распараллеливания вычислений основана на том, что большинство задач может быть разделено на набор меньших задач, которые могут быть решены одновременно. Обычно параллельные вычисления требуют координации действий. Параллельные вычисления существуют в нескольких формах: параллелизм на уровне битов, параллелизм на уровне инструкций, параллелизм данных, параллелизм задач. Параллельные вычисления использовались много лет в основном в высокопроизводительных вычислениях, но в последнее время к ним возрос интерес вследствие существования физических ограничений на рост тактовой частоты процессоров. Параллельные вычисления стали доминирующей парадигмой в архитектуре компьютеров, в основном в форме многоядерных процессоров.

С развитие многоядерных вычислительных систем с общей памятью и как следствие разработка многопоточных, потокобезопасных программ, структур данных для таких вычислительных систем возникла сложность написание многопоточных, потококобезопасных программ, структур данных. Один из вариантов решения этой проблемы является программная транзакционная память.

Транзакционная память (Software Transactional Memory, STM) - технология синхронизации конкурентных потоков. Она упрощает параллельное программирование, выделяя группы инструкций в атомарные транзакции. Конкурентные потоки работают параллельно, пока не начинают модифицировать один и тот же участок памяти.

Транзакционная память в настоящее время является одним из наиболее перспективных средств синхронизации потоков в параллельных программах. В рамках транзакционной памяти программисту предоставляются языковые конструкции для формирования в программе транзакционных секций – участков кода, в которых выполняется совместный доступ к разделяемым областям памяти.

В отличие от традиционных средств синхронизации (мьютексы, спинлоки, семафоры и т.д.), транзакционная память обеспечивает защиту не участков кода, а областей памяти в рамках транзакционных секций. Последнее позволяет выполнять транзакции без блокировки потоков, что повышает масштабируемость программ. Кроме того, в программах на основе транзакционной памяти отсутствуют взаимные блокировки (deadlocks, livelocks) и состояния гонки за данными (data race), а оптимизация выполнения транзакций реализуется на уровне компилятора. По сравнению с методами многопоточного программирования без использования блокировок на основе атомарных операций, использование транзакционной памяти значительно легче и не требует решения проблем, связанных с освобождением памяти (ABA-problem) и наличием в программе «узких мест» (bottlenecks).

# 2 ПРЕДМЕТНАЯ ОБЛАСТЬ

## 2.1 Вычислительные системы

Концептуальное представление о средствах обработки информации, получивших название (параллельных) вычислительных систем, базируется на модели коллектива вычислителей, на понятиях параллельного алгоритма, архитектуры и макроструктуры ВС.

Логика развития средств обработки информации и дуализм понятия “вычислитель” порождают понятие “коллектив вычислителей”, допускающее двойное толкование: и как ансамбль людей, занятых расчетами, и как система аппаратурно-программных средств для обработки информации. Архитектура ВС основывается на структурной и функциональной имитации коллектива (ансамбля) людей-вычислителей. Коллектив вычислителей обладает свойствами системы и является средством, которое наиболее удобно и перспективно для решения сложных (системных) задач. В дальнейшем под вычислительной системой будем понимать средство обработки информации, базирующееся на модели коллектива вычислителей. Следовательно, ВС – это композиция аппаратурно-программных средств, предназначенная для параллельной обработки данных.

При достаточно общей трактовке под вычислительной системой понимается совокупность взаимосвязанных и одновременно функционирующих аппаратурно-программных вычислителей, которая способна не только реализовать (параллельный) процесс решения сложной задачи, но и априори и в процессе работы автоматически настраиваться и перестраиваться с целью достижения адекватности между своей структурно-функциональной организацией и структурой и характеристиками решаемой задачи.

Мультипроцессорные ВС – обширная группа систем, в которую, в частности, могут быть включены конвейерные и матричные ВС (а также многомашинные ВС). Однако принято к мультипроцессорным ВС относить системы с MIMD-архитектурой, которые состоят из множества (не связанных друг с другом) процессоров и общей (возможно и секционированной, модульной) памяти; взаимодействие между процессорами и памятью осуществляется через коммутатор (общую шину и т.п.), а между процессорами – через память.

Создание ВС преследует следующие основные цели: повышение производительности системы за счет ускорения процессов обработки данных, повышение надежности и достоверности вычислений, предоставление пользователям дополнительных сервисных услуг и т.д.

Параллелизм в вычислениях в значительной степени усложняет управление вычислительным процессом, использование технических и программных ресурсов. Эти функции выполняет операционная система ВС.

### 2.1.1 Классификация вычислительных систем

Одним из наиболее распространенных способов классификации ЭВМ является систематика Флинна (Flynn), в рамках которой основное внимание при анализе архитектуры вычислительных систем уделяется способам взаимодействия

последовательностей (потоков) выполняемых команд и обрабатываемых данных. При таком подходе различают следующие основные типы систем.

SISD (Single Instruction, Single Data) – системы, в которых существует одиночный поток команд и одиночный поток данных. К такому типу можно отнести обычные последовательные ЭВМ;

SIMD (Single Instruction, Multiple Data) – системы с одиночным потоком команд и множественным потоком данных. Подобный класс составляют многопроцессорные вычислительные системы, в которых в каждый момент времени может выполняться одна и та же команда для обработки нескольких информационных элементов; такой архитектурой обладают, например, многопроцессорные системы с единым устройством управления. Этот подход широко использовался в предшествующие годы (системы ILLIAC IV или CM-1 компании Thinking Machines), в последнее время его применение ограничено, в основном, созданием специализированных систем;

MISD (Multiple Instruction, Single Data) – системы, в которых существует множественный поток команд и одиночный поток данных. Относительно этого типа систем нет единого мнения: ряд специалистов считает, что примеров конкретных ЭВМ, соответствующих данному типу вычислительных систем, не существует и введение подобного класса предпринимается для полноты классификации; другие же относят к данному типу, например, систолические вычислительные системы или системы с конвейерной обработкой данных;

MIMD (Multiple Instruction, Multiple Data) – системы с множественным потоком команд и множественным потоком данных. К подобному классу относится большинство параллельных многопроцессорных вычислительных систем.

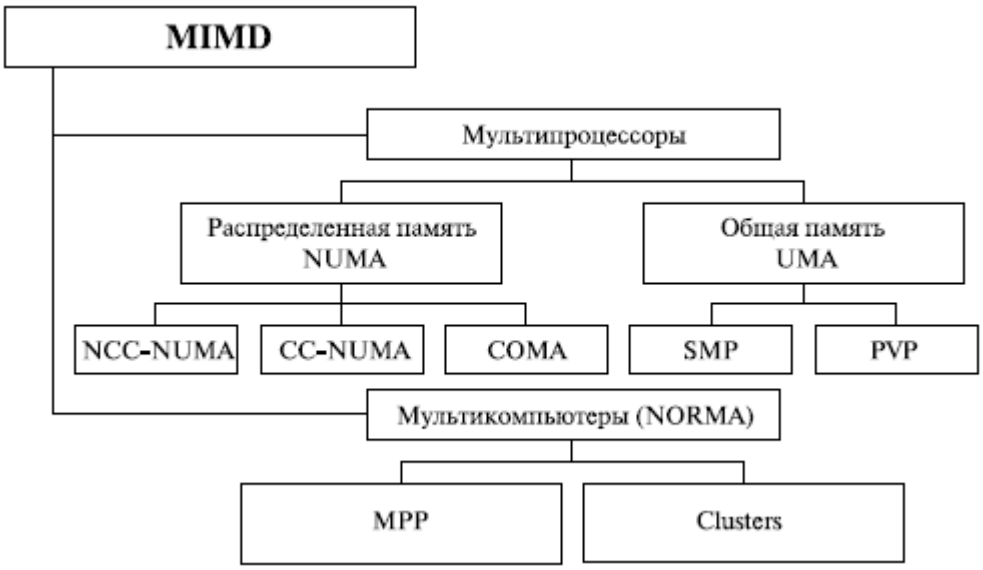


Рисунок 2.1 - Классификация многопроцессорных вычислительных систем

Следует отметить, что хотя систематика Флинна широко используется при конкретизации типов компьютерных систем, такая классификация приводит к тому, что практически все виды параллельных систем (несмотря на их существенную разнородность) оказываются отнесены к одной группе MIMD. Как результат, многими исследователями предпринимались неоднократные попытки детализации систематики Флинна. Так, например, для класса MIMD предложена

практически общепризнанная структурная схема, в которой дальнейшее разделение типов многопроцессорных систем основывается на используемых способах организации оперативной памяти в этих системах (см. рис. 1.1). Такой подход позволяет различать два важных типа многопроцессорных систем – multiprocessors ( мультипроцессоры или системы с общей разделяемой памятью) и multicomputers ( мультикомпьютеры или системы с распределенной памятью).

### 2.1.2 Мультипроцессоры

Для дальнейшей систематики мультипроцессоров учитывается способ построения общей памяти. Первый возможный вариант – использование единой (централизованной) общей памяти (shared memory) (см. рис. 1.2.2а). Такой подход обеспечивает однородный доступ к памяти (uniform memory access или UMA) и служит основой для построения векторных параллельных процессоров (parallel vector processor или PVP) и симметричных мультипроцессоров (ymmetric multiprocessor или SMP). Среди примеров первой группы - суперкомпьютер Cray T90, ко второй группе относятся IBM eServer, Sun StarFire, HP Superdome, SGI Origin и др.

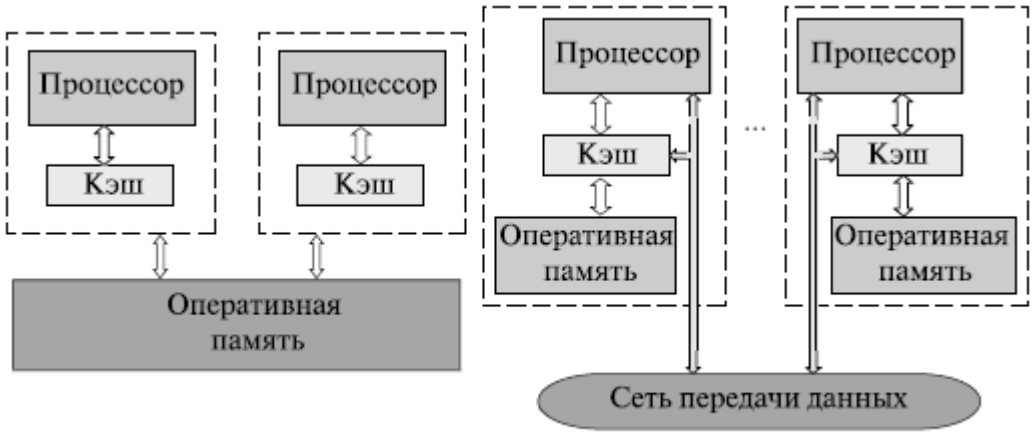


Рисунок 2.2 - Архитектура многопроцессорных систем с общей (разделяемой) памятью: системы с однородным (а) и неоднородным (б) доступом к памяти

Одной из основных проблем, которые возникают при организации параллельных вычислений на такого типа системах, является доступ с разных процессоров к общим данным и обеспечение, в связи с этим, *однозначности* (когерентности) содержимого разных кэшей (cache coherence problem). Дело в том, что при наличии общих данных копии значений одних и тех же переменных могут оказаться в кэшах разных процессоров. Если в такой ситуации (при наличии копий общих данных) один из процессоров выполнит изменение значения разделяемой переменной, то значения копий в кэшах других процессоров окажутся не соответствующими действительности и их использование приведет к некорректности вычислений. Обеспечение однозначности кэшей обычно реализуется на аппаратном уровне – для этого после изменения значения общей переменной все копии этой переменной в кэшах отмечаются как недействительные и последующий доступ к переменной потребует обязательного обращения к основной памяти. Следует отметить, что необходимость обеспечения

когерентности приводит к некоторому снижению скорости вычислений и затрудняет создание систем с достаточно большим количеством процессоров.

Наличие общих данных при параллельных вычислениях приводит к необходимости синхронизации взаимодействия одновременно выполняемых потоков команд. Так, например, если изменение общих данных требует для своего выполнения некоторой последовательности действий, то необходимо обеспечить взаимоисключение (mutual exclusion), чтобы эти изменения в любой момент времени мог выполнять только один командный поток. Задачи взаимоисключения и синхронизации относятся к числу классических проблем, и их рассмотрение при разработке параллельных программ является одним из основных вопросов параллельного программирования.

Общий доступ к данным может быть обеспечен и при физически распределенной памяти (при этом, естественно, длительность доступа уже не будет одинаковой для всех элементов памяти) (см. рис. 1.2.2 б). Такой подход именуется неоднородным доступом к памяти (non-uniform memory access или NUMA). Среди систем с таким типом памяти выделяют:

Системы, в которых для представления данных используется только локальная кэш-память имеющихся процессоров (cache-only memory architecture или COMA); примерами являются KSR-1 и DDM;

Системы, в которых обеспечивается когерентность локальных кэшей разных процессоров ( cache-coherent NUMA или CC-NUMA ); среди таких систем: SGI Origin 2000, Sun HPC 10000, IBM/Sequent NUMA-Q 2000;

Системы, в которых обеспечивается общий доступ к локальной памяти разных процессоров без поддержки на аппаратном уровне когерентности кэша (non-cache coherent NUMA или NCC-NUMA); например, система Cray T3E.

Использование распределенной общей памяти (distributed shared memory или DSM) упрощает проблемы создания мультипроцессоров (известны примеры систем с несколькими тысячами процессоров), однако возникающие при этом проблемы эффективного использования распределенной памяти (время доступа к локальной и удаленной памяти может различаться на несколько порядков) приводят к существенному повышению сложности параллельного программирования.

### 2.1.3 Мультикомпьютеры

Мультикомпьютеры (многопроцессорные системы с распределенной памятью) уже не обеспечивают общего доступа ко всей имеющейся в системах памяти (no-remote memory access или NORMA) (см. рис. 1.2.3). При всей схожести подобной архитектуры с системами с распределенной общей памятью (см. рис. 1.2.2б), мультикомпьютеры имеют принципиальное отличие: каждый процессор системы может использовать только свою локальную память, в то время как для доступа к данным, располагаемым на других процессорах, необходимо явно выполнить операции передачи сообщений (message passing operations). Данный подход применяется при построении двух важных типов многопроцессорных вычислительных систем (см. рис. 1.2.1) - массивно-параллельных систем (massively parallel processor или MPP) и кластеров (clusters). Среди представителей первого типа систем — IBM RS/6000 SP2, Intel PARAGON, ASCI Red, транспьютерные

системы Parsytec и др.; примерами кластеров являются, например, системы AC3 Velocity и NCSA NT Supercluster.

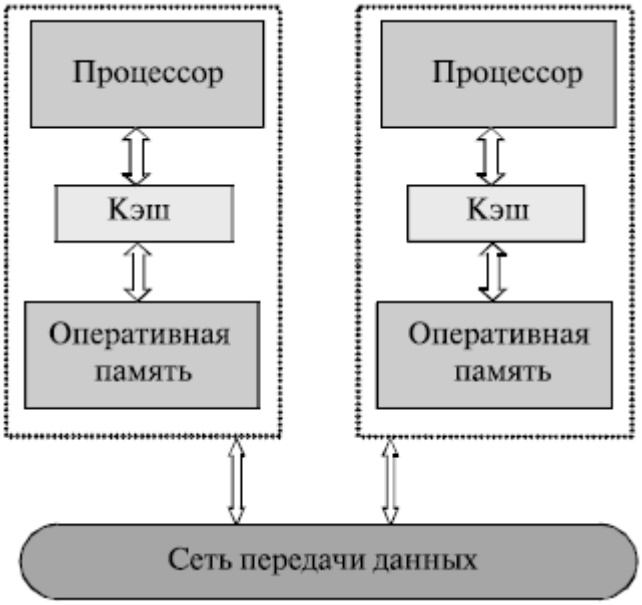


Рисунок 2.3 - Архитектура многопроцессорных систем с распределенной памятью

Следует отметить чрезвычайно быстрое развитие многопроцессорных вычислительных систем кластерного типа. Под кластером обычно понимается множество отдельных компьютеров, объединенных в сеть, для которых при помощи специальных аппаратно-программных средств обеспечивается возможность унифицированного управления ( single system image ), надежного функционирования ( availability ) и эффективного использования ( performance ). Кластеры могут быть образованы на базе уже существующих у потребителей отдельных компьютеров либо же сконструированы из типовых компьютерных элементов, что обычно не требует значительных финансовых затрат. Применение кластеров может также в некоторой степени устранить проблемы, связанные с разработкой параллельных алгоритмов и программ, поскольку повышение вычислительной мощности отдельных процессоров позволяет строить кластеры из сравнительно небольшого количества (несколько десятков) отдельных компьютеров (lowly parallel processing). Тем самым, для параллельного выполнения в алгоритмах решения вычислительных задач достаточно выделять только крупные независимые части расчетов (coarse granularity), что, в свою очередь, снижает сложность построения параллельных методов вычислений и уменьшает потоки передаваемых данных между компьютерами кластера. Вместе с этим следует отметить, что организация взаимодействия вычислительных узлов кластера при помощи передачи сообщений обычно приводит к значительным временным задержкам, и это накладывает дополнительные ограничения на тип разрабатываемых параллельных алгоритмов и программ.

### 2.1.4 Архитектура вычислительных систем

Архитектура ВС характеризует ее общую логическую организацию, техническое обеспечение, программное обеспечение, методы кодирования информации, интерфейс (способ взаимодействия) пользователя с системой.

Одной из важных разновидностей ВС является суперкомпьютер – архитектура для скоростной обработки данных. В нем используется большое число процессоров. Их совместная работа основывается на нескольких архитектурах.

Первая из них – с разделяемой оперативной памятью – осуществляет т.н. симметричную мультипроцессорную обработку.

Вторая архитектура основана на том, что каждый процессор имеет свою оперативную память, она выполняет асимметричную мультипроцессорную обработку.

Третья архитектура использует достоинства архитектур с разделяемой памятью и передачей сообщений. Здесь каждый процессор работает со своей памятью, но модули устройства памяти связаны друг с другом при помощи коммутатора. Коммутаторы, именуемые иначе узлами, могут включаться между группами процессоров и модулей памяти. Здесь сообщения между процессорами и памятью передаются через несколько узлов.

Четвертая архитектура основывается на использовании узловых блоков, каждый из которых содержит транспьютер (специализированный процессор для параллельной обработки), включающий процессор, память и средства связи. Одним из подходов является создание гиперкуба n-го порядка. Каждый блок этого куба непосредственно взаимодействует с n–1 соседними блоками. Большое будущее имеют системы с полным параллелизмом. Каждая такая система состоит из очень большого числа транспьютеров. Все процессоры по вертикали и по горизонтали обмениваются данными, образуя сложное поле обработки информации

Та или иная архитектура, реализованная на множестве компьютеров, называется вычислительным комплексом.

## 2.2 Организация управления доступом к разделяемой памяти

### 2.2.1 Блокировки

Традиционным методом организации доступа к разделяемым данным являются различные виды блокировок (мьютексы, семафоры и т.п.). Крупнозернистые блокировки относительно просты в использовании, но они могут значительно понизить производительность. С помощью мелкозернистых блокировок можно этого избежать, однако реализация эффективной синхронизации этими средствами может быть достаточно сложна, а отладка и поиск ошибок – ещё сложнее. Блокировкам присущи такие недостатки, как последовательное выполнение критических секций (serialization), взаимные блокировки (deadlocks), инверсии приоритетов (priority inversion), последовательный захват блокировки (convoying) и голодание (starvation).

### 2.2.2 Структуры данных и алгоритмы, свободные от блокировок

Также в настоящее время развивается семейство алгоритмов и структур данных, свободных от блокировок (lock-free). Данные методы обеспечивают гарантированное время выполнения вне зависимости от результата работы других потоков. Неблокирующие алгоритмы строятся на атомарных операциях, например, чтение-модификация-запись и самая значимая из них — сравнение с обменом (CAS). Написание правильно работающего неблокирующего алгоритма часто бывает очень сложно.

### 2.2.3 Транзакционная память

Альтернативной технологией синхронизации является транзакционная память. Транзакционная память (Transactional Memory, TM) – одно из современных перспективных направлений исследований в области организации управления доступом к разделяемой памяти в параллельных вычислениях, построенное на понятии транзакции (аналогично транзакциям баз данных). Под транзакцией в данном контексте понимается выполняемая потоком конечная последовательность инструкций, выполняющая чтение и запись в разделяемую память. Транзакции удовлетворяют свойствам сериализуемости и атомарности. Сериализуемость (serializability) предполагает последовательное выполнение транзакций, не приводящее к состояниям состязания (contention). Условие атомарности (atomicity) гарантирует, что транзакция вступает в силу мгновенно, при этом все изменения в памяти единовременно становятся видимыми для всех потоков. В общем случае, транзакция может либо выполниться целиком и успешно, либо не выполниться вообще. Если происходит конфликт данных, транзакция отменяется. Отмена транзакции приводит к отмене действий, которые совершал поток за время транзакции. После этого транзакция обычно перезапускается.

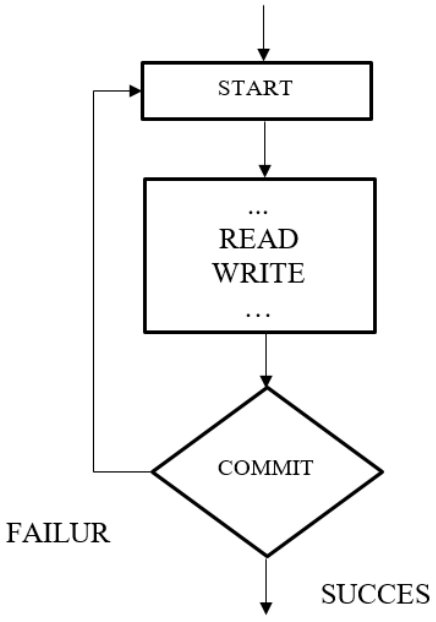


Рисунок 2.4 – Блок-схема выполнения транзакции

Транзакционная память значительно упрощает параллельное программирование, повышает уровень абстракции по сравнению с блокировками.

Работа с транзакционной памятью аналогична критическим секциям, группы инструкций выделяются в атомарные транзакции. Существенное различие заключается в том, что никакой из потоков, одновременно выполняющих транзакцию, не блокируется – все они выполняются параллельно. Такой подход называют оптимистичным – предполагается, что потоки редко изменяют одни и те же данные. Дополнительными преимуществами транзакционной памяти являются возможность вложенных транзакций – код с транзакциями может быть переиспользован в рамках другого транзакционного кода; возможность откатить транзакцию явно позволяет вернуть память в корректное состояние при обнаружении ошибки в транзакции.

Существуют как программные реализации транзакционной памяти, так и аппаратные. Например, программная реализация в C/C++ (GCC 4.7+) представляет собой библиотеку времени выполнения libitm. В язык вводятся ключевые слова, основным из которых является __transaction_atomic { … } , определяющий блок кода как транзакцию. Аппаратные реализации поддерживают транзакции на уровне процессорных команд. Компания Intel в 2013 году выпустила поколение процессоров Haswell, где вводятся расширения транзакционной синхронизации (Transactional Syncrhonization Extensions, TSX). Расширения позволяют помечать отдельные участки кода как транзакции. Ограниченная транзакционная память (Restricted Transactional Memory, RTM), являющаяся одним из расширений TSX, предполагает выполнение резервного кода (fallback code) в случае, если транзакция не может быть выполнена успешно несколько раз подряд. Недавно стало известно, что вычисления с использованием расширений TSX могут производиться с ошибками. Эту проблему обнаружил один из разработчиков, и наличие дефекта уже подтвердила компания Intel. Во избежание возможных проблем Intel выпустила программный патч, блокирующую работу процессоров с инструкциями TSX. Компания Intel работает над устранением ошибки, поэтому программисты могут и дальше работать над TSX-совместимым кодом. На данный момент времени доступна программная транзакционная память и в дальнейшем будет описываться программная транзакционная память.

#### 2.2.3.1 Что такое транзакционная память

Транзакция – это некоторая форма выполнения программ, перенятая от сообщества баз данных. Параллельно исполняемые запросы конфликтуют, когда они читают и изменяют некоторый элемент базы данных, и возникающий конфликт может привести к ошибочному результату, который не мог бы получиться при последовательном выполнении этих запросов. Транзакции гарантируют, что все запросы произведут тот же самый результат, как если бы они выполнялись последовательно в некотором порядке (serially, «сериально»; это свойство называют «сериализуемостью» (serializability)). Декомпозиция семантики транзакции приводит к четырем требованиям, обычно называемым свойствами ACID: атомарность (atomicity), согласованность (consistency), изоляция (isolation) и долговечность (durability).

TM обеспечивает механизм легковесных транзакций для потоков управления, выполняемых в общем адресном пространстве. TM гарантирует атомарность и изолированность параллельно выполняемых задач. (Вообще говоря,

согласованность и долговременность не обеспечиваются.) Атомарность гарантирует, что изменения состояния программы, производимые кодом, который выполняется в некоторой транзакции, являются невидимыми с точки зрения других, параллельно выполняемых транзакций. Другими словами, хотя код, выполняемый внутри транзакции, может изменять отдельные переменные посредством присваивания, в других вычислениях может наблюдаться состояние программы только либо непосредственно до, либо непосредственно после выполнения данной транзакции. Изолированность гарантирует, что параллельно выполняемые задачи не влияют на результат транзакции, так что транзакция производит один и тот же результат, как если бы не выполнялась никакая другая задача. Транзакции обеспечивают основу для построения параллельных абстракций, являющихся строительными блоками, которые можно комбинировать без знания их внутренних деталей, во многом подобно тому, как процедуры и объекты обеспечивают пригодные для компоновки абстракции для последовательного кода.

#### 2.2.3.2 Программая модель TM

Программная модель обеспечивает логическое обоснование конструкций языков программирования и методологические принципы конструирования программ. Подобно многим другим аспектам TM, ее программная модель все еще является предметом активных исследований. В большинстве систем TM обеспечиваются простые атомарные операторы, выполняющие блок кода (и вызываемые в нем подпрограммы) как транзакцию. Атомарный блок изолирует код от параллельно выполняемых нитей, но блок не заменяет общую синхронизацию, обеспечиваемую, например, семафорами или условными переменными. В частности, атомарные блоки сами по себе не обеспечивают средства для координации кода, выполняемого в параллельных потоках управления.

В отличие от этого, автоматическое взаимное исключение (automatic mutual exclusion, AME) выворачивает транзакционную модель «на изнанку» за счет выполнения большей части программы в транзакциях. AME поддерживает асинхронное программирование, в котором при вызове любой функции начинается одно или несколько асинхронных вычислений, и затем для получения результатов этих вызовов требуются рандеву. Эта программная модель часто применяется для решения проблемы непредвиденных задержек в управляемых пользователями и распределенных системах. Атомарность, обеспечиваемая транзакциями, гарантирует, что асинхронное вычисление, выполняемое с непредсказуемой скоростью, не мешает выполнению других асинхронных вычислений.

#### 2.2.3.3 Преимущества транзакционной памяти

При параллельном программировании приходится сталкиваться со многими трудностями, но одной из наиболее серьезных проблем при написании корректного кода является координация доступа к данным, совместно используемым в нескольких потоках управления. Последствиями попыток обеспечить взаимное исключение со слишком слабой или слишком сильной синхронизацией являются конфликты при доступе к данным (data race), взаимоблокировки (deadlock) и плохая масштабируемость. TM обеспечивает более простую альтернативу взаимного

исключения, снимая ношу корректной синхронизации с программиста и возлагая ее на систему ТМ. Теоретически от разработчика программы требуется только обозначить последовательность операций, которая должна будет выполняться атомарно по отношению к другим параллельно выполняющимся потокам управления. Это обеспечивается системой ТМ на основе многочисленных механизмов, описываемых в этой статье.

Харрис (Harris) и Пейтон-Джонс (Peyton-Jones) утверждают, что, помимо обеспечения улучшенной программной абстракции, транзакции также делают синхронизацию компонуемой, что позволяет конструировать абстракции параллельного программирования. Программная абстракция является компонуемой, если ее можно корректно комбинировать с другими абстракциями без потребности понимания ее функционирования. Простая схема блокировок не является компонуемой. ТМ позволяет прямо компоновать операции.

#### 2.2.3.4 Ограничения транзакционной памяти

Транзакции сами по себе не могут заменить всю синхронизацию в параллельной программе. Кроме взаимного исключения, синхронизация часто используется для координации независимых задач, например, путем обеспечения средств, позволяющих одной задаче дожидаться окончания другой задачи или ограничивающих число потоков управления, в которых исполняется некоторая программа.

Сами по себе транзакции не слишком помогают в координации независимых задач. Например, рассмотрим взаимосвязь программ «производитель-потребитель», когда одна задача пишет значения, которые читаются другой задачей. Транзакции могут гарантировать, что задачи, обращающиеся к одним и тем же ресурсам, не помешают друг другу. Однако схема, требуемая в данном случае, слишком дорого реализуется с использованием транзакций, назначением которых является как раз предотвращение взаимодействия задач. Если транзакция-потребитель обнаруживает, что читать еще нечего, она может только лишь аварийно завершиться и повторить попытку чтения позже. Активное ожидание (busy waiting) с использованием аварийного завершения транзакции неэффективно, поскольку при аварийном завершении транзакции «откатываются» (roll back) все проделанные ей вычисления. Было бы лучше, если бы производитель мог подать сигнал потребителю, когда значение станет готовым для чтения. Однако, поскольку в соответствии с семантикой транзакций сигнал, поданный в одной транзакции, является невидимым в других транзакциях, во многих системах ТМ обеспечивается механизм предохранителей (guard), который не дает транзакции начаться до тех пор, пока соответствующий ей предикат не примет значение true.

В Haskell ТМ поддерживаются конструкции retry и orElse, первая из которых позволяет транзакции дождаться возникновения некоторого события, а вторая – упорядочить выполнение двух транзакций Выполнение оператора retry приводит к аварийному завершению включающей его транзакции. Она не может быть выполнена повторно до тех пор, пока не изменится значение переменной, прочитанной перед выполнением retry. Это позволяет избежать использования наиболее грубой формы активного ожидания, при котором транзакция повторно считывает неизменившееся значение и аварийно завершается. Конструкция orElse

обеспечивает возможность компоновать две транзакции, позволяя второй транзакции выполняться только в том случае, когда первая завершается аварийно. Этот распространенный сценарий трудно реализовать каким-либо другим способом, поскольку аварийное завершение и повторное выполнение транзакции происходят прозрачно для всех остальных частей программы.

Плюсы и минусы программной модели TM, а также ее прагматика все еще до конца не понятны. Например, активные споры ведутся относительно вложенных транзакций. Предположим, что в коде, выполняемом в транзакции `o`, вызывается библиотечная подпрограмма, в которой начинается собственная транзакция `i`. Должны ли иметься какие-то способы взаимодействия этих транзакций, и как это влияет на реализацию TM и методику построения модульного программного обеспечения и библиотек? Пусть транзакция I успешно завершается. Должны ли ее результаты быть видимы только коду транзакции, или и другим потокам управления тоже (открытая вложенность)? При выборе второго варианта, что произойдет, если транзакция завершится аварийным образом? Аналогично, если аварийно завершится транзакция `i`, должно ли это привести к завершению и транзакции `o`, или же внутренняя транзакция должна откатиться и перезапуститься независимым образом?

Наконец, производительность систем TM пока еще не слишком высока для их широкого использования. Программные системы TM (software TM system, STM) существенно нагружают код, выполняемый внутри транзакции, что уменьшает преимущества по производительности параллельных компьютеров. Аппаратные системы TM (hardware TM systems, HTM) могут снизить накладные расходы, но их коммерческие образцы только начинают появляться, и большинство систем HTM не справляется с обработкой крупных транзакций. Применение более совершенных методов реализации, вероятно, позволит улучшить оба вида систем, и в этой области ведутся активные исследования.

#### 2.2.3.5 Реализация транзакционной памяти

TM можно реализовывать полностью программным образом (STM) или с использованием специальной аппаратной поддержки (HTM). Предлагалось много разных методов реализации.

В большинстве систем TM обоих типов реализуется оптимистическое управление параллелизмом, при котором любая транзакции выполняется в предположении, что у нее не будет конфликтов с другими транзакциями. Если две транзакции конфликтуют, поскольку одна из них модифицирует некоторую область памяти, прочитанную или модифицированную другой транзакцией, то система TM аварийно завершает одну из этих транзакций путем устранения произведенных изменений (отката). При альтернативном пессимистическом управлении параллелизмом требуется, чтобы транзакция получила монопольный доступ к требуемой области памяти (например, путем запроса ее блокировки) до выполнения операции ее модификации. При применении этого подхода транзакции также могут аварийно завершаться и откатываться в случае возникновения тупиковых ситуаций.

#### 2.2.3.6 Программная транзакционная память

В первой статье Шавита (Shavit) и Туиту (Touitou), посвященной STM, было показано, что можно полностью программным образом реализовать свободные от блокировок (lock-free) атомарные операции над несколькими переменными, но для этого в программе нужно заранее объявить, к каким ячейкам памяти будет обращаться данная транзакция.

Описанная Херлихи и др. в система Dynamic STM (DSTM) была первой системой STM, в которой это требование снималось. DSTM является системой STM с гранулированностью на уровне объектов и откладываемыми изменениями (deferred-update). Это означает, что транзакция модифицирует частные копии объектов и делает видимыми изменения другим транзакциям при своей фиксации. Транзакция имеет монопольный доступ к этим копиям без синхронизации. Однако к исходным объектам могут обращаться другие транзакции, что приводит к возникновению логического конфликта, распознаваемого системой STM и разрешаемого путем аварийного завершения одной из транзакций.

Система STM может распознать конфликт при первом обращении транзакции к объекту (early detection, раннее выявление) или в то время, когда транзакция пытается выполнить операцию фиксации (late detection, отложенное выявление). Оба подхода приводят к одним и тем же результатам, но могут выполняться по-разному, и, к сожалению, ни один из них не обладает существенными преимуществами над другим. Раннее выявление предотвращает выполнение транзакциями излишних вычислений, которые будут отброшены при последующем откате. При отложенном выявлении можно избежать излишних откатов, которые могут возникнуть, когда конфликтующая транзакция откатывается сама по причине конфликта с третьей транзакцией.

Еще одной сложностью является конфликт между транзакцией, которая только читает некоторый объект, и другой транзакцией, которая изменяет этот объект. Поскольку чтения более распространены, чем записи, системы STM клонируют только модифицируемые объекты. Для сокращения накладных расходов транзакция отслеживает прочитанные объекты и до своей фиксации убеждается в том, что эти объекты не модифицировались никакой другой транзакцией.

DSTM – это библиотека. Любой объект, к которому производятся обращения внутри транзакции, сначала регистрируется в системе DSTM, возвращающей объект-оболочку TMObject для данного объекта (рис. 1). Впоследствии программа, выполняемая внутри транзакции, может открыть TMObject только для чтения или для чтения и записи, получив в ответ указатель на исходную или клонированную версию объекта соответственно. В любом случае после этого транзакция работает с объектом напрямую без дополнительной синхронизации.

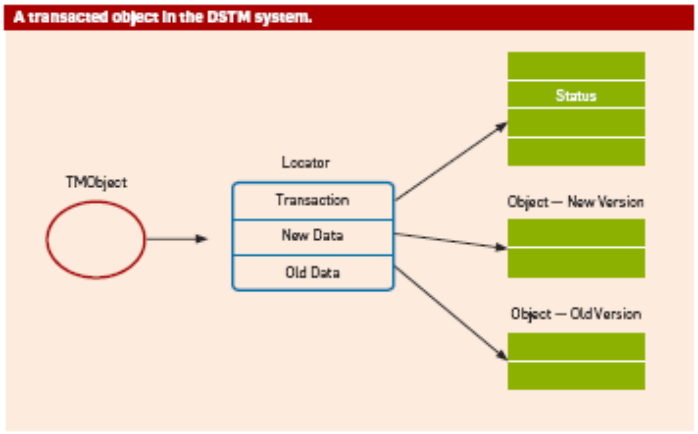


Рисунок 2.5 TMObject – это оболочка для данного объекта. Он указывает на локатор, который, в свою очередь, ссылается транзакцию, открывшую объект, исходную («old») версию объекта и частную для данной транзакции («new») копию данного объекта

Транзакция завершается, когда программа пытается зафиксировать изменения, произведенные при выполнении данной транзакции. Если транзакция успешно фиксируется, система DSTM для всех модифицированных объектов автоматически заменяет их старую структуру в локаторе соответствующей измененной версией.

Транзакция `T` может успешно зафиксироваться, если выполняются два условия. Первое условие состоит в том, что не должна существовать параллельно выполняемая транзакция, изменившая некоторый объект, который был прочитан транзакцией T. DSTM отслеживает объекты, открытые транзакцией для чтения, и проверяет элементы набора прочитанных объектов, когда транзакция пытается зафиксироваться. Объект из этого набора считается отвечающим условию фиксации, если его версия не изменилась с тех пор, как транзакция `T` открыла его в первый раз. DSTM также проверяет набор прочитанных объектов при каждом следующем открытии объекта, чтобы устранить возможность продолжения выполнения транзакции при ошибочном состоянии программы, в котором некоторые объекты были изменены после начала выполнения данной транзакции.

Вторым условием успешной фиксации является то, что транзакция `T` не должна модифицировать какой-либо объект, измененный другой транзакцией. DSTM предупреждает возникновение конфликтов этого типа, разрешая только одной транзакции открывать объект для модификации. При возникновении конфликта «write-write» DSTM аварийно завершает одну из двух конфликтующих транзакций, разрешая другой продолжать свое выполнение. DSTM откатывает одну из двух конфликтующих транзакций в ее исходное состояние и позволяет ей выполниться повторно. Политика, используемая для выбора транзакции-жертвы, может влиять на производительность системы, включая ее жизнеспособность, но не должна никаким образом воздействовать на семантику системы STM.

Производительность DSTM, как и других систем STM, зависит от деталей рабочей нагрузки. Вообще говоря, при небольшом числе процессоров накладные расходы систем STM оказываются более существенными, чем у схем с блокировками. Однако при возрастании числа процессоров повышаются и уровень

конкуренции на отдельную блокировку, и стоимость ее установки. В таких условиях при наличии редко возникающих конфликтов STM (как это показано исследователями) превосходит по производительности схемы с блокировками на тестовых наборах с мелкими транзакциями.

#### 2.2.3.7 Системы с откладываемыми изменениями

Существенное внимание уделялось исследованию политик выбора транзакции, которую следует аварийно завершить в случае конфликтов. Ни одна политика не обеспечивает наилучшее поведение во всех случаях, однако в целом неплохо работает политика «полька» (polka). При применении этой политики для каждой транзакции запоминается число открытых ей объектов, и этот счетчик используется в качестве ее приоритета. Если какая-то транзакция пытается запросить доступ к некоторому объекту, то это приводит к немедленному аварийному завершению конфликтующей транзакции с более низким приоритетом. Если приоритет у транзакции, запрашивающей доступ, оказывается ниже, чем у конфликтующей с ней транзакции, то она откатывается N раз, где N равно разнице в приоритетах, с экспоненциально возрастающим временным интервалом между попытками. Другими словами, такой транзакции дается N попыток получить доступ к требуемому объекту, и после каждой неудачной попытки она откатывается и выполняется заново.

#### 2.2.3.8 Система с непосредсвенным обменом

В системах STM с непосредственным обновлением транзакции изменяют объекты напрямую, а не через копии. Потенциально этот подход более эффективен, поскольку не требуется клонировать каждый модифицируемый объект. Однако системы с непосредственным обновлением вынуждены сохранять исходное значение каждой измененной ячейки памяти, чтобы можно было восстановить ее предыдущее состояние при аварийном завершении транзакции. Кроме того, системы с непосредственным обновлением должны предотвращать чтение транзакцией ячеек памяти, модифицированных другой, не зафиксированной транзакцией, что понижает уровень потенциально доступного параллелизма.

В системах STM с непосредственным обновлением также требуются блокировки для предотвращения одновременной модификации несколькими транзакциями одного и того же объекта. По причине высокой стоимости схемы с блокировками для чтения и записи в этих системах не блокируются только читаемые объекты, и используется проверка набора прочитанных объектов для выявления фактов модификации этих объектов параллельно выполняемыми транзакциями. Используемые блокировки порождают тот же уровень накладных расходов, что и в системах с откладываемыми изменениями.

Однако блокировки, используемые для предотвращения нескольких записей в одну и ту же область памяти, приводят к потенциальной возможности простоя многих транзакций, поскольку откладываемые транзакции продолжают удерживать блокировки. В системах STM с откладываемыми изменениями обычно используются не блокирующие структуры данных, что не дает сбойным потокам управления мешать другим потокам. В системах STM с непосредственным обновлением обеспечиваются аналогичные гарантии продвижения приложений за

счет обнаружения и аварийного завершения сбойных или блокированных потоков управления.

#### 2.2.3.9 Аппаратная поддержка транзакционной памяти

Усилия программистов, необходимые для использования параллелизма, оправдываются, если новый код выполняется лучше или обеспечивает более быструю реактивность, чем последовательный код. Хотя современные системы STM масштабируются при возрастании числа процессоров на многоядерном кристалле, накладные расходы программных систем являются существенными. Даже при использовании оптимизирующих компиляторов STM-поток может выполняться в 2-7 раз медленнее последовательного кода.

HTM могут улучшить производительность STM. Хотя в этой области продолжаются активные исследования, предложенные к настоящему времени системы можно подразделить на две широкие категории: системы, позволяющие ускорить ключевые операции STM, и системы, в которых управление транзакциями поддерживается прямо на аппаратном уровне.

#### 2.2.3.10 Аппартное ускорение STM

Основным источником накладных расходов STM является поддержка и проверка наборов прочитанных объектов. Для поддержки такого набора в системе STM обычно вызывается специальная подпрограмма при каждом чтении из совместно используемой памяти. Эта подпрограмма регистрирует адрес соответствующего объекта и, возможно, выполняет раннее обнаружение конфликта путем проверки версии объекта или наличия блокировки. Для валидации транзакции система STM должна проверить весь набор прочитанных объектов и убедиться в том, что ни для одного из них нет конфликта. Выполнение этих инструментальных действий может стать большим накладным расходом, если транзакция после каждого обращения к памяти не производит большой объем вычислений.

Система STM с аппаратным ускорением (hardware-accelerated STM, HASTM), предложенная Саха (Saha) и др., была первой системой, где для снижения накладных расходов инструментария STM использовалась аппаратная поддержка. Вспомогательная аппаратура позволяет программному обеспечению построить быстрые фильтры, которые могут ускорить действия, требуемые для поддержки наборов прочитанных объектов. HASTM обеспечивает для STM две возможности на основе поддержки битов пометки потоков управления на уровне блоков кэша. Во-первых, программное обеспечение может проверить, не был бы ранее установлен бит пометки для данного блока памяти, и не производилась ли запись в этот блок в другом потоке после того, как он был помечен (выявление конфликта). Во-вторых, программное обеспечение может узнать, производилась ли запись в других потоках управления в какой-нибудь из блоков памяти, помеченных данным потоком (валидация).

В HASTM предлагалось реализовывать биты пометки за счет использования дополнительных метаданных для каждого блока процессорных кэшей многоядерного кристалла. Аппаратура отслеживает, не стал ли недействительным какой-либо помеченный блок кэша по причине выталкивания из кэша или записи,

произведенной в него из другого потока управления. STM использует биты пометки следующим образом. При вызове подпрограммы чтения проверяется и устанавливается бит пометки для блока памяти, содержащего заголовок соответствующего объекта. Если бит пометки уже установлен, т.е. данная транзакция ранее уже обращалась к объекту, то этот объект не добавляется к набору прочитанных объектов. Для валидации транзакции STM выясняет с помощью аппаратуры, стал ли недействительным какой-либо помеченный блок кэша. Если это не произошло, то значит все объекты, к которым производился доступ через инструментарий STM, являлись частными для данного потока управления в течение всего выполнения транзакции, и никакая дальнейшая валидация не требуется. Если некоторые помеченные блоки стали недействительными, система STM должна производить программную валидацию, проверяя номера версий или наличие блокировок для всех объектов из набора прочитанных объектов. На этом дорогостоящем шаге валидации определяется, был ли помеченный блок вытолкнут из кэша из-за его ограниченной емкости или из-за наличия реального конфликта между одновременно выполняемыми транзакциями.

HASTM допускает возникновение при выполнении транзакций системных событий, таких как прерывания, переключения контекста и страничные отказы, поскольку биты пометки действуют всего лишь как фильтр. Если обработка системного события приводит к выталкиванию некоторых помеченных блоков, то незавершенная транзакция может продолжить свое последовательное выполнение без аварийного завершения. К такой транзакции просто будет применена программная валидация до ее фиксации. Аналогично, HASTM допускает приостановку транзакций и их обследование такими компонентами, как сборщик мусора или отладчик, выполняемыми в другом потоке.

Ускорить выявление конфликтов в STM можно и без модификации аппаратных кэшей. Кэши первого уровня обычно располагаются в критическом контуре процессора и взаимодействуют со сложными подсистемами, такими как подсистема поддержки протокола согласованности кэшей. Даже незначительные изменения в аппаратуре кэша могут повлиять на тактовую частоту процессора и повысить сложность его разработки и верификации. Као Минь (Cao Minh) и др. предложили подход к ускорению систем STM на основе сигнатур (SigTM). При этом подходе для пессимистического кодирования множеств прочитанных и измененных объектов программных транзакций используются аппаратные сигнатуры. Сигнатуры вычисляются вне кэшей с применением аппаратного фильтра Блюма. (Фильтр Блюма (Bloom filter) позволяет эффективно представлять надмножество элементов множества и быстро устанавливать, входит ли в это множество заданный элемент. Использование фильтров Блюма для обнаружения зависимостей при управлении выполнением нитей и транзакций было впервые предложено Сезом (Ceze) и др.) Программный инструментарий обеспечивает фильтрам адреса объектов, читаемых или изменяемых внутри транзакции. Для распознавания конфликтов аппаратура компьютера отслеживает трафик протокола поддержки согласованности кэшей, обнаруживая в нем запросы монопольного доступа к блокам кэша, что означает изменение памяти. Анализируя сигнатуры транзакции, аппаратура проверяет, не может ли относиться адрес, содержащийся в запросе, к наборам прочитанных или измененных объектов этой транзакции. Если

да, то считается, что обнаружен потенциальный конфликт, и система STM может либо аварийно завершить транзакцию, либо прибегнуть к программной валидации.

И HASTM, и SigTM ускоряют поддержку наборов прочитанных объектов и валидацию в системах STM. Тем не менее, между этими подходами имеются архитектурные различия. SigTM кодирует наборы прочитанных и измененных объектов, размеры которых превышают размеры частных кэшей. Ограниченная емкость кэша и промахи при обнаружении конфликтов не приводят к потребности программной валидации, как при использовании HASTM. С другой стороны, в SigTM используются вероятностные сигнатуры, в результате чего истинные конфликты никогда не упускаются, но из-за наложения адресов в фильтрах Блюма могут выявляться ложные конфликты. Кроме того, аппаратные сигнатуры относительно компактны, и их легко обрабатывать, так что их можно сохранять и восстанавливать при переключении контекста и обработке прерываний. В HASTM биты пометки могут быть утрачены, если процессор используется для выполнения другой задачи. С другой стороны, в сигнатурах SigTM отражаются физические адреса, и содержимое сигнатуры становится недействительным при изменении отображения виртуальных страниц.

Показано, что аппаратное ускорение управления наборами прочитанных объектов в два раза повышает производительность систем STM, основанных на блокировках, с непосредственным обновлением и отложенными изменениями. Возможны дополнительные усовершенствования с использованием аппаратных механизмов, которые поддерживают управление версиями объектов, изменяемых STM. Однако, поскольку в большинстве программ читается гораздо больше объектов, чем изменяется, получаемое повышение производительности является незначительным.

#### 2.2.3.11 Аппаратная транзакционная память

Интерес к полностью аппаратной реализации TM (HTM) появился после публикации двух первых статей про TM: статей Найта (Knight) и Херлихи и Мосса. В системах HTM не требуется программный инструментарий для обработки адресов памяти внутри кода транзакции. Аппаратура управляет версиями данных и отслеживает конфликты прозрачным образом при выполнении программами обычных операций чтения и записи. Устранение инструментария позволяет сократить накладные расходы прикладной программы и снимает потребность в специализированных телах функций, так что их можно вызывать и внутри, и вне транзакции.

В системах HTM для реализации управления версиями и выявления конфликтов используются иерархия кэшей и протокол согласования кэшей. Через кэши проходят все операции чтения и записи, задаваемые процессором, в них может буферизоваться значительный объем данных, и их ассоциативная организация позволяет производить эффективный поиск. Во всех HTM модифицируются кэши первого уровня, но этот подход может быть распространен на кэши более высокого уровня, как частные, так и совместно используемые. Для иллюстрации организации и функционирования систем HTM ниже сравнительно подробно описывается архитектура TCC, и кратко упоминаются ключевые особенности альтернативных разработок.

TCC (Transactional Coherence and Consistency) – это система HTM с откладываемыми изменениями, в которой выявление конфликтов производится на стадии фиксации транзакций. Для отслеживания наборов прочитанных и измененных объектов каждый блок кэша помечается служебными битами `R` и `W`, которые устанавливаются при первом доступе к блоку внутри транзакции по чтению или записи соответственно. Блоки кэша, входящие в набор измененных объектов, действуют как буфера записи и не выталкиваются в память до фиксации транзакции.

Для фиксации транзакций в TCC используется двухфазный протокол. Сначала аппаратура с использованием протокола поддержки согласованности кэшей получает монопольный доступ ко всем блокам кэша из набора измененных объектов. Если этот шаг проходит успешно, транзакция считается валидированной. На втором шаге аппаратура одновременно сбрасывает в кэше все биты `W`, что приводит к автоматической фиксации всех изменений данной транзакции. Новые версии данных теперь глобально доступны всем процессорам на основе обычного протокола согласования кэшей многоядерного кристалла. Если валидация не удается, поскольку другой процессор тоже пытается зафиксировать некоторую конфликтующую транзакцию, аппаратура обращается к программному обработчику, который может аварийно завершить данную транзакцию или попытаться зафиксировать ее с применением традиционной политики управления транзакциями. Когда транзакция фиксируется или завершается аварийно, все служебные биты одновременно очищаются с использованием групповой операции сброса. При отсутствии конфликтов параллельно может фиксироваться несколько транзакций.

Выявление конфликтов происходит, когда другие процессоры получают сообщения о согласовании кэшей от фиксируемой транзакции. Аппаратура проверяет, не содержится ли блок с указанным адресом в локальных кэшах. Если этот блок содержится в кэше, и для него установлены биты `R` или `W`, то это значит, что между фиксируемой и локальной транзакциями имеется конфликт «read-write» или «write-write» соответственно. Аппаратура генерирует сигнал программному обработчику, который аварийно завершает локальную транзакцию и потенциально повторяет ее с некоторой задержкой.

Аналогичные аппаратные методы могут применяться для поддержки систем HTM с непосредственным обновлением памяти и ранним обнаружением конфликтов. Для систем с непосредственным обновлением аппаратура прозрачным образом журнализует исходное значение блока памяти до его первой модификации транзакцией. Если транзакция завершается аварийно, этот журнал используется для обратного выполнения операций обновления. Для раннего обнаружения конфликтов аппаратура получает монопольный доступ к блоку кэша при первой записи и удерживает этот режим доступа до фиксации транзакции. При наличии небольшого числа конфликтов почти все системы HTM ведут себя схожим образом. Если конфликтов много, то подход с откладываемыми изменениями и обнаружением конфликтов приводит к меньшему числу патологических сценариев, с которыми можно легко справиться с применением политики задержек.

Потери производительности HTM-нити обычно составляет от 2-х до 10-ти процентов по сравнению с производительностью не транзакционного кода.

Система HTM может превосходить по производительности систему STM, основанную на блокировках, в четыре раза, а систему STM с аппаратным ускорением – в два раза. Тем не менее, системам HTM свойственно несколько проблем, которые отсутствуют в реализациях STM. Кэши, используемые для поддержки наборов прочитанных и измененных объектов, а также версий данных, обладают конечной емкостью и при выполнении длинной транзакции могут переполниться. Состояние транзакции, сохраняемое в кэше, имеет большой объем, и его трудно сохранять и восстанавливать при обработке прерываний, переключении контекста и подкачке. Длинные транзакции могут быть редкими, но их все равно необходимо выполнять с сохранением свойств атомарности и изолированности. С точки зрения программистов недопустимо устанавливать зависящие от реализации ограничения на размеры транзакций.

Простой механизм обработки переполнения кэша или системных событий заключается в том, чтобы в любом случае обеспечить возможность выполнения транзакции до конца. При возникновении одного из упомянутых выше событий система HTM может начать обновлять память напрямую, без поддержки наборов прочитанных и измененных объектов и старых версий данных. Однако в это время не могут выполняться другие транзакции, поскольку более невозможно выявление конфликтов. Кроме того, непосредственные обновления памяти без журнализации препятствуют использованию в транзакции явных операторов аварийного завершения или повторного выполнения.

Райвар (Rajwar) и др. предложили альтернативный подход VTM (Virtualized TM), который позволяет поддерживать атомарность и изолированность даже в тех случаях, когда транзакция прерывается событиями переполнения кэша или прерываниями. VTM отображает ключевые структуры данных, описывающие выполнение транзакции (наборы прочитанных и измененных данных, буфера записи или журнал откатов), в виртуальную память, которая является практически неограниченной, и на которую не влияют системные прерывания. В аппаратных кэшах сохраняется рабочий набор этих структур данных. Разработчики VTM также предлагали использовать аппаратные сигнатуры для устранения избыточного поиска в структурах в виртуальной памяти.

Наконец, для решения проблемы ограниченности аппаратных ресурсов можно использовать гибридную систему HTM–STM. Выполнение транзакции начинается в режиме HTM с использованием аппаратных механизмов выявления конфликтов и поддержки версий данных. Если ресурсы HTM исчерпываются, то транзакция откатывается и запускается заново в режиме STM с применением дополнительного инструментария. Для использования этого подхода требуется наличие двух вариантов каждой функции, но обеспечивается хорошая производительность для коротких транзакций. Проблемой гибридных систем является выявление конфликтов между одновременно выполняемыми HTM- и STM-транзакциями.

#### 2.2.3.12 Аппаратно-программный интерфейс для транзакционной памяти

Аппаратные разработки оптимизируются в расчете на то, чтобы ускорить выполнение в наиболее распространенных случаях и сократить стоимость корректной обработки редких событий. Поставщики процессоров будут следовать

этому принципу и при обеспечении аппаратной поддержки транзакционной памяти. В начальных системах, вероятно, на поддержку ТМ будут тратиться умеренные ресурсы. По мере того как транзакции начнут использовать в большем числе приложений, возможно, станут доступными более развитые аппаратные решения, включая полнофункциональные системы НТМ.

Безотносительно к объему аппаратуры, используемой для поддержки ТМ, важно то, что системы НТМ обеспечивают функциональные возможности, полезные при разработке практических программных моделей и сред выполнения. Значительная часть исследований в области НТМ посвящена аппаратно-программному интерфейсу, который может поддерживать развитые программные средства. Макдональд (McDonald) и др. предложили четыре интерфейсных механизма для систем НТМ. Первый механизм – это двухфазный протокол фиксации, который на архитектурном уровне разделяет вилидацию транзакции и фиксацию выполненных ею изменений в памяти. Второй механизм представляет собой транзакционные обработчики, позволяющие программному обеспечению реагировать на существенные события, такие как обнаружение конфликта, фиксация или аварийное завершение. Шрираман (Shriraman) и другие предлагают аналогичный механизм «alert-on-update», который вызывает программный обработчик, когда аппаратура НТМ обнаруживает конфликт, или в ней возникает переполнение. Третий механизм заключается в поддержке замкнутых и открыто-вложенных транзакций. Открытая вкладываемость позволяет программному обеспечение прерывать выполняемую транзакцию и исполнять некоторый служебный код (например, системный вызов) с независимыми гарантиями атомарности и изолированности. Другие исследователи полагают, что для обслуживания системных вызовов и операций ввода-вывода достаточно приостанавливать НТМ-транзакции. Тем не менее, если в служебном коде для доступа к совместно используемым данным используются транзакции, требования задержки выполнения транзакции не существенно отличаются от требований открыто-вложенных транзакций. Наконец, и Макдональд, и Шрираман предлагают несколько типов инструкций чтения и записи, позволяющих компиляторам отличать доступ к частным данным потока управления, неизменяемым и идемпотентным данным от доступа к истинно разделяемым данным. При наличии таких механизмов системы НТМ могут поддерживать широкий диапазон программных средств, от средств традиционной синхронизации и ограниченного ввода-вывода внутри транзакций до высокоуровневых моделей параллелизма, позволяющих избегать аварийного завершения транзакций при конфликтах по памяти, если не нарушается семантика уровня приложений.

#### 2.2.3.13 Открытые проблемы

Кроме обсуждавшихся выше реализационных проблем, в области ТМ имеется ряд проблем, являющихся предметом активных исследований. Одной из серьезных трудностей оптимистической ТМ является то, что в результате конфликта может откатиться транзакция, выполняющая операцию ввода-вывода. Под вводом-выводом в данном случае понимается любое взаимодействие с миром, расположенным вне системы ТМ. Если транзакция аварийно завершается, ее операции ввода-вывода также требуется откатить, что в общем случае сделать

очень трудно, если не невозможно. Некоторые откаты позволяет производить буферизация данных, прочитанных или записанных транзакцией, но буферизация не удается даже в простых ситуациях, например, когда транзакция выдает приглашение и ожидает ввода данных от пользователя. Более общий подход состоит в назначении одной из транзакций привилегированного статуса, которой всегда гарантируется возможность завершения в ущерб всем конфликтующим с ней транзакцией. Ввод-вывод может производиться только привилегированной транзакцией (но эта привилегия может передаваться между транзакциями), что, к сожалению, ограничивает объем ввода-вывода, который может произвести программа.

Другой важной проблемой является сильная и слабая атомарность. В системах STM, вообще говоря, реализуется слабая атомарность, когда не транзакционный код не изолируется от кода транзакций. С другой стороны, в системах HTM реализуется сильная атомарность, обеспечивающая более детерминированную программную модель, в которой не транзакционный код не влияет на атомарность транзакции. Из-за этого различия возникает несколько проблем. Кроме потребности в ответе на основной вопрос (какая модель является лучшим базисом для написания программного обеспечения?), это семантическое различие затрудняет разработку программного обеспечения, которое может выполняться в системах обоих типов. Наименьшим общим знаменателем является слабая модель, но ошибочные программы будут производить на разных системах отличающиеся результаты. Альтернативная точка зрения состоит в том, что доступ к общим данным в двух потоках управления без синхронизации вообще является ошибкой, и если только в одном потоке выполняется транзакция, то это не является достаточной синхронизацией между потоками. Следовательно, языки программирования, инструментальные средства, системы поддержки времени выполнения или аппаратура должны предотвращать или выявлять совместную работу с данными без синхронизации транзакционного и не транзакционного кода, и программисты должны исправлять этот дефект.

Системы со слабой атомарностью также сталкиваются с трудностями, когда некоторый объект совместно используется транзакционным и не транзакционным кодом. Когда некоторый поток управления делает некоторый объект видимым для других потоков (например, путем добавления его к некоторой глобальной очереди), происходит публикация этого объекта. Когда некоторый поток управления удаляет некоторый объект из разделяемого глобального пространства, происходит приватизация этого объекта. Частные (приватные) данные можно обрабатывать за пределами транзакции без синхронизации, но изменение состояния объекта с публичного на частный и наоборот должно координироваться системой TM, чтобы она не пыталась откатить состояние объекта, в то время как другой поток управления предполагает, что имеет к этому объекту исключительный, частный доступ.

Наконец, TM должна сосуществовать и взаимодействовать с существующими программами и библиотеками. Непрактично требовать от программистов начать все с начала и полльзоваться новым набором транзакционных библиотек, чтобы использовать преимущества TM. Должна иметься возможность корректно исполнять в транзакции существующий

последовательный код, возможно, с небольшим числом аннотаций и специальной компиляцией. Существующий параллельный код, в котором используются блокировки и другие формы синхронизации, должен продолжать выполнять правильно, даже если в некоторых потоках управления выполняются транзакции.

#### 2.2.3.14 Заключение

Вряд ли транзакционная память сама по себе сделает многоядерные компьютеры полностью пригодными для программирования. Для этого также требуются многочисленные усовершенствования языков программирования, инструментальных средств, систем поддержки времени выполнения и компьютерных архитектур. Однако TM обеспечивает проверенную временем модель для взаимной изоляции одновременно проводимых вычислений. Эта модель повышает уровень абстракции для понимания параллельных задач и помогает избежать многочисленных коварных ошибок параллельного программирования. Однако многие аспекты семантики и реализации TM все еще остаются предметом активных исследований. Если эти трудности удастся своевременно преодолеть, то TM, вероятно, станет основным принципом параллельного программирования.

## 2.3 Параллельное программирование

Параллельные вычисления — способ организации компьютерных вычислений, при котором программы разрабатываются как набор взаимодействующих вычислительных процессов, работающих параллельно. Термин охватывает совокупность вопросов параллелизма в программировании, а также создание эффективно действующих аппаратных реализаций.

Понятие параллельного алгоритма (Parallel Algorithm) относится к фундаментальным в теории вычислительных систем. Это понятие, прежде всего, ассоциируется с вычислительными системами с массовым параллелизмом. Параллельный алгоритм – это описание процесса обработки информации, ориентированное на реализацию в коллективе вычислителей. Такой алгоритм, в отличие от последовательного, предусматривает одновременное выполнение множества операций в пределах одного шага вычислений и как последовательный алгоритм сохраняет зависимость последующих этапов от результатов предыдущих.

Методы и алгоритмы обработки информации, решения задач, как правило, – последовательные. Процесс “приспособления” методов к реализации на коллективе вычислителей или процесс “расщепления” последовательных алгоритмов решения сложных задач называется распараллеливанием (Paralleling or Multisequencing). Теоретическая и практическая деятельность по созданию параллельных алгоритмов и программ обработки информации называется параллельным программированием (Parallel or Concurrent Programming).

В информатике параллельный алгоритм, противопоставляемый традиционным последовательным алгоритмам, — алгоритм, который может быть реализован по частям на множестве различных вычислительных устройств с последующим объединением полученных результатов и получением корректного результата.

Параллельные алгоритмы весьма важны ввиду постоянного совершенствования многопроцессорных систем и увеличения числа ядер в современных процессорах. Обычно проще сконструировать компьютер с одним быстрым процессором, чем с множеством медленных процессоров (при условии достижения одинаковой производительности). Однако производительность процессоров увеличивается главным образом за счет совершенствования техпроцесса (уменьшения норм производства), чему мешают физические ограничения на размер элементов микросхем и тепловыделение. Указанные ограничения могут быть преодолены путем перехода к многопроцессорной обработке, что оказывается эффективным даже для малых вычислительных систем.

Параллельное программирование является более сложным по сравнению с последовательным как в написании кода, так и в его отладке. Основная сложность при проектировании параллельных программ — обеспечить правильную последовательность взаимодействий между различными вычислительными процессами, а также координацию ресурсов, разделяемых между процессами. При использовании систем с общей памятью, если потоки обращаются к одним и тем же данным, возможны повреждения этих данных в результате асинхронных изменений (состояние гонки). Во избежание коллизий необходима синхронизация доступа потоков к разделяемым данным.

# 3 ПРОЕКТИРОВАНИЕ И РЕАЛИЗАЦИЯ

## 3.1 Постановка задачи

Целью работы является разработка потокобезопасной параллельной программы для анализа эффективности использования транзакционной памяти. Было решено разработать потокобезопасный ассоциативный контейнер для оценки производительности транзакционной памяти. Для этого необходимо проанализировать потокобезопасную структуру на основе транзакционной памяти путём замера времени выполнения программы в целом. Далее требовалось произвести анализ производительности алгоритмов выполнения транзакций, сравнить реализацию с другими средствами синхронизации потоков (PThread mutex, PThread spinlock) и провести тесты на многоядерной вычислительной системе для определения эффективности реализации алгоритмов транзакционной памяти и ассоциативного контейнера в целом.

## 3.2 Инструментарий

Для решения поставленной задачи использовался язык C/C++14 из пакета GCC 5.3.0 (на момент реализации последняя версия), библиотека GNU libitm, алгоритмы транзакционной памяти gl_wt, ml_wt, serial, serialirr, serialirr_onwrite из библиотеки GNU libitm, так же для сравнения транзакционной памяти были использованы примитивы синхронизации потоков mutex, spinlock из библиотеки GNU libpthread. Для замера времени выполнения программы в целом была использована функция wtime() из библиотеки GNU sys/time.h, для выбора случайных операций при проведении тестов была использована функция rand() из библиотеки GNU stdlib.h.

Для реализации ассоциативного контейнера была выбрана структура данных – красно-черное дерево. Эта структура была выбрана потому что она используется в ассоциативных контейнерах, ядре GNU Linux, поисковых системах.

## 3.3 Библиотрека libitm

Начиная с версии 4.7, GCC поддерживает транзакционную память. Реализация представляет собой библиотеку времени выполнения libitm, для компиляции указывается флаг -fgnu-tm (-mrtm, -mhle). Библиотека разрабатывалась с оглядкой на черновик транзакционных конструкций для C (предлагается включение в эталон языка).

Множество реализаций аппаратной транзакционной памяти применяют правило наибольшего усилия. Следственно практичные реализации применяют объединение спецтехнологий аппаратурной и программной транзакционной памяти. Такие системы называют системами «гибридной транзакционной памяти». К ним относится, в частности, реализация GCC.

В язык вводятся ключевые слова:

- `__transaction_atomic{…}` — указание, что блок кода — транзакция;
- `__transaction_relaxed{…}` — указание, что опасный код внутри блока не приводит к побочным результатам;

- `__transaction_cancel` — очевидная отмена транзакции;
- `attribute((transaction_safe))` — указание транзакционно неопасной функции;
- `attribute((transaction_pure))` — указание функции без побочных результатов.

## 3.4 Красно-черное дерево

Красно-черные деревья - один из способов балансировки деревьев. Название происходит от стандартной раскраски узлов таких деревьев в красный и черный цвета. Цвета узлов используются при балансировке дерева. Во время операций вставки и удаления поддеревья может понадобиться повернуть, чтобы достигнуть сбалансированности дерева. Оценкой как среднего время, так и наихудшего является *O(log n)*.

Красно-черное дерево - это бинарное дерево с следующими свойствами:

- Каждый узел покрашен либо в черный, либо в красный цвет.
- Листьями объявляются NIL-узлы (т.е. "виртуальные" узлы, наследники узлов, которые обычно называют листьями; на них "указывают" NULL указатели). Листья покрашены в черный цвет.
- Если узел красный, то оба его потомка черны.
- На всех ветвях дерева, ведущих от его корня к листьям, число черных узлов одинаково.

Количество черных узлов на ветви от корня до листа называется черной высотой дерева. Перечисленные свойства гарантируют, что самая длинная ветвь от корня к листу не более чем вдвое длиннее любой другой ветви от корня к листу. Чтобы понять, почему это так, рассмотрим дерево с черной высотой 2. Кратчайшее возможное расстояние от корня до листа равно двум - когда оба узла черные. Длиннейшее расстояние от корня до листа равно четырем - узлы при этом покрашены (от корня к листу) так: красный, черный, красный, черный. Сюда нельзя добавить черные узлы, поскольку при этом нарушится свойство 4, из которого вытекает корректность понятия черной высоты. Поскольку согласно свойству 3 у красных узлов непременно черные наследники, в подобной последовательности недопустимы и два красных узла подряд. Таким образом, длиннейший путь, который мы можем сконструировать, состоит из чередования красных и черных узлов, что и приводит нас к удвоенной длине пути, проходящего только через черные узлы. Все операции над деревом должны уметь работать с перечисленными свойствами. В частности, при вставке и удалении эти свойства должны сохраниться.

## 3.5 Реализация

Ассоциативный контейнер на основе транзакционной памяти с использованием библиотеки GNU libitm в представлении красно-черного дерева, реализован в виде статической библиотеки (librbt-tm.a) на языке C++ с использованием 14 стандарта.

Для проведения тестов разработан алгоритм, который случайным образом выбирает операции – вставить, удалить случайные значения из ассоциативного контейнера.

Для автоматизации выполнения тестирования разработан скрипт на языке bash, позволяющий проводить тесты с заданным количеством потоков, операций, повторений и выбором примитивов синхронизаций потоков. Доступны следующие виды синхронизации потоков: транзакционная память (алгоритмы: gl_wt, ml_wt, serial, serialirr, serialirr_onwrite), PThread mutex, PThread spinlock. Результат выполнения теста сохраняется в файл с именем примитива синхронизации потоков.

## 3.6 Описание библиотеки librbt-tm

Реализация библиотеки состоит из двух файлов, одного заголовочного файла (rbt-tm.h) содержащий класс потокобезопасного ассоциативного контейнера на основе транзакционной памяти и одного файла с исходным кодом класса (rbt-tm.cpp).

В библиотеке реализованы следующий интерфейс для работы с потокобезопасным ассоциативным контейнером:

- `void Insert(int)` – метод для вставки в ассоциативный контейнер;
- `void Remove(int)` – метод для удаления из ассоциативного контейнера;
- `void Show()` – метод для отображения ассоциативного контейнера.

Для использования библиотеки librbt-tm необходимо подключить заголовочный файл rbt-tm.h, так же при компиляции необходимо указать путь до статической библиотеки с помощью флага –L, указать флаг подключаемой библиотеки -lrbt-tm и указать флаг библиотеки содержащая алгоритмы программной транзакционной памяти – -fgnu-tm.

Листинг 3.1 - Пример подключения библиотеки rbt-tm.h

```
#include "../library/rbt-tm.h"
```

Листинг 3.2 - Пример команды компиляции

```
$g++ -std=c++14 main-tm.cpp -L../library -lrbt-tm -o main-
tm -fgnu-tm
```

## 3.7 Сборка библиотеки librbt-tm

Для сборки библиотеки предусмотрен makefile позволяющий собирать библиотеку (make all), удалять библиотеку (make clean), устанавливать в заданную директорию (make install PREFIX=указать_директорию), удалять из заданной директории (make uninstall PREFIX=указать_директорию).

Листинг 3.3 - Пример сборки библиотеки

```
$make
$make install
```

В данном примере будет собрана библиотека и установлена в текущую папку с makefile, в папке librbt-tm содержащая файлы – rbt-tm.h, librbt-tm.a

Листинг 3.4 - Содержимое makefile

```
.PHONY: all clean install uninstall
TARGET = librbt-tm.a
PREFIX = .
NAMEDIR = librbt-tm
all: rbt-tm.cpp
g++-5 -std=c++14 -Wall -c rbt-tm.cpp -fgnu-tm
ar rc $(TARGET) rbt-tm.o
rm -rf *.o
clean:
rm -rf $(TARGET)
install:
mkdir $(PREFIX)/$(NAMEDIR)
install rbt-tm.h $(PREFIX)/$(NAMEDIR)
install $(TARGET) $(PREFIX)/$(NAMEDIR)
uninstall:
rm -rf $(PREFIX)/$(NAMEDIR)
```

## 3.8 Алгоритм выбора случайных операций потоками

Данный алгоритм позволяет выбирать потокам случайные операции вставки и удаления случайных значений.

Листинг 3.5 - Исходный код алгоритмы

```
static void func1(RBtree &tree, size_t ITER_NUM){
for(size_t i(0); i < ITER_NUM; ++i)
          tree.Insert(0 + rand() % ITER_NUM);
}

static void func2(RBtree &tree, size_t ITER_NUM){
for(size_t i(0); i < ITER_NUM; ++i)
          tree.Remove(0 + rand() % ITER_NUM);
}
…
for(auto &thr: pool){
    auto r = 0 + rand() % 2;
    if(r == 0)
     thr = thread(func1, std::ref(tree), ITER_NUM);
    else
     thr = thread(func2, std::ref(tree), ITER_NUM);
}
```

## 3.9 Скрипт для автомаизации тестирования

Скрипт позволяет выбирать до какого количества потоков выпонять тест, количество операция на все потоки, количество замеров времени выполнения программы, алгоритм транзакций или других методов синхронизации потоков.

Листинг 3.6 - Исходный код

```
#!/bin/bash
#До скольки потоков выполнять тест
THR_NUM=$1
#Количество операций
ITER_NUM=$2
#Количество тестов
LIMIT=$3
#Выбор алгоритма транзакций
ITM_METHOD=$4
#Количество операций на поток
SIZE_THREAD=$(echo "scale=0; $ITER_NUM/$THR_NUM" | bc)

#echo "До $THR_NUM потоков"
#echo "Количество операций на все потоки: $ITER_NUM"
#echo "Количество тестов: $LIMIT"

PROG=main-tm

#Выбор метода
case $ITM_METHOD in

     gl_wt)
          #echo "Метод: gl_wt"
          echo "Метод: gl_wt" > "$ITM_METHOD.txt"
          export ITM_DEFAULT_METHOD=gl_wt
     ;;
     ml_wt)
          #echo "Метод: ml_wt"
          echo "Метод: ml_wt" > "$ITM_METHOD.txt"
          export ITM_DEFAULT_METHOD=ml_wt
     ;;
     serial)
          #echo "Метод: serial"
          echo "Метод: serial" > "$ITM_METHOD.txt"
          export ITM_DEFAULT_METHOD=serial
     ;;
     serialirr)
          #echo "Метод: serialirr"
          echo "Метод: serialirr" > "$ITM_METHOD.txt"
          export ITM_DEFAULT_METHOD=serialirr
     ;;
     serialirr_onwrite)
          #echo "Метод: serialirr_onwrite"
```

```
          echo "Метод: serialirr_onwrite" >
"$ITM_METHOD.txt"
          export ITM_DEFAULT_METHOD=serialirr_onwrite
     ;;
     mutex)
          #echo "Метод: PThread mutex"
          echo "Метод: PThread mutex" >
"$ITM_METHOD.txt"
          PROG=main-me
     ;;
     spinlock)
          #echo "Метод: PThread spinlock"
          echo "Метод: PThread spinlock" >
"$ITM_METHOD.txt"
          PROG=main-sl
     ;;
     *)
          echo "Выбран неправельный метод!"
esac

#Запуск теста
var0=0
var1=1
while [ "$var1" -le "$THR_NUM" ]
do
     echo "Количество потоков: $var1" >>
"$ITM_METHOD.txt"
     while [ "$var0" != "$LIMIT" ]
     do
          ./$PROG $var1 $ITER_NUM >> "$ITM_METHOD.txt"
          var0=$(($var0+1))
     done
     var0=0
     var1=$(($var1+1))
done
```

Листинг 3.7 - Пример использования скрипта

```
$./test 50 1000000 30 serial
$./test 65 1000000 30 serialirr
$./test 15 1000000 30 mutex
$./test 8 1000000 30 spinlock
```

- Первый аргумент скрипта – количество потоков;
- Второй аргумент скрипта – количество операций на все потоки;
- Третий аргумент скрипта – количество замеров времени выполнения;
- Четвертый аргумент скрипта – выбор алгоритма.

По завершению выполнения скрипта создается файл с именем алгоритма транзакции/других примитивов синхронизации, содержащий поля со следующие данными: количество потоков, время выполнения программы.

## 3.10 Описание тестов

Тестирование проводилось с использованием скрипта, описанного в разделе 3.10 При тестировании использовались следующие параметры скрипта: до 100 потоков, 1000000 операций, 30 замеров времени выполнения программы, алгоритмы транзакционной памяти (gl_wt, ml_wt, serial, serialirr, serialirr_onwrite) и другие средства синхронизации потоков (PThread mutex, PThread spinlock).

## 3.11 Описание системы на которой проводились эксперименты

Эксперименты проводились на вычислительной системе Jet ФГОБУ ВПО «СибГУТИ».

Характеристики узла вычислительного кластера Центра параллельных вычислительных технологий СибГУТИ:

- Процессор: 2 x Intel Xeon E5420(2,5 ГГц, 4 ядра);
- Оперативная память: 8 GB;
- Количество потоков: *n* варьировалось от 1 до 100.

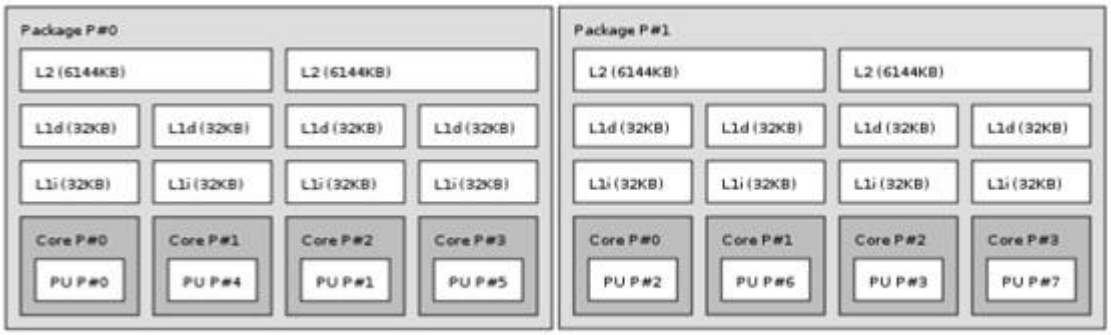


Рисунок 3.1 – Структурная схема вычислительного узла Jet

## 3.12 Результаты моделирования

Проведен сравнительный анализ эффективности выполнения транзакций с помощью различных алгоритмов программной транзакционной памяти из библиотеки GNU libitm и стандартных блокировок из библиотеки GNU libpthread.

Результаты моделирования представлены в виде графиков на рисунках 3.2-3.6. Для анализа эффективности использовался разработанный потокобезопасный ассоциативный контейнер с использование примитивов синхронизации: транзакционная память (алгоритмы: gl_wt, ml_wt, serial, serialirr, serialirr_onwrite), PThread mutex, PThread spinlock.

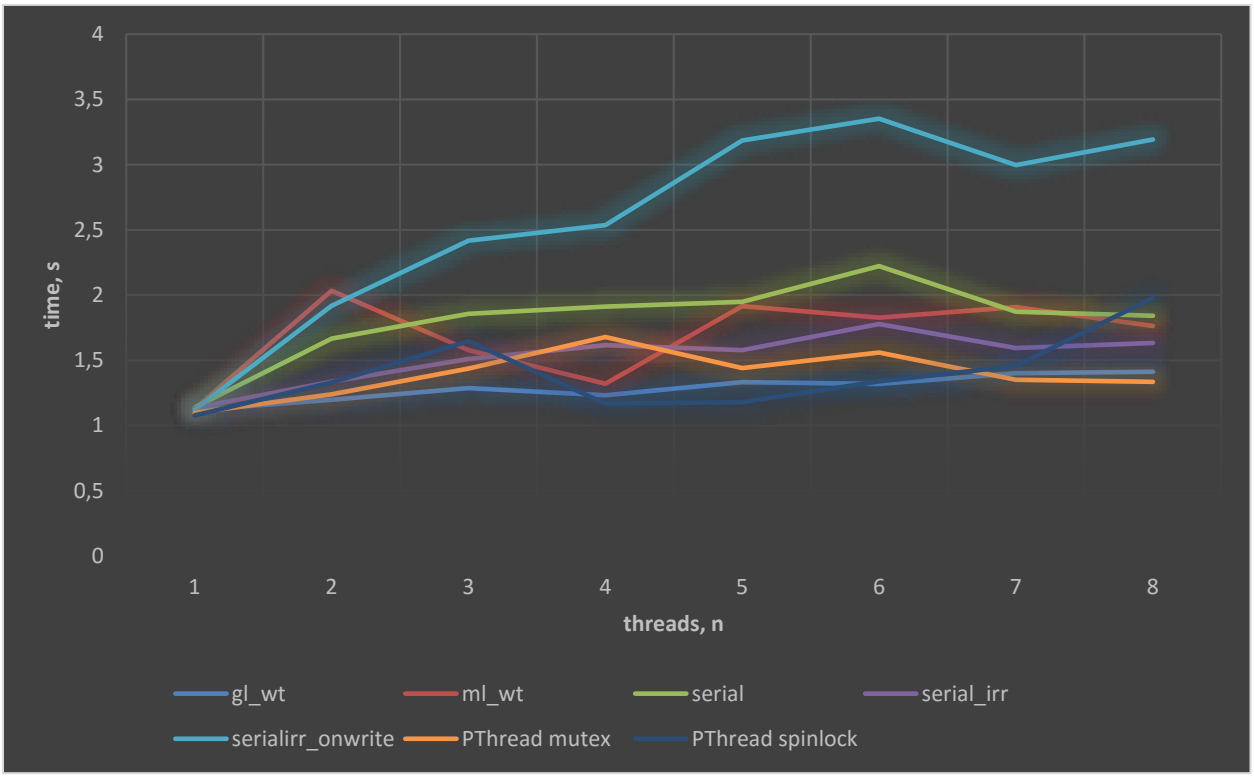


Рисунок 3.2 - Сравнение алгоритмов реализации транзакционной памяти в GCC 5.3.0 с PThread mutex, PThread spinlock до 8 потоков

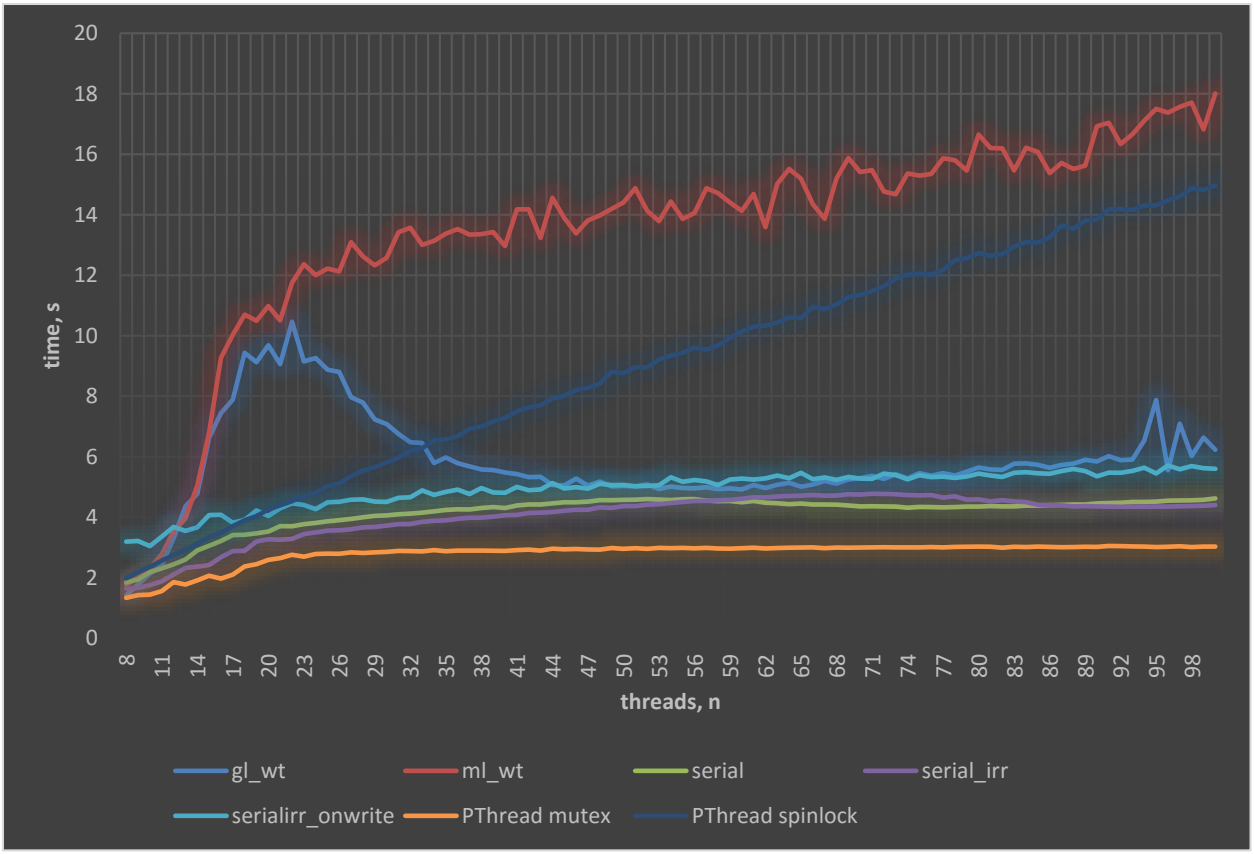


Рисунок 3.3 - Сравнение алгоритмов реализации транзакционной памяти в GCC 5.3.0 с PThread mutex, PThread spinlock от 8 до 100 потоков

На рисунке 3.2 показан результат теста выполнения алгоритмов транзакций до 8 потоков, из данного рисунка 3.2 очевидно, что алгоритм gl_wt транзакционной памяти является более эффективным по производительности и стабильным по времени выполнения операций чем Pthread spinlock и Pthread mutex. По данному рисунку 3.2 можно сделать вывод, что алгоритм gl_wt транзакционной памяти выполняет транзакции параллельно используя каждое вычислительное ядро вычислительной системы и не создает нагрузку из очереди потоков на вычислительном узле. Таким образом исходя из результатов проведенного теста алгоритмов транзакционной памяти до 8 потоков, а также сравнение с Pthread spinlock и Pthread mutex алгоритм gl_wt транзакционной памяти является более эффективным, что потвердел тест.

На рисунке 3.3 показан результат теста выполнения алгоритмов транзакций до 100 потоков, из данного рисунка 3.3 очевидно, что алгоритмы транзакционной памяти при использовании более 8 потоков не настолько эффективны по сравнению с Pthread mutex, но стоит отметить что алгоритмы транзакционной памяти serial, serialirr, serialirr_onwrite хоть и показывают менее эффективную производительность по времени выполнения операций, но являются стабильными как и Pthread mutex. Что же касается алгоритмов сравнения транзакционной памяти с Pthread spinlock, то тут можно отметить что транзакционная память является более эффективной по производительности и времени выполнения операций за исключением алгоритма ml_wt транзакционной памяти. Алгоритм ml_wt транзакционной памяти показал наименее эффективные результаты в данном тесте.

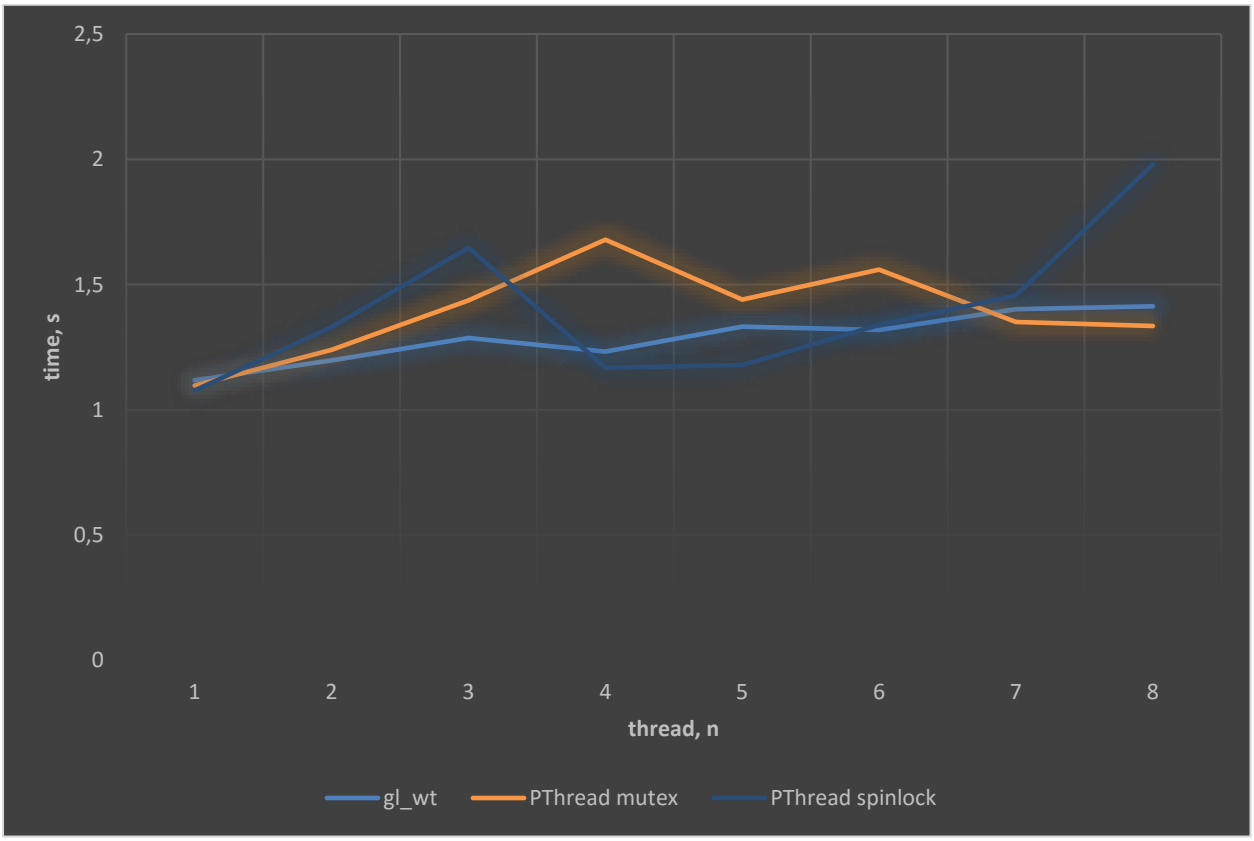


Рисунок 3.4 - Сравнение потокобезопасного красно-черного дерева на основе GCC TM (global lock, gl), PThread mutex, PThread spinlock до 8 потоков

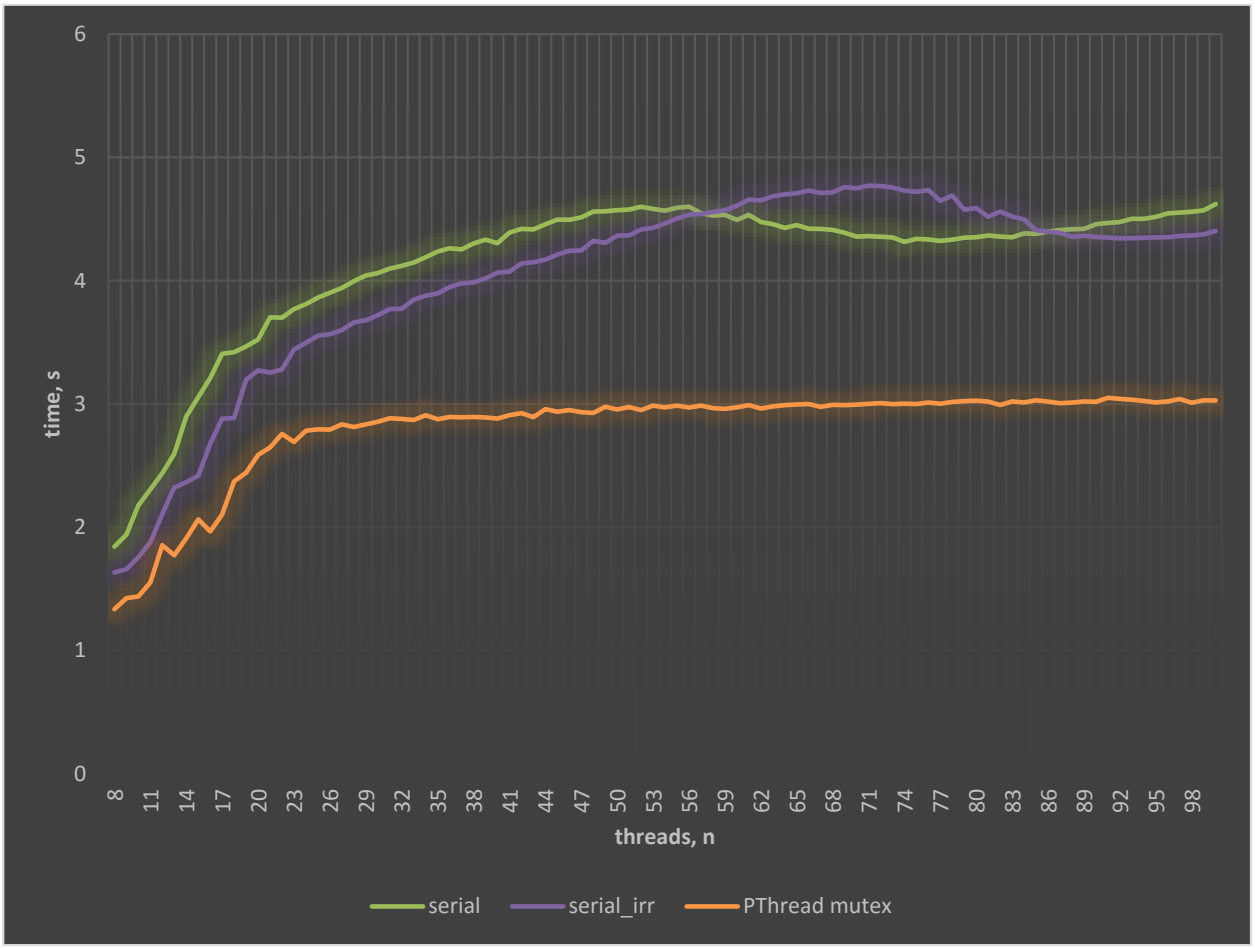


Рисунок 3.5 - Сравнение потокобезопасного красно-черного дерева на основе GCC TM (serial, serialirr), PThread mutex от 8 до 100 потоков

На рисунке 3.4 и 3.5 представлены более эффективные алгоритмы транзакций до 8 потоков и после 8 потоков исходя из проведенных тестов

## 3.13 Обработка результатов измерений

Проведена обработка результатов измерений путём расчёта доверительного интервала для ассоциативного контейнера на основе транзакционной памяти и PThread mutex, PThread spinlock.

На рисунке 3.6 представлен рассчитанный доверительный интервал для всех алгоритмов транзакционной памяти, а так же для Pthread mutex, Pthread spinlock. Опираясь на полученные показания доверительного интервала на рисунке 3.6 можно сделать вывод что самые точные и эффективные показания даёт алгоритм gl_wt транзакционной памяти до 8 потоков, алгоритмы serial, serialirr транзакционной памяти до 100 потоков. Так же точный и эффективный доверительный интервал у Pthread mutex от 1 потока до 8 и от 8 до 100 потоков.

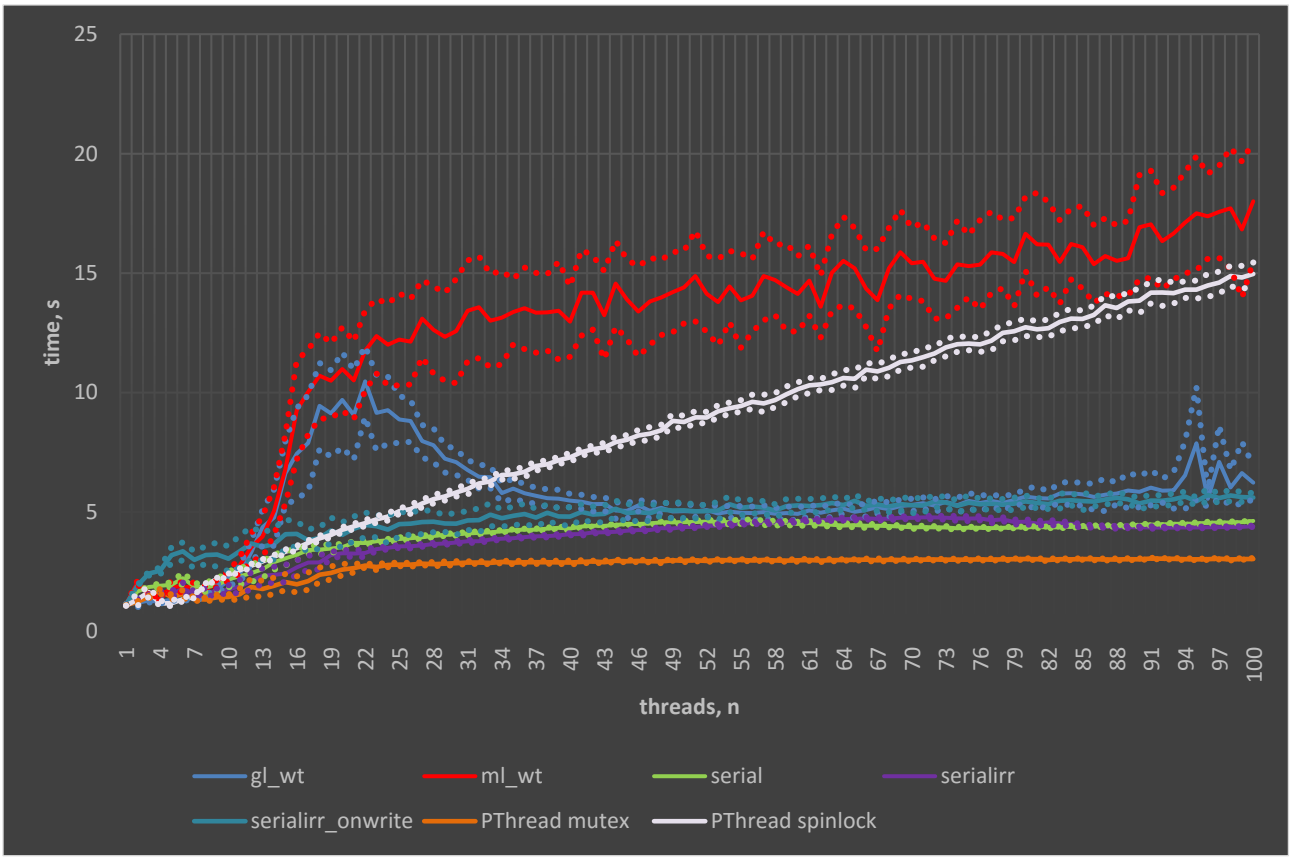


Рисунок 3.6 – Доверительный интервал для потокобезопасного ассоциативного контейнера на основе транзакционной памяти и PThread mutex, PThread spinlock до 100 потоков

# 4 ЗАКЛЮЧЕНИЕ

В ходе выполнения работы реализован потокобезопасный ассоциативный контейнер на основе транзакционной памяти с использованием языка C/C++14 из пакета GNU GCC 5.3.0 и библиотеки GNU libitm. Проведен сравнительный анализ эффективности алгоритмов выполнения транзакций, так же проведено сравнение транзакционной памяти с PThread mutex, PThread spinlock.

Были проведены эксперименты, в ходе которых было установлено, что использование программной транзакционной памяти эффективно до 8 потоков по сравнению с PThread mutex, PThread spinlock. Использование транзакционной памяти более 8 потоков не эффективно по сравнению с PThread mutex.

# ПРИЛОЖЕНИЕ А

(справочное)

Библиография


1 Хорошевский, В.Г. Архитектура вычислительных систем : Учеб. пособие / В.Г. Хорошевский. – 2-е изд., перераб. и доп. – М.: Изд-во МГТУ им. Н.Э. Баумана, 2008. – 520 с. – (Информатика в техническом университете, ISBN 987-5-7038-3175-5)

2 Токарь Т. М. Оптимизация потокобезопасных красно-чёрных деревьев на основе транзакционной памяти // Материалы 54-й Международной научной студенческой конференции МНСК-2016 "Информационные технологии" / Новосиб. гос. ун-т. Новосибирск, 2016. С. 153.

3 Токарь Т.М., Пазников А.А. Исследование эффективности потокобезопасного красно-чёрного дерева на основе транзакционной памяти // Материалы Российской научно-технической конференции “Обработка информации и математическое моделирование”. – 2016. – С. 373-379.

4 Курносов М.Г. Введение в структуры и алгоритмы обработки данных (учебное пособие). - Новосибирск: Автограф, 2015. - 179 с. (ISBN 978-5-9906983-4-5)

5 Кулагин И.И., Курносов М.Г. О подходах к реализации программной транзакционной памяти // Российская научно-техническая конференция «Обработка информации и математическое моделирование», 2016. – Новосибирск: СибГУТИ, 2016. – С. 331-338.

6 Patrick Marlier. Brief Transactional Memory GCC tutorial. URL: http://pmarlier.free.fr/gcc-tm-tut.html (дата обращения 24.02.2016)

7 Gitu Jain. Memory models for embedded multicore SoCs, Part 6 - Transactional memory URL: http://www.edn.com/design/systems-design/4421606/3/Memory-models-for-embedded-multicore-SoCs--Part-6---Transactional-memory (дата обращения 16.02.2016).

8 Herlihy M., Moss J. E. B. Transactional memory: Architectural support for lock-free data struc-tures. ACM. 1993. № 2. 289-300 с.

9 Shavit N., Touitou D. Software transactional memory. 1997. № 2. 99-116 с.

10 Felber P., Fetzer C., Riegel T. Dynamic performance tuning of word-based software transactional memory. ACM. 2008. 237-246 с.

11 Riegel T., Fetzer C., Felber P. Time-based transactional memory with scalable time bases. ACM. 2007. 221-228 с.

12 T. Riegel. Software Transactional Memory Building Blocks. PhD thesis. Technischen Universi-tat Dresden, geborenam 1.3.1979 in Dresden. 2013.

13 Herlihy M., Shavit N. The Art of Multiprocessor Programming. Morgan Kaufmann, NY, USA. 2008. P. 245-286

14 Shavit N., Touitou D. Software transactional memory. 1997, № 2, P. 99-116.

15 T. Riegel. Software Transactional Memory Building Blocks. PhD thesis, Technischen Universitat Dresden, geborenam 1.3.1979 in Dresden, 2013.

16 Adl-Tabatabai, A.R., Lewis, B.T., Menon, V., Murphy, R., Saha, B., and Shpeisman, T. Abstract compiler and runtime support for efficient software transactional memory. In Proceedings of the 2006 ACM SIGPLAN Conference on

Programming Language Design and Implementation (Ottawa, Ontario, Canada, 2006). ACM, NY 26–37.
17 Blundell, C., Lewis, E.C., and Martin, M.M.K. Subtleties of transactional memory atomicity semantics. IEEE Computer Architecture Letters 5 (Nov. 2006).
18 Carlstrom, B. D., McDonald, A., Carbin, M., Kozyrakis, C., and Olukotun, K. Transactional collection classes. In Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (San Jose, CA, 2007). ACM, NY, 56–67.
19 Carlstrom, B.D., McDonald, A., Chafi, H., Chung, J., Minh, C.C., Kozyrakis, C., and Olukotun, K. The Atomos transactional programming language. In Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation (Ottawa, Ontario, Canada, 2006). ACM, NY, 1–13.
20 Damron, P., Fedorova, A., Lev, Y., Luchangco, V., Moir, M., and Nussbaum, D. Hybrid transactional memory. In Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (San Jose, CA, 2006). ACM, NY, 336–346.
21 Gray, J. and Reuter, A. Transaction Processing: Concepts and Techniques. Morgan Kaufmann Publishers, San Francisco, CA, 1992.
22 Grossman, D. The transactional memory / garbage collection analogy. In Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (Montreal, Canada, 2007). ACM, NY, 695–706.
23 Harris, T., Marlow, S., Peyton-Jones, S., and Herlihy, M. Composable memory transactions. In Proceedings of the 10th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (Chicago, IL, 2005), ACM, NY, 48–60.
24 Harris, T., Plesko, M., Shinnar, A., and Tarditi, D. Optimizing memory transactions. In Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation (Ottawa, Ontario, Canada, 2006). ACM, NY, 14–25.
25 Kumar, S., Chu, M., Hughes, C.J., Kundu, P., and Nguyen, A. Hybrid transactional memory. In Proceedings of the 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. ACM, NY, 2006, 209–220.
26 Larus, J.R. and Rajwar, R. Transactional Memory. Morgan & Claypool, 2006.
27 McDonald, A., Chung, J., Brian, D.C., Minh, C.C., Chafi, H., Kozyrakis, C., and Olukotun, K. Architectural semantics for practical transactional memory. In Proceedings of the 33rd International Symposium on Computer Architecture. ACM, 2006, 53–65.
28 Minh, C. C., Trautmann, M., Chung, J., McDonald, A., Bronson, N., Casper, J., Kozyrakis, C., and Olukotun, K. An effective hybrid transactional memory system with strong isolation guarantees. In Proceedings of the 34th International Symposium on Computer Architecture (San Diego, CA, 2007) ACM, NY, 69–80.
29 Rajwar, R., Herlihy, M., and Lai, K. Virtualizing transactional memory. In Proceedings of the 32nd International Symposium on Computer Architecture (Madison, WI, 2005). ACM. NY, 494–505.
30 Saha, B., Adl-Tabatabai, A. R., and Jacobson, Q. Architectural support for software transactional memory. In Proceedings of the 39th International Symposium on Microarchitecture (Orlando, FL, 2006). IEEE, 185–196.

31 Scherer III, W.N., and Scott, M.L. Advanced contention management for dynamic software transactional memory. In Proceedings of the Twentyfourth Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (Las Vegas, NV, 2005). ACM Press, 240–248.
32 Shavit, N. and Touitou, D. Software transactional memory . In Proceedings of the 14th ACM Symposium on Principles of Distributed Computing (Ottawa, Canada, 1995). ACM, NY, 204–213.
33 Shpeisman, T., Menon, V., Adl-Tabatabai, A.-R., Balensiefer, S., Grossman, D., Hudson, R.L., Moore, K.F., and Saha, B. Enforcing isolation and ordering in STM. In Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation (San Diego, CA, 2007). ACM, NY, 78–88.
34 Shriraman, A., Spear, M.F., Hossain, H., Marathe, V.J., Dwarkadas, S., and Scott, M.L. An integrated hardware-software approach to flexible transactional memory. In Proceedings of the 34th International Symposium on Computer Architecture (San Diego, CA, 2007). ACM, NY, 104–115.
35 Zilles, C. and Baugh, L. Extending hardware transactional memory to support nonbusy waiting and nontransactional actions. In Proceedings of the First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing (Ottawa, Canada, 2006). ACM, NY.

# ПРИЛОЖЕНИЕ Б

(рекомендуемое)

Наиболее употребляемые текстовые сокращения

ВС – вычислительная система

ПЗ – пояснительная записка

СибГУТИ – Сибирский государственный университет телекоммуникаций и информатики

ТМ – Transactional Memory

# ПРИЛОЖЕНИЕ В

(Исходный код)
Реализация статической библиотеки librbt-tm

Листинг В.1 – Исходный код заголовочного файла rbt-tm.h

```
#ifndef RBT_TM_H
#define RBT_TM_H
class RBtree
{
private:
    struct node_st{node_st *p1, *p2; int value; bool red;};
    node_st *tree_root;
    //Создание узла
    node_st *NewNode(int);
    //Удление узла
    void DelNode(node_st*);
    //Удаление дерева
    void Clear(node_st*);
    //Поворот 21
    node_st *Rotate21(node_st*);
    //Поворот 12
    node_st *Rotate12(node_st*);
    //Балансировка дерева
    void BalanceInsert(node_st**);
    //Вставка в дерево
    bool Insert(int value,node_st**);
    //Балансировка при удалении 1
    bool BalanceRemove1(node_st**);
    //Балансировка при удалении 2
    bool BalanceRemove2(node_st**);
    //Минимальный элемент
    bool GetMin(node_st**, node_st**);
    //Удаление из дерева
    bool Remove(node_st**, int);
    //Вывод дерева
    void Show(node_st*, int, char);
public:
    //Конструктор
    RBtree();
    //Вставка в дерево
    void Insert(int);
    //Удаление из дерева
    void Remove(int);
    //Вывод дерева
    void Show();
    //Деструктор
    ~RBtree();
};
#endif
```

Листинг В.2 – Исходный код реализации rbt-tm.cpp

```
#include <iostream>
#include <cstdio>
#include "rbt-tm.h"

using std::endl;
using std::cout;

    //Создание узла
    RBtree::node_st* RBtree::NewNode(int value){
        node_st *node = new node_st;
        node->value = value;
        node->p1 = 0;
        node->p2 = 0;
        node->red = true;
        return node;
    }

    //Удление узла
    void RBtree::DelNode(node_st *node){
        delete [] node;
    }

    //Удаление дерева
    void RBtree::Clear(node_st *node){
        if(!node)
            return;
        Clear(node->p1);
        Clear(node->p2);
        DelNode(node);
    }

    //Поворот 21
     RBtree::node_st* RBtree::Rotate21(node_st *node){
        node_st *p2 = node->p2;
        node_st *p21 = p2->p1;
        p2->p1 = node;
        node->p2 = p21;
        return p2;
    }

    //Поворот 12
    RBtree::node_st* RBtree::Rotate12(node_st *node){
        node_st *p1 = node->p1;
        node_st *p12=p1->p2;
        p1->p2=node;
        node->p1=p12;
        return p1;
    }

    //Балансировка дерева
    void RBtree::BalanceInsert(node_st **root){
        node_st *p1,*p2,*px1,*px2;
        node_st *node=*root;
```

```
        if(node->red) return;
        p1=node->p1;
        p2=node->p2;
        if(p1 && p1->red) {
            px2=p1->p2;
            if(px2 && px2->red) p1=node->p1=Rotate21(p1);
            px1=p1->p1;
            if(px1 && px1->red) {
                node->red=true;
                p1->red=false;
                if(p2 && p2->red) {
                    px1->red=true;
                    p2->red=false;
                    return;
                }
                *root=Rotate12(node);
                return;
            }
        }
        // тоже самое в другую сторону
        if(p2 && p2->red) {
            px1=p2->p1;
            if(px1 && px1->red) p2=node->p2=Rotate12(p2);
            px2=p2->p2;
            if(px2 && px2->red) {
                node->red=true;
                p2->red=false;
                if(p1 && p1->red) {
                    px2->red=true;
                    p1->red=false;
                    return;
                }
                *root=Rotate21(node);
                return;
            }
        }
    }

    //Вставка в дерево
    bool RBtree::Insert(int value,node_st **root){
        node_st *node=*root;
        if(!node) *root=NewNode(value);
        else {
            if(value==node->value) return true;
            if(Insert(value,value<node->value?&node->p1:&node->p2))
return true;
            BalanceInsert(root);
        }
        return false;
    }


//Балансировка при удалении 1
    bool RBtree::BalanceRemove1(node_st **root){
```

```
        node_st *node=*root;
            node_st *p1=node->p1;
            node_st *p2=node->p2;
            if(p1 && p1->red) {
                p1->red=false;return false;
            }
            if(p2 && p2->red) { // случай 1
                node->red=true;
                p2->red=false;
                node=*root=Rotate21(node);
                if(BalanceRemove1(&node->p1)) node->p1->red=false;
                return false;
            }
            unsigned int mask=0;
            node_st *p21=p2->p1;
            node_st *p22=p2->p2;
            if(p21 && p21->red) mask|=1;
            if(p22 && p22->red) mask|=2;
            switch(mask)
            {
            case 0:     // случай 2 - if((!p21 || !p21->red) && (!p22
|| !p22->red))
                p2->red=true;
                return true;
            case 1:
            case 3:     // случай 3 - if(p21 && p21->red)
                p2->red=true;
                p21->red=false;
                p2=node->p2=Rotate12(p2);
                p22=p2->p2;
            case 2:     // случай 4 - if(p22 && p22->red)
                p2->red=node->red;
                p22->red=node->red=false;
                *root=Rotate21(node);
            }
        return false;
    }

    //Балансировка при удалении 2
    bool RBtree::BalanceRemove2(node_st **root){
        node_st *node=*root;
        node_st *p1=node->p1;
        node_st *p2=node->p2;
        if(p2 && p2->red) {p2->red=false;return false;}
        if(p1 && p1->red) { // случай 1
            node->red=true;
            p1->red=false;
            node=*root=Rotate12(node);
            if(BalanceRemove2(&node->p2)) node->p2->red=false;
            return false;
        }
        unsigned int mask=0;
        node_st *p11=p1->p1;
        node_st *p12=p1->p2;
```

```
        if(p11 && p11->red) mask|=1;
        if(p12 && p12->red) mask|=2;
        switch(mask) {
        case 0:     // случай 2 - if((!p12 || !p12->red) && (!p11 ||
!p11->red))
            p1->red=true;
            return true;
        case 2:
        case 3:     // случай 3 - if(p12 && p12->red)
            p1->red=true;
            p12->red=false;
            p1=node->p1=Rotate21(p1);
            p11=p1->p1;
        case 1:     // случай 4 - if(p11 && p11->red)
            p1->red=node->red;
            p11->red=node->red=false;
            *root=Rotate12(node);
        }
        return false;
    }

    //Минимальный элемент
    bool RBtree::GetMin(node_st **root,node_st **res)
    {
        node_st *node=*root;
        if(node->p1) {
            if(GetMin(&node->p1,res)) return BalanceRemove1(root);
        } else {
            *root=node->p2;
            *res=node;
            return !node->red;
        }
        return false;
    }

    //Удаление из дерева
    bool RBtree::Remove(node_st **root,int value){
        node_st *t,*node=*root;
        if(!node) return false;
        if(node->value<value) {
            if(Remove(&node->p2,value))return BalanceRemove2(root);
        } else if(node->value>value) {
            if(Remove(&node->p1,value))return BalanceRemove1(root);
        } else {
            bool res;
            if(!node->p2) {
                *root=node->p1;
                res=!node->red;
            } else {
                res=GetMin(&node->p2,root);
                t=*root;
                t->red=node->red;
                t->p1=node->p1;
                t->p2=node->p2;
```

```
                if(res) res=BalanceRemove2(root);
            }
            DelNode(node);
            return res;
        }
        return 0;
    }

    //Вывод дерева
    void RBtree::Show(node_st *node,int depth,char dir){
        int n;
        if(!node) return;
        for(n=0; n<depth; n++) putchar(' ');
        printf("%c[%d]                (%s)\n",dir,node->value,node-
>red?"red":"black");
        Show(node->p1,depth+2,'-');
        Show(node->p2,depth+2,'+');
    }

    RBtree::RBtree() {
        tree_root = 0;
    }

    //Вставка в дерево
    void RBtree::Insert(int value)__transaction_atomic{
        Insert(value,&tree_root);
        if(tree_root) tree_root->red=false;
    }
    //Удаление из дерева
    void RBtree::Remove(int value)__transaction_atomic{
        Remove(&tree_root,value);
    }
    //Вывод дерева
    void RBtree::Show(){
        cout << "[tree]" << endl;
        Show(tree_root,0,'*');
    }

    RBtree::~RBtree(){
        Clear(tree_root);
    }
```