Министерство науки и высшего образования Российской Федерации

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО»

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА

ТЕОРЕТИЧЕСКИЙ АНАЛИЗ ПРОИЗВОДИТЕЛЬНОСТИ МНОГОПОТОЧНЫХ СТРУКТУР ДАННЫХ

Автор: Болотов Даниил Ильич _____

Направление подготовки: 01.03.02 Прикладная математика и информатика

Квалификация: Бакалавр

Руководитель ВКР: Аксёнов В.Е., PhD, науки _____

Санкт-Петербург, 2021 г.

Обучающийся Болотов Даниил Ильич
Группа М3435 Факультет ИТиП

Направленность (профиль), специализация
Информатика и программирование

Консультанты:

а) Кузнецов П.В, Professor, PhD _____

ВКР принята «____» _____ 20___ г.

Оригинальность ВКР _____%

ВКР выполнена с оценкой _____

Дата защиты «____» _____ 20___ г.

Секретарь ГЭК Павлова О.Н. _____

Листов хранения _____

Демонстрационных материалов/Чертежей хранения _____

Министерство науки и высшего образования Российской Федерации

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО»

УТВЕРЖДАЮ

Руководитель ОП

проф., д.т.н. Парфенов В.Г. _____

«____» _____ 20___ г.

ЗАДАНИЕ

НА ВЫПУСКНУЮ КВАЛИФИКАЦИОННУЮ РАБОТУ

Обучающийся Болотов Даниил Ильич

Группа M3435 **Факультет** ИТиП

Квалификация: Бакалавр

Направление подготовки: 01.03.02 Прикладная математика и информатика

Направленность (профиль) образовательной программы: Информатика и программирование

Тема ВКР: Теоретический анализ производительности многопоточных структур данных

Руководитель Аксёнов В.Е., PhD, науки, программист ФИТиП

2 Срок сдачи студентом законченной работы до: «____» _____ 20___ г.

3 Техническое задание и исходные данные к работе

Как нам узнать, что конкурентная программа эффективна? Обычно, эффективность конкурентной структуры данных оценивается с помощью экспериментов, и, как известно, чрезвычайно сложно учесть все экспериментальные параметры, чтобы результаты были значимыми. Цель этой работы состоит в том, чтобы дополнить экспериментальную оценку аналитической моделью, которую можно использовать для прогнозирования производительности, а не для ее измерения. В качестве первого шага к этой цели была предложена модель прогнозирования пропускной способности класса алгоритмов, использующих грубую синхронизацию с блокировкой CLH ([1] Исходные материалы и пособия). В этом проекте требуется расширить анализ прогнозирования пропускной способности класса алгоритмов, использующих грубую синхронизацию. А также попытаться проанализировать структуры данных, которые используют тонкую синхронизацию.

4 Содержание выпускной квалификационной работы (перечень подлежащих разработке вопросов)

1 Анализ существующих аналитических подходов ([1],[3],[4] Исходные материалы и пособия).
2 Реализованная модель прогнозирования производительности для алгоритма взаимного исключения MCS.
3 Реализованная модель прогнозирования производительности для тонкой синхронизации на примере связного списка.
4 Сравнение предложенной модели с тестами производительности.

5 Перечень графического материала (с указанием обязательного материала)

Графические материалы и чертежи работой не предусмотрены

6 Исходные материалы и пособия

1 Vitaly Aksenov, Dan Alistarh, Petr Kuznetsov: Performance Prediction for Coarse-Grained Locking. CoRR abs/1904.11323 (2019) https://arxiv.org/pdf/1904.11323.pdf;

2 John M Mellor-Crummey and Michael L Scott. 1991. Algorithms for scalable synchronization on shared-memory multiprocessors. ACM Transactions on Computer Systems (TOCS) 9, 1 (1991), 21–65. http://web.mit.edu/6.173/www/currentsemester/readings/R06-scalable-synchronization-1991.pdf

3 Aras Atalar, Paul Renaud-Goud, and Philippas Tsigas. 2015. Analyzing the Performance of Lock-Free Data Structures: A Conflict-Based Model. In Distributed Computing - 29th International Symposium, DISC 2015, Tokyo, Japan, October 7-9, 2015, Proceedings. 341–355. https://arxiv.org/pdf/1508.03566.pdf

4 Aras Atalar, Paul Renaud-Goud, and Philippas Tsigas. 2016. How Lock-free Data Structures Perform in Dynamic Environments: Models and Analyses. In 20th International Conference on Principles of Distributed Systems, OPODIS 2016, December 13-16, 2016, Madrid, Spain. 23:1–23:17. https://arxiv.org/pdf/1611.05793.pdf

7 Дата выдачи задания «____» _____ 20___ г.

Руководитель ВКР _____

Задание принял к исполнению _____ «____» _____ 20___ г.

Министерство науки и высшего образования Российской Федерации

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО»

АННОТАЦИЯ
ВЫПУСКНОЙ КВАЛИФИКАЦИОННОЙ РАБОТЫ

Обучающийся: Болотов Даниил Ильич

Наименование темы ВКР: Теоретический анализ производительности многопоточных структур данных

Наименование организации, в которой выполнена ВКР: Университет ИТМО

ХАРАКТЕРИСТИКА ВЫПУСКНОЙ КВАЛИФИКАЦИОННОЙ РАБОТЫ

1 Цель исследования: Разработка аналитической модели для оценки пропускной способности многопоточных структур данных, использующих грубые и тонкие механизмы синхронизации.

2 Задачи, решаемые в ВКР:

а) анализ существующих аналитических подходов;

б) реализация модели прогнозирования производительности для алгоритма взаимного исключения MCS;

в) реализация модели прогнозирования производительности для тонкой синхронизации на примере связного списка.

г) сравнение предложенной модели с тестами производительности;

3 Число источников, использованных при составлении обзора: 0

4 Полное число источников, использованных в работе: 23

5 В том числе источников по годам:

Отечественных			**Иностранных**		
Последние 5 лет	От 5 до 10 лет	Более 10 лет	Последние 5 лет	От 5 до 10 лет	Более 10 лет
0	0	0	9	3	11

6 Использование информационных ресурсов Internet: да, число ресурсов: 22

7 Использование современных пакетов компьютерных программ и технологий:

Пакеты компьютерных программ и технологий	**Раздел работы**
Пакет `tabularx` для чуть более продвинутых таблиц	**??**, Приложения **??**, **??**
Пакет `biblatex` и программное средство `biber`	Список использованных источников

8 Краткая характеристика полученных результатов

Получилось расширить имеющуюся аналитическую модель для многопоточных структур данных, использующих грубые механизмы синхронизации, алгоритмом блокировки MCS.

9 Гранты, полученные при выполнении работы

Грантов при выполнении работы получено не было.

10 Наличие публикаций и выступлений на конференциях по теме выпускной работы

Нет.

Обучающийся Болотов Д.И. _____

Руководитель ВКР Аксёнов В.Е. _____

«____» _____ 20___ г.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Как нам узнать, что конкурентная программа эффективна? Обычно эффективность конкурентной структуры данных оценивается с помощью экспериментов, и, как известно, чрезвычайно сложно учесть все экспериментальные параметры, чтобы результаты были значимыми.

Цель этой работы заключалась в том, чтобы дополнить экспериментальную оценку производительности многопоточных структур данных аналитической моделью, которую можно использовать для прогнозирования производительности, а не для ее измерения.

В работе стояли следующие задачи:

— Изучить существующие аналитические подходы [23], [3], [4], [13].
— Выбрать подходящую модель исполнения для дальнейшего анализа.
— Расширить имеющуюся аналитическую модель, использующуюся в [23], аналитической моделью для алгоритма взаимного исключения MCS[13].
— Разработать аналитическую модель производительности для lock-free алгоритма Treiber Stack [21].
— Сравнение предложенных моделей с тестами производительности.

Актуальность задачи обусловлена существованием большого числа конкурентных структур данных, использующих алгоритмы взаимного исключения для обеспечения грубой синхронизации между потоками. А также, помимо структур данных, использующих алгоритмы блокировки, существуют lock-free конкурентные структуры данных, не использующие алгоритмы взаимного исключения. Представителем последнего класса является одна из самых популярных реализаций многопоточного стека —— Treiber Stack. Но для таких алгоритмов отсутствуют аналитические модели, позволяющие оценить производительность структуры данных до экспериментов.

Практическая значимость работы обусловлена тем, что она предоставит возможность еще до реализации необходимой конкурентной стуктуры данных оценить ее производительность.

Работа имеет следующую структуру:

— В первой главе работы проводится обзор предметной области: рассматривается модель с общей памятью для исполнения конкурентных про-

грамм и описываются основные метрики для измерения производительности. Также, в главе рассмотрена практическая значимость конкурентных объектов. Помимо этого, в главе описаны требования к механизмам синхронизации и рассматриваются гарантии прогресса этих механизмов.

— Во второй главе разрабатывается аналитическая модель для предсказания производительности конкурентных структур данных, использующих грубые механизмы синхронизации. В частности, в качестве такого механизма рассматривается алгоритм взаимного исключения MCS. Кроме того, в главе, разработанные аналитические модели для имеющихся алгоритмов сравниваются с тестами производительности

— В третьей главе разрабатывается аналитическая модель для предсказания производительности lock-free структуры данных на примере классической реализации Treiber Stack [21]. Кроме того, в этой главе разработанная аналитическая модель для имеющегося алгоритма сравнивается с тестами производительности.

ГЛАВА 1. ОБЗОР ПРЕДМЕТНОЙ ОБЛАСТИ

1.1. Терминология

В этой секции рассмотрены основные определения, которые используются в дальнейшем обзоре.

В основе конкурентного программирования лежат термины **поток** и **процесс**. С практической точки зрения термины имеют следующие определения:

Процесс [16] — единица владения памятью и ресурсами, у которой есть свое собственное адресное пространство. Процесс состоит из нескольких потоков. Каждый процесс имеет иллюзию монопольного использования памяти, которая достигается за счет использования виртуальной памяти.

Поток [20] — контекст исполнения внутри процесса, который оркестрируется планировщиком операционной системы. Потоки процесса разделяют единое адресное пространство процесса, в котором они созданы. У потока имеется свой собственный стек и имеется свое собственное процессорное время.

Отметим, что в теории конкурентного программирования данные термина являются синонимами. Поэтому в дальнейшем **поток** и **процесс** будут иметь определение, которое указано для термина **поток**.

Произошло до (англ. Happens-before) [11] — модель, использующая отношение строгого частичного порядка, предложенная Лесли Лампортом, которая в конкурентном программировании означает следующее. Рассмотрим две операции a и b (например, чтение, запись в ячейку памяти). Операцию представляет из себя два события: $inv(a)$ — вызов операции a, а $res(a)$ — возврат результата на операцию a. Тогда, операция a **произошла до** операции b, тогда и только тогда, когда $res(a)$ случился до вызова операции $inv(b)$.

Линеаризация (англ. Linearization) [9] — Условие корректности, которое формализуется следующим образом: высокоуровневая история H исполнения α — это подпоследовательность α, состоящая из всех событий $inv()$ и $res()$ высокоуровневых операций. Полная высокоуровневая история H линеаризуема относительно объекта t, если существует такая последовательная высокоуровневая история S эквивалентная H, что:

— S сохраняет отношение ”произошло до” истории H;

— S соответствует последовательной спецификации объекта t.

Таким образом, высокоуровневая история *H линеаризуема*, если может быть дополнена до полной линеаризуемой высокоуровневой истории путем добавления соответствующих событий завершения подмножеству незавершенных операций и удалением оставшихся.

Критическая секция [7] — часть программного кода, которая может исполняться одновременно только одним потоком. Как правило, в критической секции происходит взаимодействие с разделяемым ресурсом, доступ к которому безопасно может получать только один поток одновременно. Одновременность в этом случае объясняется тем, что если два события нельзя упорядочить отношением ”произошло до”, то такие события происходят одновременно.

Параллельная секция — часть программного кода, которая может исполняться одновременно любым числом потоков.

Синхронизация — механизм, с помощью которого обеспечивается корректный доступ потоков к общей памяти. Доступ процессов к критическому участку контролируется с помощью алгоритмов синхронизации, которые могут быть как блокирующими, так и использующими неблокирующие подходы с помощью атомарных операций.

Алгоритм взаимного исключения — алгоритм для предотвращения параллельности, гарантией корректности которого является выполнение критической секции одновременно одним потоком.

CAS (Compare-And-Swap) [5] – атомарная универсальная операция, использующаяся для синхронизации межпоточного взаимодействия. Семантика операции заключается в проверке содержимого ячейки памяти с заданным значением и, только если они совпадают, изменяет содержимое этой ячейки памяти на новое заданное значение.

SWAP (Get-And-Set) — атомарная операция с консенсусным числом два, использующаяся для синхронизации межпоточного взаимодействия. Семантика операция заключается в атомарном получении старого значения в указанной ячейке памяти и установки нового значения в эту ячейку.

Конкурентная структура данных [9] — это структура данных, совместно используемая разными потоками. Алгоритмы для реализации парал-

лельных стуктур данных лежат в основе многих важных проблем в параллельных системах. Традиционный подход к реализации подобных объектов сосредотачивается вокруг использования критических секций, где реализация критических секций достигается при помощи алгоритмов взаимного исключения.

1.2. Причины развития конкурентных программ

Согласно [18], начиная, приблизительно, с 2005 года, число ядер в новых процессорах стало экспоненциально расти, в этот же момент, рост частоты процессоров остановился ввиду физических ограничений. Если ранее, учитывая закон Мура [14], для масштабируемости программного обеспечения было достаточно запустить ее на более мощном аппаратном обеспечении, то после замедления роста частоты, этого стало недостаточно. Основное следствие из [18] является то, что сейчас для обеспечения масштабируемости нужно использовать параллелизм, чтобы более выгодно использовать CPU.

Также, фактом того, что для масштабируемости нужно больше параллелизма (см. Рисунок 1), является сформулированный в 1967 году ученым Джином Амдалом закон [1], с помощью которого можно оценить максимальное ускорение программы в зависимости от количества потоков и процента параллельной части программы.

Формальное определение Закона Амдала выглядит следующим образом.

Пусть N — число потоков, и P —— доля параллельного кода, тогда максимальное ускорение программы S при использовании N потоками можно найти по формуле:

$$S = \frac{1}{1 - P + \frac{P}{N}}.$$

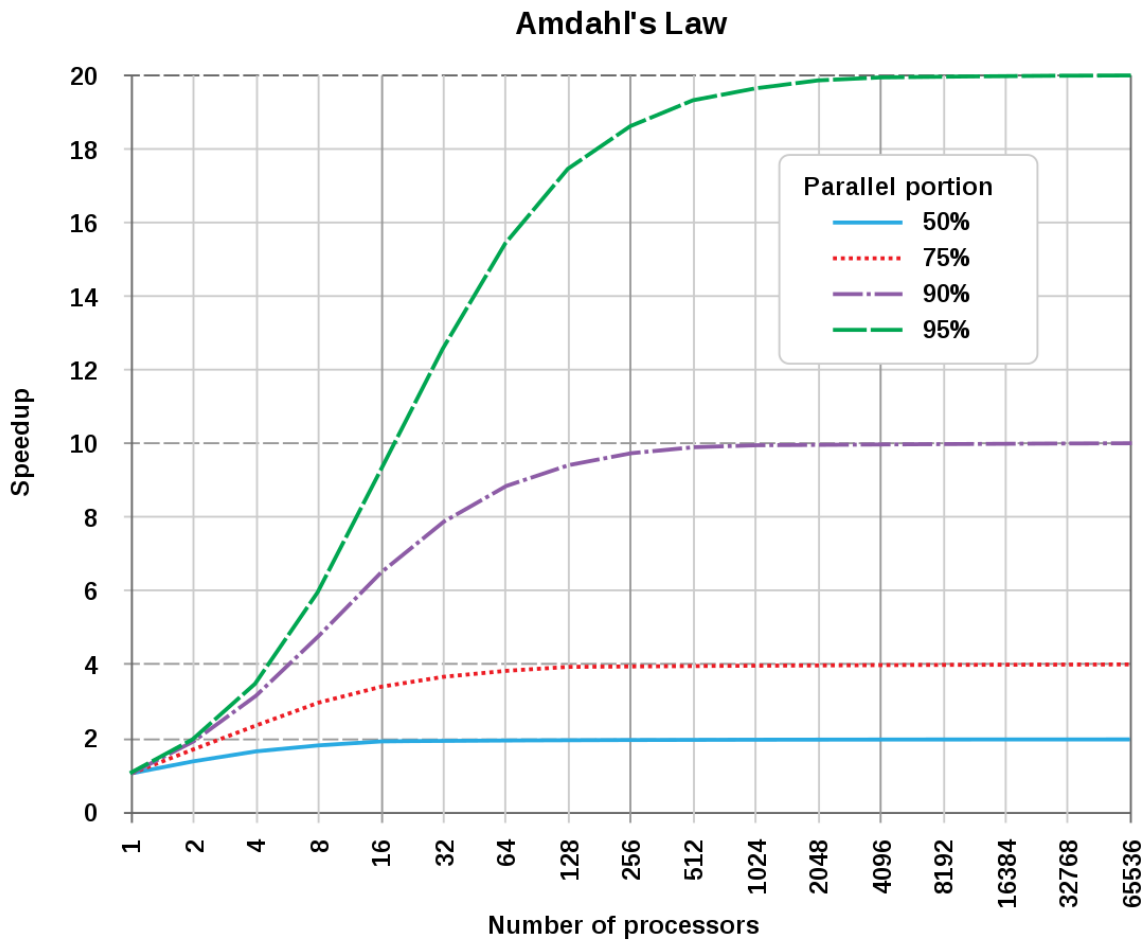

Рисунок 1 – Ускорение кода в зависимости от количества потоков и доли параллельной части. Источник [1]

Именно эти факторы послужили развитию многопоточных структур данных.

1.3. Модель исполнения

В этой секции описана модель исполнения конкурентных программ, которая далее будет использована в работе.

Многопоточное программирование позволяет использовать ресурсы многоядерной системы в рамках решения одной задачи.

Существуют следующие варианты для формализации многопоточного программирования:

— Модель с общей памятью [2];

— Модель с передачей сообщений [19].

В рамках этой работы мы используем модель с общей памятью. Формализуем основные требования к данной модели. В системе существуют потоки $\{p_i\}_{i=1}^{n}$,

которые имеют своё внутреннее состояние, и также имеется общая память, в которой находятся общие объекты $\{O_j\}_{j=1}^{m}$. Каждый общий объект доступен каждому потоку.

В этой модели не важны операции внутри потоков:

— вычисления;
— обновления регистров процессора;
— обновления стека потока;
— обновления любой другой локальной памяти потока.

В моделе с общей памятью важна только коммуникация между потоками. В предложенной модели единственный способ коммуникации между ними — работа над общими объектами (см. Рисунок 2).

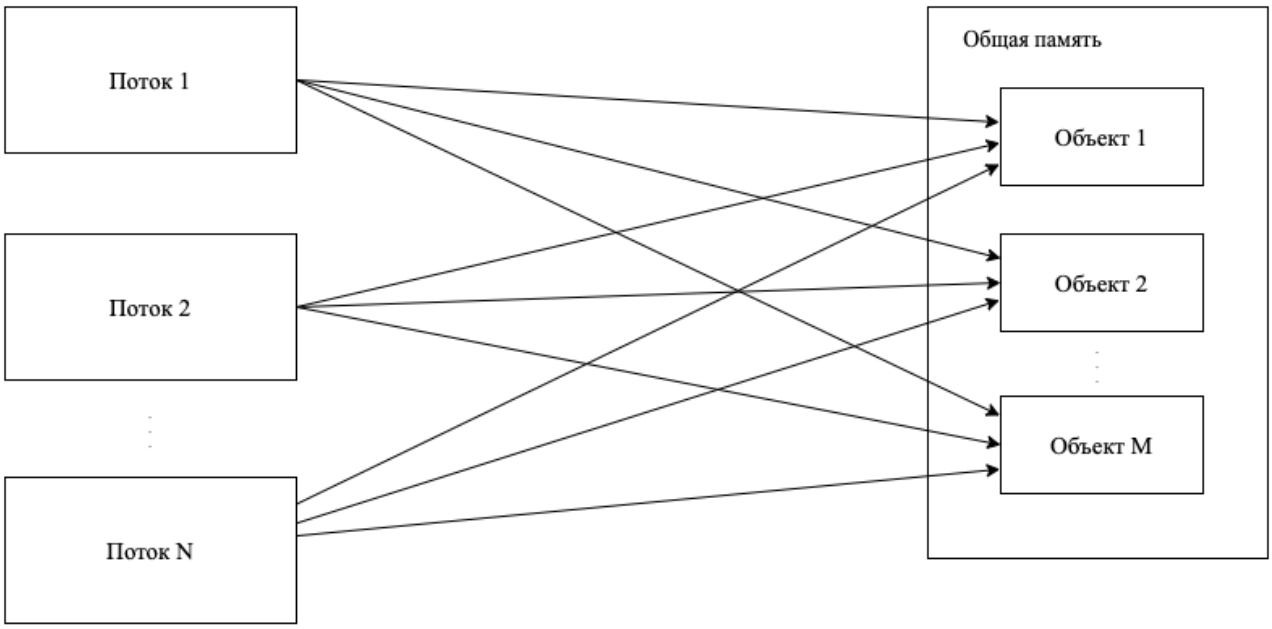

Рисунок 2 – Потоки и общие объекты

Как правило, общие объекты поддерживают следующие операции над ними:

— чтение (R);
— запись (W);
— чтение-модификация-запись (RMW).

Операция чтения-модификация-запись атомарно читает необходимую ячейку памяти и записывает в нее новое значение либо с совершенно новым значением, либо с некоторой функцией f, примененной к текущему значению в этой ячейки памяти.

Примерами таких операций являются выше упомянутые операции Compare-And-Swap, SWAP (также именуемая как Get-And-Set). Операции чтение-модификация-запись, их классификация и практическая значимость описаны в работе [8].

1.4. Механизмы синхронизации

В предыдущей секции была описана модель исполнения, которая применима к рассматриваемым в этой работе конкурентным структурам данных. Как было упомянуто выше, конкурентные структуры данных используются разными потоками. Коммуникация потоков в таких объектах происходит через общую память.

Однако, существуют **конфликтующие операции** — операции над общим объектом, одна операция из которых является запись (W).

Ситуация, при которой две конфликтующие операции не упорядочены отношением **произошло до** называется **состоянием гонки данных** (англ. data race). Пример гонки данных рассмотрен на Рисунке 3.

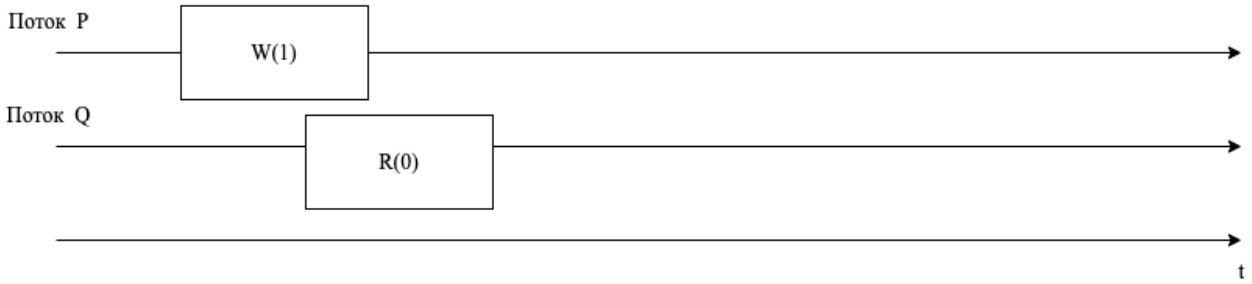

Рисунок 3 – Состояние гонки данных

Корректной многопоточной структурой данных будем называть программу, любые исполнения которой линеаризуемо. В корреткной многопоточной структуре данных в любом допустимом исполнении нет гонок данных.

Именно для обеспечения корректности программы необходим механизм синхронизации. С помощью синхронизации достигается линеаризация.

1.5. Гранулярность синхронизации

Как известно, использование синхронизации между потоками необходима для обеспечения корректности многопочной структуры данных, с другой стороны — излишняя синхронизация может существенно снизить процент параллельной части программы, что, согласно закону Амдала [1], отрицательно влияет на итоговую производительность.

— Грубая синхронизации: при работе в критической секции вся структура данных блокируется.
— Тонкая синхронизация: увеличивает параллелизм в коде, что по закону Амдала, влечет за собой больший теоретический прирост производительности. Однако, сложнее в реализации.

1.6. Алгоритмы взаимного исключения

В качестве механизма синхронизации над конкурентным объектом зачастую используются алгоритмы взаимного исключения (блокировки). Главной гарантией корректности данного алгоритма является взаимное исключение — в критической секции одновременно может находиться только один поток.

Помимо требования корректности, для алгоритмов взаимного исключения существуют **условные** гарантии прогресса. Они называются условными, так как эти гарантии могут выполняться, если работа в критической секции выполняется за конечное время.

Ниже рассмотрены условные гарантии прогресса в порядке от самой слабой гарантии к самой сильной:

— **Отсутствие взаимной блокировки (deadlock-freedom)** — хотя бы один поток в системе должен совершить работу в критической секции (при условии, что эта работа выполняется за конечное время).
— **Отсутствие голодания (starvation-freedom)** — если какой-то поток пытается войти в критическую секцию, то он войдет в критическую секцию за конечное время (если критическая секция выполняется за конечное время).

Также алгоритмы блокировок характеризуются свойством честности (англ. fairness)[9]. Ниже описаны свойства честности в порядке от самого слабого к самому сильному свойству:

— **Квадратичное ожидание** — поток может ждать пока другие потоки $O(N^2)$ раз войдут в критическую секцию.
— **Линейное ожидание** — поток может ждать пока другие потоки $O(N)$ раз войдут в критическую секцию.
— **Первый пришел, первый обслужен (англ. FCFS — First Come First Served)** – наиболее сильное свойство честности алгоритма взаимного исключения. Требование First Come, First Served формализуется следующим образом:

- Метод `lock()` должен состоять из двух последовательных секций **doorway, waiting**.
- Секция **doorway** должна быть `wait-free`, то есть, каждый поток должен выполнять работу в этой секции за конечное число шагов, независимо от действия других потоков.
- Секция **waiting** должна удовлетворять следующему условию. Пусть DW_i — работа в секции **doorway**, а WT_i — работа в **waiting**. Если DW_i произошло до DW_j, тогда `res(`WT_i`)` произошло до `res(`WT_j`)`, что также означает, что i-ый поток войдет в критическую секцию раньше, чем j-ый.

Подробнее это свойство честности описано в [9]. Если алгоритм удовлетворяет этому свойство, то этот алгоритм также удовлетворяет гарантиям прогресса **deadlock-freedom**, **starvation-freedom**.

1.7. Измерение производительности

В большинстве теоретических работ, есть две основные метрики для оценивания производительности.

Задержка (англ. Latency) — метрика производительности, так же известная как время ответа на операцию `(англ. Response Time)`. Эта метрика равна количеству времени, которое требуется системе для обработки запроса/операции.

Пропускная способность (англ. Throughput) — метрика производительности, используемая для обозначения количества обработанных операций, которые система может обработать в течение определенного периода времени.

В работе [22] проводятся многочисленные тесты производительности для популярных алгоритмов синхронизации, где используются обе метрики.

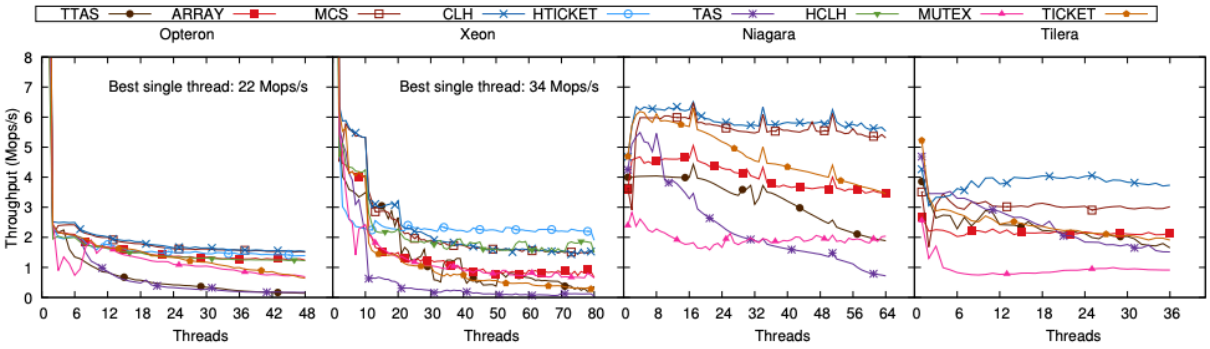

Рисунок 4 – Пример измерения метрики производительности пропускной способности для алгоритмов взаимного исключения. Источник [22]

В этой работе разработанные аналитические модели позволяют предсказать именно пропускную способность (англ. throughput).

1.8. MESI

В качестве протокола когерентности кэша в анализе рассматривается протокол MESI [15]. Переходы между этими состояниями описаны на Рисунке 5.

Ниже рассмотрены важные для анализа переходы состояний. Во время чтения, состояние кэш-линии изменяется с любого состояния до Shared, и, в случае если состояния было Invalid, отправляется запрос на чтение на шину. Во время записи, состояние кэш-линии становится Modified, и если состояния до записи было Shared или Invalid, то к шине отправляется запрос на инвалидацию указанной кэш-линии.

Далее предполагается, что в среде исполнения используются симметричные кэши: для каждого состояния MESI, существуют две константы R_{st} и W_{st} такие, что чтение из любой кэш-линии в состоянии st будет занимать R_{st} времени, и любая запись в кэш-линию в состоянии st будет занимать W_{st} времени.

В работе [22] указано, что для семейства Intel Xeon процессоров выполняются следующие утверждения:

— Для записи требуется одинаковое время работы независимо от состояния кэш-линии, имеется в виду, что $W = W_E = W_M = W_I = W_S$.

— При малой конкуренции потоков при взятии блокировки для входа в критическую секцию, для любой операции эксклюзивного атомарного обмена (SWAP или getAndSet) так же требуется W времени для совершения операции.

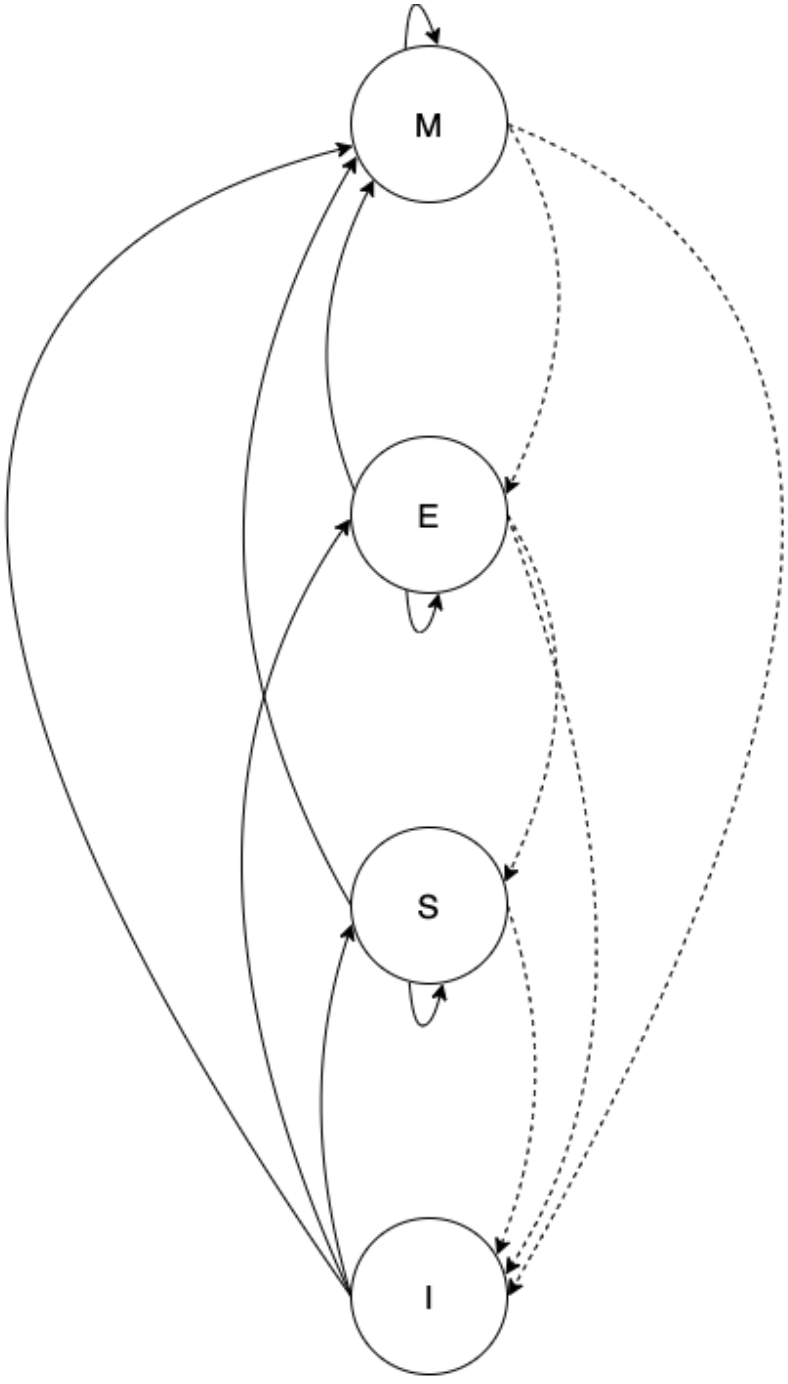

Рисунок 5 – Диаграмма состояний протокола MESI

Выводы по главе 1

В этой главе был проведён анализ предметной области. Были рассмотрены необходимые термины, использующиеся в следующих главах и в дальнейшем обзоре. Также рассмотрены основополагающие факторы, способствующие развитию конкурентных структур данных. Кроме того, было дано определение модели исполнения с общей памятью. Также рассмотрена необходимость синхронизации. Далее были определены алгоритмы взаимного исключения и их свойства. Помимо этого, было дано определение понятию **пропускная способность** — метрики для оценивания производительности, которая будет использована в следующих главах работы.

ГЛАВА 2. АНАЛИЗ ПРОИЗВОДИТЕЛЬНОСТИ АБСТРАКТНОЙ МНОГОПОТОЧНОЙ СТРУКТУРЫ ДАННЫХ, ИСПОЛЬЗУЮЩЕЙ ГРУБЫЕ МЕХАНИЗМЫ СИНХРОНИЗАЦИИ

В этой главе расширен имеющийся анализ для структур данных с грубой синхронизацией [23] алгоритмом взаимного исключения MCS [13].

В начале главы описывается различие между алгоритмами, преимущество алгоритма MCS перед алгоритмом CLH на NUMA [12]-архитектурах. Далее следует обзор алгоритма MCS. После чего в главе описана среда исполнения. Рассмотрев среду исполнения, на следующем шаге в анализе представлена абстрактная структура данных, использующая в качестве грубой блокировки MCS. В той же секции рассмотрены необходимые для построения модели операции, которые присутствуют в имеющейся структуре данных, где используются явно функции `lock()` и `unlock()` алгоритма MCS. Дав оценку операциям, рассмотрены возможные случаи исполнения, которые зависят от количества работы в параллельной и критической секции. После рассмотрения случаев исполнения, описывается получившаяся аналитическая модель. В конце главы получившаяся модель сравнивается с тестами производительности.

2.1. Анализ существующего решения для CLH. Сравнение MCS и CLH.

В представленном ранее анализе [23] была разработана аналитическая модель для многопоточных структур данных, использующих грубые механизмы синхронизации, при помощи алгоритма взаимного исключения CLH [6]. В этой секции произведён обзор таких алгоритмов, их свойства, а также описан главный недостаток CLH перед MCS.

Рассмотрим для начала алгоритм CLH:

— Гарантия честности FCFS.
— Разработан с целью устранения лишних инвалидаций, которые присутствуют в Test-and-Set и Test-and-Test-and-Set. lock [22]
— Хранится логическая очередь ждущих потоков.

```
1 class Node:
2   bool locked // shared, atomic
3
4 class CLHLock:
5   tail = Node() // shared, global
6   threadlocal myNode = Node() // per process
7   threadlocal pred // per process
8
9   lock():
10    myNode.locked = true
11    pred = tail.getAndSet(myNode)
12    while pred.locked:
13      //pass
14
15  unlock():
16    myNode.locked = false
17    myNode = pred
```

Рисунок 6 – Операции lock и unlock алгоритма взаимного исключения CLH

Далее рассмотрим алгоритм MCS:

— Гарантия честности FCFS.

— В отличие от CLH, потоки «ждут» на своей памяти в строке 15.

```
1 class Node:
2   bool locked //shared, atomic
3   Node next = null
4
5 class MCSLock:
6   tail = null // shared, global
7   threadlocal myNode // per process
8
9   lock():
10    myNode = Node()
11    myNode.locked = true
12    pred = tail.getAndSet(myNode)
13    if pred != null:
14        pred.next = myNode
15        while my.locked:
16          // pass
17
18  unlock():
19   if myNode.next == null:
20     if tail.CAS(myNode,null): // в случае если в очереди единственный
     поток и нет следующих CAS=true
21       return
22     else:
23       while myNode.next == null: // иначе, какой-то поток успел присвоить
     на себя ссылку tail, но не обновил next у вызывающего unlock() потока,
     в таком случае происходит, когда появится ссылка на ждущий поток
24         //pass
25   myNode.next.locked = false
```

Рисунок 7 – Операции lock и unlock алгоритма взаимного исключения MCS

Заметим, что эти два алгоритмы достаточно похожи. Оба алгоритма решают проблему инвалидации (англ. invalidation storm), которая встречается у более простых алгоритмов Test-and-Set lock и Test-and-Test-and-Set lock, так как обе блокировки используют очереди, в первом случае используется «логическая» очередь, во втором случае очередь указывается явно. Именно из-за использования в реализации очереди, во время освобождения блокировки, запрос на инвалидацию кэш-линии получает только один поток, который находится за текущим в очереди. Такое решение лучше масштабируется в среде исполнения с высокой конкуренцией (англ. high contention) [22]

Однако, стоит заметить, что в реализации CLH алгоритма, ждущий поток вынужден ждать на «чужой» памяти. В строке 12 показано активное ожидание потока, который находится в очереди. Такой недостаток особенно критичен при использовании в NUMA-архитектурах (Рисунок 8). Так как предположим, что поток, освобождающий блокировку исполняется на CPU 4, а следующий в очереди поток исполняется на другом CPU. Такое ожидание является дорогостоящей операцией ввиду того, что запрос от CPU, на котором исполняется ждущий поток, маршрутизируется на память, расположенную с CPU, где исполняется поток, владеющей блокировкой в текущий момент. Пропускная способность упирается в скорость передачи сообщения через **interconnection network** [12]. В отличие от CLH, в MCS все потоки, ждущие взятия блокировки, ждут на «своей» памяти (строка 15), а поток, освобождающий блокировку вынужден только один раз обратиться к чужой памяти, чтобы сообщить ждущему потоку о возможности войти в критическую секцию. Это решение лучше сказывается на производительность в NUMA-архитектурах.

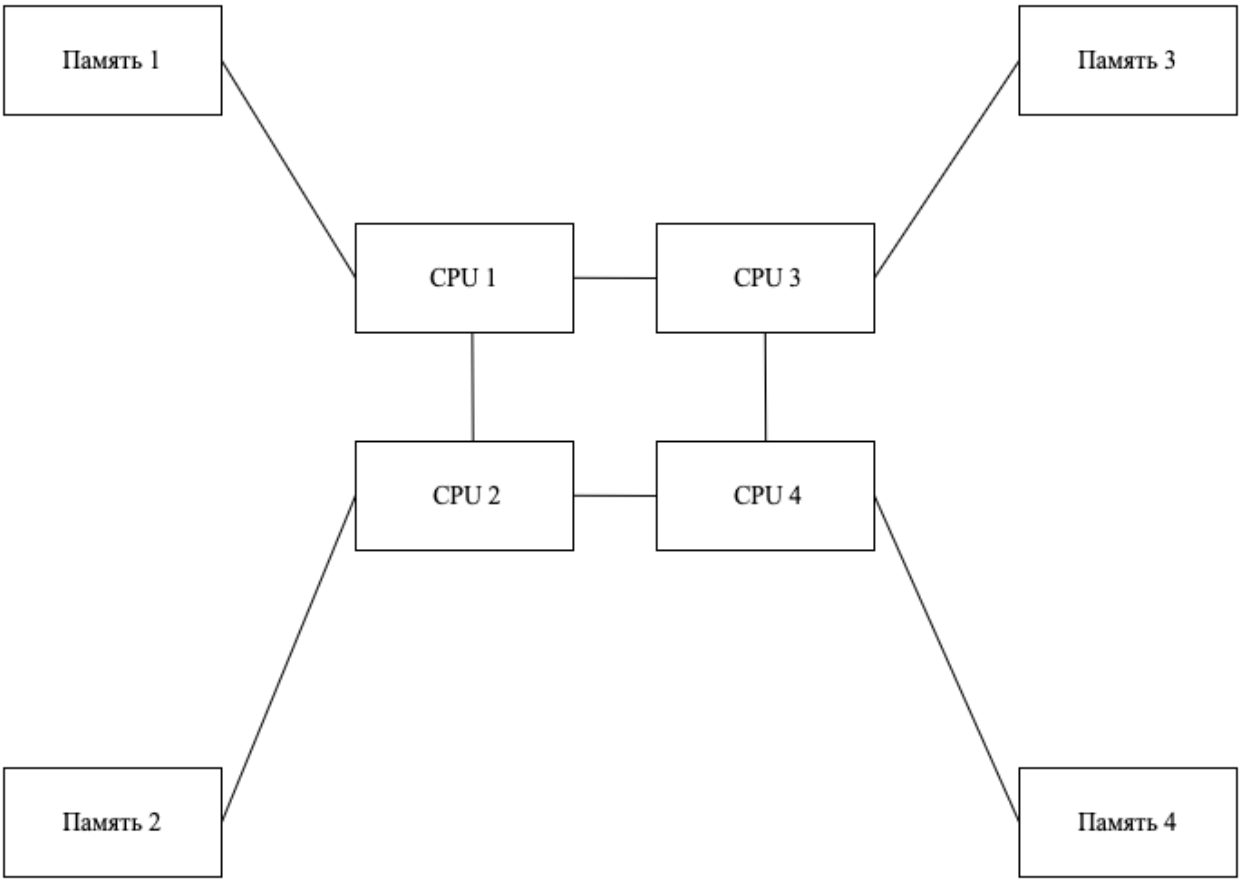

Рисунок 8 – NUMA

Как правило, в мультисокетных NUMA-архитектурах для лучшей масштабируемости используется техника **Lock Cohortion** (Рисунок 9). В этой технике используется объединение двух видов блокировок. К сожалению, у такой схемы сложно промоделировать всевозможные варианты исполнения и разработать для нее аналитическую модель. Поэтому, для анализа был выбран алгоритм взаимного исключения MCS, так как, во-первых, как и CLH, MCS имеет максимальное свойство честности FCFS, но кроме этого лучше масштабируется на NUMA-архитектурах.

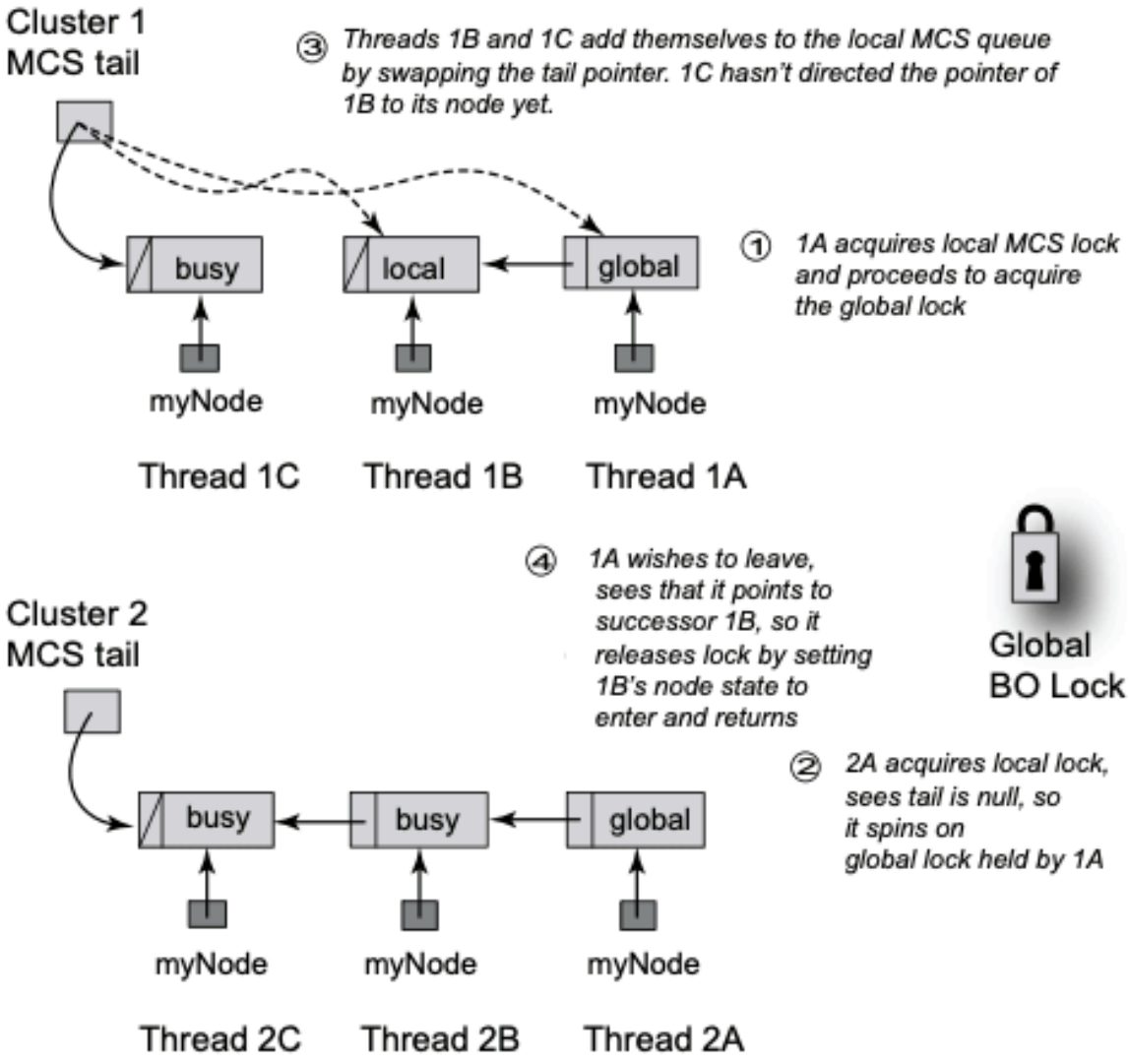

Рисунок 9 – Техника Lock Cohortion. Источник [12]

2.2. Среда исполнения

Для упрощения моделирования, сделаны основные предположения о среде, в которой работает многопоточная структура данных. Для начала, в качестве планировщика используется простой симметричный планировщик (англ. Uniform Scheduler), обладающий следующими свойствами:

— Каждый поток совершает одну операцию за один такт.

— Всем потокам выдается одинаковое процессорное время.

Ввиду того, что используется простой симметричный планировщик, для моделирования возможных случаев при использовании разными потоками рассматриваемой многопоточной структуры данных, удобно использовать модель PRAM [10] — абстрактная модель для параллельных вычислений, кото-

рая предполагает, что все процессы работают синхронно под одним тактовым сигналом.

Также как и в предложенном ранее анализе в [23], были введены следующие константы для оценивания пропускной способности при разработке модели. α — число тактов в единицу времени. α вычисляется экспериментально с помощью программы, в которой отсутствует критическая секция, все вычисления происходят в параллельной секции в течение T времени. Тогда $\alpha = P \cdot F/(T \cdot N)$, где F — суммарное число операций, совершенных N потоками за время T. P — число тактов для совершения операции в параллельной части. Чем больше T, тем более точна оценка a.

Например, если для выполнения параллельной работы требуется P тактов, то пропускная способность для одного потока будет вычисляться как a/P, а для N потоков как $N \cdot a/P$

2.3. Оценка производительности

На рисунке 10, в рассматриваемой многопоточной структуре данных с грубой синхронизацией явно используются функции `lock` и `unlock` алгоритма взаимного исключения MCS.

```
1 class Node:
2   bool locked // shared, atomic
3   Node next = null
4
5   tail = null // shared, global
6   threadlocal myNode = null // per process
7 operation():
8   myNode = Node()
9   myNode.locked = true
10  pred = tail.getAndSet(myNode) // W or X
11  if pred != null:
12    pred.next = myNode
13    while myNode.locked: // pass // R_I
14 // CS started
15 for i in 1..C: // C
16    //nop
17 // CS finished
18 if myNode.next == null: // R_I
19   if tail.CAS(myNode,null): // W or X
20     return
21   else:
22     while myNode.next == null: // R_I
23       //pass
24 myNode.next.locked = false // W
25 //Parallel section
26 for i in 1..P: // P
27   //nop
```

Рисунок 10 – Структура данных с грубой синхронизацией, использующая алгоритм взаимного исключения MCS

Далее рассмотрены по отдельности стоимость конкретных операций в структуре данных, с псевдокодом на рисунке 10.

Операция взятия блокировки начинается с операции `getAndSet` в строке 10. Для совершения операции требуется W тактов, в случае если эта операции эксклюзивная, в противном случае, требуется X тактов.

Затем, в строке 11 происходит проверка, есть ли в очереди перед потоком в критической секции другие потоки. В первом случае, если ссылка на хвост очереди оказалась `null`, то процесс входит в критическую секцию.

В противном случае, перед текущим потоком есть другой поток, который выполняет работу в критической секции. Тогда, процесс записывает себя в эту очередь и далее, в цикле `while` в строке 13 возможен один кэш-мисс: если значение `locked` изменилось и стало `false`, значит, другой процесс в строке 24 произвел операцию записи и инвалидировал кэш-линию с `myNode`. В таком случае, потребуется R_I тактов.

В следующих строках 15-16 происходит работа в критической секции, которая занимает C тактов.

После работы в критической секции происходят операции освобождения блокировки. В строке 18 также возможна ситуация с кэш-миссом. Так как поток, следующий за текущим, в строке 12 мог инвалидировать кэш-линию, в этом случае, потребуется R_I тактов для совершения операции. Затем, возможны два случая. В очереди оказался следующий поток за текущим потоком, который хочет выполнить работу в критической секции:

1. В первом случае, результат проверки в строке 18 будет отрицательным и будет выполнен переход к строке 24, для которой потребуется W тактов.
2. Результат проверки в строке 18 положительный. Тогда, поток с помощью операции CAS в строке 19 пытается обновить ссылку на хвост текущей очереди, что влечет за собой W времени выполнения. Результат CAS так же рассматривается в двух случаях:
   2.1. Результат операции CAS вернул `true`. Операция `unlock` завершена, переход к параллельной части.
   2.2. Результат операции CAS вернул `false`. Следующий поток в очереди успел обновить голову списка, но еще не успел записаться в очередь за текущим потоком. В такой ситуации текущий поток ожидает в цикле в строке 22 возможен один кэш-мисс. В случае, если `myNode.next != null`, значит, другой процесс в строке 12 произвел операцию записи и инвалидировал кэш-линию с `myNode`. В таком случае, потребуется R_I тактов для выполнения операции. В конце выполняется переход к строке 24, для которой потребуется W времени работы.

После освобождения блокировки процесс выполняет работу в параллельной части (строки 26-27), для выполнения которой требуется P тактов. Затем, поток вновь начинает выполнять операцию со строки 9.

После того, как была дана оценка необходимым для анализа операция, с помощью модели PRAM ниже рассмотрены следующие варианты исполнения, при этом стоит отметить, что присутствие параллельной части необходима, чтобы эмулировать клиентское приложение [3], которое пользуется такой структурой данных. Также, критическая секция зафиксирована, тогда как варьируется только работа параллельной части:

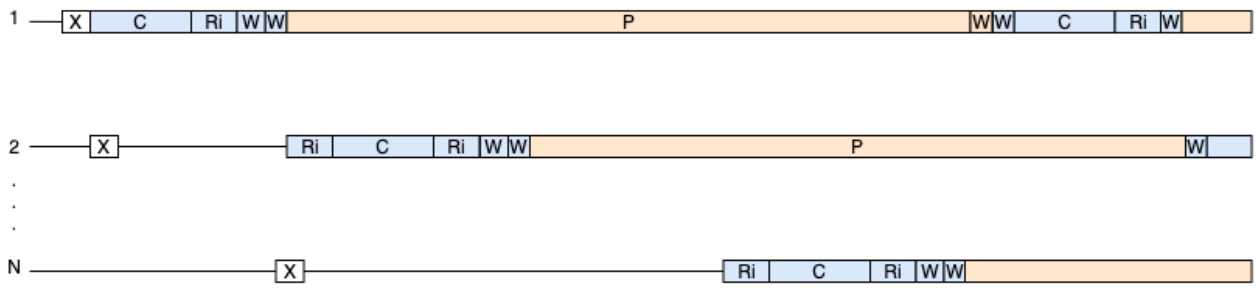

Рисунок 11 – Случай 1:$P \geqslant (N-1) \cdot C$

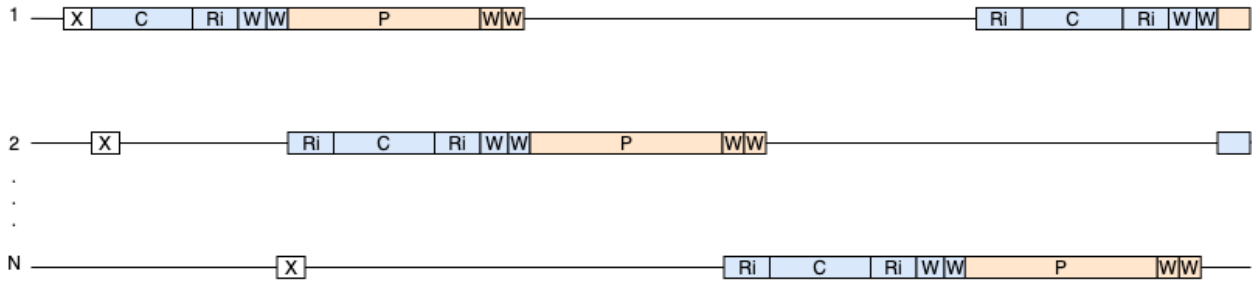

Рисунок 12 – Случай 2:$P << (N-1) \cdot C$

В первом случае исполнения (Рисунок 11, при достаточно большой параллельной части, процесс не ждет захвата блокировки и совершает операцию. В таком случае, в любой момент времени поток не ждет освобождения блокировки другим процессом и в каждый момент времени выполняет работу. Таким образом, пропускную способность для этого случая можно оценить как число операций в момент времени, а именно:

$$\frac{\alpha \cdot N}{(2W + C + R_I) + (P + W)}.$$

Во втором случае (Рисунок 12) второй процесс будет вынужден ждать первого, N-ый процесс ждать $N-1$, который находятся в критической секции. Таким образом, всегда есть процесс, который находится в критической секции и тогда пропускная способность для данного случая будет оцениваться как:

$$\frac{\alpha}{2R_I + C + 2W}.$$

В третьем случае, следующий процесс успел обновить ссылку на хвост, но не успел записаться в очередь. В таком случае, аналогично второму, второй

процесс будет вынужден ждать первого, N-ый процесс будет ждать $N-1$, который находятся в критической секции. Таким образом, всегда есть процесс, который находится в критической секции и тогда пропускная способность для этого случая будет оцениваться как:

$$\frac{\alpha}{3R_I + C + 2W}.$$

Однако, учитывая, что для анализа в среде исполнения используется симметричный планировщик, то этот случай не включен в итоговую модель.

Для моделирования других случаев, где $P < (N-1) \cdot C$ также использовалась модель PRAM. Варьируя работу в параллельной части P, все рассмотренные случаи сводились к рассмотренному случаю выше на рисунке 12.

Таким образом, была получена следующая аналитическая модель для прогнозирования пропускной способности для классов алгоритмов, использующих грубый механизм синхронизации при помощи алгоритма взаимного исключения MCS:

$$\begin{cases} \frac{\alpha}{2R_I+C+2W} & \text{, если } P + W \leqslant (N-1) \cdot (2W + C + R_I) \\ \frac{\alpha \cdot N}{(2W+C+R_I)+(P+W)} & \text{, иначе} \end{cases}$$

2.4. Эксперименты

Эксперименты производились на двух серверах, оснащенными процессорами Intel® Xeon® Gold 6230 и AMD® Opteron® 6378 соответственно. Такие процессоры принадлежат к семейству самых популярных CPU Intel Xeon и AMD Opteron, которые используются на современных серверах.

В первом случае использовался процессор Intel Xeon Gold 6230, где было выделено 16 ядер. Исполнение происходило на ОС Ubuntu 20.04.1 LTS (Focal Fossa). В качестве компилятора кода был выбран MinGW GCC 5.2.0 с указанием опции -O0, чтобы избежать оптимизаций со стороны компилятора, которые могут внести помехи при измерениях. Исходный код доступен по следующему адресу: `https://github.com/exyfi/complexity-lock-with-libslock`.

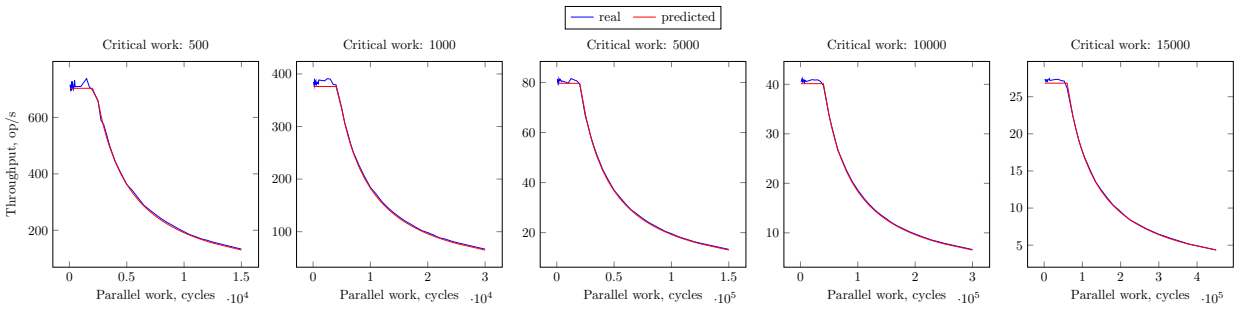

Рисунок 13 – Результаты сравнения полученной аналитической модели с тестами производительности на Intel® Xeon® Gold 6230: 5 потоков

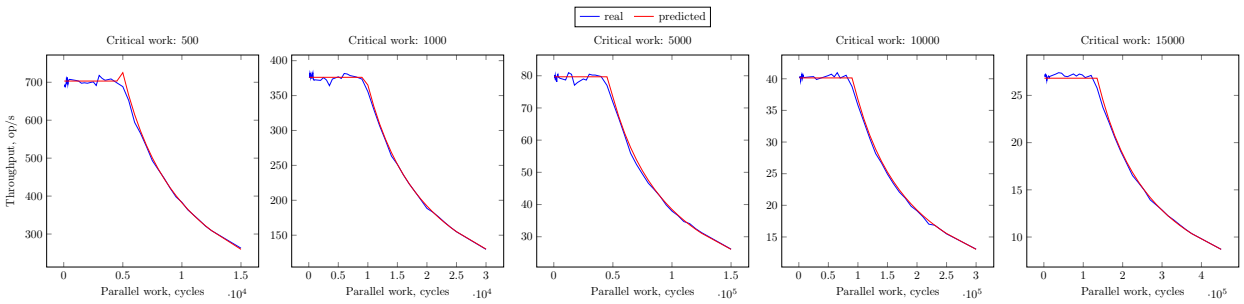

Рисунок 14 – Результаты сравнения полученной аналитической модели с тестами производительности на Intel® Xeon® Gold 6230: 10 потоков

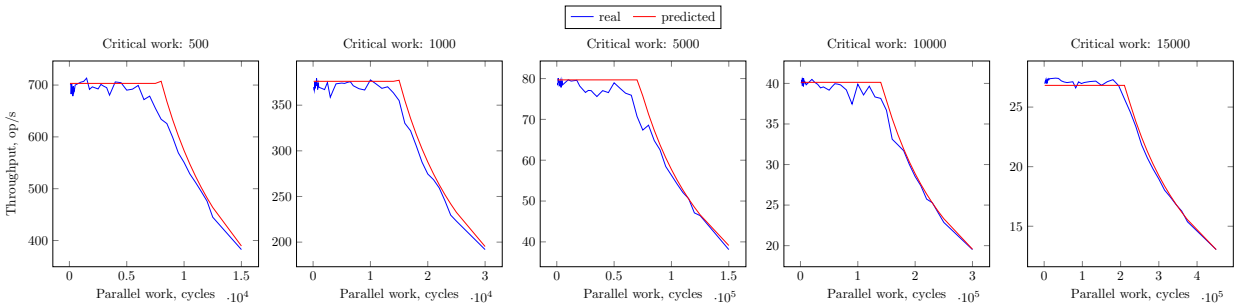

Рисунок 15 – Результаты сравнения полученной аналитической модели с тестами производительности на Intel® Xeon® Gold 6230: 15 потоков

На рисунках 13-15 показаны результаты экспериментов. Эксперименты производились для 5, 10, 15 потоков, соответственно. Были зафиксированы критические секции $C \in \{500, 1000, 5000, 10000, 15000\}$. Параллельная секция в экспериментах зависела от работы критической секции и вычислялась следующим образом: $P = x \cdot C$, где $x \in \{0.1, 0.2, 0.25, 0.3, 0.4, 0.45, 0.5, 0.55, 0.6, 0.7, 0.8, 0.9, 1, 2, 2.5, 3, 3.5, 4, 5, 5.5, 6, 6.5, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 30\}$.

Для каждого значения тройки параметров N, C, P был произведен запуск эксперимента на 10 секунд. На оси ординат указана пропускная способность (операций в секунду), на оси абсцисс — работа в параллельной секции. Синим цветом на графике обозначена пропускная способность реального исполнения системы. Красным обозначена предсказанная пропускная способность, использующая итоговую модель 2.3.

Экспериментально были получены значения констант $\alpha \approx 4.04 \cdot 10^5$, $W \approx 15$, $R \approx 15$. На графике также видно, что при достаточно малой работе в параллельной секции, до точки перегиба, предсказанная производительность не до конца описывает реальную пропускную способность. Это объясняется тем, что модель упрощена и не берет во внимание тот факт, что при достаточно большой конкуренции (англ. contention), происходят попытки записи из разных состояний кэш-линии. Также, в получившейся модели для упрощения исключены из оценки возможные предсказания переходов (англ. branch prediction), а также параллелизм на уровне инструкций (англ. instruction level parallelism), которые могут влиять на итоговоую пропускную способность.

Во втором случае на сервере использовался процессор AMD® Opteron® 6378 2.4 GHz, 256 GB RAM, где было выделено 16 ядер. Исполнение происходило на ОС Ubuntu 18.04.2 LTS (Bionic Beaver). В качестве компилятора кода был выбран MinGW GCC 5.2.0 с указанием опции -O0, чтобы избежать оптимизаций со стороны компилятора, которые могут внести помехи при измерениях. Исходный код также доступен по следующему адресу: https://github.com/exyfi/complexity-lock-with-libslock.

Как и в случае 2.4, были использованы те же значения для количества потоков, работы в критической и параллельной секции. Результаты экспериментов представлены на рисунках 16-18.

Экспериментально были получены значения констант $\alpha \approx 1.24 \cdot 10^5$, $W \approx 15, R \approx 15$.

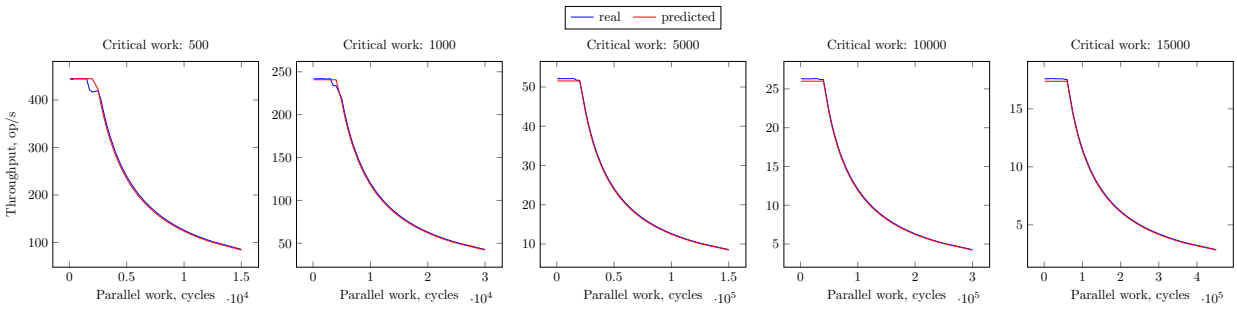

Рисунок 16 – Результаты сравнения полученной аналитической модели с тестами производительности на AMD® Opteron® 6378: 5 потоков

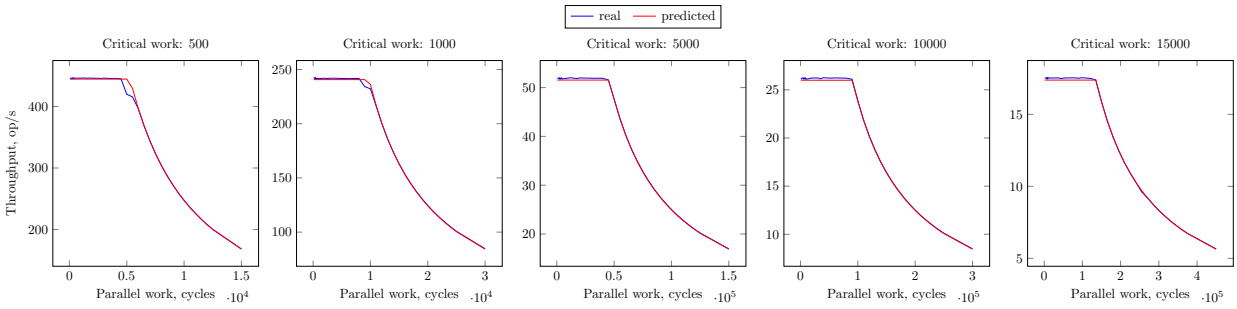

Рисунок 17 – Результаты сравнения полученной аналитической модели с тестами производительности на AMD® Opteron® 6378: 10 потоков

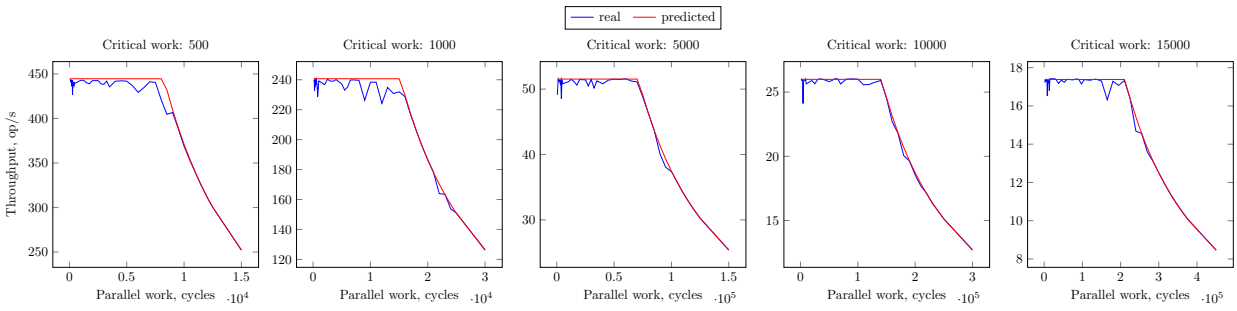

Рисунок 18 – Результаты сравнения полученной аналитической модели с тестами производительности на AMD® Opteron® 6378: 15 потоков

Выводы по главе 2

В рамках главы была рассмотрена среда исполнения, в которой проводятся эксперименты. Описаны все необходимые для анализа условия. Также, было проведено сравнение двух алгоритмов взаимного исключения MCS и CLH. После этого, был расширен имеющийся анализ [23] для классов алгоритмов, использующих грубые механизмы синхронизации, для алгоритма блокировки MCS. Получившаяся аналитическая модель была сравнена с тестами

производительности. Применимость получившейся модели была протестирована на представителях двух популярных семейств процессоров Intel Xeon и AMD Opteron. Помимо этого, в главе были описаны ограничения имеющейся модели.

ГЛАВА 3. АНАЛИЗ ПРОИЗВОДИТЕЛЬНОСТИ LOCK-FREE СТРУКТУРЫ ДАННЫХ НА ПРИМЕРЕ TREIBER STACK

В этой главе рассмотрен анализ теоретической пропускной способности для lock-free структур данных на примере популярной реализации многопоточного стека — Treiber Stack [21].

В начале главы описывается различие между конкурентными структурами данных, использующих алгоритмы взаимного исключения, и lock-free структурами данных. Описаны преимущества lock-free алгоритмов. Далее происходит непосредственный обзор самого алгоритма Treiber Stack. На следующем шаге в анализе представлена абстрактная структура данных, использующая операции многопоточного стека и имеющая параллельную секцию. В той же секции рассмотрены необходимые для построения модели операции, которые присутствуют в имеющейся структуре данных. Дав оценку операциям и опираясь на среду исполнения, описанную в секции 2.2, рассмотрены возможные случаи исполнения, которые зависят от количества работы в параллельной секции. После рассмотрения случаев исполнения, описывается получившаяся аналитическая модель. В конце главы получившаяся модель сравнивается с тестами производительности.

3.1. Lock-free структуры данных

В отличие от структур данных, использующие алгоритмы взаимного исключения, многопоточные структуры данных имеют безусловные гарантии прогресса. Безусловность в данном случае означает, что, даже в том случае, если во время работы в критической секции, поток будет вытеснен планировщиком операционной системы, то прогресс в системе продолжится, тогда как, в случае с условным прогрессом, потоки будут находиться в фазе активного ожидания и ждать до тех пор, пока вытесненный поток вновь продолжит исполнение и закончит работу в критической секции.

Свойства безусловного прогресса от самого слабого к самому сильному:

— **Отсутствие помех (англ. `obstruction-freedom`)**. Если несколько потоков пытаются выполнить операцию, то любой из них должен выполнить ее за конечное время, если все другие потоки остановить.

— **Отсутствие блокировок (англ. lock-freedom)**. Если несколько потоков пытаются выполнить операцию, то хотя бы один из них должен выполнить свою операцию за конечное время независимо от действия/-бездействия других потоков.

— **Отсутствие ожидания (англ. wait-freedom)**. Если какой-то поток пытается выполнить операцию, то он ее выполнит за конечное время независимо от действия/бездействия других потоков. Включает в себя все выше описанные свойства.

3.2. Treiber Stack

На рисунке 19 представлен псевдокод многопоточной lock-free структуры данных Treiber Stack.

```
1  class Node:
2    T data;
3    Node next
4
5  head = null //shared, atomic
6
7  push(data):
8    newHead = Node(data)
9    while !success:
10     oldHead = atomic_read(head) // M or X
11     newHead.next = oldHead
12     success = head.compareAndSet(oldHead, newHead) // W
13
14 pop():
15   Node oldHead
16   while !success:
17      oldHead = atomic_read(head) // M or X
18      if (oldHead == null) {
19            return DEFAULT_VALUE // corner case
20      }
21      newHead = oldHead.next
22      success = head.compareAndSet(oldHead, newHead) // W
23
24   return oldHead.data
```

Рисунок 19 – Псевдокод lock-free алгоритма Treiber Stack

Заметим, что для анализа, можно свести операции `pop/push` к одной мета-операции и в дальнейшем анализе моделировать использование многопоточного стека, используя только одну, например, `push`. Такое возможно, так как обе операции имеют одинаковую структуру:

— В обоих случаях сначала происходит атомарное чтение `head`.

— Далее в цикле выполняется операция присвоение нового значения `head`.

— Для выхода из цикла выполняется CAS-операция над общим объектом `head`.

Такая мета-операция будет выглядеть, как показано на Рисунке 20.

```
1 pop_or_push_operation():
2   while !success do
3     current = atomic_read(head)
4     new = critical_work(current)
5     success = head.compareAndSet(current, new)
```

Рисунок 20 – Абстрактная lock-free операция

3.3. Оценка производительности

На рисунке 21, в рассматриваемой многопоточной структуре данных явно используется операция `push` алгоритма lock-free стека, а также присутствует параллельная часть.

```
1 class Node:
2   T data;
3   Node next
4
5 head = null //shared, atomic
6
7 operation():
8   newHead = Node(data)
9   while !success:
10    oldHead = atomic_read(head) // M or X
11    newHead.next = oldHead
12    success = head.compareAndSet(oldHead, newHead); // W
13
14  for i in 1..P: / / P
15    //nop
```

Рисунок 21 – Структура данных, использующая операции Treiber Stack

Далее рассмотрены по отдельности количество работы конкретных операций в структуре данных, изображенной на Рисунке 21.

Операция начинается с атомарного чтения общего объекта head (строка 10) в переменную `oldHead`. Для совершения операции требуется M тактов — количество тактов, которые требуются для атомарного чтения, в случае если эта операции эксклюзивная, в противном случае, требуется X тактов. Далее, происходит присвоение локальному объекту `newHead.next` ссылки на текущий `head` стека (строка 11) (в случае `pop`, объекту `newHead` присваивается значение `oldHead.next`). Операция совершается каждый раз в случае

неуспешного выполнения `compareAndSet` в строке 12 и для выполнения это операции при неуспешном `CAS` требуется также M тактов, так как заново происходит чтение общего объекта `head`. Для операции `CAS`, как и в секции 2.3, требуется W работы. После успешного выполнения `CAS` поток выходит из цикла и переходит к исполнению работы в параллельной секции (строки 14-15), для которой требуется P тактов. В противном случае, если операция `CAS` была неуспешной, поток заново в цикле пытается обновить ссылку на `head` до тех пор, пока результат выполнения `CAS` не станет успешным.

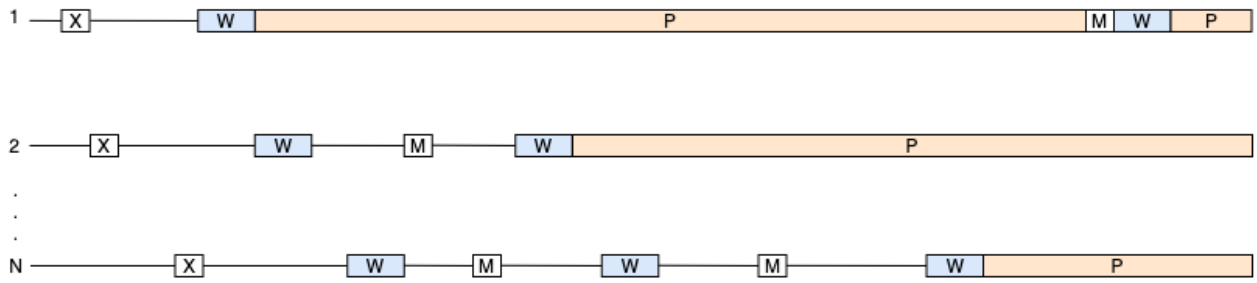

Рисунок 22 – Случай 1:$P \geqslant (N-1) \cdot (M+W)$

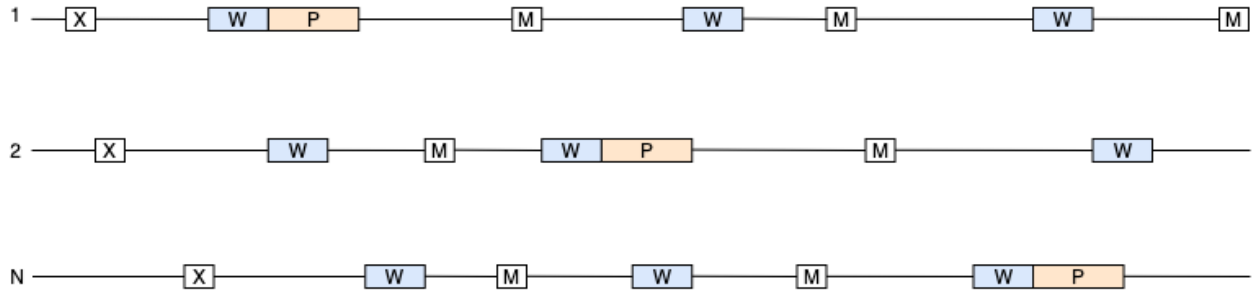

Рисунок 23 – Случай 2:$P < (N-1) \cdot (M+W)$

В первом случае исполнения (Рисунок 22), при достаточно большой параллельной части, ввиду отсутствия конкуренции за исполнение `CAS` операции, процесс совершает успешную операцию `pop/push` с первой попытки и далее производит работу в параллельной части. В таком случае, в любой момент времени поток, выполняющий операцию `pop/push` не совершает повторных попыток в CAS-цикле и в каждый момент времени выполняет полезную работу. Таким образом, пропускную способность для данного случая можно оценить как число операций в момент времени, а именно:

$$\frac{\alpha \cdot N}{(P+W+M)}.$$

Во втором случае (Рисунок 23) при малой работе в параллельной секции, возникает большая конкуренция за общий ресурс `head`. В каждый момент времени только один поток успешно выполняет операцию `CAS`, тогда как другие потоки, конкурирующие с этим процессом, вынуждены повторить свою попытку. Учитывая, что в каждый момент времени существует поток, сумевший успешно совершить операцию `CAS`, такое утверждение гарантирует свойство `lock-freedom`, то итоговая пропускная способность для случая оценивается как

$$\frac{\alpha}{M+W}.$$

Итоговая аналитическая модель:

$$\begin{cases} \frac{\alpha}{M+W} & \text{, если } P \leqslant (N-1)\cdot(M+W) \\ \frac{\alpha\cdot N}{(P+M+W)} & \text{, иначе} \end{cases}$$

3.4. Эксперименты

Сравнения с тестами производительности были выполнены на двух серверах, оснащенными процессорами Intel® Xeon® Gold 6230 и AMD® Opteron® 6378, соответственно.

На сервере с процессором Intel® Xeon® Gold 6230 было выделено 16 ядер. Исполнение происходило на ОС Ubuntu 20.04.1 LTS (Focal Fossa). В качестве компилятора кода был выбран MinGW GCC 5.2.0 с указанием опции -O0, чтобы избежать оптимизаций со стороны компилятора, которые могут внести помехи при измерениях.

На рисунке 24 показаны результаты экспериментов на сервере с Xeon® Gold 6230. Эксперименты производились для 5, 10, 15 потоков, соответственно. Параллельная секция в экспериментах вычислялась следующим образом: $P = x \cdot C$, где $C = 100$, $x \in \{1, 2, \ldots, 50\}$.

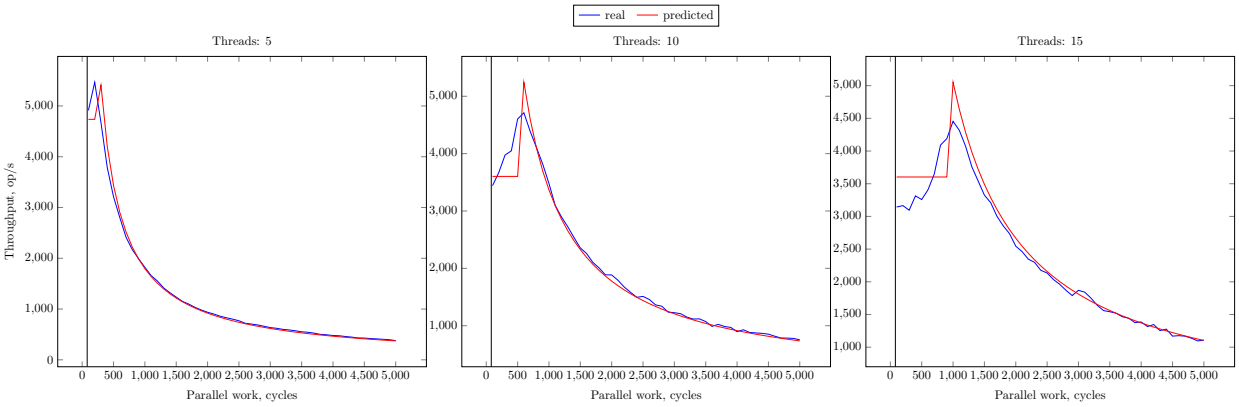

Рисунок 24 – Результаты сравнения полученной аналитической модели с тестами производительности на Intel® Xeon® Gold 6230 для 5, 10 и 15 потоков

Для каждого значения пары N и P был произведен запуск эксперимента на 10 секунд. На оси ординат указана пропускная способность (операций в секунду), на оси абсцисс — работа в параллельной секции. Синим цветом на графике обозначена пропускная способность реального исполнения системы. Красным обозначена предсказанная пропускная способность, использующая итоговую модель 3.3.

Экспериментально были получены значения констант $\alpha \approx 4.04 \cdot 10^5$, $W \approx 15$, $M \approx 50$. Эти константы соответствуют тем, что были получены в экспериментах 2.4.

Аналогично, были проведены эксперименты на сервере с процессором AMD® Opteron®. На данном сервере использовался процессор AMD® Opteron® 6378 2.4 GHz, 256 GB RAM, где было выделено 16 ядер. Исполнение происходило на ОС Ubuntu 18.04.2 LTS (Bionic Beaver). В качестве компилятора кода был выбран MinGW GCC 5.2.0 с указанием опции -O0, чтобы избежать оптимизаций со стороны компилятора, которые могут внести помехи при измерениях.

Как и в случае 3.4, были использованы те же значения для количества потоков и работы в параллельной секции. Результаты экспериментов представлены на рисунке 25.

Экспериментально были получены значения констант $\alpha \approx 1.24 \cdot 10^5$, $W \approx 15, M \approx 30$. Эти константы соответствуют тем, что были получены в экспериментах 2.4.

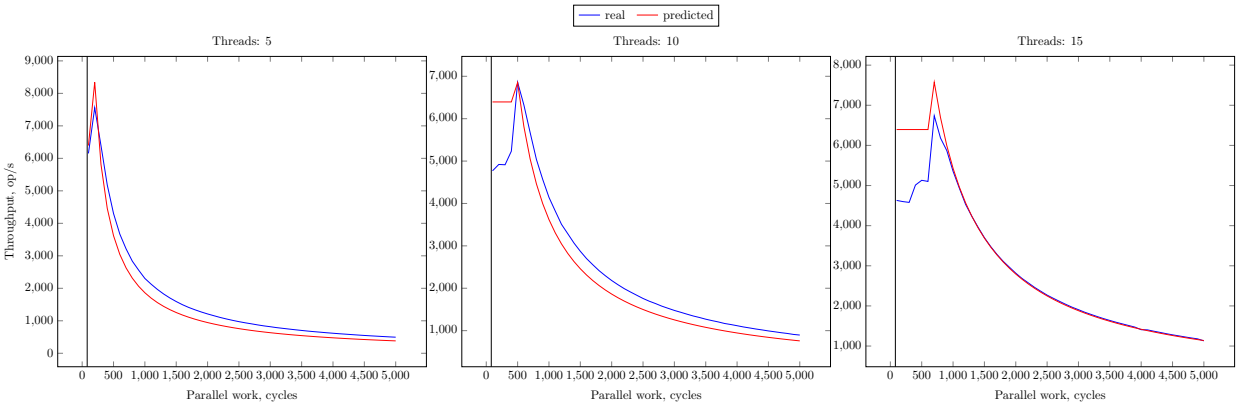

Рисунок 25 – Результаты сравнения полученной аналитической модели с тестами производительности на AMD® Opteron® 6378 для 5, 10 и 15 потоков

На графиках видно, что при достаточно малой работе в параллельной секции, до точки перегиба, предсказанная производительность не до конца описывает реальную пропускную способность. Это объясняется тем, что модель упрощена и не берет во внимание тот факт, что при достаточно большой конкуренции (англ. contention) происходят попытки записи из разных состояний кэш-линии. Помимо этого, есть ограничения, которые накладывает на получившуюся модель, модель PRAM. На графике видно, что до того, как P меньше, чем работа $(N-1)\cdot(M+W)$, производительность с увеличением P возрастает и достигает максимума при $P = (N-1)\cdot(M+W)$. Возрастание производительности при росте P может свидетельствовать о том, что с увеличением P увеличивается число потоков, которые в сразу же выполняют полезную работу, не делая повторные попытки в CAS-цикле. Также, в получившейся модели для упрощения исключены из оценки возможные предсказания переходов (англ. branch prediction), а также параллелизм на уровне инструкций (англ. instruction level parallelism), которые могут влиять на итоговую пропускную способность.

Но также стоит отметить, что lock-free структуры данных являются, во-первых спекулятивными [23], а также в lock-free структурах данных присутствуют logical conflicts и hardware conflicts, описанные в смежной работе [3], которые влияют на итоговую реальную пропускную способность.

Выводы по главе 3

В рамках этой главы был расширен имеющийся анализ [23] для многопоточных lock-free структур данных на примере алгоритма многопоточного стека, Treiber Stack. Были проанализированы особенности lock-free структур данных. Далее, была разработана аналитическая модель для многопоточной структуры данных, использующей операции Treiber Stack. Получившаяся аналитическая модель была сравнена с тестами производительности. Применимость получившейся модели была протестирована на представителе семейства процессоров Intel Xeon. Также, в главе были описаны основные трудности при оценивании производительности lock-free структур данных и описаны ограничения получившейся аналитической модели.

ЗАКЛЮЧЕНИЕ

В рамках работы были получены следующие результаты:

— Была реализована модель прогнозирования производительности для многопоточных структур данных, использующих грубые механизмы синхронизации, с помощью алгоритма взаимного исключения MCS.
— Реализована модель прогнозирования производительности для lock-free структур данных на примере стека Трайбера.
— Удалось сравнить предложенные модели с тестами производительности. Для первого случая, полученная модель была протестирована на представителях семейств популярных CPU Intel Xeon и AMD Opteron, которые используются на современных серверах. Во втором случае, аналитическая модель была протестирована только на представителе семейства CPU Intel Xeon.

В рамках улучшения этой работы предлагается сделать следующее:

— Расширить аналитическую модель для прогнозирования производительности для многопоточных структур данных, использующих грубые механизмы синхронизации, с помощью других алгоритмов взаимной блокировки, например, `ticket lock`, `test-and-test-and-set lock`.
— Реализовать модель прогнозирования производительности для lock-free очереди Майкла Скотта [17].

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1 Amdahl's law [Электронный ресурс]. — 2021. — URL: https://en.wikipedia.org/wiki/Amdahl%27s_law.

2 *Anderson R. J.*, *Snyder L.* A Comparison of Shared and Nonshared Memory Models of Parallel Computation [Электронный ресурс]. — 1991. — URL: https://core.ac.uk/download/pdf/190853994.pdf.

3 *Aras Atalar P. R.-G.*, *Tsigas P.* Analyzing the Performance of Lock-Free Data Structures: A Conflict-based Model [Электронный ресурс]. — 2015. — URL: https://arxiv.org/pdf/1508.03566.pdf.

4 *Aras Atalar P. R.-G.*, *Tsigas P.* How Lock-free Data Structures Perform in Dynamic Environments: Models and Analyses [Электронный ресурс]. — 2016. — URL: https://arxiv.org/pdf/1611.05793.pdf.

5 Compare-and-swap [Электронный ресурс]. — 2021. — URL: https://en.wikipedia.org/wiki/Compare-and-swap.

6 *Craig T.* Building FIFO and priorityqueuing spin locks from atomic swap [Электронный ресурс]. — 1993. — URL: ftp://ftp.cs.washington.edu/tr/1993/02/UW-CSE-93-02-02.pdf.

7 Critical Section [Электронный ресурс]. — 2021. — URL: https://en.wikipedia.org/wiki/Critical_section.

8 *Herlihy M.* Wait-Free Synchronization [Электронный ресурс]. — 1991. — URL: http://cs.brown.edu/~mph/Herlihy91/p124-herlihy.pdf.

9 *Herlihy M.*, *Shavit N.* The Art of Multiprocessor Programming. — San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 2008. — ISBN 0123705916.

10 *Jaja J. F.* An introduction to parallel algorithms [Электронный ресурс]. — 1992. — URL: http://www.cs.utah.edu/~hari/teaching/bigdata/book92-JaJa-parallel.algorithms.intro.pdf.

11 *Lamport L.* Time, Clocks, and the Ordering of Events in a Distributed System [Электронный ресурс]. — 1978. — URL: https://www.microsoft.com/en-us/research/uploads/prod/2016/12/Time-Clocks-and-the-Ordering-of-Events-in-a-Distributed-System.pdf.

12 *Manchanda N.*, *Anand K.* Non-Uniform Memory Access (NUMA) [Электронный ресурс]. — 2013. — URL: https://web.archive.org/web/20131228092942/http://www.cs.nyu.edu/~lerner/spring10/projects/NUMA.pdf.

13 *Mellor-Crummey J. M.*, *Scott M. L.* Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors [Электронный ресурс]. — 1991. — URL: http://web.mit.edu/6.173/www/currentsemester/readings/R06-scalable-synchronization-1991.pdf.

14 Moore's law [Электронный ресурс]. — 2021. — URL: https://en.wikipedia.org/wiki/Moore%27s_law.

15 *Papamarcos M. S.*, *Patel J. H.* A Low-overhead coherence solution for multiprocessors with private cache memories [Электронный ресурс]. — 1984. — URL: https://dl.acm.org/doi/pdf/10.1145/800015.808204.

16 Process [Электронный ресурс]. — 2021. — URL: https://en.wikipedia.org/wiki/Process_(computing).

17 *Scott M. L.* Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms [Электронный ресурс]. — 1996. — URL: https://www.cs.rochester.edu/~scott/papers/1996_PODC_queues.pdf?.

18 *Sutter H.* The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software [Электронный ресурс]. — 2005. — URL: http://www.gotw.ca/publications/concurrency-ddj.htm.

19 The Shared Memory and Message Passing Models of Interprocess Communication [Электронный ресурс]. — 2021. — URL: http://www.umsl.edu/~siegelj/CS4740_5740/Overview/MPSM.html.

20 Thread [Электронный ресурс]. — 2021. — URL: https://en.wikipedia.org/wiki/Thread_(computing).

21 *Treiber R. K.* Systems programming: Coping with parallelism [Электронный ресурс]. — 1968. — URL: https://dominoweb.draco.res.ibm.com/reports/rj5118.pdf.

22 *Tudor David Rachid Guerraoui V. T.* Everything You Always Wanted to Know About Synchronization but Were Afraid to Ask [Электронный ресурс]. — 2013. — URL: https://dl.acm.org/doi/pdf/10.1145/2517349.2522714.

23 *Vitalii Aksenov Dan Alistarh P. K.* Performance Prediction for Coarse-Grained Locking [Электронный ресурс]. — 2018. — URL: https://arxiv.org/pdf/1904.11323.pdf.