Моделирование и оптимизация выполнения транзакционных секций на примере потокобезопасных хеш-таблиц и деревьев поиска

В. А. Смирнов[1], А. Р. Омельниченко[2], А. А. Пазников[3]
Санкт-Петербургский государственный электротехнический университет
«ЛЭТИ» им. В.И. Ульянова (Ленина)
[1]smirnov3308@gmail.com, [2]omelnichenko3308@gmail.com, [3]apaznikov@gmail.com

***Аннотация*. В работе предложены алгоритмы реализации потокобезопасных ассоциативных массивов (красно-чёрное дерево, хеш-таблица с открытой адресацией на основе метода Hopscotch hashing разрешения коллизий) с использованием программной транзакционной памяти (software transactional memory). Представлен анализ эффективности ассоциативных массивов при разном количестве задействованных потоков и процессорных ядер, приведено сравнение с аналогичными структурами данных на основе крупнозернистых и мелкозернистых блокировок, также сформулированы рекомендации по выбору алгоритмов выполнения транзакций. Описаны принципы работы программной транзакционной памяти, различные политики обновления объектов в памяти и стратегии обнаружения конфликтов. Представлены различные методы блокировки при использовании транзакционной памяти, реализованной в компиляторе GCC 5.4.0. Коротко рассмотрены альтернативы, применяемые в настоящее время, выделены их достоинства и недостатки.**

Ключевые слова: транзакционная память; потокобезопасные структуры данных; красно-чёрное дерево; хеш-таблица; многопоточное программирование; синхронизация многопоточных программ

I. Введение

На Многоядерные вычислительные системы (ВС) с общей памятью в настоящее время используются для решения различных сложных задач. К таким ВС относятся SMP и NUMA системы. В связи с ростом количества процессорных ядер остро стоит проблема обеспечения масштабируемого доступа параллельных потоков к разделяемым структурам данных.

Синхронизация доступа к разделяемым ресурсам является одной из важных и сложных задач при разработке многопоточных программ. На сегодняшний день основными методами решения данной проблемы являются:

- Применение блокировок (mutex, spinlock, critical section и т.д.).
- Алгоритмы и структуры данных, свободные от блокировок (lockless, lock-free).
- Транзакционная память (transactional memory).

При использовании традиционных примитивов синхронизации (семафоры, мьютексы, спинлоки и др.), необходимо обеспечить не только корректность программы – отсутствие взаимных блокировок и состояний гонок за данными (data race) – но и минимизировать время ожидания доступа к критическим секциям (разделяемым ресурсам). Классические методы синхронизации, основанные на механизме блокировок, позволяют организовывать в параллельных программах критические секции, выполнение которых возможно только одним потоком в каждый момент времени [1].

При использовании блокировок имеет место альтернатива использования крупнозернистых и мелкозернистых блокировок. Потокобезопасные структуры на основе крупнозернистых блокировок (coarse-grained lock) легко реализовывать, но они не обеспечивают предельных показателей эффективности, так как обладают ограниченным параллелизмом выполнения операций. Мелкозернистые блокировки (fine-grained lock) обеспечивают хорошую производительность, но их использование – сложная задача [2].

Структуры данных, свободные от блокировок, строятся на базе атомарных операций, таких как запись (atomic store), чтение (atomic load), сравнение с обменом (compare and swap, CAS) и др. Данный подход позволяет полностью избавиться от взаимных блокировок, однако может привести к возникновению активных блокировок (livelock) – ситуаций, когда два потока одновременно пытаются изменить структуру данных, но каждый из них циклически выполняет свою операцию из-за изменений, произведённых другим потоком. Также к недостаткам данного подхода можно отнести сложность реализации структур, свободных от блокировок [3], и проблема ABA освобождения памяти.

II. Программная транзакционная память

На сегодняшний день транзакционная память является одним из наиболее перспективных механизмов

Работа выполнена при поддержке Совета по грантам Президента РФ для государственной поддержки молодых российских ученых (проект СП-4971.2018.5). Публикация выполнена в рамках государственной работы «Инициативные научные проекты» базовой части государственного задания Министерства образования и науки Российской Федерации (ЗАДАНИЕ № 2.6553.2017/БЧ).

синхронизации, её использование позволяет выполнять не конфликтующие между собой операции параллельно. Транзакционная память упрощает параллельное программирование, выделяя группы инструкций в атомарные транзакции – конечные последовательности операций транзакционного чтения/записи памяти [4]. Изменения, вносимые потоком внутри транзакционных секций, незаметны другим потокам до тех пор, пока транзакция не будет зафиксирована (commit). Если во время выполнения транзакции потоки обращаются к одной области памяти, и один из потоков совершает операцию записи, то транзакция одного из потоков отменяется (cancel, rollback). При выполнении транзакционной секции одним потоком, другие потоки наблюдают состояние либо непосредственно до, либо непосредственно после выполнения транзакции. Важным свойством транзакционной памяти является линеаризуемость выполнения транзакций: ряд успешно завершённых транзакция эквивалентен некоторому последовательному их выполнению. Для выполнения транзакционных секций runtime-системой компилятора создаются транзакции. Операция транзакционного чтения выполняет копирование содержимого указанного участка общей памяти в соответствующий участок локальной памяти потока. Транзакционная запись копирует содержимое указанного участка локальной памяти в соответствующий участок общей памяти, доступной всем потокам [1].

Основными различиями, определяющими эффективность программной транзакционной памяти, являются политика обновления объектов в памяти и стратегия обнаружения конфликтов.

Политика обновления объектов в памяти определяет, когда изменения объектов внутри транзакции будут зафиксированы. Существуют две основные политики – ленивая и ранняя [5]. В случае ленивой политики (lazy version management) все операции с объектами откладываются до момента фиксации транзакции. Все операции записываются в специальном журнале изменений (redo log), который при фиксации транзакции используется для отложенного выполнения операций. Использование журнала изменений замедляет операцию фиксации, но существенно упрощает процедуры её отмены и восстановления.

Ранняя политика обновления объектов в памяти (eager version management) предполагает, что все изменения объектов сразу записываются в память. В случае возникновения конфликта оригинальное состояние восстанавливается с помощью журнала отката (undo log). Характеризуется быстрой фиксацией транзакции, но медленным выполнением процедуры её отмены [5].

Момент времени, когда инициируется алгоритм обнаружения конфликта, определяется стратегией обнаружения конфликтов [5]. При отложенной стратегии (lazy conflict detection) данная процедура запускается на этапе фиксации транзакции. Недостатком этой стратегии является позднее обнаружение конфликта и как следствие выполнение лишних операций.

Пессимистичная стратегия обнаружения конфликтов (eager conflict detection) запускает алгоритм их обнаружения при каждой операции обращения к памяти. Такой подход позволяет избежать недостатков отложенной стратегии, но может привести к значительным накладным расходам.

В данной работе используется реализация транзакционной памяти в компиляторе GCC (библиотека libitm), в котором используется ранняя политика обновления объектов в памяти и реализован комбинированный подход к обнаружению конфликтов – отложенная стратегия используется совместно с пессимистической [1].

В данной работе рассматривается применение программной транзакционной памяти для реализации потокобезопасных структур данных. К одним из наиболее распространённых структур данных в многопоточных программах относятся потокобезопасные ассоциативные массивы. В настоящее время существует множество реализаций ассоциативных массивов, например: контейнер map в библиотеке STL языка C++, тип Dictionary в C#, класс Hash в Ruby, тип словаря в Python и Java.util.map в Java.

В статье предлагаются алгоритмы реализации потокобезопасных хеш-таблиц (на основе алгоритма Hopscotch hashing разрешения коллизий) и красно-чёрных деревьев поиска с использованием транзакционной памяти.

III. Алгоритм хеширования Hopscotch hashing

Хеш-таблица – одна из наиболее широко используемых структур данных при реализации ассоциативных массивов. Повышение её эффективности позволит сократить время выполнения большого числа параллельных программ [6]. В хеш-таблице с открытой адресацией, в отличие от хеш-таблицы с закрытой адресацией, в ячейке хранится не указатель на связный список, а один элемент (ключ, значение). Если при вставке элемента соответствующая ячейка занята, алгоритм вставки проверяет следующие ячейки до тех пор, пока не будет найдена свободная. Данный подход характеризуется хорошей локальностью кэш-памяти, поскольку каждая кэш-линия содержит сразу несколько записей хеш-таблицы. Существенным недостатком этого подхода является снижение производительности по мере заполнения хеш-таблицы.

Hopscotch hashing [6] объединяет в себе преимущества трёх подходов: Cuckoo hashing [7], метод цепочек [8] и метод линейного хеширования [8]. Алгоритм обладает высоким коэффициентом попадания в кэш-память. В худшем случае временная сложность операции добавления – $O(n)$, в лучшем случае – $O(1)$. Операции поиска и удаления выполняются за константное время.

Основная идея Hopscotch hashing заключается в использовании свойства пространственной локальности кэш-памяти. Искомый элемент находится в окрестности ячейки, на которую указывает хеш-функция. В данной

работе размер окрестности $H = 32$. Время поиска элемента в окрестности близко к времени поиска в одной ячейке. Это достигается при вставке элемента вытеснением других элементов.

Каждая ячейка содержит информацию о том, какие ячейки в окрестности имеют ключ с таким же значением хеш-функции. В данной реализации эта информация представлена в виде связного списка: в ячейке содержится относительная позиция следующей и первой ячейки в списке.

Ниже представлена функция добавления в хеш-таблицу. Критическая секция выделена в транзакцию (строки 4-23). Это позволяет потокам выполнять добавление в хеш-таблицу параллельно. Если элемент с заданным ключом уже находится в хеш-таблице, функция возвращает false (строка 6). Если в пределах *ADD_RANGE* (в данной реализации *ADD_RANGE* = 256) пустую ячейку найти не удалось, происходит возврат false, а функция Resize() изменяет размер хеш-таблицы и производит рехеширование (строки 21-22). Если элемент успешно добавлен в таблицу, функция возвращает true (строка 19).

```
1: procedure HopscotchInsert
2:   hash = HashFunc(key)
3:   start_bucket = segments_arys + hash
4:   transaction
5:     if Contains(key) then
6:       return false
7:     end if
8:     free_bucket_index = hash
9:     free_bucket = start_bucket
10:    distance = 0
11:    FindFreeBucket(free_bucket, distance)
12:    if distance < ADD_RANGE then
13:      if distance > HOP_RANGE then
14:        FindCloser(free_bucket, distance)
15:      end if
16:      start_bucket.hop_info |= (1 << distance)
17:      free_bucket.data = data
18:      free_bucket.key = key
19:      return true
20:    end if
21:    Resize()
22:    return false
23: end transaction
```

Рис. 1. Функция добавления в хеш-таблицу

Пример выполнения операции вставки представлен на рис. 2.

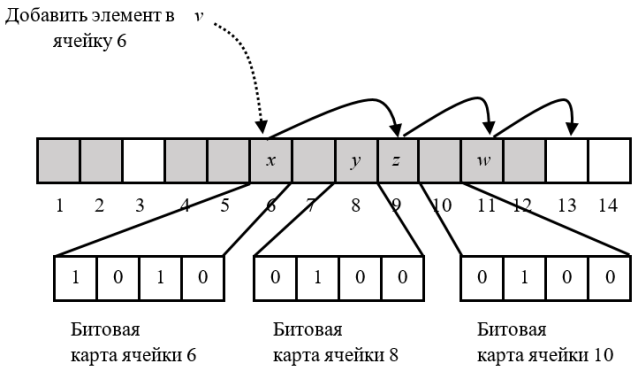

Рис. 2. Пример выполнения операции вставки

Ниже представлена функция удаления элемента из хеш-таблицы. Критическая секция функция удаления элемента также выделена в транзакцию (строки 4-19). Если удаление элемента прошло успешно, функция возвращает true (строка 14), в противном случае – false (строка 18). Для обеспечения максимальной производительности был выбран минимально возможный размер транзакционной секции.

```
1: procedure HopscotchRemove
2:   hash = HashFunc(key)
3:   start_bucket = segments_arys + hash
4:   mask = 1
5:   transaction
6:     hop_info = start_bucket.hop_info
7:     for i = 0 to HOP_RANGE do
8:       mask <<= 1
9:       if mask & hop_info then
10:        check_bucket = start_bucket + i
11:      if key = check_bucket.key then
12:        check_bucket.key = NULL
13:        check_bucket.data = NULL
14:        start_bucket.hop_info &= ~(1 << i)
15:        return true
16:      end if
17:    end if
18:  end for
19:  return false
20: end transaction
```

Рис. 3. Функция удаления из хеш-таблицы

IV. Потокобезопасное красно-чёрное дерево

В данной работе также предлагается реализация потокобезопасного красно-черного дерева на основе транзакционной памяти. Красно-черные деревья широко применяется при реализации ассоциативных массивов, обеспечивают логарифмический рост высоты дерева в зависимости от числа узлов и выполняющее основные операции дерева поиска: добавление, удаление и поиск узла за $O(\log n)$. Сбалансированность достигается за счёт введения дополнительного атрибута узла дерева – «цвета» [9].

Ниже представлена функция добавления элемента в красно-чёрное дерево. Критическая секция выделена в транзакцию (строки 3-10), это позволяет потокам выполнять добавление элементов, не нарушая целостность данных и баланс дерева.

```
1: procedure RBTREEINSERT
2:   x = NEWNODE(data)
3:   transaction
4:     if !FINDPARENT(x) then
5:       return false
6:     end if
7:     INSERTNODE(x)
8:     INSERTBALANCE(x)
9:   return true
10: end transaction
```

Рис. 4. Функция добавления в красно-чёрное дерево

Функция INSERTNODE вставки узла (строка 8) определяет поля left или right родительского узла. Если родительский узел отсутствует, добавляемый узел становится корнем дерева. Функция создания нового узла NEWNODE (строка 9) вынесена за пределы транзакционной секции, это позволяет сократить её размер, а следовательно, уменьшить количество конфликтов между разными транзакциями.

V. РЕЗУЛЬТАТЫ ЭКСПЕРИМЕНТОВ

Эксперименты проводились на вычислительной системе, оборудованной 4-ядерным процессором Intel i5-3470 (отсутствует поддержка аппаратной транзакционной памяти) и объёмом кэш-памяти L1 – 32 КБ, L2 – 256 КБ, L3 – 6 МБ. Размер оперативной памяти – 8 ГБ. Используемое программное обеспечение: Linux Mint 18.1, компилятор GCC 5.4.0.

Тест представлял собой запуск p потоков, выполняющих операции добавления, удаления и поиска элемента. Вид операции выбирался случайным образом. Количество операций поиска – 40%, вставки элемента – 30%, удаления – 30%. Число потоков p в тестовых программах варьировалось от 2 до 32. На первом этапе моделирование проводилось для $p = 1, \ldots, 4$ потоков, что не превышает количество ядер процессора. На втором этапе количество потоков доходило до 32. В данном эксперименте использовалось четыре метода выполнения транзакций, реализованных в компиляторе GCC:

- Метод глобальной блокировки (gl_wt) – потоки выполняют транзакции параллельно, глобальная блокировка возникает, когда потоки начинают изменять один участок памяти.
- Метод множественной блокировки (ml_wt) – потоки выполняют транзакции параллельно, пока не выполняют запись в один участок памяти; множественная блокировка транзакций возникает, когда потоки выполняют запись в один участок памяти.
- Последовательные методы (serial, serialirr) – в serial все транзакции выполняются последовательно. В serialirr чтение идёт параллельно, а при появлении операции записи транзакция переходит в irrevocable режим, предотвращая несанкционированные записи.

Также для экспериментов использовались реализации структур данных (красно-черное дерево и хеш-таблица) на основе крупнозернистых (coarse-grained) и мелкозернистых (fine-grained) блокировок (только для хеш-таблицы).

В качестве показателя эффективности использовалась пропускная способность $b = N / t$, где N – количество выполненных операций, а t – время выполнения всех операций.

Результаты моделирования хеш-таблицы показаны на рис. 5 и 6.

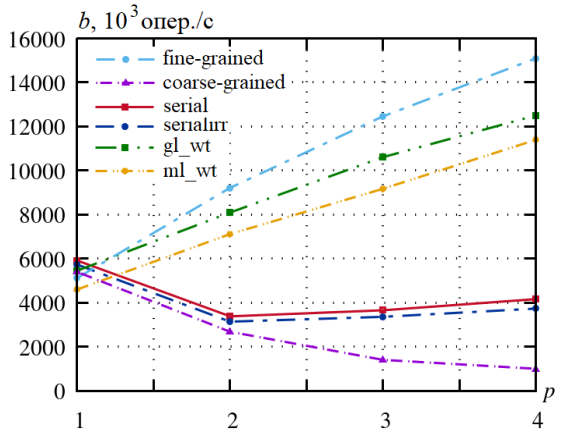

Рис. 5. Пропускная способность хеш-таблицы при выполнении случайных операций для 1–4 потоков

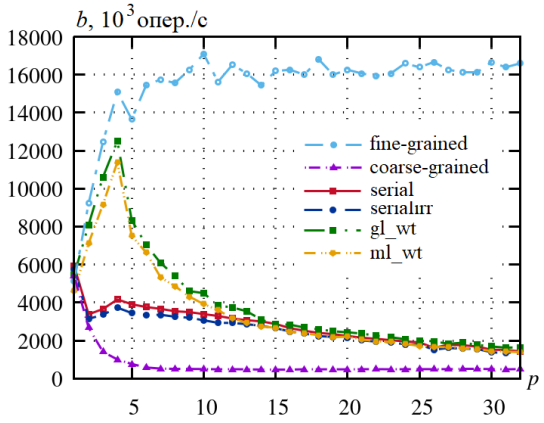

Рис. 6. Пропускная способность хеш-таблицы при выполнении случайных операций для 1–32 потоков

Хеш-таблица на основе транзакционной памяти обеспечивает большую пропускную способность, по сравнению с реализацией на основе крупнозернистых блокировок, при любом количестве потоков (рис. 6). Методы gl_wt и ml_wt демонстрируют рост пропускной

способности с увеличением числа потоков и практически не уступают в производительности реализации на основе мелкозернистых блокировок при условии, что число потоков не превышает число процессорных ядер (рис. 5). В случае, если число потоков больше 16, эффективность всех методов выполнения транзакций сопоставима. При количестве потоков, превышающем количество процессорных ядер, любой из четырёх представленных методов выполнения транзакций уступает fine-grained алгоритмам.

На рис. 7 и 8 представлены результаты моделирования потокобезопасного красно-чёрного дерева.

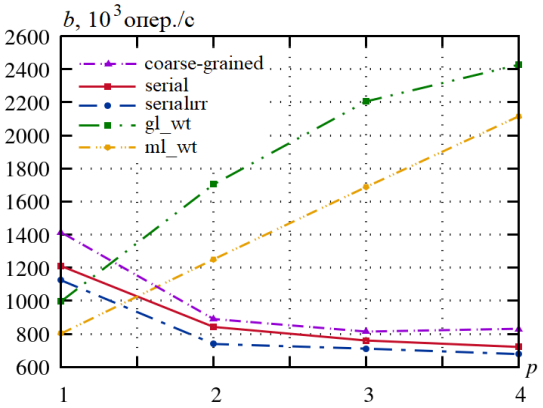

Рис. 7. Пропускная способность красно-чёрного дерева при выполнении случайных операций для 1–4 потоков

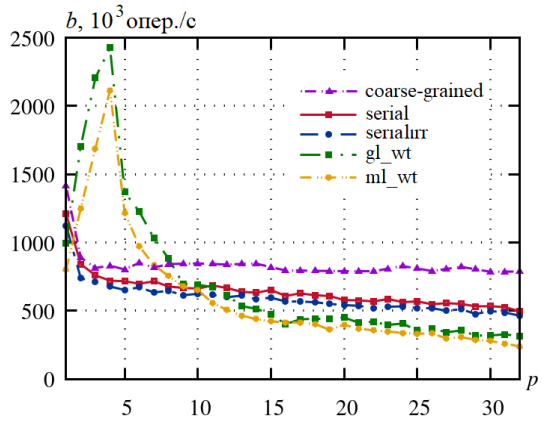

Рис. 8. Пропускная способность красно-чёрного дерева при выполнении случайных операций для 1–32 потоков

Пропускная способность красно-чёрного дерева на основе транзакционной памяти выше реализации на основе блокировок только при количестве потоков меньшем или равным 8 и для методов gl_wt и ml_wt выполнения транзакций (рис. 8). Последовательные методы выполнения транзакций (serialirr и serial) уступают блокировкам при любом количестве потоков.

VI. Заключение

В работе с помощью транзакционной памяти реализованы потокобезопасные красно-чёрное дерево и хеш-таблица на основе метода разрешения коллизий Hopscotch hashing.

Моделирование показало, что эффективность хеш-таблицы на основе транзакционной памяти превосходит аналогичные реализации на основе крупнозернистых блокировок. Красно-черное дерево уступает по пропускной способности аналогу на основе мелкозернистых блокировок при реализации с количеством потоков, превышающим количество ядер, это связано с множеством конфликтов между транзакциями и, как следствие, их множественных отмен. Рекомендуемым методом выполнения транзакций для хеш-таблицы является метод глобальных блокировок (gl_wt). Большую пропускную способность красно-черного дерева обеспечивают методы выполнения транзакций gl_wt (при количестве потоков, не превышающем число процессорных ядер) и serial (в случае, если количество потоков превосходит количество процессорных ядер).

Список литературы

[1] Кулагин И.И., Курносов М.Г. Оптимизация обнаружения конфликтов в параллельных программах с транзакционной памятью // Вестник ЮУрГУ. Серия: Вычислительная математика и информатика.-2016. Т. 5; № 4. С.46-60.

[2] Kwiatkowski J. Evaluation of Parallel Programs by Measurement of Its Granularity // Parallel Processing and Applied Mathematics, Naleczow, Poland, September 2001. pp. 145–153.

[3] Fraser K. Practical lock freedom. PhD thesis, Cambridge University, 2003. 116 p.

[4] Shavit N., Touitou D. Software Transactional Memory // In PODC'95: Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing, New York, NY, USA, Aug. 1995. ACM, pp. 204–213.

[5] Spear M., Marathe V., Scherer W., Scott M. (2006) Conflict Detection and Validation Strategies for Software Transactional Memory. In: Dolev S. (eds) Distributed Computing. DISC 2006. //Lecture Notes in Computer Science, vol 4167. Springer, Berlin, Hedelberg Symposium on Parallelism in Algorithms and Architectures, June 2008, P. 275–284.

[6] Herlihy M., Shavit N., Tzafrir M. Hopscotch Hashing // Proceedings of the 22nd international symposium on Distributed Computing. Arcachon, France: Springer-Verlag. pp. 350–364.

[7] Pagh R., Rodler F.F. Cuckoo hashing // Journal of Algorithms. 2004. №51, С. 122–144.

[8] Knuth D.E. The art of computer programming, volume 1 (3rd ed.): fundamental algorithms. Addison Wesley Longman Pulishing Co., Inc., Redwood City, CA, USA, 1997.

[9] Introduction to Algorithms. / Cormen T., Leiserson C., Rivest R., Stein C.; MIT Press, 2001.