Тема работы

Потокобезопасные структуры данных для вычислительных систем с общей памятью на основе программной транзакционной памяти

Состав коллектива

1. Пазников Алексей Александрович, к.т.н., с.н.с., доцент СПбГЭТУ «ЛЭТИ», руководитель
2. Смирнов Вадим Александрович, магистрант, СПбГЭТУ «ЛЭТИ», исполнитель
3. Омельниченко, Артур Романович, магистрант, СПбГЭТУ «ЛЭТИ», исполнитель

Информация о гранте

РФФИ, проект № 19-07-00784 «Разработка методов, алгоритмов и программного обеспечения масштабируемой синхронизации для многопроцессорных вычислительных систем», руководитель – Пазников А.А., 2019-2021

Научное содержание работы

1. Постановка задачи

Ставится задача создания масштабируемых потокобезопасных структур данных на базе программной транзакционной памяти (transactional memory). Необходимо создать масштабируемые потокобезопасные линейные списки (очереди, стеки), пулы и ассоциативные массивы (хеш-таблицы, деревья поиска). Требуется разработать алгоритмы и программные средства, реализующие потокобезопасные структуры данных, с целью минимизации времени выполнения операций (обеспечение максимальной пропускной способности). Необходимо провести анализ эффективности современных реализаций программной транзакционной памяти и дать рекомендации по оптимизации алгоритмов реализации транзакционной памяти.

2. Современное состояние проблемы

На сегодняшний день транзакционная память является одним из наиболее перспективных механизмов синхронизации, её использование позволяет выполнять не конфликтующие между собой операции параллельно. Транзакционная память позволяет выделить группы инструкций в атомарные транзакции – конечные последовательности операций транзакционного чтения/записи памяти [1]. 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). Важным свойством транзакционной памяти является линеаризуемость выполнения транзакций: ряд успешно завершённых транзакция эквивалентен некоторому последовательному их выполнению. Транзакции обладают качествами атомарности, согласованности, изолированности, устойчивости (atomicity, consistency, isolation, durability – ACID) [2].

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 аспектами реализации, 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 [3]. 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 [4-6]. Использование журнала изменений замедляет операцию фиксации, но существенно упрощает процедуры ее отмены и восстановления.

Ранняя политика обновления объектов в памяти (eager version management) предполагает, что все изменения объектов сразу записываются в память [3, 7-9]. В журнале отката (undo log) фиксируются все выполненные операции с памятью. Он используется для восстановления оригинального состояния модифицируемых участков памяти в случае возникновения конфликта. Эта политика характеризуется быстрым выполнением операции фиксации транзакции, но медленным выполнением процедуры ее отмены.

Момент времени, когда инициируется алгоритм обнаружения конфликта, определяется стратегией обнаружения конфликтов. При отложенной стратегии (lazy conflict detection) процедура обнаружения конфликтов запускается на этапе фиксации транзакции [3, 4, 10]. Недостатком этой стратегии является то, что временной интервал между возникновением конфликта и его обнаружением может быть достаточно большим.

Пессимистичная стратегия обнаружения конфликтов (eager conflict detection) запускает алгоритм их обнаружения при каждой операции обращения к памяти [8, 9, 11]. Такой подход позволяет избежать недостатков отложенной стратегии, но может привести к значительным накладным расходам.

1. Shavit N., Touitou D. Software transactional memory // Distributed Computing. 1997. Vol. 10. no. 2. pp. 99-116.
2. Larus, J., Kozyrakis, C. Transactional memory, Communications of the ACM, v.51 n.7, July 2008
3. Spear, M., Marathe, V., Scherer, W., Scott M. (2006) Conflict Detection and Validation Strategies for Software Transaction-al Memory. In: Dolev S. (eds) Distributed Computing. DISC 2006. Lecture Notes in Computer Science, vol 4167. Springer, Berlin, Heidelberg Symposium on Parallelism in Algorithms and Architectures, June 2008, P. 275–284.
4. Rochester Software Transactional Memory Runtime. Project web site [Electronic resource]. — URL: www.cs.rochester.edu/research/synchronization/rstm/
5. Spear, Michael F. RingSTM: scalable transactions with a single atomic instruction [Text] / Michael F. Spear, Maged M. Michael, Christoph von Praun // In SPAA ’08: Proc. 20th Annual Symposium on Parallelism in Algorithms and Architectures, June 2008. — 2008. — P. 275–284.
6. A comprehensive strategy for contention management in software transactional memory [Text] / Michael F. Spear, Luke Dalessandro, Virendra J. Marathe, Michael L. Scott. // In PPoPP ’09: Proc. 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, February 2009. — [S. l. : s. n.], 2009. — P. 141–150.
7. LogTM: Log-based transactional memory [Text] / Kevin E. Moore, Jayaram Bobba, Michelle J. Moravan [et al.] // In HPCA ’06: Proc. 12th International Symposium on High-Performance Computer Architecture, February 2006. — [S. l. : s. n.], 2006. — P. 254–265.
8. Riegel, Torvald. A Lazy Snapshot Algorithm with Eager Validation [Text] / Torvald Riegel, Pascal Felber, Christof Fetzer // 20th International Symposium on Distributed Computing (DISC). — 2006.

9. Time-based Software Transactional Memory [Text] / Pascal Felber, Christof Fetzer, Patrick Marlier, Torvald Riegel // IEEE Transactions on Parallel and Distributed Systems. — Vol. 21(12). — 2010. — P. 1793–1807.
10. A comprehensive strategy for contention management in software transactional memory [Text] / Michael F. Spear, Luke Dalessandro, Virendra J. Marathe, Michael L. Scott. // In PPoPP ’09: Proc. 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, February 2009. — 2009. — P. 141–150.
11. Dice, Dave. Transactional locking II [Text] / Dave Dice, Ori Shalev, Nir Shavit // In DISC ’06: Proc. 20th International Symposium on Distributed Computing, September 2006. — Vol. 4167. — 2006. — P. 194–208.

3. Подробное описание работы, включая используемые алгоритмы

В проекте предложены алгоритмы реализации потокобезопасных ассоциативных массивов (красно-чёрное дерево, дерево ван Эмде Боаса, хеш-таблица с открытой адресацией на основе метода Hopscotch hashing разрешения коллизий) с использованием программной транзакционной памяти (software transactional memory), реализующей спекулятивное выполнение критических секций. Представлен анализ эффективности ассоциативных массивов в зависимости от числа задействованных потоков, приведено сравнение с аналогичными структурами данных на основе крупнозернистых и мелкозернистых блокировок, сформулированы рекомендации по выбору алгоритмов выполнения транзакций. Представлены различные методы выполнения транзакций, реализованные в компиляторе GCC 4.8.0. Моделирование показало, что структуры данных на основе транзакционной памяти превосходит аналогичные реализации на основе крупнозернистых блокировок, но уступают реализациям на основе мелкозернистых блокировок.

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

Хеш-таблицы с открытой адресацией характеризуются хорошей локальностью кэш-памяти. Hopscotch hashing объединяет в себе преимущества трёх подходов: Cuckoo hashing, метод цепочек и метод линейного хеширования. 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 окрестности каждой ячейки массива.

Пример выполнения операции вставки представлен на рис. 1.

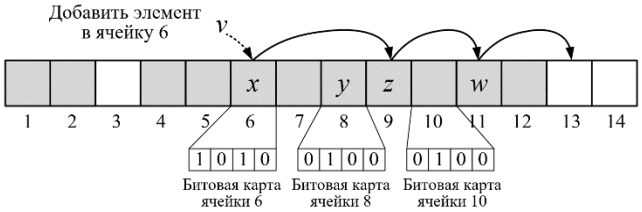

Рис. 1. Пример выполнения операции вставки

Основная идея Hopscotch hashing заключается в использовании свойства пространственной локальности кэш-памяти. Искомый элемент находится в окрестности ячейки, на которую указывает хеш-функция. На рис. 2а 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
```

а

```
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
```

б

Рис. 2. Функция добавления в хеш-таблицу

На рис. 2б 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.

Также предлагается реализация потокобезопасного красно-черного дерева на основе транзакционной памяти. Красно-черные деревья обеспечивают логарифмический рост высоты дерева в зависимости от числа узлов, что позволяет выполнять основные операции за время $O(\log_2 n)$. На рис. 3 представлена функция добавления элемента в красно-чёрное дерево. Критическая секция выделена в транзакцию (строки 4-11), это позволяет потокам выполнять добавление элементов, не нарушая целостность данных и баланс дерева.

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.

Кроме того, в данном проекте также предложена потокобезопасная реализация дерева ван Эмде Боаса на основе транзационной памяти. Дерево ван Эмде Боаса – это дерево поиска для хранения целочисленных m-битных ключей. Основные операции (Insert, Delete, Lookup, Min, Max) выполняются за время $O(\log_2(\log_2 U))$, где U – это размер универсума (множество всех

возможных элементов), что асимптотически лучше, чем логарифмическая сложность в сбалансированных бинарных деревьях поиска.

```
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. 3. Функция добавления элемента в красно-чёрное дерево

В ходе экспериментов 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. 4.

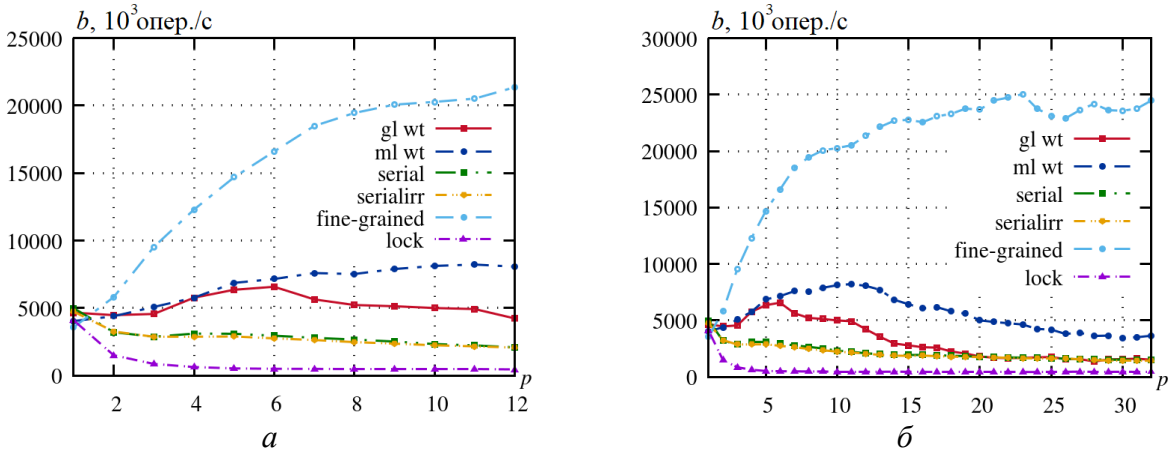

Рис. 4. Результаты анализа эффективности хеш-таблицы

$а$ – $p = 1, \ldots, 12$, $б$ – $p = 1, \ldots, 32$

Хеш-таблица на основе транзакционной памяти обеспечивает большую пропускную способность, по сравнению с реализацией на основе крупнозернистых блокировок, при любом количестве потоков (рис. 4.5б). 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. 4). 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. 5 представлены результаты моделирования потокобезопасного красно-чёрного дерева.

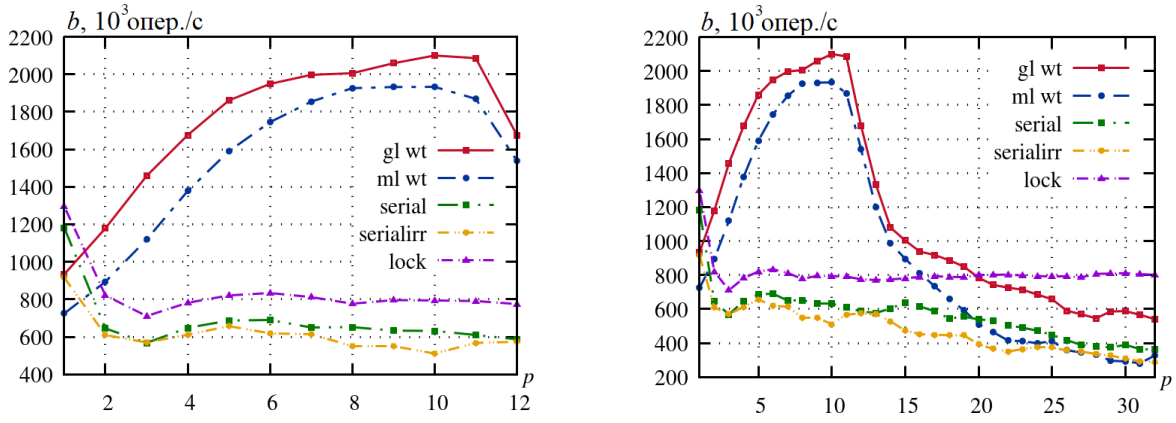

Рис. 5. Результаты анализа эффективности красно-черного дерева
а – $p = 1, \ldots, 12$, *б* – $p = 1, \ldots, 32$

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. Последовательные методы выполнения транзакций (serialirr и serial) уступают блокировкам при любом количестве потоков.

Результаты экспериментов для дерева ван Эмде Боаса аналогичны при использовании всех методов кроме gl wt. Эффективность метода gl wt существенно не зависит от числа потоков, при условии, что количество потоков не превышает число процессорных ядер (рис. 6).

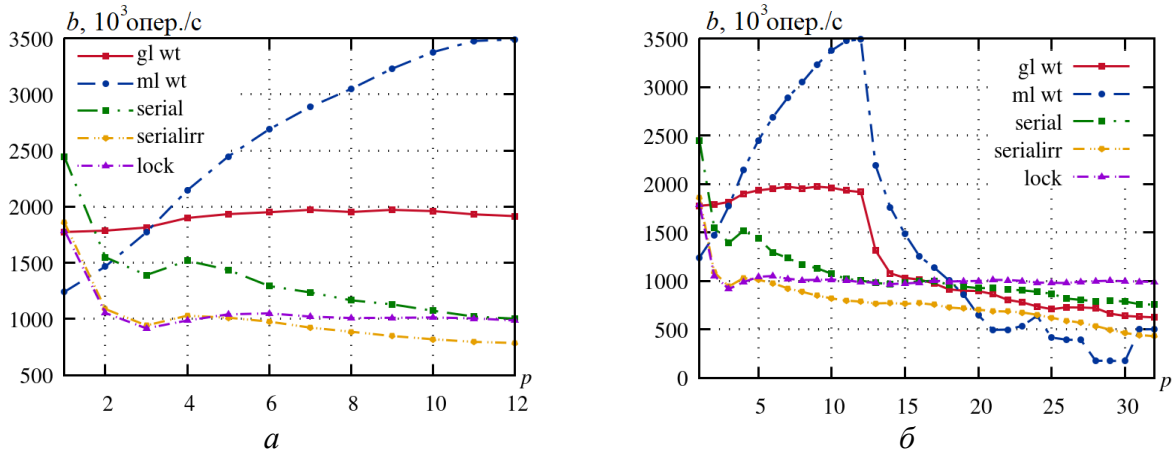

Рис. 6. Результаты анализа эффективности дерева ван Эмде Боаса
а – $p = 1, \ldots, 12$, *б* – $p = 1, \ldots, 32$

4. Эффект от использования кластера в достижении целей работы

Ресурсы кластерной вычислительной системы использовались в натурном моделировании созданных структур данных. Эксперименты проводились на многоядерных вычислительных узлах (ВС с общей памятью). Использование кластера позволило проанализировать эффективность структур данных в зависимости от выбора алгоритма реализации транзакционной памяти и количества потоков (используемых процессорных ядер).

Перечень публикаций, содержащих результаты работы

1. Paznikov A.A., Smirnov V.A., Omelnichenko A.R. Towards Efficient Implementation of Concurrent Hash Tables and Search Trees Based on Software Transactional Memory // Proc. Of the 2019 International Multi-Conference on Industrial Engineering and Modern Technologies (FarEastCon), 2019. – pp. 1-5. DOI: 10.1109/FarEastCon.2019.8934131
2. Смирнов В. А., Омельниченко А. Р., Пазников А. А. Алгоритмы реализации потокобезопасных ассоциативных массивов на основе транзакционной памяти // Известия СПбГЭТУ «ЛЭТИ». – 2018. – № 1. – С. 12-18.
3. Смирнов В. А., Омельниченко А. Р., Пазников А. А. Моделирование и оптимизация выполнения транзакционных секций на примере потокобезопасных хеш-таблиц и деревьев поиска // Международная конференция по мягким вычислениям и измерениям. – 2018 – Т.1. – С. 578-582.