УДК 004.272

Исследование эффективности потокобезопасного красно-чёрного дерева на основе транзакционной памяти

Т. М. Токарь, А. А. Пазников[1]

В работе выполнено исследование эффективности реализации потокобезопасного красно-чёрного дерева на основе программной транзакционной памяти (software transactional memory, STM). Суть подхода заключается в выделении участков кода, в которых реализуется доступ к разделяемым областям памяти, в транзакционные блоки. Приведено описание потокобезопасного красно-чёрного дерева и результаты натурных экспериментов на вычислительном кластере с использование реализации программной транзакционной памяти в компиляторе GCC 5.3.0. Сформулированы рекомендации по выбору транзакционных секций и алгоритмов реализации транзакций.

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

1. Введение

Одна из ключевых проблем, которые возникают при разработке многопоточных программ для вычислительных систем с общей памятью [1], – это гонки данных (data race), возникающие при совместном доступе параллельных потоков к общими ресурсами (память, сетевая, файловая система и т.д.). На сегодняшний день основными методами решения данной проблемы являются:

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

Транзакционная память относится к одним из наиболее перспективных средств синхронизации потоков и предоставляет механизм, позволяющий частям программы выполняться независимо от других параллельно выполняемых задач. Данный метод предполагается организацию в программе транзакционных блоков– участков кода, в которых выполняется доступ к разделяемым данным. Внутри транзакционных блоков потоки могут обращаться к глобальным структурам данных, не нарушая их целостности, создавая иллюзию последовательного выполнения потоков.

Транзакционный блок (transactional block) в процессе своего выполнения называется транзакцией (transaction). В общем случае транзакция определяется как группа последовательных операций, которая предоставляет собой логическую единицу работы с данными. Транзакция гарантирует, что все запросы к разделяемым данным приведут к тому же результату, как если бы они выполнялись последовательно в некотором порядке. Данный механизм

[1] Работа выполнена при поддержке РФФИ (гранты № 15-07-02693, 15-37-20113, 15-07-00653, 16-07-00712, 15-07-00048)

базируется на двух основных принципах: управления версиями данными и обнаружения конфликтов. Если транзакция, выполняемая в данном потоке, выполняется успешно (т. е. никакая другая транзакция не использовала общие с данной транзакцией ресурсы), то она фиксируется (commit), и все изменения, производимые с данными считаются видимыми другим потокам. Иначе – если произошёл конфликт доступа к ресурсам – производится отмена (cancel) транзакции, при этом изменения в данных восстанавливаются, и транзакция отмечается для повторного выполнения. При этом политика выбора откатываемой транзакции и условия повторного выполнения транзакции влияют на производительность всей систем в целом. В общем случае транзакционная память обеспечивает атомарность (atomicity) и изолированность (isolation) параллельно выполняемых задач; при этом согласованность (consistency) и долговечность (durability) не гарантируются.

Отказ от блокировок и использование механизмов транзакционной памяти (transactional memory) при написании многопоточных приложений имеет следующие преимущества:

1. Облегчает процесс разработки многопоточного программного обеспечения.
2. Обеспечивает прозрачное для пользователя решение проблем блокировок (взаимные блокировки, «живые» блокировки, инверсия приоритетов и т. п.).
3. Повышение эффективности использования вычислительных ресурсов.
4. Возможность простой композиции программного кода.

Также данная технология не исключена отрицательных сторон, к ним можно отнести:

1. При неправильном использовании возможно снижение производительности и некорректная работа.
2. Ограниченность применения: в транзакции нельзя выполнять операции, действие которых невозможно отменить.
3. Сложность отладки – поставить точку останова (breakpoint) при отладке программы внутри транзакции невозможно.

Реализация систем транзакционной памяти может быть чисто программной (Software Transactional Memory, STM), аппаратной (Hardware Transactional Memory, HTM) и гибридной, сочетающей возможности STM и HTM. В данной работе рассматривается программная транзакционная память.

2. Программная транзакционная память

Международным комитетом ISO по стандартизации языка C++ в рамках рабочей группы WG21 ведутся работы по внедрению транзакционной памяти в стандарт языка. Окончательное внедрение планируется в стандарте C++17. На сегодняшний день предложен черновой вариант спецификации поддержки транзакционной памяти в C++. Она реализована в компиляторе GCC начиная с версии 4.8 и представлена ключевыми словами __transaction_atomic, __transaction_relaxed для создания транзакционных секций и __transaction_cancel для принудительной отмены транзакции. В компиляторе в настоящий момент реализованы методы выполнения транзакций ml, gl, serial, serialirr, serialirr_onwrite, htm [5].

Для выполнения транзакционных секций runtime-системой создаются транзакции. Транзакция – это конечная последовательность операций транзакционного чтения/записи памяти. 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 [4].

Инструкции транзакций выполняются потоками параллельно. После завершения выполнения транзакция может быть либо зафиксирована (commit), либо отменена (cancel). Фиксация транзакции подразумевает, что все сделанные в рамках нее изменения памяти становятся

необратимыми. При отмене транзакции ее выполнение прерывается, а состояние всех модифицированных областей памяти восстанавливается в исходное с последующим перезапуском транзакции (откат транзакции, rollback) [2].

3. Потокобезопасные структуры на основе транзакционной памяти

Применение транзакционной памяти упрощает разработку многопоточных структур данных за счёт более простого использования транзакций, так как в данной технологии нет необходимости определять области критических секций, и отсутствует возможность возникновения тупиковых ситуаций (deadlocks, livelocks). Также данная технология позволяет достичь большей масштабируемости, так как выполнение транзакций происходит без блокирования потоков [3].

В работе рассматривается создание потокобезопасной структуры данных «Красно-Черное дерево» (Red-Black Tree, RBT) на основе транзакционной памяти. Для красно-чёрного дерева потоки могут выполнять операции добавления и удаления элементов без возникновения состояния гонки, то есть каждый поток модифицирует свой независящий от других потоков участок памяти. Тем не менее в некоторых случаях потоки могут конкурировать между собой при обращении к одному участку памяти. Такие операции, как поиск элементов, поиск максимального (минимального) элемента в RBT-TM потоки выполняют независимо друг от друга и не конфликтуют друг с другом, так как эти операции не вносят изменения в структуру данных и могут быть выполнены параллельно. На рис. 1 представлено RBT-TM с *n* потоками, выполняющими операции над структурой данных.

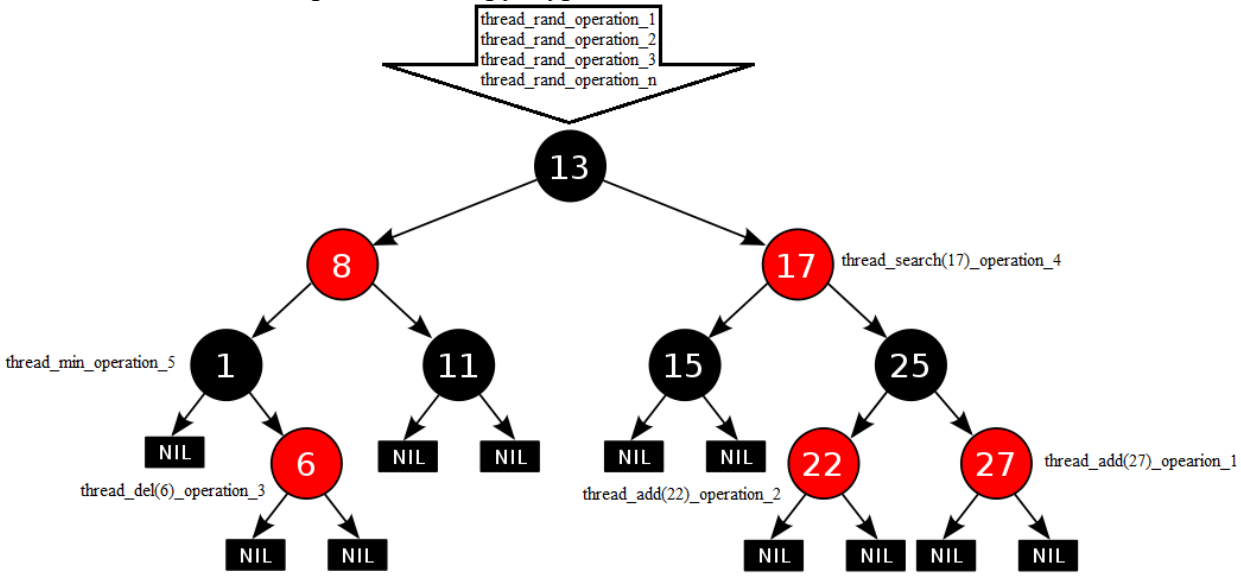

Рис. 1. Графическая структура RBT-TM и выполняемые операции потоками над структурой

В ходе выполнения потоками операций над структурой данных предполагается, что балансировка дерева выполняется за константное время $O(1)$ и не мешает потокам выполнять случайные операции в структуре, при этом сохраняя баланс дерева.

Рассмотрим пример операции добавления в RBT-TM выполняемая потоками (рис. 2). В данной операции транзакционный блок __transaction_atomic{. . .} обеспечивает выполнение данной операции атомарно потоками в транзакции в случае успешного добавления данных в дерево. В случае возникновения коллизий с конкурентными потоками, транзакция отменяется и выполняется повторно, пока не будет завершена удачно, то есть пока данные не будут добавлены в структуру.

```
int move(list *from, list *to) {
    __transaction_atomic {
        node *n = pop(from);
        push(to, n);
    }
}
```

Рис. 2. Функция добавления в RBT-TM.

3. Оптимизация потокобезопасных красно-чёрных деревьев на основе транзакционной памяти

Красно-черное дерево представляет собой бинарное дерево поиска с одним дополнительным битом цвета в каждом узле. Цвет может быть либо красным (RED), либо черным (BLACK). В соответствии с накладываемыми на узлы дерева ограничениями ни один простой путь от корня в красно-черном дереве не отличается по длине от другого более чем в два раз, так что красно-чёрные деревья является приближено сбалансированным.

Бинарное дерево поиска является красно-чёрным деревом, если оно удовлетворяет следующим свойствам:

– каждый узел является либо красным, либо чёрным;

– корень дерева является черным узлом;

– каждый лист дерева является черным узлом;

– если узел красный, то оба его дочерних узла черные.

Так как красно-чёрное дерево является сбалансированным, все операции над ним выполняются за время $T = O(\log n)$ в среднем и худшем случае.

Рассмотрим алгоритмы операций красно-черного дерева с применением транзакционной памяти.

```
bool Insert(int value,node_st **root) __transactional_atomic {
        node_st *node=*root;
        if(!node)
                *root=NewItem(value);
        else {
                if(value==node->value) return true;
                if(Insert(value,value<node->value?&node->p1:&node->p2)) return true;
                BalanceInsert(root);
        }
        return false;
}
```

Рис 3. Функция добавления в красно-чёрное дерево

Функция добавления в красно-чёрное дерево (рис. 3) выделена в транзакцию, что позволяет потокам выполнять добавление элементов, не нарушая целостности данных и баланс дерева. В случаи неудачного добавления данных потоком в структуру функция (транзакция) отменится и повторится.

```
bool Remove(node_st **root,int value) __transactional_atomic {
        node_st *t,*node=*root;
        if(!node) return false;
        if(node->value<value) {
                if(Remove(&node->p2,value))     return BalanceRemove2(root);
        } else if(node->value>value) {
                if(Remove(&node->p1,value))     return BalanceRemove1(root);
        } else {
                bool res;
                if(!node->p2) {
                        *root=node->p1;
                        res=!node->red;
                        DelItem(node);
                        return res;
                }
                res=GetMin(&node->p2,root);
                t=*root;
                t->red=node->red;
                t->p1=node->p1;
                t->p2=node->p2;
                DelItem(node);
                if(res) res=BalanceRemove2(root);
                return res;
        }
        return 0;
}
```

Рис 4. Функция удаления из красно-чёрного дерева

Аналогичным образом, функция удаления (рис. 4) не нарушает целостность структуры и баланс дерева. Данная функция также выделена в транзакцию.

Функции поиска, поиск минимального (максимального) элемента не изменяют структуру и данные и могут выполнятся потоками независимо друг от друга.

В библиотеке GCC libitm представлено 6 методов выполнения транзакций:

ml_wt – метод выполнения транзакций на основе множественных блокировок (multiple lock);

gl_wt – метод выполнения транзакций на основе глобальной блокировки (global lock);

serial, serialirr, serialirr_onwrite – методы последовательного выполнения транзакций;

htm – аппаратный, последовательный метод выполнения транзакций; отличается от serial тем, что использует аппаратное выполнение транзакций.

4. Результаты экспериментов

Эксперименты проводились на узле вычислительного кластера Центра параллельных вычислительных технологий СибГУТИ со следующей конфигурацией:

Процессор: 2 x Intel Xeon E5420 (2,5 ГГц, 4 ядра).

Оперативная память: 8 GB (4 x 2GB PC-5300).

Операционная система: GNU/Linux Fedora 20.

Компилятор: GCC 5.3.0.

Количество потоков p варьировалось от 1 до 8.

Значение переменной ITM_DEFAULT_METHOD среды окружения, определяющей используемый метод выполнения транзакций: gl_wt, ml_wt, serial, serialirr, serialirr_onwrite, htm.

Метод глобальной блокировки (global lock, gl_wt) – потоки выполняют транзакции параллельно, глобальная блокировка возникает, когда потоки начинают изменять один участок памяти. В данном методе блокируется структура данных.

Метод множественной блокировки (multiple lock, ml_wt) транзакций – потоки выполняют транзакции параллельно, пока не редактируют один участок памяти, множественная блокировка транзакций возникает, когда потоки редактируют один участок памяти. В данном методе блокируются потоки.

Последовательный метод выполнения (serial, serialirr, serialirr_onwrite) – потоки выполняют транзакции последовательно друг за другом.

На рис. 5 представлен график зависимости пропускной способности потокобезопасного красно-чёрного дерева (thread-safe Red-Black tree) на основе программной транзакционной памяти (software transactional memory) от количества потоков. Применялись различные методы выполнения транзакций (transactional method).

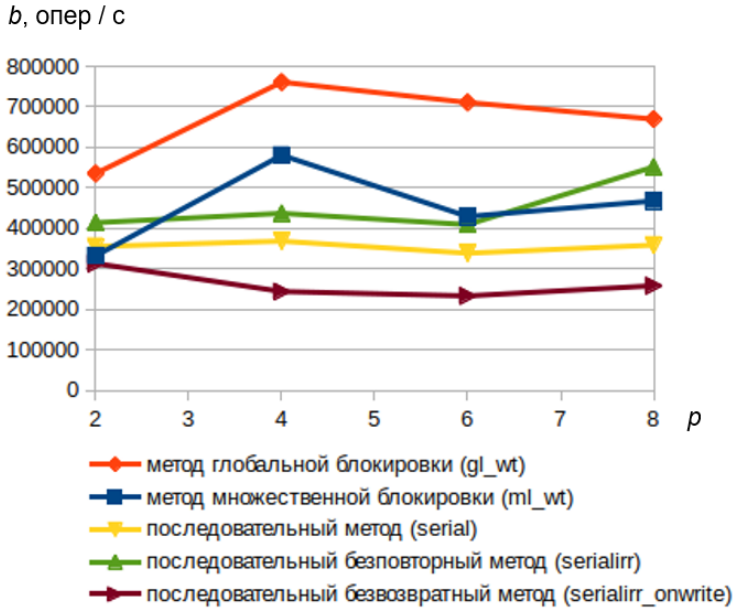

Рис 5. Тест производительности методов выполнения транзакций (transactional method)

Пропускная способность красно-чёрного дерева зависит от выбранного метода выполнения транзакций. Результаты тестов показывают, что метод глобальной блокировки (global lock, gl_wt) структуры данных обеспечивает большую пропускную способность, так как этот метод позволяет выполнять транзакции (операции) параллельно над структурой. Блокировка структуры выполняется при внесении изменения в один участок памяти, что происходит не так часто.

Литература

1. *Хорошевский В. Г.* Распределенные вычислительные системы с программируемой структурой. Вестник СибГУТИ. 2010. №2. 10 с.
2. *Кулагин И. И. Курносов М. Г.* Инструментация и оптимизация выполнения транзакционных секций многопоточных программ. Труды ИСП РАН. № 27. 2015.
3. *Herlihy M., Moss J. E. B.* Transactional memory: Architectural support for lock-free data structures. ACM. 1993. № 2. 289-300 с.

4. *Shavit N., Touitou D.* Software transactional memory. 1997. № 2. 99-116 с.
5. *Felber P., Fetzer C., Riegel T.* Dynamic performance tuning of word-based software transactional memory. ACM. 2008. 237-246 с.
6. *Riegel T., Fetzer C., Felber P.* Time-based transactional memory with scalable time bases. ACM. 2007. 221-228 с.
7. *T. Riegel.* Software Transactional Memory Building Blocks. PhD thesis. Technischen Universitat Dresden, geborenam 1.3.1979 in Dresden. 2013.

Статья поступила в редакцию 14.03.2016

Пазников Алексей Александрович
к.т.н., доцент кафедры вычислительных систем СибГУТИ (630102, Новосибирск, ул. Кирова, 86) тел. (383) 2-698-286, e-mail: apaznikov@gmail.com.

Токарь Тимур Михайлович
студент 3 курса факультета информатики и вычислительной техники СибГУТИ (630102, Новосибирск, ул. Кирова, 86) тел. +7(913)-371-53-79, e-mail: tokar.t.m@gmail.com

Research of efficiency of a thread-safe red-black tree-based transactional memory

T. Tokar, A. Paznikov

This paper represents the study of the effectiveness of the implementation of thread-safe red-black tree-based software transactional memory (software transactional memory, STM). The essence of the approach is to allocate parts of the code, which is executed access to shared memory regions in transaction blocks. The description of a thread-safe red-black tree and the results of field experiments on a computing cluster using a software implementation of transactional memory compiler GCC 5.3.0. Recommendations on the choice of transactional algorithms for the implementation of sections are represented.

Keywords: transactional memory, multi-threaded programming, synchronization of multi-threaded programs thread-safe data structures.