Инструментация и оптимизация выполнения транзакционных секций многопоточных программ

[1]И.И. Кулагин <ivan.i.kulagin@gmail.com>
[2]М.Г. Курносов <mkurnosov@gmail.com>
[1] Сибирский государственный университет телекоммуникаций и информатики, 630102, Россия, г. Новосибирск, ул. Кирова, дом 86.
[2]Санкт-Петербургский государственный электротехнический университет «ЛЭТИ», 197376, Россия, г. Санкт-Петербург, ул. Профессора Попова, дом 5.

Аннотация. В работе выполнено исследование эффективности реализации программной транзакционной памяти (software transactional memory) в компиляторе GCC, предложен метод инструментации параллельных программ, использующих транзакционную память, для осуществления задач профилирования, а также предложен подход к сокращению числа ложных конфликтов, возникающих при выполнении транзакционных секций. Суть подхода заключается в варьировании параметров реализации транзакционной памяти в runtime-библиотеке компилятора GCC по результатам предварительного профилирования программы (profile-guided optimization). Предложенный метод инструментации позволяет оптимизировать динамические характеристики выполнения транзакционных секций. Эффективность подхода к сокращению числа ложных конфликтов исследована на тестовых многопоточных программах из пакета STAMP.

Ключевые слова: программная транзакционная память, инструментация, оптимизация по результатам предварительного профилирования, многопоточное программирование, компиляторы.

DOI: 10.15514/ISPRAS-2015-27(6)-9

Для цитирования: Кулагин И.И., Курносов М.Г. Инструментация и оптимизация выполнения транзакционных секций многопоточных программ. Труды ИСП РАН, том 27, вып. 6, 2015 г., стр. 135-150. DOI: 10.15514/ISPRAS-2015-27(6)-9.

1. Введение

Синхронизация доступа к разделяемым ресурсам является одной из важных и сложных задач при разработке параллельных алгоритмов для многопоточных программ. Используя примитивы синхронизации (семафоры, мьютексы и др.) программист должен обеспечить не только корректность программы – отсутствие взаимных блокировок (deadlock, livelock) 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 создавать в коде программы критические секции, выполнение которых возможно только одним потоком в каждый момент времени [1]. Такие примитивы синхронизации позволяют *защитить участок кода программы*, одновременно выполняющийся множеством потоков.

Анализ многопоточных программ показывает, что при одновременном выполнении потоками одной критической секций в ней может происходить обращения по разным адресам памяти и, следовательно, возникновение состояния гонки данных маловероятно. Например, такая ситуация наблюдается если в критическую секцию помещен код добавления элемента в хеш-таблицу (рис. 1). Если все потоки в один момент времени обратятся к функции для добавления нового элемента с хеш-кодом *i*, то с большой долей вероятности ключи *key* будут иметь разные хеш-коды и, как следствие, каждый поток будет выполнять добавления нового узла в отдельный связный список *h*[*i*]. Здесь блокировка мьютексом всей функции является избыточной и неэффективной.

```
function hashtable_add(h, key, value)
    lock_acquire()
    i = hash(key)
    list_add_front(h[i], key, value)
    lock_release()
end function
```

Рис. 1. Добавление пары (key, value) в хеш-таблицу h.

Активно развиваются два альтернативных подхода к созданию потокобезопасных и масштабируемых многопоточных программ – это *неблокирующие алгоритмы и структуры данных* (lock-free data structures) [2] и транзакционная память (ТП, transactional memory) [3, 4].

Использование неблокирующих структур данных, как правило, требует глубокой переработки многопоточных программ и совместно используемых потоками объектов в памяти [2].

Менее трудоемким и прозрачным для программиста видится использование технологии транзакционной памяти, основная идея которой заключается в *защите от конкурентного доступа области памяти* программы, а не участка кода, как в случае использования блокировок на базе мьютексов.

Известны как программные реализации транзакционной памяти (software transactional memory – STM): LazySTM, TinySTM, GCC TM, DTMC, RSTM, STMX, STM Monad, так и аппаратные реализации в процессорах (hardware transactional memory): Intel TSX, AMD ASF, Oracle Rock, IBM POWER8, IBM PowerPC A2.

В рамках программной транзакционной памяти программисту предоставляются языковые конструкции или API для формирования в программе *транзакционных секций* (transactional section) – участков кода, в которых осуществляется защита совместно используемых областей памяти. Выполнение потоками таких секций осуществляется без их блокирования. На среду выполнения (runtime) ложатся задачи по контролю за корректностью выполнения транзакций. Если во время выполнения транзакции другие потоки одновременно с ней не модифицировали защищенную область памяти, то транзакция считается корректной, и она фиксируется. Если же два или более потока, при выполнении транзакций, обращаются к одной и той же области памяти и как минимум один из них выполняет операцию записи, то возникает *конфликт* (аналог состояния гонки данных). Для его разрешения выполнение одной или нескольких транзакций может быть либо приостановлено (до завершения конфликтующей транзакции), либо прервано, а все модифицированные ими (их потоками) области памяти приведены в исходное состояние (на момент старта транзакции) – *отмена транзакции и восстановление* (cancel and rollback).

Для того, чтобы обнаруживать конфликты runtime-система должна отслеживать попытки одновременного доступа к одной и той-же области памяти. Это реализуется путем поддержки информации о состоянии защищаемых регионов памяти. Возможны два уровня гранулярности контролируемых областей: *уровень программных объектов* (object-based STM) и *уровень слов памяти* (word-based STM).

Уровень программных объектов подразумевает поддержку runtime-системой метаданных о состоянии каждого объекта программы. Например, объектов в C++-программе.

Для реализации уровня слов памяти в простейшем случае требуется каждый байт линейного адресного пространства процесса сопровождать метаданными, что является практически невозможным. Вместе этого линейное адресное пространство процесса разбивается на фиксированные блоки, каждый из которых сопровождается метаданными о состоянии (подход, подобный прямому отображению физических адресов на кеш-память процессора) [5, 6]. Это приводит к тому, что множеству областей памяти соответствуют одни метаданные, что является источником возникновения ложных конфликтов. *Ложный конфликт* (false conflict) – это ситуация, при которой два или более потока во время выполнения транзакции обращаются к разным участкам линейного адресного пространства, но отображаемым на одни и те же

метаданные. Поэтому runtime-система воспринимает такую ситуацию как конфликт (data race), хотя на самом деле таковой отсутствует.

Ложные конфликты существенно снижают эффективность параллельных STM-программ. Поэтому остро стоит задача разработки алгоритмов обнаружения и сокращения числа ложных конфликтов в реализациях STM.

В данной работе предлагается метод оптимизации ложных конфликтов по результатам предварительного профилирования C/C++ STM-программы. Для чего разработан модуль расширения компилятора GCC, который реализует инструментацию оптимизируемой STM-программы. Результаты выполнения инструментированной STM-программы поступают на вход созданного модуля анализа и варьирования параметров реализации runtime-библиотеки STM в компиляторе GCC (libitm).

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 стандарт C++17. На сегодняшний день предложен черновой вариант спецификации поддержки транзакционной памяти в C++ [7]. Она реализована в компиляторе GCC начиная с версии 4.8 и предоставляет ключевые слова `__transaction_atomic`, `__transaction_relaxed` для создания транзакционных секций, а также `__transaction_cancel` для принудительной отмены транзакции.

Для выполнения транзакционных секций runtime-системой создаются транзакции. *Транзакция* (transaction) – 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.

Инструкции транзакций выполняются потоками параллельно (конкурентно). 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).

Отмена транзакции происходит в случае *обнаружения конфликта* – ситуации, при которой два или более потока обращаются к одному и тому же участку памяти и как минимум один из них выполняет операцию записи.

Для разрешения конфликта разработаны различные походы, например, можно приостановить на некоторое время или отменить одну из конфликтующих транзакций.

На рис. 2 представлен пример создания транзакционной секции, в теле которой выполняется добавление элемента в хэш-таблицу множеством потоков. После выполнения тела транзакционной секции каждый поток приступит к выполнению кода, следующего за ней, в случае отсутствия конфликтов. В противном случае поток повторно будет выполнять транзакцию до тех пор, пока его транзакция не будет успешно зафиксирована.

Основными аспектами реализации транзакционной памяти в runtime-системах являются:

- политика обновления объектов в памяти;
- стратегия обнаружения конфликтов;
- метод разрешения конфликтов.

```
/* Совместно используемая хеш-таблица */
hashtable_t *h;

/* Код потоков */
void *thread_start(void *arg) {
    struct data *d = (struct data *)arg;
    prepareData(d);

    /* Транзакционная секция */
    __transaction_atomic {
        /* Добавление элемента в хеш-таблицу */
        struct data *d = (struct data *)arg;
        hashtable_insert(h, d);
    }

    saveData(d);
    return NULL;
}
```

Рис. 2. Использование транзакционной памяти в языке C/C++ (GCC libitm).

Политика обновления объектов в памяти определяет, когда изменения объектов в рамках транзакции будут записан в память. Распространение получили две основные политики – ленивая и ранняя. *Ленивая* политика обновления объектов в памяти (lazy version management) откладывает все операции с объектами до момента фиксации транзакции. Все операции записываются в специальном журнале (redo log), который при фиксации используется для отложенного выполнения операций. Очевидно, что это замедляет операцию фиксации, но существенно упрощает процедуры ее отмены и восстановления. Примером реализаций ТП, использующих данную политику являются RSTM-LLT [8] и RSTM-RingSW[9].

Ранняя политика обновления (eager version management) предполагает, что все изменения объектов сразу записываются в память. 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. Примерами реализаций, использующими раннюю политику обновления данных являются GCC (libitm), TinySTM [5], LSA-STM [6], Log-TM [10], RSTM [8] и др.

Момент времени, когда инициируется алгоритм обнаружения конфликта, определяется *стратегией обнаружения конфликтов*. При *отложенной стратегии* (lazy conflict detection), алгоритм обнаружения конфликтов запускается на этапе фиксации транзакции [11]. Недостатком этой стратегии является то, что временной интервал между возникновением конфликта и его обнаружением может быть достаточно большим. Эта стратегия используется в RSTM-LLT [8] и RSTM-RingSW [9].

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, а также, в некоторых случаях, может привести к увеличению числа откатов транзакций. Стратегия реализована в TinySTM [5], LSA-STM [6] и TL2 [12].

В компиляторе GCC (libitm) реализован комбинированный подход к обнаружению конфликтов – отложенная стратегия используется совместно с пессимистической.

3. Обнаружение конфликтов

Выбор гранулярности обнаружения конфликтов – один из ключевых моментов при реализации программной транзакционной памяти. На сегодняшний день используются два уровня гранулярности: уровень программных объектов (object-based STM) и уровень слов памяти (word-based STM). Уровень программных объектов подразумевает отображение объектов модели памяти языка (объекты C++, Java, Scala) на метаданные runtime-библиотеки. При использовании уровня слов памяти осуществляется отображение блоков линейного адресного пространства процесса на метаданные. Метаданные хранятся в таблице, каждая строка которой соответствует объекту программы или области линейного адресного пространства процесса. В строке содержатся номер транзакции, выполняющей операцию чтения/записи памяти; номер версии отображаемых данных; их состояние и др. Модификация метаданных выполняется runtime-системой с помощью атомарных операций процессора.

В данной работе рассматривается реализация программной транзакционной памяти в компиляторе GCC, использующий уровень слов памяти (в версиях GCC 4.8+ размер блока – 16 байт).

На рис. 3 представлен пример организации метаданных транзакционной памяти с использованием уровня слов памяти (GCC 4.8+). Линейное адресное пространство процесса фиксированными блоками циклически отображается на строки таблицы, подобно кешу прямого отображения. Выполнение операции записи приведет к изменению поля «состояние» соответствующей строки таблицы на «заблокировано». Доступ к области линейного адресного пространства, у которой соответствующая строка таблицы помечена как «заблокировано», приводит к конфликту.

Основными параметрами транзакционной памяти с использованием уровня слов памяти являются число S строк таблицы и количество B адресов линейного адресного пространства, отображаемых на одну строку таблицы. От выбора этих параметров зависит число ложных конфликтов – ситуаций аналогичных ситуации ложного разделения данных при работе кеша процессора. В текущей реализации GCC (4.8-5.1) эти параметры фиксированы [13].

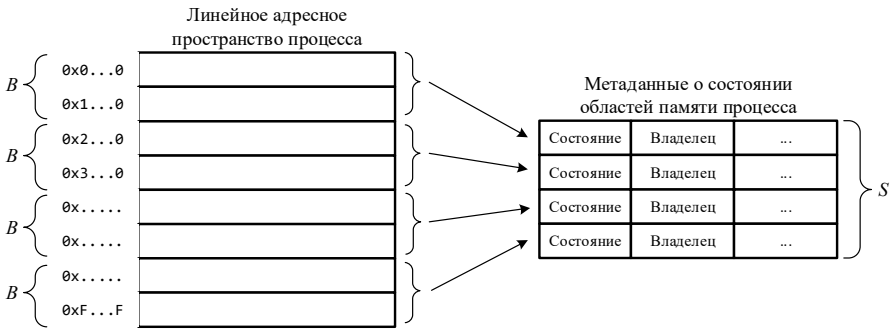

Рис. 3. Таблица с метаданными транзакционной памяти GCC 4.8+ (word-based STM): $B = 16$, $S = 2^{19}$.

4. Ложные конфликты

При отображении блоков линейного адресного пространства процесса на метаданные runtime-библиотеки возникают коллизии. Это неизбежно, так как размер таблицы метаданных гораздо меньше размера линейного адресного пространства процесса. Коллизии приводят к возникновению ложных конфликтов. *Ложный конфликт* – ситуация, при которой два или более потока во время выполнения транзакции обращаются к разным участкам линейного адресного пространства, но сопровождаемые одними и теме же метаданными о состоянии, и как минимум один поток выполняет операцию записи. Таким образом, ложный конфликт – это конфликт, который

происходит не на уровне данных программы, а на уровне метаданных runtime-библиотеки.

Возникновение ложных конфликтов приводит к откату транзакций, так же, как и возникновение обычных конфликтов, несмотря на то, что состояние гонки за данными не возникает, что влечет за собой увеличение времени выполнения STM-программ. Сократив число ложных конфликтов можно существенно уменьшить время выполнения программы.

На рис. 4 показан пример возникновения ложного конфликта в результате коллизии отображения линейного адресного пространства на строку таблицы. Поток 1 при выполнение операции записи над областью памяти с адресом A1 захватывает соответствующую строку таблицы. Выполнение операции чтения над областью памяти с адресом A2 потоком 2 приводит к возникновению конфликта, несмотря на то что операции чтения и записи выполняются над различными адресами. Последнее обусловлено тем, что 1 и 2 отображены на одну строку таблицы метаданных.

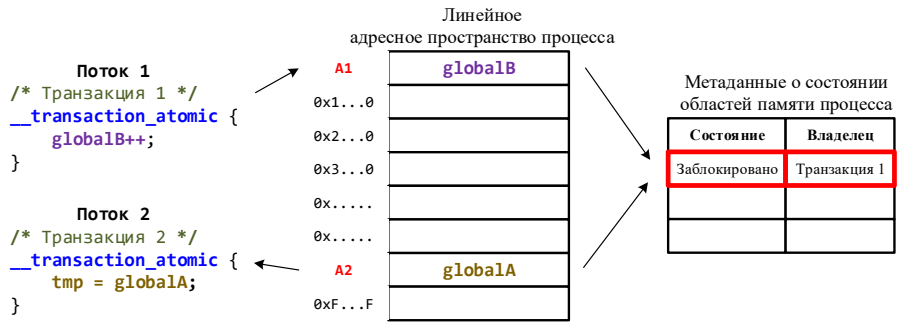

Рис. 4. Пример возникновения ложного конфликта при выполнении двух транзакций (GCC 4.8+).

5. Сокращение числа ложных конфликтов

В работе [14] для минимизации числа ложных конфликтов предлагается использовать вместо таблицы с прямой адресацией (как в GCC 4.8+), в которой индексом является часть линейного адреса, хеш-таблицу, коллизии в которой разрешаются методом цепочек. В случае отображения нескольких адресов на одну запись таблицы каждый адрес добавляется в список и помечается тэгом для идентификации (рис. 5). Такой подход позволяет избежать ложных конфликтов, однако накладные расходы на синхронизацию доступа к метаданным существенно возрастают, так как значительно увеличивается количество атомарных операций «сравнение с обменом» (compare and swap – CAS).

Рис. 5. Хеш-таблица для хранения метаданных.

Авторами предложен метод, позволяющий сократить число ложных конфликтов в STM-программах. Предполагается, что метаданные организованы в виде таблицы с прямой адресацией. Суть метода заключается в автоматической настройке параметров S и B таблицы под динамические характеристики конкретной STM-программы. Метод включает три этапа.

Этап 1. Инструментация транзакционных секций с целью профилирования. На первом этапе выполняется компиляция C/C++ STM-программы с использованием разработанного модуля инструментации транзакционных секций (модуль расширения GCC). В ходе статического анализа транзакционных секций STM-программ выполняется внедрение кода для регистрации обращений к функциям Intel TM ABI (_ITM_beginTransaction, _ITM_comitTransaction, _ITM_LU4, _ITM_WU4 и др.). Детали реализации модуля инструментации описаны ниже.

Этап 2. Профилирование программы. На данном этапе выполняется запуск STM-программы в режиме профилирования. Профилировщик регистрирует все операции чтения/записи памяти в транзакциях. В результате формируется протокол (trace), содержащий информацию о ходе выполнения транзакционных секций:

- адрес и размер области памяти, над которой выполняется операция;
- временная метка (timestamp) начала выполнения операции.

Этап 3. Настройка параметров таблицы. По протоколу определяются средний размер W читаемой/записываемой области памяти во время выполнения транзакций. По значению W подбираются субоптимальные параметры B и S таблицы, с которыми STM-программа компилируется.

Эксперименты с тестовыми STM-программами из пакета STAMP (6 типов STM-программ), позволили сформулировать эвристические правила для подбора параметров B и S по значению W.

Значение параметра S целесообразно выбирать из множества $\{2^{18}, 2^{19}, 2^{20}, 2^{21}\}$. Значение параметра B выбирается следующим образом:

- если $W = 1$ байт, то $B = 2^4$ байт;

- если $W = 4$ байт, то $B = 2^6$ байт;
- если $W = 8$ байт, то $B = 2^7$ байт;
- если $W >= 64$ байт, то $B = 2^8$ байт.

6. Инструментация транзакционных секций

STM-компилятор осуществляет трансляцию транзакционных секций в последовательность вызовов функций runtime-системы поддержки TM [15].

Компания Intel предложила спецификацию ABI для runtime-систем поддержки транзакционной памяти – Intel TM ABI [16]. Компилятор GCC, библиотека libitm, реализует этот интерфейс начиная с версии 4.8.

На рис. 6 представлен пример трансляции компилятором GCC транзакционной секции в обращения к функциям Intel TM ABI.

```
int a, b;
...
__transaction_atomic {
    if (a == 0)
        b = 1;
    else
        a = 0;
}
...
```

```
    ...
    state = _ITM_beginTransaction()
<L1>:
    if (state & a_abortTransaction)
        goto <L3>;
    else
        goto <L2>;
<L2>:
    if (_ITM_LU4(&a) == 0)
        _ITM_WU4(&b, 1);
    else
        _ITM_WU4(&a, 0);
    _ITM_commitTransaction();
<L3>:
    ...
```

Рис. 6. Инструментация транзакционной секции компилятором GCC.

В общем случае последовательность выполнения транзакции следующая:

1. Создание транзакции (вызов `_ITM_beginTransaction`) и анализ ее состояния. Если состояние транзакции содержит флаг принудительной отмены, то выполнение продолжается с метки `<L3>`, т.е. осуществляется выход из транзакции, иначе выполнение тела транзакции начинается с метки `<L2>`.
2. Выполнение транзакции. Если выполняется принудительная отмена транзакции, то в состоянии устанавливается флаг принудительной отмены (`a_abortTransaction`) и управление передается метке `<L1>`.
3. Попытка фиксации транзакции (вызов `_ITM_commitTransaction`), в случае возникновения конфликта транзакция отменяется, в состояние

транзакции записывается причина отмены и выполнение транзакции повторяется начиная с метки `<L1>`.

Разработанный модуль анализирует промежуточное представления GIMPLE транзакционных секций и добавляет функции регистрации обращений к функциям Intel TM ABI: регистрация начала транзакции и ее фиксации, транзакционное чтения/запись областей памяти. Функции регистрации заносят в протокол адреса и размер областей памяти, над которыми выполняются операции, а также время начала выполнения операций.

На рис. 7 представлен пример инструментации транзакционных секций. Функции с префиксом `tm_prof_` выполняют регистрацию событий.

```
int a, b;
...
__transaction_atomic {
    if (a == 0)
        b = 1;
    else
        a = 0;
}
...
```

→

```
    ...
    state = _ITM_beginTransaction()
    tm_prof_begin(state);
<L1>:
    if (state & a_abortTransaction)
        goto <L3>;
    else
        goto <L2>;
<L2>:
    tm_prof_operation(sizeof(a));
    if (_ITM_LU4(&a) == 0) {
        tm_prof_operation(sizeof(b));
        _ITM_WU4(&b, 1);
    } else {
        tm_prof_operation(sizeof(a));
        _ITM_WU4(&a, 0);
    }
    _ITM_commitTransaction();
    tm_prof_commit();
<L3>:
    ...
```

Рис. 7. Инструментация транзакционной секции.

Модуль инструментации в связке с библиотекой профилирования предоставляют достаточно сведений о динамических характеристиках транзакционных секций для того чтобы ответить на вопрос: «Фиксации каких транзакций или операции над какими данными приводят к отмене других транзакций?». Кроме этого, метод позволяет определить значения субоптимальных значений параметров реализации runtime-системы ТП, а именно число строк таблицы метаданных о состоянии областей памяти и количество адресов линейного адресного пространства, отображаемых на одну строку таблицы.

8. Эксперименты

Экспериментальное исследование проводилось на вычислительной системе, оснащенной двумя четырехъядерными процессорами Intel Xeon E5420. В

данных процессорах отсутствует поддержка аппаратной транзакционной памяти (Intel TSX).

В качестве тестовых программ использовались многопоточные STM-программы из пакета STAMP [9, 11, 12]. Число потоков варьировалось от 1 до 8. Тесты собирались компилятором GCC 5.1.1. Операционная система GNU/Linux Fedora 21 x86_64.

В рамках экспериментов измерялись значения двух показателей:

- время t выполнения STM-программы;
- количество C ложных конфликтов в программе.

На рис. 8 и 9 показана зависимость количества C ложных конфликтов и времени t выполнения теста от числа потоков при различных значениях параметров B и S. Результаты приведены для программы genome из пакета STAMP. В ней порядка 10 транзакционных секций, реализующих операции над хеш-таблицей и связными списками. Видно, что увеличение значений параметров S и B приводит к уменьшению числа возможных коллизий (ложных конфликтов), возникающих при отображении адресов линейного адресного пространства процесса на записи таблицы.

При размере таблицы 2^{21} записей, на каждую из которых отображается 2^6 адресов линейного адресного пространства, достигается минимум времени выполнения теста genome, а также минимум числа ложных конфликтов.

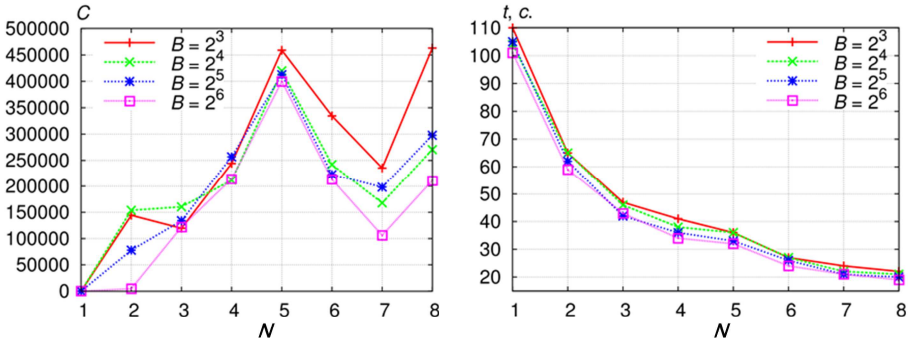

Рис. 8. Зависимость числа C ложных конфликтов (слева) и времени t выполнения теста (справа) от числа N потоков: $S = 2^{19}$.

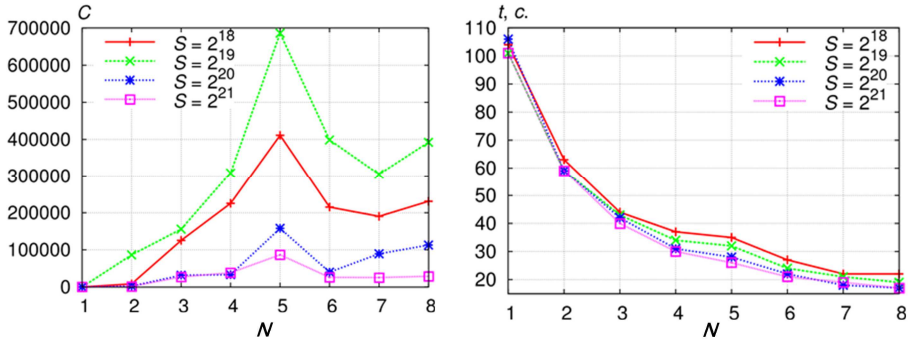

Рис. 9. Зависимость числа C ложных конфликтов (слева) и времени t выполнения теста (справа) от числа N потоков: $B = 2^6$.

Время выполнения теста genome удалось сократить в среднем на 20% за счет минимизации числа ложных конфликтов.

9. Заключение

В рамках данной работы создан модуль компилятора GCC для инструментации транзакционных секций STM-программ. Предложена библиотека профилирования STM-программ и оптимизации параметров внутренних структур данных runtime-библиотеки транзакционной памяти компилятора GCC (libitm) под конкретное приложение. Используя предложенный метод время выполнения теста genome удалось сократить на 20% за счет минимизации числа ложных конфликтов.

В будущем, планируется разработать алгоритмы выбора способа реализации программной транзакционной памяти во время компиляции программы. Дополнительно планируется провести исследование реализаций программной транзакционной памяти без централизованного хранения метаданных о состоянии областей памяти процесса.

Работа выполнена при поддержке РФФИ (гранты 15-37-20113, 15-07-00653), а также Министерства образования и науки Российской Федерации в рамках договора 02.G25.31.0058 от 12.02.2013.

Список литературы

[1]. M. Herlihy, N. Shavit. The Art of Multiprocessor Programming. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[2]. Hendler D., Shavit N., Yerushalmi L. A scalable lock-free stack algorithm // Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures. SPAA '04. 2004. P. 206–215.

[3]. Кузнецов С.Д. Транзакционная память [HTML]. http://citforum.ru/programming/digest/transactional_memory/.

[4]. N. Shavit, D. Touitou. 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, 204–213.

[5]. Pascal Felber, Christof Fetzer, Patrick Marlier, and Torvald Riegel, Time-based Software Transactional Memory, IEEE Transactions on Parallel and Distributed Systems, Volume 21, Issue 12, pp. 1793-1807, December 2010.

[6]. Torvald Riegel, Pascal Felber, and Christof Fetzer. A Lazy Snapshot Algorithm with Eager Validation, 20th International Symposium on Distributed Computing (DISC), 2006.

[7]. Victor Luchango, Jens Maurer, Mark Moir. Transactional memory for C++ [PDF]. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3718.pdf

[8]. Rochester Software Transactional Memory Runtime. Project web site [HTML]. www.cs.rochester.edu/research/synchronization/rstm/.

[9]. Michael F. Spear, Luke Dalessandro, Virendra J. Marathe, and Michael L. Scott. A comprehensive strategy for contention management in software transactional memory. In PPoPP '09: Proc. 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, February 2009, P – 141-150.

[10]. Kevin E. Moore, Jayaram Bobba, Michelle J. Moravan, Mark D. Hill, and David A. Wood. LogTM: Log-based transactional memory. In HPCA '06: Proc. 12th International Symposium on High-Performance Computer Architecture, February 2006, P. – 254-265.

[11]. Michael F. Spear, Maged M. Michael, and Christoph von Praun. RingSTM: scalable transactions with a single atomic instruction. In SPAA '08: Proc. 20th Annual Symposium on Parallelism in Algorithms and Architectures, June 2008, P. 275–284.

[12]. Dave Dice, Ori Shalev, and Nir Shavit. Transactional locking II. In DISC '06: Proc. 20th International Symposium on Distributed Computing, September 2006. Springer Verlag Lecture Notes in Computer Science volume 4167, P. – 194-208.

[13]. Pascal Felber, Christof Fetzer, Torvald Riegel. Dynamic performance tuning of word-based software transactional memory. PPOPP 2008. P. – 237-246.

[14]. Craig Zilles and Ravi Rajwar. Implications of false conflict rate trends for robust software transactional memory. In IISWC '07: Proc. 2007 IEEE.

[15]. Olszewski M., Cutler J., Steffan J. G. JudoSTM: A Dynamic Binary-Rewriting Approach to Software Transactional Memory. Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques. PACT '07. 2007. P. 365–375.

[16]. Intel Corporation. Intel Transactional Memory Compiler and Runtime Application Binary Interface. Revision: 1.0.1, November 2008.

Instrumentation and Optimization of Transactional Sections Execution in Multithreaded Programs

[1]*I. Kulagin <ivan.i.kulagin@gmail.com>*
[2]*M. Kurnosov <mkurnosov@gmail.com>*
[1]*SibSUTIS, 86 Kirova Str., Novosibirsk, 630102, Russian Federation*
[2] *Saint Petersburg Electrotechnical University "LETI", 5 Professora Popova Str., St. Petersburg, 197376, Russian Federation*

Abstract. Software transactional memory (STM) is approach to develop thread-safe programs. In contrast to locking a block of code by mutex, the main idea of this approach is to protect memory areas from concurrent access by threads. In this paper, we investigate efficiency of software transactional memory implementation in GCC compiler. Software tools for instrumentation and profiling STM-programs are proposed. Profile-guided method for reducing false conflicts. in STM-programs is presented. False conflict is conflict that exist on the level of runtime library but not when the memory accessing happens. The instrumentation module analyzes the code of transactional sections and puts calls for registration of the some events (begin transactions, transactional read/write, commit transactions and abort transactions). The profiling of the instrumented program allows get dynamic properties of execution transactional code like size of used data, read/write addresses, timestamp of events, etc. The static instrumentation allows to optimize dynamic properties of execution transactional sections. The method of reducing false conflicts performs the tuning of transactional memory parameters value in GCC implementation (libitm runtime-library) by using the profiling results (profile-guided optimization). The efficiency of reducing false conflicts is investigated on the STAMP benchmarks. These benchmarks contains eight tests that operate with hash tables, lists, arrays protected by software transactional memory. Using the proposed method of the tests time is reduced approximately to 20%.

Keywords: software transactional memory; instrumentation; profile-guided optimization; multithreaded programming; compilers.

DOI: 10.15514/ISPRAS-2015-27(6)-9

For citation: Kulagin I.**,** Kurnosov M. Instrumentation and Optimization of Transactional Sections Execution in Multithreaded Programs. *Trudy ISP RAN/Proc. ISP RAS, vol. 27, issue 6, 2015,* pp. 135-150 (in Russian). DOI: 10.15514/ISPRAS-2015-27(6)-9

References

[1]. M. Herlihy, N. Shavit. The Art of Multiprocessor Programming. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.
[2]. Hendler D., Shavit N., Yerushalmi L. A scalable lock-free stack algorithm // Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures. SPAA ’04. 2004. P. 206–215.
[3]. Kuznetsov S.D. Transaktsionnaya pamat. [Transactional memory]. http://citforum.ru/programming/digest/transactional_memory/. (in Russian).
[4]. N. Shavit, D. Touitou. 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, 204–213.
[5]. Pascal Felber, Christof Fetzer, Patrick Marlier, and Torvald Riegel, Time-based Software Transactional Memory, IEEE Transactions on Parallel and Distributed Systems, Volume 21, Issue 12, pp. 1793-1807, December 2010.
[6]. Torvald Riegel, Pascal Felber, and Christof Fetzer. A Lazy Snapshot Algorithm with Eager Validation, 20th International Symposium on Distributed Computing (DISC), 2006.
[7]. Victor Luchango, Jens Maurer, Mark Moir. Transactional memory for C++ [PDF]. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3718.pdf
[8]. Rochester Software Transactional Memory Runtime. Project web site [HTML]. www.cs.rochester.edu/research/synchronization/rstm/.
[9]. Michael F. Spear, Luke Dalessandro, Virendra J. Marathe, and Michael L. Scott. A comprehensive strategy for contention management in software transactional memory. In PPoPP ’09: Proc. 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, February 2009, P – 141-150.
[10]. Kevin E. Moore, Jayaram Bobba, Michelle J. Moravan, Mark D. Hill, and David A. Wood. LogTM: Log-based transactional memory. In HPCA ’06: Proc. 12th International Symposium on High-Performance Computer Architecture, February 2006, P. – 254-265.
[11]. Michael F. Spear, Maged M. Michael, and Christoph von Praun. RingSTM: scalable transactions with a single atomic instruction. In SPAA ’08: Proc. 20th Annual Symposium on Parallelism in Algorithms and Architectures, June 2008, P. 275–284.
[12]. Dave Dice, Ori Shalev, and Nir Shavit. Transactional locking II. In DISC ’06: Proc. 20th International Symposium on Distributed Computing, September 2006. Springer Verlag Lecture Notes in Computer Science volume 4167, P. – 194-208.
[13]. Pascal Felber, Christof Fetzer, Torvald Riegel. Dynamic performance tuning of word-based software transactional memory. PPOPP 2008. P. – 237-246.
[14]. Craig Zilles and Ravi Rajwar. Implications of false conflict rate trends for robust software transactional memory. In IISWC ’07: Proc. 2007 IEEE.
[15]. Olszewski M., Cutler J., Steffan J. G. JudoSTM: A Dynamic Binary-Rewriting Approach to Software Transactional Memory. Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques. PACT ’07. 2007. P. 365–375.
[16]. Intel Corporation. Intel Transactional Memory Compiler and Runtime Application Binary Interface. Revision: 1.0.1, November 2008.