СРАВНЕНИЕ БЛОКИРУЮЩЕЙ И НЕБЛОКИРУЮЩЕЙ СИНХРОНИЗАЦИИ В ТРАНЗАКЦИОННОЙ ПАМЯТИ УРОВНЯ ОБЪЕКТА

А.Ю. Бахиркин
Московский Государственный Технический Университет им. Н.Э. Баумана
Россия, 105005, Москва, 2-я Бауманская ул., д. 5
E-mail: abakhirkin@gmail.com

Выбор между блокирующей и неблокирующей синхронизацией считается важным решением при проектировании реализации транзакционной памяти, т.к. необходимо понимать, принесут ли пользу усилия по обеспечению для транзакционной памяти определенных гарантий прогресса.
В рамках работы реализована программная библиотека транзакционной памяти на платформе CLR в двух вариантах: с использованием блокирующей и неблокирующей синхронизации. Проведено сравнение двух вариантов реализации в синтетических тестах, основанных на реализации с помощью транзакционной памяти потокобезопасных вариантов распространенных структур данных.

A COMPARISON OF BLOCKING AND NON-BLOCKING SYNCHRONIZATION IN OBJECT-BASED SOFTWARE TRANSACTIONAL MEMORY / A.Yu. Bakhirkin (Bauman Moscow State Technical University, 2-nd Baumanskaya, 5, 105005, Moscow, Russia). The choice between blocking and nonblocking synchronization techniques is a major design decision when implementing software transactional memory. It should be clearly visible to the implementers whether the efforts made to provide nonblocking progress guarantees are going to pay off.
In this work two similar CLR-based software transactional memory implementations are presented: a blocking one and a lock-free one. These two implementations are compared in a number of synthetic benchmarks employing common STM-based thread-safe data structures.

1 Введение

Противоречием в области транзакционной памяти является выбор между блокирующими и неблокирующими техниками синхронизации. Ряд ранних реализаций транзакционной памяти, например WSTM и OSTM[1], фактически являлся обобщением современных на тот момент неблокирующих техник параллельного программирования. Так, в [1] наглядно показан переход от неблокирующих техник к транзакционности. Позднее, вслед за [2], где неблокирующий подход подвергается критике, интерес исследователей и разработчиков сместился в сторону блокирующих техник. В частности, к выводам [2] апеллируют [3, 4]. Однако в [5, 6, 7, 8], тезисы [2] ставятся под сомнение и производится сравнение блокирующих и неблокирующих техник для транзакционной памяти уровня слова.

2 Блокирующая и неблокирующая синхронизация

Выбор между блокирующей и неблокирующей синхронизацией является важным при проектировании реализации транзакционной памяти, т.к. необходимо понимать, принесут ли пользу усилия по обеспечению для транзакционной памяти определенных гарантий прогресса.

Гарантии прогресса, предоставляемые неблокирующим реализациями, помогают справиться с livelock-ами и инверсией приоритетов, которые в случае блокирующей синхронизации могут приводить к заметному падению производительности, особенно при росте количества рабочих потоков по сравнению с количеством процессоров или ядер в системе[3].

С другой стороны, в качестве проблем неблокирующей транзакционной памяти отмечается[2] значительное усложнение алгоритмов, а также дополнительные накладные расходы из-за особенностей неблокирующих алгоритмов, снижающих локальность обращений к памяти и требующих в ряде случаев использования атомарных операций, таких как compare-and-swap, вместо операций чтения/записи. Для транзакционной памяти уровня слова заметное усложнение алгоритмов при переходе к неблокирующей синхронизации можно наблюдать в [5].

Вопрос выбора между блокирующей и неблокирующей синхронизацией для транзакционной памяти уровня объекта на данный момент слабо рассмотрен в литературе. Как правило, авторы заранее принимают сторону блокирующей или неблокирующей реализации. В итоге, например, не существует двух аналогичных реализаций транзакционной памяти уровня слова, отличающихся только применением блокирующей, либо неблокирующей синхронизации.

Для исследования этой проблемы в рамках данной работы на платформе CLR реализуется программная библиотека транзакционной памяти уровня объекта в двух вариантах:

– первый вариант — блокирующая библиотека транзакционной памяти уровня объекта;

– второй вариант — код алгоритмов модифицируется для придания им неблокирующих свойств.

Для сравнения производительности двух вариантов библиотеки реализуются синтетические тесты на основе потокобезопасных структур данных, использующих транзакционную память.

Для реализации была выбрана платформа CLR как платформа, поддерживающая сборку мусора. Это позволило упростить реализацию: избежать отслеживания времени жизни вспомогательных структур данных, а также предотвратить проблемы, вызываемые параллельными конфликтующими операциями записи и управления памятью, подробно описанные в [9, 3].

3 Реализация блокирующей транзакционной памяти

В качестве основы для блокирующей реализации были выбраны TL2[9] — высокопроизводительная реализация на языке C — и OSTM[1].

В качестве основных унаследованных от TL2 моментов стоит отметить следующие.

– При открытии объекта на запись создается приватная для транзакции копия, заменяющая оригинальный объект при фиксации транзакции (т.н. *поздняя запись*).

– Объекты блокируются только на время фиксации транзакции, когда происходит замена глобально видимых объектов на их модифицированные транзакцией копии (т.н. *позднее разрешение конфликтов*, или, применительно к блокирующей реализации — *блокировка времени фиксации*). В литературе позднюю запись и позднее разрешение конфликтов иногда относят к неперспективным подходам[10], однако пример TL2 показывает, что это не так.

– Ведется глобальный счетчик *времени*, инкрементируемый при фиксации каждой транзакции-писателя. Для каждой транзакции отмечается время ее начала, а для объекта — время его публикации. Разрешая транзакции открывать на чтение и запись только объекты, *опубликованные* раньше ее начала, можно исключить возможность нахождения транзакции в неконсистентном состоянии, т.е. наблюдать такую комбинацию состояний объектов, которая невозможна для линеаризуемых транзакций. В данной работе, как и в TL2, данный прием реализован в упрощенном варианте. Более полно и подробно он описан в [11, 12, 13].

Для доступа к объектам, аналогично DSTM[14] и в отличии от OSTM, используется два уровня косвенности. На верхнем уровне находится дескриптор объекта. Именно дескрипторы видимы транзакциям — и их необходимо открывать на чтение или запись — для получения данных объекта, с которыми можно непосредственно работать.

В каждый момент времени дескриптор либо свободен — и ссылается на *версию* объекта — либо заблокирован транзакцией — и тогда ссылается на нее. Версия объекта содержит время своей публикации и ссылку на собственно данные объекта. В ходе работы выяснилось, что для объекта, поддерживающего транзакционность, все равно приходится реализовывать операцию клонирования, т.е. модифицировать его по сравнению с аналогичным объектом, используемым в нетранзакционном коде. Поэтому допустимо было бы не выделять дополнительный уровень косвенности и располагать поле со временем публикации в данных объекта.

Необходимо выделить основные операции, реализуемые транзакционной памятью.

Фиксация транзакции. Процедура фиксации выполняется аналогично TL2 и выглядит следующим образом.

– Блокируются дескрипторы из множества записанных объектов.

– Увеличивается счетчик времени.

– Транзакция переходит в состояние *валидации*. Данное состояние и все, что с ним связано здесь и далее — необходимо для линеаризации транзакций, что наглядно показано в [1].

– Транзакция убеждается, что текущие версии прочитанных объектов не изменились с момента открытии на чтение.

– Транзакция переходит в *зафиксированное* состояние. Таким образом, за счет особенностей алгоритма получения текущей версии, меняются текущие версии всех модифицированных транзакцией объектов.

– Транзакция освобождает дескрипторы, проставляя в них новые версии модифицированных объектов.

Код фиксации для блокирующей реализации представлен в приложении А на листинге 1.

Получение текущей версии объекта. Получая текущую версию, транзакция извлекает ее либо из дескриптора, либо из множества записанных объектов заблокировавшей его транзакции (берется копия, либо оригинал, в зависимости от статуса транзакции-владельца). Если транзакция-владелец находится в состоянии валидации, необходимо дождаться ее отката или успешной фиксации. Код получения текущей версии объекта для блокирующей реализации представлен в приложении А на листинге 2.

Открытие объекта на чтение или запись. Открывая объект, транзакция запоминает его текущую версию (или ее копию со ссылкой на оригинал — в случае открытия на запись) во множестве прочитанных, либо записанных объектов.

При данной реализации возможны взаимоблокировки, которые можно считать одним из проявлений конфликта между транзакциями.

– При блокировке объектов, что говорит о конфликте *запись-запись*.

– При получении текущей версии на этапе валидации, что говорит о двойном конфликте *чтение-запись*.

В блокирующей реализации взаимоблокировки разрешаются путем отката транзакции по таймауту. Стоит отметить, что вопрос выбора таймаута практически не рассмотрен в литературе.

4 Реализация неблокирующей транзакционной памяти

При переходе от блокирующей к неблокирующей реализации модификации подвергаются только те алгоритмы, которые связаны с вероятными взаимоблокировками и их предотвращением. Модифицируется код фиксации транзакции и получения текущей версии объекта, остальной же программный код, например, открытие объекта остается без изменений. Таким образом, разработанная неблокирующая реализация объединяет неблокирующие техники из OSTM[1] с механизмом исключения возможности неконсистентного состояния, реализованным в TL2[9]. Код фиксации и получения текущей версии объекта для неблокирующей реализации представлен в приложении А на листингах 3 и 4.

Взаимоблокировки при блокировке объектов можно избежать тривиальным образом — блокируя дескрипторы в определенном порядке. В языках C/C++ упорядочивать объекты можно согласно их адресам, являющимся уникальными числовыми идентификаторами. Особенностью платформы CLR в данном случае является то, что с объектом изначально не связан уникальный числовой идентификатор. Сборщик мусора может перемещать объекты в памяти, и платформа не предоставляет разработчику доступ к значениям их адресов. Вместо этого платформа CLR связывает с объектом его т.н. хеш-код — некоторое псевдослучайное неуникальное целое число. В данной работе для упорядочения дескрипторов решено было использовать пару, состоящую из хеш-кода *дескриптора* и очередного значения глобального счетчика, причем значения счетчика резервируются для пары дескрипторов только в случае коллизии их хеш-кодов.

Взаимоблокировки при получении текущей версии объекта, заблокированного валидирующейся транзакцией, представляют больший интерес. Для их разрешения данная реализация использует стратегию *помощи*, описанную применительно к транзакционной памяти в [1]. Вместо того, чтобы ожидать окончания валидации транзакции, текущая транзакция начинает ей *помогать* — запускает в своем потоке код ее фиксации. Для предотвращения *рекурсивной помощи* (эта проблема приходит в неблокирующем алгоритме на место взаимоблокировки) вводится приоритет транзакций. Транзакция помогает только транзакциям, имеющим больший приоритет или не находящимся на стадии валидации, а валидирующуюся транзакцию с меньшим приоритетом откатывает.

Задача выбора способа вычисления приоритета транзакции в неблокирующем случае приходит на смену проблеме выбора таймаутов перед откатом в блокирующем случае. В данной работе в целях упрощения выбрано тривиальное отношение порядка, аналогичное тому, что используется для упорядочения блокировки объектов. В целом же данный вопрос, называемый *управлением конфликтами*, подробно освещен в литературе, например в [15, 16, 17, 18].

Для реализации помощи потребовалось модифицировать код получения текущей версии объекта, а также гарантировать корректное поведение кода фиксации при его параллельном выполнении для одной и той же транзакции несколькими потоками (тривиальный процесс, заключающихся в замене ряда операций записи на compare-and-swap). Таким образом для неблокирующей реализации, аналогично OSTM, были достигнуты гарантии lock-free.

В целом изменения в алгоритмах, необходимые для придания им неблокирующих свойств оказались локальными. Был модифицирован код получения текущей версии объекта и фиксации транзакции, большая же часть кода реализации осталась неизменной.

По результатам данной работы нельзя говорить, что для транзакционной памяти уровня объекта при переходе к неблокирующей синхронизации имеет место серьезное усложнение реализации. Во-первых, изменения в алгоритмах локальны, поэтому выбор между блокирующей и неблокирующей синхронизацией не влияет на общую структуру реализации и ее программный интерфейс. Так, реализация может предлагать разработчику выбор между блокирующим и неблокирующим алгоритмом. Либо на начальном этапе разработки может быть реализована одна из техник, а в дальнейшем, при необходимости, можно реализовать другую. Также стоит отметить, что ряд аспектов неблокирующей транзакционной памяти (в частности — управление конфликтами) изучен в литературе лучше аналогичных аспектов блокирующих реализаций. Поэтому может оказаться целесообразным применять неблокирующую синхронизацию в прототипе реализации транзакционной памяти, а в дальнейшем при необходимости переходить к блокирующей технике.

5 Тестирование производительности

В данной работе тестировании производительности разработанных реализаций осуществлялось в синтетических тестах. С помощью транзакционной памяти были реализованы два варианта структуры данных «множество»: на основе несбалансированного дерева и сортированного связного списка. В ходе тестирования в нескольких потоках в цикле запускались операции поиска, вставки и удаления, при этом варьировалось: количество потоков, диапазон ключей (как следствие — средний размер множества) и процентное соотношение операций поиска и модификации.

Данная работа следует подходу к синтетическим тестам, принятому в [4]: вместо реализации сложных сценариев и структур данных используются простые алгоритмы, моделирующие отдельные аспекты реальных сценариев использования транзакционной памяти. Так, связный список демонстрирует поведение транзакционной памяти в случае длинных транзакций, читающих большое количество объектов, а несбалансированное дерево — наоборот в случае коротких транзакций. В обоих случаях каждая транзакция модифицирует небольшое количество объектов.

Тестирование производилось на стенде следующей конфигурации:

– Core i5-540, 2 физических ядра, hyperthreading (4 логических ядра).

– 3Gb RAM.

На рис. 1 — 3 представлены наиболее наглядные графики производительности, полученные в данной работе. По ним видно, что производительность разработанных блокирующей и неблокирующей реализаций различается незначительно. В частности, для обеих имеет место спад производительности при росте количества потоков и превышении им количества процессоров. Стоит отметить, что в данной работе, в отличии от, например, [3, 4], блокирующая реализация не использует подсказки планировщику потоков. В целом, в данной работе не было выявлено сценария, где блокирующая или неблокирующая реализация показала бы более высокую производительность. Предположительно это связано с тем, что разница в производительности между блокирующим и неблокирующим алгоритмом незначительна по сравнению с накладными расходы на последовательное выполнение, сходными для обеих реализаций.

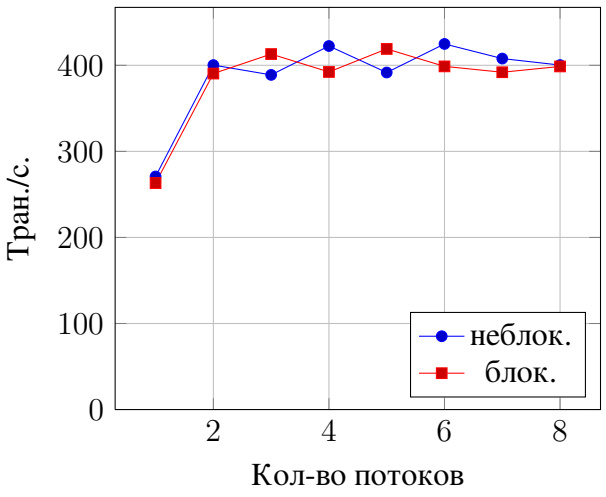

Рис. 1: Несбаланс. дерево, 10000 ключей

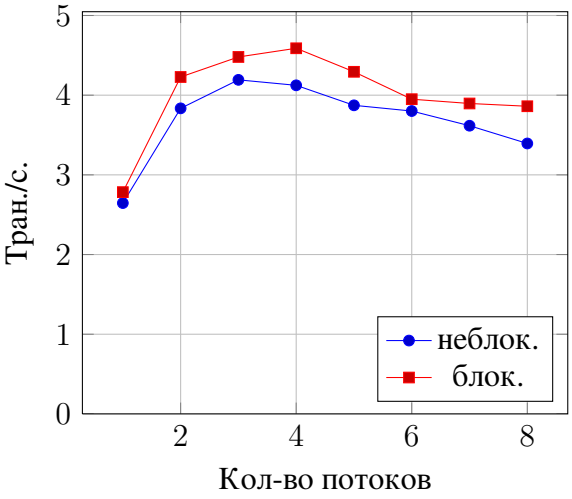

Рис. 2: Связный список, 10000 ключей

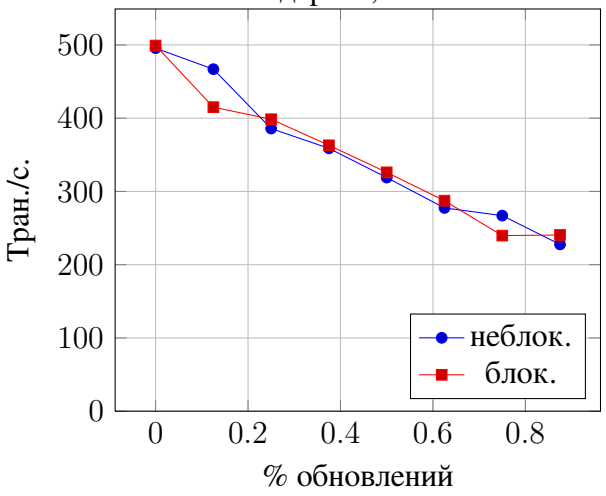

Рис. 3: Несбаланс. дерево, 10000 ключей

6 Заключение

В данной работе описана реализация транзакционной памяти уровня объекта на платформе CLR в двух в остальном идентичных вариантах: с использованием блокирующей и неблокирующей синхронизации. Реализация объединяет преимущества ряда ранее описанных систем, таких как TL2 и OSTM. Продемонстрирован переход он блокирующего к неблокирующему алгоритму, отмечена его невысокая трудоемкость, показан ряд сложностей, с которыми сталкиваются разработчики транзакционной памяти уровня объекта в управляемой среде. Проведено тестирование производительности реализаций в ряде синтетических тестов на основе потокобезопасных структур данных, где реализации показали схожие результаты.

Список литературы

[1] *Fraser, K.* Concurrent programming without locks / K. Fraser, T. Harris // *ACM Transactions on Computer Systems (TOCS)*. — 2007. — may. — Vol. 25, no. 2.

[2] *Ennals, R.* Software transactional memory should not be obstruction-free: Tech. Rep. IRC-TR-06-052 / R. Ennals: Intel Research Cambridge Tech Report, 2006. — Jan.

[3] *Dice, D.* Understanding tradeoffs in software transactional memory / D. Dice, N. Shavit // CGO '07: Proceedings of the International Symposium on Code Generation and Optimization. — Washington, DC, USA: IEEE Computer Society, 2007. — Pp. 21–33.

[4] Mcrt-stm: a high performance software transactional memory system for a multi-core runtime / B. Saha, A.-R. Adl-Tabatabai, R. L. Hudson et al. // Proc. 11th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP '06). — 2006. — Mar. — Pp. 187–197.

[5] *Marathe, V. J.* Toward high performance nonblocking software transactional memory / V. J. Marathe, M. Moir // PPoPP '08: Proc. 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming. — 2008. — feb. — Pp. 227–236.

[6] Lowering the overhead of software transactional memory: Tech. Rep. TR 893 / V. J. Marathe, M. F. Spear, C. Heriot et al.: Computer Science Department, University of Rochester, 2006. — Mar. — Condensed version submitted for publication.

[7] *Marathe, V. J.* Design tradeoffs in modern software transactional memory systems / V. J. Marathe, W. N. S. Iii, M. L. Scott // In Proceedings of the 7th Workshop on Languages, Compilers, and Run-time Support for Scalable Systems. — ACM Press, 2004. — Pp. 1–7.

[8] *Marathe, V. J.* Efficient nonblocking software transactional memory / V. J. Marathe, M. Moir // PPoPP '07: Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming. — New York, NY, USA: ACM, 2007. — Pp. 136–137.

[9] *Dice, D.* Transactional locking ii / D. Dice, O. Shalev, N. Shavit // In Proc. of the 20th Intl. Symp. on Distributed Computing. — 2006. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.90.811.

[10] *Larus, J.* Transactional Memory (Synthesis Lectures on Computer Architecture) / J. Larus, R. Rajwar. — Morgan & Claypool Publishers, 2007.

[11] *Riegel, T.* A lazy snapshot algorithm with eager validation / T. Riegel, P. Felber, C. Fetzer // Proceedings of the 20th International Symposium on Distributed Computing, DISC 2006. — Springer, 2006. — Sep. — Vol. 4167 of *Lecture Notes in Computer Science*. — Pp. 284–298.

[12] *Riegel, T.* Snapshot isolation for software transactional memory / T. Riegel, C. Fetzer, P. Felber // Proceedings of the First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing. — 2006. — Jun.

[13] *Riegel, T.* Time-based transactional memory with scalable time bases / T. Riegel, C. Fetzer, P. Felber // 19th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA). — 2007. — Jun.

[14] Software transactional memory for dynamic-sized data structures / M. Herlihy, V. Luchangco, M. Moir, I. William N. Scherer // PODC '03: Proc. 22nd ACM Symposium on Principles of Distributed Computing. — 2003. — Jul. — Pp. 92–101.

[15] A comprehensive strategy for contention management in software transactional memory / M. F. Spear, L. Dalessandro, V. J. Marathe, M. L. Scott // PPoPP '09: Proc. 14th ACM SIGPLAN symposium on Principles and practice of parallel programming. — 2009. — feb. — Pp. 141–150.

[16] *Guerraoui, R.* Polymorphic contention management / R. Guerraoui, M. Herlihy, B. Pochon // DISC '05: Proceedings of the nineteenth International Symposium on Distributed Computing. — LNCS, Springer, 2005. — Sep. — Pp. 303–323.

[17] Robust contention management in software transactional memory / R. Guerraoui, M. Herlihy, M. Kapalka, B. Pochon // Proceedings of the OOPSLA Workshop on Synchronization and Concurrency in Object-Oriented Languages (SCOOL). — 2005. — October.

[18] *Scherer III, W. N.* Advanced contention management for dynamic software transactional memory / W. N. Scherer III, M. L. Scott // Proceedings of the 24th ACM Symposium on Principles of Distributed Computing. — Las Vegas, NV: 2005. — Jul.

A Примеры исходного кода на языке C#

Листинг 1: Фиксация в блокирующей реализации

```
internal void Commit()
{
    if (this.written.Count == 0)
    {
        status = Txn.COMMITED;
        return;
    }
    var desiredStatus = Txn.RUNNING;
    var acquired = new List<KeyValuePair<Obj, Ver>>(this.written.Count);
    foreach (var pair in written)
    {
        int i = 0;
        while (i < TM.acquireRetries)
        {
            var obj = pair.Key;
            var myVer = pair.Value;
            var origVer = myVer.orig;
            var currentVer = Interlocked.CompareExchange(ref obj.ver, this,
                origVer);
            if (ReferenceEquals(currentVer, origVer))
            {
                acquired.Add(pair);
                break;
            }
            else if (currentVer is Ver)
            {
                desiredStatus = Txn.ABORTED;
                goto decisionPoint;
            }
            ++i;
        }
        if (i == TM.acquireRetries)
        {
            desiredStatus = Txn.ABORTED;
            goto decisionPoint;
        }
    }
    this.status = Txn.READ_CHECK;
    this.commitTime = Interlocked.Increment(ref TM.time);
    foreach (var pair in read)
    {
        var obj = pair.Key;
        var myVer = pair.Value;
        if (ReferenceEquals(myVer, null))
            continue;
        var currentVer = obj.ver;
        if (currentVer is Txn)
        {
            var owner = (Txn)currentVer;
            int i = 0;
            while (owner.status == Txn.READ_CHECK)
            {
                if (++i > TM.readRetries)
                {
                    desiredStatus = Txn.ABORTED;
                    goto decisionPoint;
```

```
56                  }
57              }
58              var ownedVer = owner.written[obj];
59              if (owner.status == Txn.COMMITED)
60                  currentVer = ownedVer;
61              else
62                  currentVer = ownedVer.orig;
63          }
64          if (!ReferenceEquals(currentVer, myVer))
65          {
66              desiredStatus = Txn.ABORTED;
67              goto decisionPoint;
68          }
69      }
70      desiredStatus = Txn.COMMITED;
71      foreach (var pair in acquired)
72          pair.Value.time = this.commitTime;
73  decisionPoint:
74      int eventualStatus = -1;
75      this.status = desiredStatus;
76      eventualStatus = desiredStatus;
77      if (eventualStatus == Txn.COMMITED)
78      {
79          foreach (var pair in acquired)
80              Interlocked.CompareExchange(ref pair.Key.ver, pair.Value, this);
81      }
82      else
83      {
84          foreach (var pair in acquired)
85              Interlocked.CompareExchange(ref pair.Key.ver, pair.Value.orig, this)
                    ;
86          Rollback1();
87      }
88  }
```

Листинг 2: Получение текущей версии в блокирующей реализации

```
1   internal Ver ReadVer(Txn txn)
2   {
3       var ver = this.ver;
4       if (ver is Ver)
5           return (Ver)ver;
6       else
7       {
8           var owner = (Txn)ver;
9           while (owner.status == Txn.READ_CHECK)
10          {
11          }
12          var ownedVer = owner.written[this];
13          if (owner.status == Txn.COMMITED)
14              return ownedVer;
15          else
16              return ownedVer.orig;
17      }
18  }
```

Листинг 3: Фиксаиця в неблокирующей реализации

```
1   internal void Commit(CommitMode mode = CommitMode.Normal)
2   {
3       if (this.written.Count == 0)
```

```
4       {
5           status = Txn.COMMITED;
6           return;
7       }
8       var desiredStatus = Txn.RUNNING;
9       var acquired = new List<KeyValuePair<Obj, Ver>>(this.written.Count);
10      var sortedWritten = this.written.ToList();
11      sortedWritten.Sort(_writeSetComparer);
12      foreach (var pair in sortedWritten)
13      {
14          while (true)
15          {
16              var obj = pair.Key;
17              var myVer = pair.Value;
18               var origVer = myVer.orig;
19              var currentVer = Interlocked.CompareExchange(ref obj.ver, this,
                    origVer);
20              if (ReferenceEquals(currentVer, origVer))
21              {
22                  acquired.Add(pair);
23                  break;
24              }
25                      else if(ReferenceEquals(currentVer, this))
26                  break;
27              else if (currentVer is Ver)
28              {
29                  desiredStatus = Txn.ABORTED;
30                  goto decisionPoint;
31              }
32              else
33              {
34                  var owner = (Txn)currentVer;
35                  owner.Commit(CommitMode.Help);
36              }
37          }
38      }
39      Interlocked.CompareExchange(ref this.status, Txn.READ_CHECK, Txn.RUNNING);
40      var newTime = Interlocked.Increment(ref TM.time);
41      Interlocked.CompareExchange(ref this.commitTime, newTime, -1);
42      foreach (var pair in read)
43      {
44          var obj = pair.Key;
45          var myVer = pair.Value;
46          if (ReferenceEquals(myVer, null))
47              continue;
48          var currentVer = obj.ReadVer(this);
49          if (!ReferenceEquals(currentVer, myVer))
50          {
51              desiredStatus = Txn.ABORTED;
52              goto decisionPoint;
53          }
54      }
55      desiredStatus = Txn.COMMITED;
56      foreach (var pair in acquired)
57          Interlocked.CompareExchange(ref pair.Value.time, commitTime, -1);
58  decisionPoint:
59      int eventualStatus = -1;
60  while (((eventualStatus = this.status) != Txn.ABORTED) && (eventualStatus != Txn
        .COMMITED))
61      Interlocked.CompareExchange(ref this.status, desiredStatus, eventualStatus);
```

```
62      if (eventualStatus == Txn.COMMITED)
63      {
64          foreach (var pair in acquired)
65              Interlocked.CompareExchange(ref pair.Key.ver, pair.Value, this);
66      }
67      else
68      {
69          foreach (var pair in acquired)
70              Interlocked.CompareExchange(ref pair.Key.ver, pair.Value.orig, this)
                    ;           if (mode == CommitMode.Normal)
71              Rollback1();
72      }
73  }
```

Листинг 4: Получение текущей версии в неблокирующей реализации

```
1   internal Ver ReadVer(Txn txn)
2   {
3       var ver = this.ver;
4       if (ver is Ver)
5           return (Ver)ver;
6       else
7       {
8           var owner = (Txn)ver;
9           if (owner.status == Txn.READ_CHECK)
10          {
11              if ((txn.status != Txn.READ_CHECK) || (txn.CompareTo(owner) > 0))
12                  owner.Commit(CommitMode.Help);
13              else
14              {
15                  Interlocked.CompareExchange(ref owner.status, Txn.ABORTED, Txn.
                        READ_CHECK);
16              }
17          }
18          var ownedVer = owner.written[this];
19          if (owner.status == Txn.COMMITED)
20              return ownedVer;
21          else
22              return ownedVer.orig;
23      }
24  }
```