СПИСОК ЛИТЕРАТУРЫ

1. The Internet of Things. URL: http://www.theinternetofthings.eu/.

2. Как интернет вещей меняет современную медицину. URL: https://meduza.io/feature/2015/12/14/bolnitsy-bez-vrachey-i-umnoe-mylo.

3. W3C. SemanticWeb. URL: http://www.w3.org/standards/semanticweb/.

4. Deep learning applications and challenges in big data analytics / M. M. Najafabadi, F. Villanustre, T. M. Khoshgoftaar, N. Seliya, R. WaldEmail, E. Muharemagic // J. of Big Data. 2015. № 2. URL: http://journalofbigdata.springeropen.com/articles/10.1186/s40537-014-0007-7.

5. Медицинский центр им. В. А. Алмазова. URL: http://www.almazovcentre.ru/.

6. InterSystems. URL: http://www.intersystems.com

D. A. Korobov, M. V. Lapaev
ITMO University (Saint Petersburg National Research University of Information Technologies, Mechanics and Optics)

A. I. Vodyaho, N. A. Zhukova
Saint Petersburg Electrotechnical University «LETI»

DATA MODELS FOR MEDICAL DOMAIN

A multilevel information model for adaptive information systems for data processing and analysis is discussed. The model is to be used for building of real world object models taking into account operational and historical data. The model has 3 levels of abstraction: abstract conceptual model, model of subject domain and models of target system. In the article problems of usage of this model in medicine domain are discussed. The models are realized as ontologies. Suggested approach was tested while building the system of medical data processing for Almazov Center in Saint Petersburg

Information models, medical data processing and analysis, deep learning, semantic technologies

УДК 681.3

М. Ф. Галимуллин, Е. Л. Калишенко,
Санкт-Петербургский государственный электротехнический университет «ЛЭТИ» им. В. И. Ульянова (Ленина)

Н. А. Рапоткин
АО «Информационно-телекоммуникационные технологии» (Санкт-Петербург)

Анализ производительности стратегий синхронизации потоков в структурах данных, основанных на flat-combining

Рассматриваются некоторые сценарии использования конкурентных структур данных, показывающие повышение производительности при увеличении времени работы одного потока, которому остальные потоки делегируют свои задачи. Данный подход получил название flat-combining (FC) [1]. Представлены несколько разработанных стратегий синхронизации, описаны их преимущества и область применения.

Конкурентные структуры данных, анализ производительности, flat-combining, многопоточность

Идея подхода flat-combining (на примере стека) заключается в следующем (рис. 1): со стеком связан мьютекс и список анонсов (publication list) размером, пропорциональным количеству потоков, работающих со стеком. Каждый поток при первом обращении к стеку добавляет в список анонсов свою запись (*1* на рис. 1). Если потоку необходимо выполнить операцию над контейнером, то он публикует в своей записи запрос – операцию, например push или pop для стека, и ее аргументы – и пытается захватить мьютекс. Если мьютекс захвачен, поток становится комбайнером

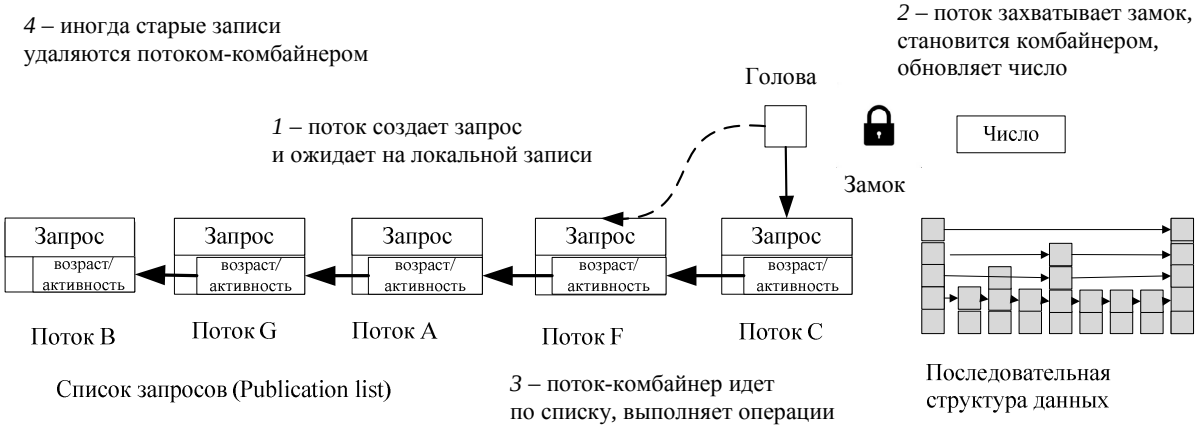

Рис. 1

(*англ.* combiner) (*2* на рис. 1), он просматривает список аннотаций, выполняет все запросы из него, записывает результат в элементы списка аннотаций и, наконец, освобождает мьютекс (*3* на рис. 1). Если же попытка захвата мьютекса не удалась, поток ожидает (spinning) на своем анонсе, когда комбайнер выполнит его запрос и поместит результат в запись аннотации.

Список анонсов построен таким образом, чтобы уменьшить накладные расходы на управление им. Ключевым моментом является то, что список анонсов редко изменяется (*4* на рис. 1), иначе возникнет ситуация, когда помимо управления доступом к последовательной структуре данных необходимо управлять доступом и целостностью lock-free publication list, что вряд ли как-то ускорит доступ к последовательной структуре данных. Запросы на операцию вносятся в уже существующую запись списка анонсов, которая является собственностью потоков и находится в TLS. Для того чтобы упростить управление lock-free-списком [2], голова списка не меняется и является фиктивным элементом, не относящимся ни к одному потоку, существующему внутри ядра FC, служащим лишь точкой входа. Новые записи добавляют строго в конец списка.

Некоторые записи списка могут иметь статус «empty», означающий, что соответствующий поток не выполняет в настоящий момент никаких действий с последовательной структурой данных. Время от времени комбайнер прореживает список анонсов, исключая из него записи потоков, которые были не активны в течение определенного времени, точнее, в течение определенного числа проходов потока-комбайнера. Тем самым снижаются затраты времени на обработку пустых аннотаций (давно неактивных потоков). При этом вместо фактического удаления происходит логическое удаление записи из списка, т. е. записи помечаются как «inactive» и тем самым исключаются из обработки потоком-комбайнером. Физическое удаление происходит гораздо реже, тем самым уменьшая накладные расходы на управление lock-free-списком, но все же контролируя размер списка, не давая ему разрастись.

Стратегии синхронизации. Однако активное ожидание выполнения задач потоками мешает работе потока-комбайнера, также потребляя ресурсы процессора. Для уменьшения влияния ожидающих потоков на работу потока-комбайнера можно использовать различные стратегии синхронизации. Выбранные для разработки и исследования стратегии основаны на механизме wait/notify и отличаются конфигурацией использования мьютекса и условной переменной. Стандартный алгоритм работы wait/notify, часто используемый для реализации шаблона Publish/Subscriber (Производитель/Потребитель), состоит из следующих шагов:

• Поток-потребитель ожидает, пока какое-то условие не станет истинным (в данном случае это поток, анонсирующий свою операцию и ожидающий ее завершения). Ожидающий поток должен захватить примитив синхронизации. Эта блокировка передается методу wait(), который освобождает мьютекс и приостанавливает поток, пока не будет получен сигнал от условной переменной. Когда это произойдет, поток пробудится и снова выполнится захват примитива синхронизации.

FC Kernel
global_mutex : Mutex waitStrategyMutex : boost::mutex waitStrategyCondVar: boost::cond_var
...

Рис. 2

• Поток-производитель сигнализирует о том, что условие стало истинным (в данном случае это поток-комбайнер, выполняющий анонсированную операцию и выставляющий флаг завершенности операции).

Рассмотрим реализованные стратегии более подробно и проведем их сравнительный анализ.

Back-off-стратегия. Данная стратегия используется в оригинальной реализации flat-combining в библиотеке libcds [3]. Когда N потоков борются за критический ресурс, доступ к которому организован при помощи CAS-операций, только один из них получит к нему доступ, остальные $N - 1$ будут впустую потреблять процессорное время и мешать друг другу. Чтобы разгрузить процессор при обнаружении такой ситуации, потоки могут отступить (back off) от выполнения основной задачи и сделать что-либо полезное или просто подождать. Именно для этого предназначены back-off-стратегии.

Использование back-off-стратегии, реализованной в виде задержки на 2 мс на каждой итерации цикла, согласно докладу [4] (рис. 2), значительно повысило производительность FC и позволило показать хорошие результаты на высоких нагрузках.

Wait/notify-стратегия на основе глобального мьютекса и условной переменной (рис. 2). Данная стратегия основана на использовании примитивов синхронизации, агрегированных в ядре FC и разделяемых всеми потоками.

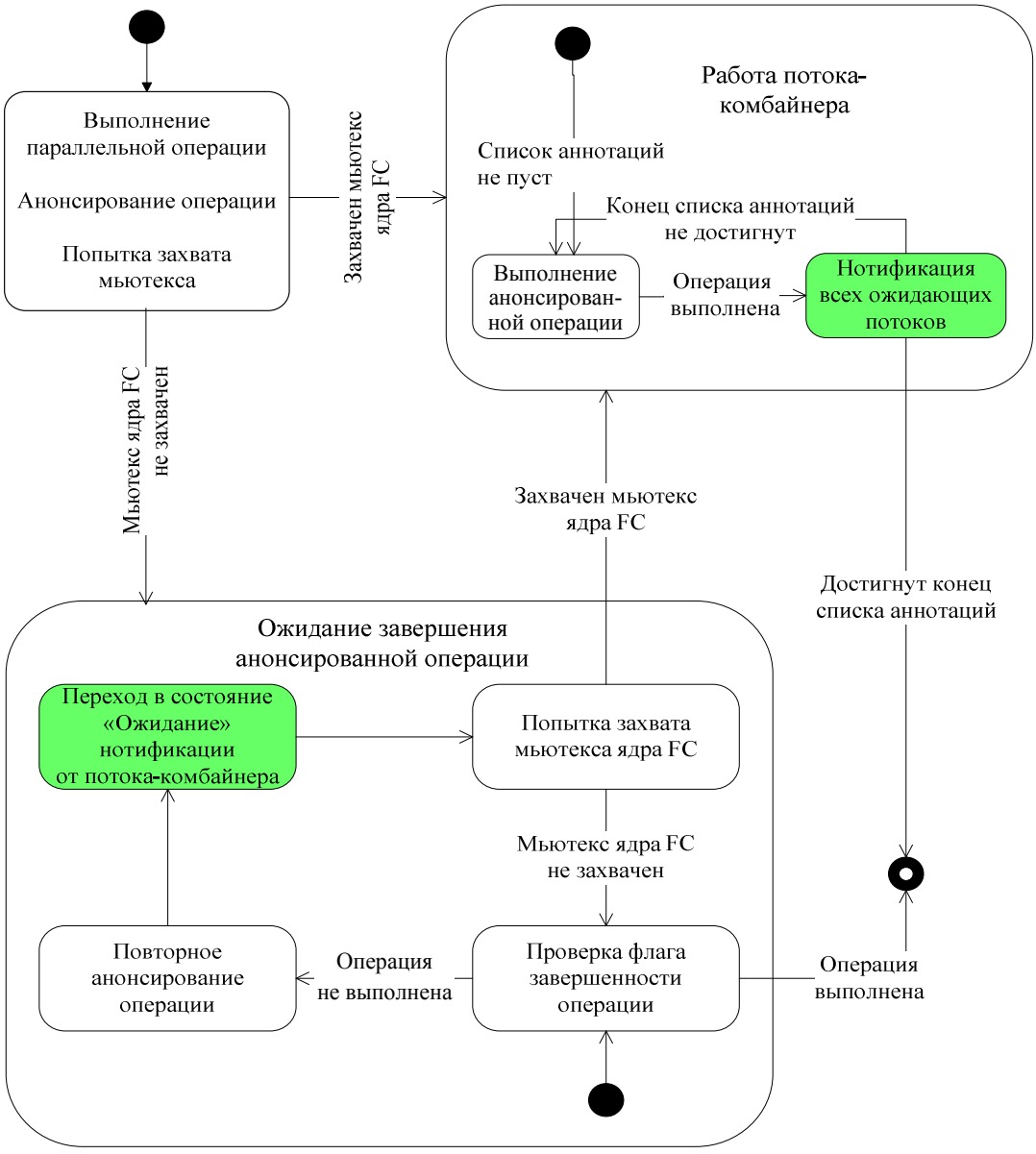

Рис. 3

Алгоритм 1. Алгоритм взаимодействия потоков заключается в следующем (рис. 3):

• Если какой-то поток хочет выполнить операцию над контейнером, он публикует запись и пытается захватить мьютекс ядра FC.

• Если потоку не удается захватить мьютекс ядра FC и стать комбайнером, то поток ожидает уведомления о выполнении операции от потока-комбайнера на условной переменной, разделяемой всеми потоками.

• Если потоку удалось захватить мьютекс ядра FC, то поток становится комбайнером. Обрабатывая каждую из анонсированных записей, он уведомляет об этом все потоки, ожидающие выполнения анонсированной ими операции.

• Поток, получивший уведомление о завершении какой-то операции, переходит в состояние «Готовность» и попадает на обработку планировщику. После перехода в состояние «Выполнение» и фактического возобновления работы поток проверяет флаг завершенности своей операции. Если его операция по-прежнему не выполнена, поток снова засыпает в ожидании следующего уведомления.

Wait/notify-стратегия на основе глобального мьютекса и локальной для потока условной переменной (рис. 4). Модификацией стратегии, описанной выше, является стратегия, использующая один дополнительный мьютекс, агрегированный в ядре FC, и по условной переменной для каждого потока, агрегированных в каждой записи (*англ.* publication record) списка анонсов (*англ.* publication list), как показано на рис. 4. Данная модификация должна минимизировать в худшем и исключить вовсе в лучшем случаях избыточные итерации в цикле ожидания выполнения анонсированной операции.

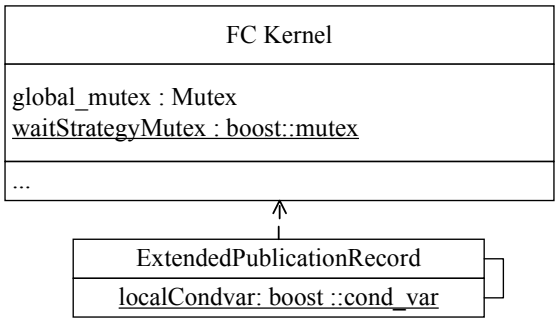

Рис. 4

Алгоритм 2. Алгоритм взаимодействия потоков заключается в следующем (рис. 5):

• Если какой-то поток хочет выполнить операцию над контейнером, он публикует запись и пытается захватить мьютекс ядра FC.

• Если потоку не удается захватить мьютекс ядра FC и стать комбайнером, то поток ожидает сигнала от потока-комбайнера на условной переменной, агрегированной в анонсированной записи.

• Если потоку удалось захватить мьютекс ядра FC, то поток становиться комбайнером. Обрабатывая каждую из анонсированных записей, он уведомляет об этом только тот поток, которому принадлежит обработанная запись, при помощи условной переменной, агрегированной в этой записи.

• Поток, получивший уведомление о завершении его операции, возобновляет работу и проверяет флаг завершенности своей операции. Если анонсированная операция не выполнена, поток снова засыпает.

Несмотря на то что в этой стратегии, в отличие от предыдущей, осуществляется нотификация только одного потока, которому принадлежит обработанная запись, все равно необходимо после возобновления работы потока проверять флаг завершенности операции, так как существует вероятность ложного пробуждения потоков (*англ.* spurious wakeup). Дело в том, что существует механизм предотвращения live и dead-lock'ов средствами ОС. Если операционная система предполагает, что потоки находятся в live и dead-lock'e, то она может разбудить все потоки, для того чтобы перегруппировать потоки и дать возможность другим потокам захватить примитив синхронизации.

Кроме того, наличие глобального мьютекса приводит к последовательной обработке потоками событий завершения запрошенных ими операций, что, опять же, может сказаться на производительности в целом.

Wait/notify-стратегия на основе локальных для потока мьютекса и условной переменной (рис. 6). Последняя из рассматриваемых стратегий синхронизации также основана на связке mutex/conditional variable, но локальных для каждого потока. Данная стратегия схожа с алгоритмами тонкой синхронизации конкурентных структур данных [5]. Стратегия должна свести к нулю количество избыточных итераций в цикле ожидания завершения анонсированной операции, а также уменьшить конкурентную борьбу потоков за примитив синхронизации стратегии. Однако эта стратегия требует большего объема памяти, отводимого для анонсируемой записи, нежели предыдущая, в связи с необходимостью агрегировать внутри записи 2 примитива синхронизации.

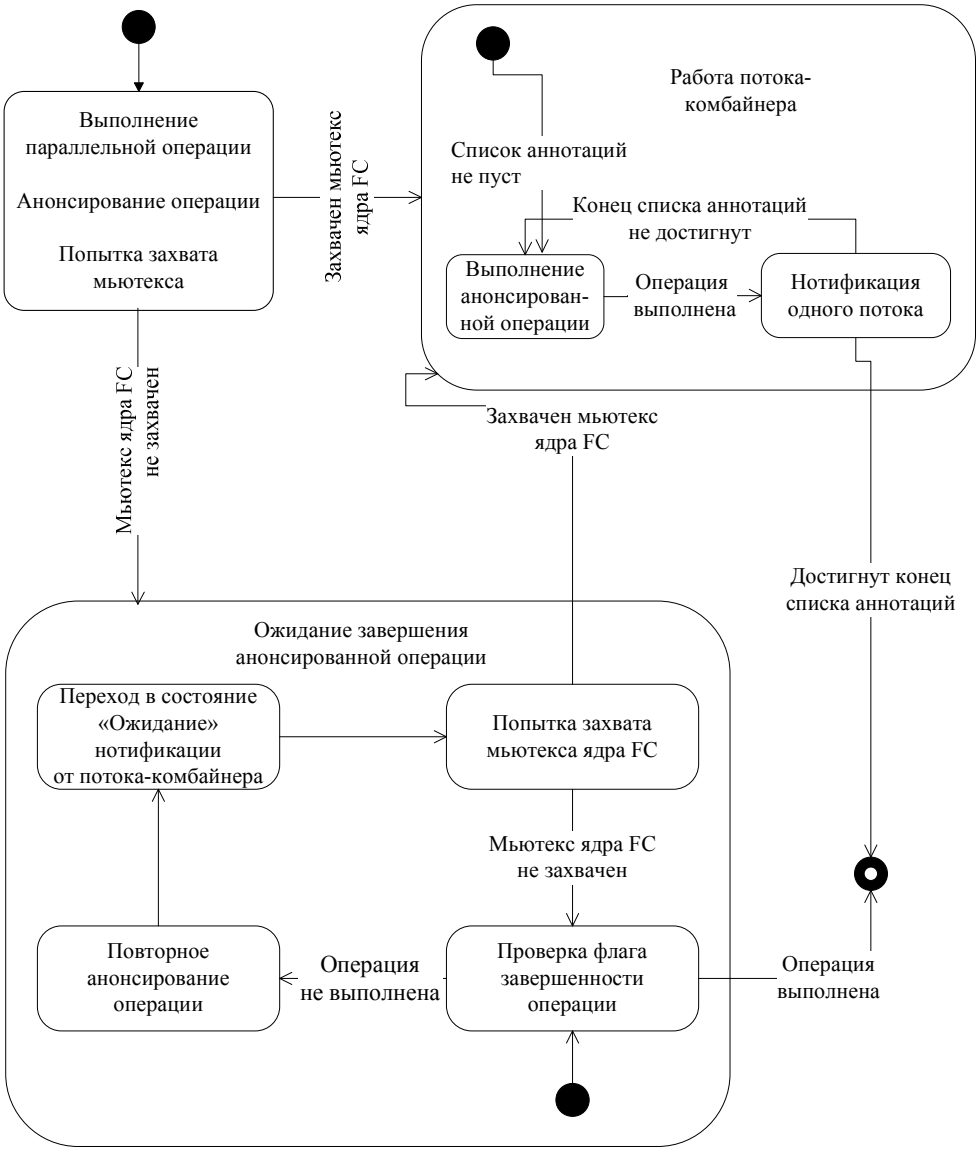

Рис. 5

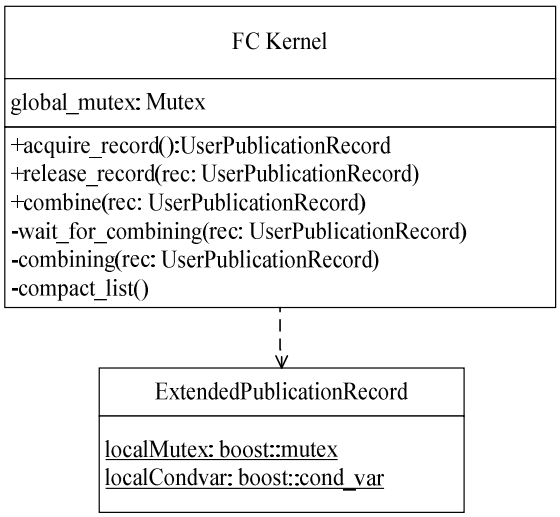

Рис. 6

Алгоритм 3. Алгоритм взаимодействия потоков заключается в следующем:

- Если какой-то поток хочет выполнить операцию над контейнером, он публикует запись и пытается захватить мьютекс ядра FC.
- Если потоку не удается захватить мьютекс ядра FC и стать комбайнером, то поток ожидает сигнала от потока-комбайнера на условной переменной, агрегированной в анонсированной записи.
- Если потоку удалось захватить мьютекс ядра FC, то поток становиться комбайнером. Обрабатывая каждую из анонсированных записей, он нотифицирует об этом лишь тот поток, которому принадлежит обработанная запись.
- Поток, ожидающий выполнения своей операции, получивший уведомление о завершении его операции, возобновляет свою работу и прове-

ряет флаг завершенности своей операции, чтобы избежать ситуации, описанной ранее.

Диаграмма состояний и переходов алгоритма работы описанной стратегии синхронизации представлена на рис. 5.

Адаптивная стратегия (рис. 7). В ходе тестов, описанных далее, было выяснено, что стратегии ведут себя по-разному при работе с контейнерами, наполненными «тяжелыми» или «легкими» элементами. Для автоматизации выбора стратегии была разработана обобщенная стратегия на базе шаблона метапрограммирования Int2Type [6].

Стратегия работает как back-off, если контейнер содержит «легкие» элементы, или как MMMCV-стратегия, если контейнер содержит «тяжелые» элементы. Выбор стратегии происходит на этапе компиляции.

Реализация стратегий синхронизации. Рассмотренные ранее стратегии синхронизации реализованы в библиотеке «libcds» на основе шаблона проектирования «стратегия» [7]; SMSCV – Single Mutex Single Conditional Variable; SMMCV – Single Mutex Multiple Conditional Variable; MMMCV – Multiple Mutex Multiple Conditional Variable (рис. 7).

Алгоритм 4. Описание алгоритмов синхронизации (рис. 8):

• При создании объекта ядра FC создается объект класса выбранной стратегии.

• Происходит расширение пользовательской записи (publication record) до ExtendedPublicationRecord посредством одиночного открытого наследования.

• Объект класса ExtendedPublicationRecord используется в ядре FC.

Тестирование стратегий синхронизации. Для тестирования разработанных стратегий синхронизации необходимо построить конкурентную

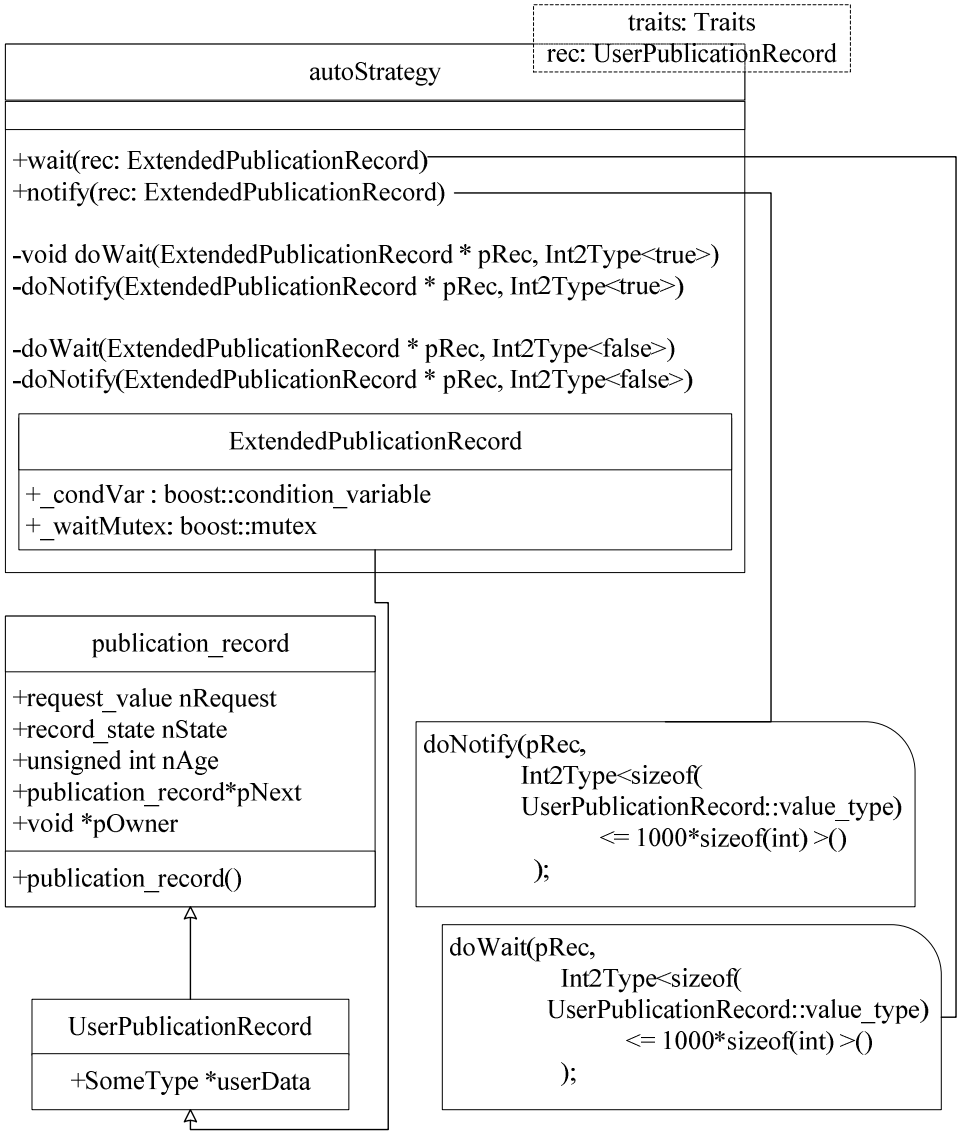

Рис. 7

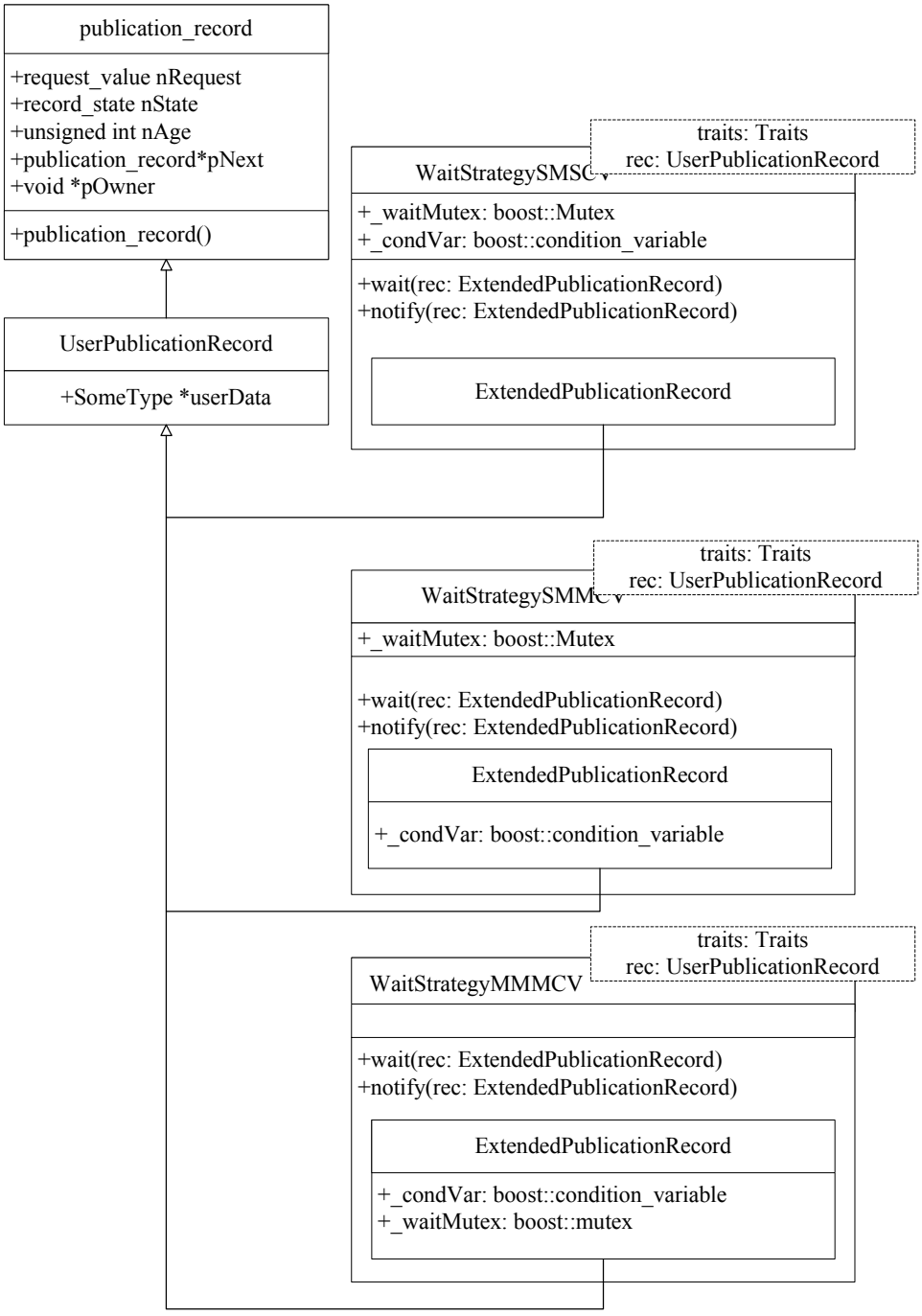

Рис. 8

структуру данных в рамках подхода FC, например конкурентную очередь (FIFO).

Для этого необходимо реализовать класс, являющийся наследником класса cds::algo::flat_combining::container, определенного в ядре FC, для реализации интерфейса взаимодействия с ядром FC, а также агрегирующий в себе объект ядра FC и объект стандартной последовательной очереди, например std::queue. При этом объект ядра FC инстанцируется расширенной пользователем анонсируемой записью fc_record и одной из реализованных стратегий синхронизации.

Для тестирования реализованных стратегий были построены следующие тесты:

– reader/writer – половина потоков осуществляет только добавление элементов в контейнер, а вторая половина – только удаление;

– random – случайная последовательность push/pop-операций;

– pop – преобладают операции добавления элементов в контейнер;

– push – преобладают операции удаления элементов из контейнера.

Анализ эффективности использования реализованных стратегий основывается на измерении следующих величин:

– duration – среднего времени выполнения операций push/pop (ms/op). Данный параметр является основным показателем эффективности использования разработанных стратегий синхронизации, так как показывает быстродействие работы FC в целом;

– combining factor – отношения количества выполненных операций к количеству вызовов метода комбайнера. Другими словами, combining factor определяет эффективность FC в целом: чем большее число операций выполняется одним комбайнером и чем реже меняется поток-комбайнер, тем эффективней используются вычислительные ресурсы ядра процессора;

– redundant iterations – среднего количества избыточных итераций в цикле при ожидании выполнения анонсированной операции. Основной задачей разработанных стратегий является минимизация именно данного параметра, так как, во-первых, каждая итерация цикла содержит обращение к примитиву синхронизации ядра FC; во-вторых, поток находится в активном состоянии и потребляет ресурсы процессора, тем самым мешая потоку-комбайнеру и увеличивая накладные расходы на управление потоками планировщиком.

Данный набор параметров позволяет наиболее полно оценить эффективность использования разработанных стратегий синхронизации с точки зрения производительности и потребления ресурсов.

Результаты тестирования очереди с «легкими» элементами.

Структура «легкого» элемента:

```
struct SimpleValue {
 size_t nNo;
 size_t nThread;
}
```

Как видно из рис. 9, разработанные стратегии даже при жесткой конкурентной борьбе уступают оригинальной реализации и реализации с использованием back-off-стратегии в производительности, несмотря на то, что в худшем случае имеют в 10 раз меньше избыточных операций, а в лучшем – в 10^7 раз (рис. 10).

Копирование восьми байт памяти занимает ничтожно малое время, поэтому операции pop/push с такой структурой происходят намного быстрее, чем переключение контекста и управление состояниями потоков. В связи с этим механизм wait/notify в разработанных стратегиях только мешает работе FC.

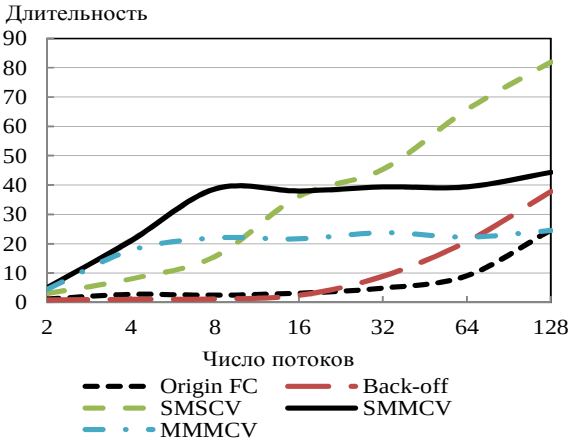

Рис. 9

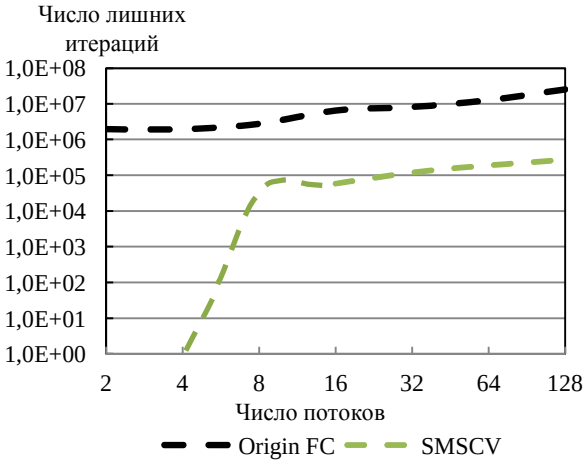

Рис. 10

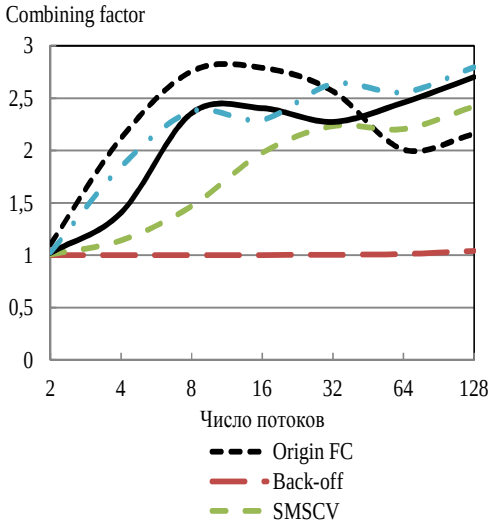

Рис. 11

Если проанализировать значения combining factor при выполнении FC с использованием back-off-стратегии (рис. 11), становится понятной причина таких высоких результатов: среднее значение combining factor для back-off-стратегии не превышает 1.1. Это значит, что почти каждый поток, анонсировавший свою запись, становится

потоком-комбайнером и успевает выполнить, чаще всего только свою операцию. Это также подтверждается отсутствием избыточных операций (redundant iterations = 0), т. е. параллельная работа потоков сводится к последовательному выполнению операций над контейнером.

Результаты тестирования очереди с «тяжелыми» элементами. Чтобы избежать ситуации, описанной ранее, был построен тест на основе параллельной очереди с «тяжелыми» структурами в качестве элементов этой очереди вида

```
    struct HeavyValue {
    size_t nNo;
    size_t nWriterNo;
    static int pop_buff[1000];

    HeavyValue() :nNo(0), nWriterNo(0){}
    size_t getNo() const { return  nNo;
}
    };
```

Как видно из графиков длительности операций (рис. 12), разработанные стратегии показывают гораздо лучшие результаты при «тяжелом» тестировании, особенно при большом числе потоков и жесткой конкурентной борьбе за ресурс. Это связанно с тем, что в данном случае время переключения контекста мало по сравнению со временем выполнения операции и слабо влияет на производительность в целом, в отличие от случая со списком «легких» элементов.

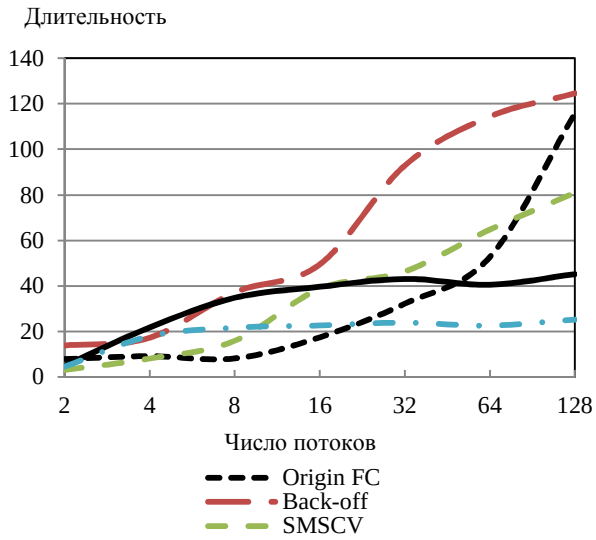

Рис. 12

Производительность реализации оригинального алгоритма FC, как и реализации с использованием back-off-стратегии, значительно проигрывает разработанным стратегиям, особенно при большом числе потоков, в связи с частым обращением к примитиву синхронизации и использованием ресурсов процессора, что повышает накладные расходы планировщика на управление потоками.

Как видно из рис. 13, число избыточных операций при тестировании реализации оригинального алгоритма FC (график Origin FC) и реализации с использованием back-off-стратегии (график Back-off) в худшем случае больше на порядок, а в лучшем – в 10^9 раз.

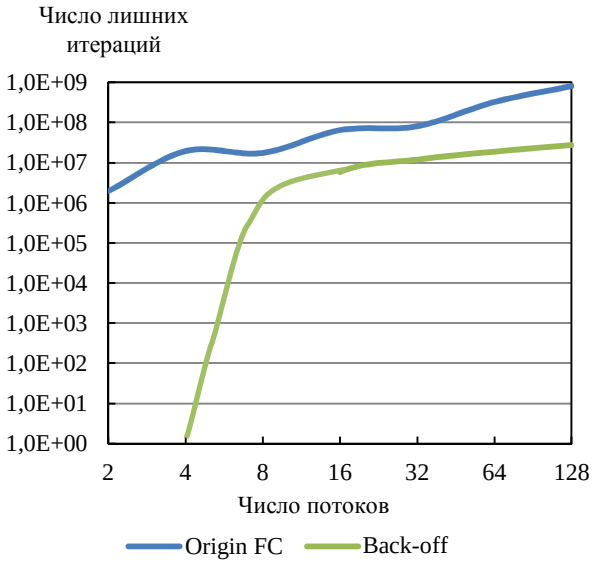

Рис. 13

Из анализа полученных результатов следует, что разработанные стратегии синхронизации на основе механизма wait/notify эффективны при построении контейнеров с элементами большого размера (сопоставимыми по размеру с sizeof(int)*1000) и алгоритмов, требующих длительного выполнения операций.

Использование разработанных стратегий с контейнерами, содержащими простые элементы, напротив, крайне неэффективно в связи с большими накладными расходами на частое переключение контекста и управление состояниями потоков, которые много больше времени выполнения единичной операции над параллельной структурой данных.

Наилучшие результаты при тестировании показала стратегия, основанная на агрегировании мьютекса и условной переменной в каждой анонсируемой записи. Это обусловлено тем, что захваты агрегированных мьютексов происходят параллельно в отличие от других двух стратегий с одним, разделяемым всеми потоками мьютексом. Multiple Mutex Multiple Conditional Variable-стратегия благодаря агрегированию условной пере-

менной в каждой анонсируемой записи и детерминированной нотификации исключает избыточные итерации в цикле ожидания завершения анонсируемой операции.

Таким образом, в данной статье представлены некоторые аспекты FC и стратегии синхронизации, описаны некоторые проблемы, касающиеся эффективности разработанных стратегий, зависящих от размера элемента контейнера. Но некоторую работу еще стоит провести. Прежде всего адаптивную стратегию следует проанализировать более тщательно, принимая во внимание возможность работы с другими контейнерами, а не только с элементами разного размера. Интересно проанализировать некоторые существующие системы, которые используют методы FC, например программную платформу эмуляции единого адресного пространства в распределенной системе [8], а также применить реализацию адаптивной стратегии.

СПИСОК ЛИТЕРАТУРЫ

1. Flat Combining and the Synchronization-Parallelism Tradeoff / D. Hendler, I. Incze, N. Shavit, M. Tzafrir //Proc. of the 22nd Annual ACM Symp. on Parallelism in Algorithms and Architectures, Thira, Santorini, June 13–15, 2010.

2. Herlihy M., Shavit N. The Art of Multiprocessor Programming. San Francisco: Morgan Kaufmann, 2012.

3. Хижинский М. Разработка Lock-free структуры данных. Эволюция стека. URL: http://habrahabr.ru/company/ifree/blog/216013/.

4. Хижинский М. C++ developers meeting. URL: http://video.yandex.ru/users/ya-events/view/2932.

5. Gamma E., Johnson R., Vlissides J. Design Patterns: Elements of Reusable Object-Oriented Software. Boston: Addison-Wesley Professional, 1994.

6. Alexandrescu A. Modern C++ Design: Generic Programming and Design Patterns Appl. Boston: Addison-Wesley Professional, 2001.

7. Vandevoorde D., Josuttis N. C++ Templates: The Complete Guide. Boston: Addison-Wesley Professional, 2002.

8. Flat Combining Synchronized Global Data Structures / B. Holt, J. Nelson, B. Myers, P. Briggs, L. Ceze, S. Kahan, M. Oskin // Proc. of 7th Intern. Conf. on PGAS Programming Models, Edinburgh, 2013.

M. F. Galimullin, E. L. Kalishenko
Saint Petersburg Electrotechnical University «LETI»

N. A. Rapotkin
Department of Software Engineering and Computer Applications (Saint Petersburg)

PERFORMANCE ANALYSIS OF THREAD SYNCHRONIZATION STRATEGIES IN CONCURRENT DATA STRUCTURES BASED ON FLAT-COMBINING

Deals with the development of threads synchronizing strategies based on the creation of concurrent «flat-combining» data structures as well as research of their performance. The paper considers «flat-combining» approach and its implementation in the library libcds, the development of thread synchronization strategy and its possible implementations. The efficiency of synchronization strategies usage is researched on the example of the open source library libcds. The research revealed the strategy with the lowest operation execution time on a container and the lowest amount of CPU resources, and identifies use cases of the developed strategies. A mechanism with the developed synchronization strategy to build concurrent data structures was implemented. The implemented strategies were integrated in the cross-platform open source library libcds.

Multithreading, performance analysis, flat-combining, concurrency