Санкт-Петербургский государственный университет
Математическое обеспечение и администрирование
информационных систем

Системное программирование

Моисеенко Евгений Александрович

Библиотека параллельной сборки мусора для C++

Бакалаврская работа

Научный руководитель:
к.ф.-м.н. Булычев Д. Ю.

Рецензент:
асп. Березун Д. А.

Санкт-Петербург
2016

SAINT-PETERSBURG STATE UNIVERSITY
Information Systems Administration and Mathematical Support

Software Engineering

Moiseenko Evgenii

Parallel garbage collection library for C++

Graduation Thesis

Scientific supervisor:
Dmitri Boulytchev

Reviewer:
Daniil Berezun

Saint-Petersburg
2016

Оглавление

Введение **4**

Постановка задачи **6**

1. Обзор **7**

1.1. Обзор алгоритмов сборки мусора 7

1.1.1. Подсчёт ссылок 7

1.1.2. Трассирующая сборка мусора 8

1.1.3. Сжимающая сборка мусора 10

1.1.4. Инкрементальная параллельная маркировка . . . 11

1.2. Сборка мусора в C++ 13

1.3. Существующие решения 14

2. Анализ предыдущей версии библиотеки **17**

3. Реализация **24**

3.1. Остановка мира 24

3.1.1. Сигналы ОС Linux 25

3.1.2. Блокирование сигналов 25

3.1.3. Асинхронно-сигналобезопасные примитивы синхронизации 27

3.1.4. Алгоритм остановки мира 30

3.2. Закрепление объектов 30

3.3. Инкрементальная параллельная маркировка 32

4. Апробация **35**

Результаты **40**

Список литературы **43**

Введение

C++ является популярным языком общего назначения, разработанным в 1980-х годах как улучшение широко используемого языка C. Главным новшеством языка было введение объектно-ориентированных конструкций, позволявшей моделировать предметную область программы на языке объектов. С самого начала C++ разрабатывался с учетом высоких требований к производительности, таким образом, чтобы введение новых языковых конструкций не нарушало обратную совместимость и не накладывало дополнительных накладых расходов (принцип "don't pay for what you don't use"). C++ используется в приложениях, где критичными являются требования ко времени выполнения, потребляемой памяти, размеру исполняемых файлов. Среди таких приложений можно выделить финансовые и банковские системы (High-Frequency Trading), приложения, активно работающие с графикой (графические редакторы, системы виртуальной реальности, компьютерные игры), программы, работающие на встраиваемых платформах (Embedded System)[1].

Одним из основных ресурсов приложения является память. Большинство современных языков программирования активно использует *динамическое распределение памяти*, при котором выделение памяти осуществляется во время исполнения программы. Динамическое управление памятью вводит два основных примитива — функции выделения и освобождения блоков памяти. Существуют два способа управления динамической памятью — *ручное* и *автоматическое*. При *ручном управлении памятью* программист должен следить за освобождением выделенной памяти, что приводит к возможности возникновения труднообнаружимых ошибок. Более того, в некоторых ситуациях (например, при программировании на функциональных языках или в многопоточной среде) время жизни объекта не всегда очевидно для разработчика. *Автоматическое управление памятью* избавляет программиста

[1] С обзором современного использования C++ можно ознакомиться здесь: `http://blog.jetbrains.com/clion/2015/07/infographics-cpp-facts-before-clion/`

от необходимости вручную освобождать выделенную память, устраняя тем самым целый класс возможных ошибок и увеличивая безопасность исходного кода программы. *Сборка мусора* (garbage collection) давно стала стандартом в области автоматического управления памятью, хотя её использование может накладывать дополнительные расходы по памяти и времени исполнения.

На сегодняшний день среды времени выполнения многих популярных языков программирования, таких как Java, C#, Python, Ruby и другие, активно используют сборку мусора. Язык C++ разрабатывался с расчетом на использование ручного управления памятью. Некоторые языковые возможности, такие как адресная арифметика и приведение типов указателей, затрудняют реализацию сборщиков мусора для этого языка. Несмотря на это, существует ряд подходов к сборке мусора для языка C++.

В рамках проекта лаборатории JetBrains была реализована сборка мусора для языка C++ [1, 13, 15] на уровне библиотеки. Сборщик приостанавливает работу приложения на все время сборки мусора, из-за чего в работе программы могут возникать длительные паузы. Продолжительные остановки на сборку мусора неприемлимы для целого класса приложений, для которых важна низкая латентность, то есть малое время отклика на запросы пользователей. Существуют подходы, призванные уменьшить время паузы на сборку мусора, в частности инкрементальная параллельная маркировки. Идея данного подхода заключается в том, чтобы выполнять часть работы по сборке мусора параллельно с приложением, без его полной остановки.

Постановка задачи

В данной работе были поставлены следующие задачи:

- проанализировать ограничения и недостатки текущей версии библиотеки сборки мусора;
- предложить и реализовать новые методы для модернизации библиотеки;
- реализовать и встроить в библиотеку алгоритм инкрементальной параллельной маркировки;
- протестировать функциональность и производительность новой версии библиотеки.

1. Обзор

Существуют различные подходы к автоматическому управлению памятью, каждый из которых имеет свои преимущества и недостатки [9,10]. В данной главе будут рассмотрены основные алгоритмы сборки мусора, а также подходы к автоматическому управлению памятью, актуальные в контексте C++. Кроме того, будет выполнен обзор существующих решений для C++.

1.1. Обзор алгоритмов сборки мусора

В данном разделе приводится описание основных алгоритмов сборки мусора. Различают два принципиально различных подхода к автоматическому управлению памятью: *подсчёт ссылок* (*reference counting*) и *трассирующая сборка мусора* (*tracing garbage collection*). Существует множество различных разновидностей трассирующей сборки мусора, мы рассмотрим *сжимающую сборку мусора* (*compacting garbage collection*) и *инкрементальную сборку* (*incremental garbage collection*).

1.1.1. Подсчёт ссылок

Одним из наиболее простых методов автоматического управления памятью является *подсчёт ссылок*. С каждым объектом, выделенным в куче, связывается целое число — *счётчик ссылок*. Изначально значение счётчика инициализируется единицей. При копировании указателя на объект его счётчик увеличивается на единицу, а при удалении указателя — уменьшается на единицу. Когда значение счётчика падает до нуля, объект может быть удален.

К достоинствам данного подхода можно отнести то, что он прост для понимания, легко реализуем и, кроме того, момент освобождения памяти строго детерминирован: как только в программе не остается ссылок на определенный объект, занимаемая им память может быть освобождена и переиспользована. Однако данный подход обладает рядом недостатков.

1. Поддержка счётчика ссылок накладывает дополнительные накладные расходы на все операции с указателями.

2. Имеется вероятность *лавинного освобождения памяти*, т.е. ситуации, при которой уничтожение одной ссылки на "корень" структуры данных приводит к цепному освобождению целой области памяти, вызывая тем самым *непредсказуемую* по продолжительности паузу в процессе исполнения программы.

3. *Циклические структуры данных* (множества объектов, содержащих взаимные ссылки) не могут быть корректно обработан без модификации алгоритма (наличие таких структур приводит к утечке памяти).

Ясно, что второй недостаток прямо связан с детерминированным моментом освобождения памяти, и попытка преодолеть его также приведёт к потери этого преимущества. Для решения третьей проблемы были предложены различные модификации алгоритма, однако они либо достаточно сложны и вычислительно трудоёмки, либо перекладывают ответственность за разрешение циклических ссылок на программиста [10]. Например, может быть введён новый примитив — *невладеющий указатель* (weak pointer). Невладеющий указатель не участвует в подсчёте ссылок. Используя этот указатель для хранения ссылок, замыкающих цикл, можно предотвратить утечку памяти.

1.1.2. Трассирующая сборка мусора

Принципиально другой подход к сборке мусора основан на использовании критерия доступности объекта. *Доступность* определяется индуктивно, а именно, область памяти называется *доступной*, если либо указатель на неё принадлежит корневому множеству, либо существует указатель на неё из другой области памяти, являющейся доступной. *Корневое множество* представляет собой множество априори доступных объектов. Конкретное определение корневого множества зависит от языка программирования и среды времени выполнения. Зачастую

корневым множеством является множество указателей, расположенных в регистрах, на стеке и в статической области памяти. Все объекты, которые не являются доступными, объявляются *мусором*. Сборщики мусора, использующие данный подход, называются *трассирующими*. Инвариантом данного подхода является то, что память, занимаемая "мусором", в любой момент может быть безопасно освобождена. Тем самым доступность является эвристическим приближением понятия *используемости*. Заметим, что проблема используемости памяти неразрешима[2].

Базовым алгоритмом трассирующий сборки мусора является алгоритм *"пометить и освободить"* (*mark-and-sweep*). Сборщик мусора обходит граф достижимых объектов, начиная с корневого множества, помечая все достижимые объекты как живые (для хранения метки с каждым объектом связывается специальный бит). Затем память из-под всех непомеченных объектов освобождается. Компонент, выполняющий пометку и освобождение, называют *сборщиком*, а компонент, модифицирующий память и запрашивающий её выделение — *мутатором*.

Ввиду того, что мутатор может модифицировать объекты и менять структуру графа достижимых объектов, перед запуском маркировки и освобождения он должнен быть остановлен. Алгоритмы сборки мусора, требующие полной остановки работы приложения на время своей работы, известны как *алгоритмы с полной остановкой мира* (*stop-the-world algorithms*).

Трассирующие сборщики мусора можно разделить на два класса:

- *точные* (*precise*/*accurate*) сборщики мусора, которые способны обнаружить все недоступные объекты;
- *консервативные* (*conservative*) — сборщики мусора, не являющиеся точными.

Для точной трассирующей сборки мусора необходимо выполнение следуюших условий:

[2]Может быть показано, что проблема используемости памяти сводится к проблеме остановки: `https://en.wikipedia.org/wiki/Tracing_garbage_collection#Reachability_of_an_object`

- возможность построения корневого множества;
- возможность определить все указатели из любого объекта на другие области управляемой памяти.

Часто в силу особенности языка программирования, устройства среды времени выполнения или по другим специфическим причинам выполнение этих условий невозможно. В таком случае, без наложения дополнительных ограничений на использование языковых средств, возможна только консервативная сборка мусора. При этом используются различные эвристические методы построения корневого множества или идентификации ячейки памяти как указателя. Это приводит к тому, что для каждого консервативного сборщика мусора существует целый класс программ, на которых выбранные эвристики работают недостаточно хорошо [2]. Следствием этого могут стать длительные задержки в работе сборщика или же исчерпание памяти.

1.1.3. Сжимающая сборка мусора

Сжимающая сборка (*compacting garbage collection*) — это разновидность трассирующей сборки, предназначенная для решения проблемы *фрагментации памяти*. Фрагментацией называется ситуация, при которой в куче выделено большое число блоков, не образующих непрерывную область памяти. В таком случае попытка выделения объекта большого размера может закончиться неудачей несмотря на то, что суммарного количества свободной памяти вполне достаточно для удовлетворения запроса.

Алгоритм сжимающей сборки *"пометить и сжать"* (*mark-and-compact*) после этапа маркировки запускает процедуру сжатия кучи. Все живые объекты перемещаются на место непереживших сборку, а указатели на них модифицируются, чтобы указывать на новое расположение объектов (*pointer forwarding*).

Для сжатия кучи могут применяться различные алгоритмы. Рассмотрим одну из модификаций *двухпальцевого алгоритма* (*two-finger-*

compact), которая может применяться для сжатия области памяти, содержащей объекты фиксированного размера. Заметим, что существуют вариации двухпальцевого алгоритма, позволяющие сжимать кучу с объектами произвольного размера, но мы не рассматриваем их, так как в текущей версии библиотеки они не применяются. Алгоритм поддерживает два указателя — указатель на свободный объект и указатель на перемещаемый объект. Первый указатель инициализируется адресом начала сжимаемой области памяти, а второй — адресом конца. Затем выполняется поиск очередного свободного блока (первым указателем) и очередного перемещаемого объекта (вторым указателем). Память, занимаемая перемещаемым объектом, копируется в свободный блок, а на её место записывается так называемый *"перемещающий" указатель* (*forward pointer*) — указатель на новое расположение объекта. Перемещающий указатель затем используется в фазе модификации указателей на перемещенные объекты. Алгоритм повторяет этот процесс, пока указатель на свободный блок меньше указателя на перемещаемый объект.

1.1.4. Инкрементальная параллельная маркировка

Алгоритмы, позволяющие чередовать работу коллектора и мутатора называются *инкрементальными*. Инкрементальные алгоритмы разбивают цикл сборки мусора на несколько частей, при этом снижается время паузы мутатора и уменьшаются задержки в работе приложения, вызванные остановкой на сборку мусора. Однако инкрементальная сборка, как правило, привносит накладные расходы, так что суммарное время работы приложения может увеличиться по сравнению с подходами с полной остановкой мутатора. Алгоритмы сборки, способные выполнять большую часть своей работы (часто, всю работу лишь за исключением сканирования корневого множества) по сборке мусора одновременно с работой мутатора, называются *почти паралелльными* (*mostly-concurrent*) сборщиками.

Далее мы рассмотрим модификацию алгоритма "пометить и сжать", которая позволит производить маркировку без остановки мутатора, остановка мира необходима только для сканирования корневого мно-

жества, а также на этапе сжатия кучи. Чтобы рассмотреть такую модификацию, введем несколько определений. Разделим все объекты на три класса:

- "*белые*" — объекты, которые еще не были сканированы сборщиком мусора;
- "*серые*" — объекты, которые были помечены сборщиком, но не все поля которого были просканированы;
- "*чёрные*" — объекты, которые были помечены сборщиком и все их поля были просканированы.

При инкрементальной маркировке может возникнуть ситуация, в которой чёрный объект будет содержать ссылку на белый. После завершения маркировки белый объект будет освобожден, а чёрный будет хранить некорректный указатель. Такая ситуация может возникнуть при выполнении одновременно двух условий:

- мутатор сохраняет указатель на белый объект в чёрном;
- не существует серого объекта, такого, что существует путь от него до рассматриваемого белого объекта.

Для предотвращения ошибочной ситуации сборщик и мутатор должны поддерживать один из двух инвариантов.

1. *Слабый трехцветный инвариант* (*weak tricolour invariant*): для любого белого объекта, на который указывает чёрный объект, существует серый объект, такой что существует путь от этого серого объекта до рассматриваемого белого.
2. *Сильный трехцветный инвариант* (*strong tricolour invariant*): не существует чёрных объектов, указывающих на белые.

Для поддержки инвариантов могут быть использованы *барьеры чтения* или *барьеры записи*. Барьером чтения называется перехват чтения полей белого объекта, барьером записи называется прехват записи

ссылок на белый объект в чёрный. Заметим, что последовательность инструкций, реализующая барьер, выполняется при каждой операции чтения/записи во время фазы маркировки объектов, накладывая тем самым дополнительные расходы на время выполнения работы операций с указателями. Барьеры могут быть реализованы либо программно, либо с помощью аппаратной поддержки, либо с использованием функций операционной системы. Далее мы будем рассматривать только барьер записи, так как именно он используется в нашей работе. Этот выбор обоснован тем, что операции записи, как правило, выполняются реже, следовательно и барьер записи вызывается реже, а значит и накладные расходы меньше. Существуют различные реализации барьеров записи, каждая из которых поддерживает один из двух упомянутых выше инвариантов. Приведем псевдокод барьера записи, впервые предложенного в работе [12]:

Листинг 1: Барьер записи Дейкстры

```
1 Write(src, field, ptr):
2       src->field = ptr
3       Shade(ptr)
4
5 Shade(ptr):
6       if white(ptr)
7           colour(ptr) = grey
```

Заметим, что для корректной работы сборщика мусора необходимо, чтобы обе операции в функции *Write* были атомарны.

1.2. Сборка мусора в C++

Как уже говорилось, язык C++ разрабатывался с расчетом на использование ручного управления памятью. Поддержка сборки мусора не была изачально добавлена разработчиками языка и по сей день не включена в стандарт [8]. В [3] было упомянуто три подхода для добавления сборки мусора в язык, подробно описанных ниже.

1. Подход, основанный на *умных указателях* (*smart pointer*) — объ-

ектах специального класса, оборачивающих “сырые” указатели и добавляющие к ним некоторую функциональность. Зачастую умные указатели используются для реализации подсчёта ссылок, однако могут быть реализованы и другие алгоритмы автоматического управления памятью.

2. Подход, основанный на введении нового типа данных для хранения указателей на управляемые объекты. Экземпляры этого нового типа имеют более бедный интерфейс по сравнению с обычными указателями, например, запрещена адресная арифметика. Существенное отличие от предыдущего подхода — использование поддержки со стороны компилятора. Хорошим примером реализации данного подхода служит нестандартное расширение C++/CLI [6], разработанное компанией Microsoft.

3. *“Прозрачная”* (transparent) сборка мусора. Этот подход подразумевает использование обычных указателей с возможностью переиспользования памяти. Примером реализации “прозрачной” сборки является сборщик мусора Бёма-Демерса-Вайзера [4].

В языках C/C++ указатели хранятся в памяти как целые числа. Иными словами, представление указателей в памяти никак не отличается от представления других примитивных типов данных. Из сказанного выше ясно, что без наложения дополнительных ограничений или введения новых типов данных для хранения указателей, точная сборка невозможна в C/C++. Поэтому, третий подход подразумевает использование консервативного сборщика. Также стоит отметить, что в рамках третьего подхода затруднительно реализовать сжимающий сборщик мусора.

1.3. Существующие решения

Рассмотрим более подробно существующие решения для автоматического управления памятью в C++.

Шаблонный класс `shared_ptr<T>` из стандартной библиотеки C++11 [8] — это умный указатель, реализующий алгоритм подсчёта ссылок для автоматического управления памятью. *Умный указатель* (*smart pointer*) — это класс-декоратор, добавляющий некоторую функциональность к обычному интерфейсу указателей. Умные указатели используют целый ряд языковых возможностей C++ (в том числе новые возможности, добавленные в стандарте C++11), такие как переопределение операторов, семантика перемещения (move semantics) и другие, для предоставления удобного и интуитивно понятного интерфейса. Также, как правило, используется идиома RAII (Resource Acquisition Is Initialization). В самом общем смысле, использование этой идиомы позволяет связывать некоторые операции с конструированием и уничтожением объекта. Для этого создается специальный класс-обёртка над каким-либо ресурсом, в конструкторе этого класса вызываются функции, связанные с инициализацией ресурса, а в деструкторе — с освобождением ресурса. Например, конструктор `shared_ptr<T>` инициализирует контрольный блок (специальный объект, хранящий счётчик ссылок и указатель на функцию зачистки), а деструктор выполняет проверку счётчика ссылок и при необходимости освобождает память, занятую объектом и его контрольным блоком.

Как отмечалось в разделе 1.1.1, один из главных недостатков наивного подсчёта ссылок — некорректная обработка циклических ссылок. Для преодоления этой проблемы программисту предлагается использовать объекты класса `weak_ptr<T>` — невладеющие указатели. Перекладывание проблемы обработки циклических ссылок на пользователя библиотеки приводит к возможности возникновения в программе труднообнаружимых ошибок.

Расширение C++/CLI было разработано компанией Microsoft для возможности интеграции программ на C++ с программной платформой .Net. .Net использует общеязыковую среду исполнения (Common Language Runtime), управление памятью в которой происходит при помощи трассирующего сборщика мусора. Для того чтобы обеспечить совместимость C++ с CLR в язык необходимо было добавить поддержку

управляемых объектов, что и было сделано. Для хранения указателей на управляемые объекты был введён новый тип данных T^, который, однако, предоставляет более бедный интерфейс по сравнению с обычными указателями. Подход C++/CLI позволяет совмещать ручное и автоматическое управление памятью, сборка мусора является точной и сжимающей. Но скомпилировать программы, написанные на C++/CLI, можно только специальным компилятором, поддерживающим нестандартные расширения языка.

Среди консервативных сборщиков мусора для C++ наиболее популярным является BoehmGC [4]. Одна из главных целей, которую преследовали разработчики этого сборщика — возможность использования сборки мусора в существующих проектах с минимальной модификацией исходного кода и минимальными ограничениями на использование языковых возможностей. BoehmGC использует алгоритм mark-sweep. Для выделения памяти программист должен использовать функцию `GC_malloc`, которая сохраняет необходимую для сборщика метаинформацию. BoehmGC используется в таких проектах, как Mono, Portable.NET, GNU Compiler for Java и другие. Также он может быть использован для поиска утечек памяти, и в таком качестве он применяется в продуктах компании Mozilla. BoehmGC не требует поддержки от компилятора. К его недостаткам можно отнести консервативность и отсутствие поддержки сжимающей сборки мусора.

2. Анализ предыдущей версии библиотеки

В рамках проекта лаборатории JetBrains была реализована библиотека сборки мусора для языка C++ [1, 13, 15]. Реализованный сборщик мусора является почти точным, копирующим, не требует поддержки от компилятора и позволяет совмещать ручное и автоматическое управление памятью. Используется модификация алгоритма mark-compact, мутатор останавливается на время маркировки и сжатия. От пользователя библиотеки ожидается выполнение некоторых соглашений, необходимых для корректной работы сборщика. Библиотека может быть скомпилирована только компилятором gcc для 64 битной архитектуры и операционной системы Linux, однако в дальнейшем возможен перенос на другие платформы, который потребует переработки лишь небольшой части платформозависимого кода.

Далее в данной главе будет приведён детальный обзор реализации предыдущей версии библиотеки, будут рассмотрены интерфейс библиотеки, реализация корневого множества и кучи, структура метаинформации, необходимой для сборки мусора, используемый алгоритм остановки мира и механизм взаимодействия с “сырыми” указателями.

Интерфейс. Библиотека основана на подходе, использующем “умные указатели” для поддержки автоматического управления памятью. Пользователю предоставляется шаблонный класс `gc_ptr<T>` для хранения умных указателей и шаблонная функция `gc_new<T>` для создания управляемый объектов в куче. Класс `gc_ptr<T>` определяет конструктор по умолчанию, конструктор от нулевого указателя (`nullptr`), конструктор копирования, оператор присваивания, оператор доступа к члену класса (`operator->()`), оператор разыменования (`operator*()`), оператор преобразования в тип bool (для проверки, является ли указатель нулевым), метод reset для обнуления указателя. Пример использования примитивов библиотеки приводится в листинге 2.

Листинг 2: Пример использования примитивов библиотеки

```
1  struct Node {
2      Node(const gc_ptr<Node>& left,
3           const gc_ptr<Node>& right)
4          : left_(left)
5          , right_(right)
6      {}
7
8      gc_ptr<Node> left_;
9      gc_ptr<Node> right_;
10 };
11 ...
12 gc_ptr<Node> a, b;
13 ...
14 gc_ptr<Node> c = gc_new<Node>(a, b);
```

Корневое множество. Корневым множеством полагается множество указателей на управляемые объекты на стеке и в статической памяти. Корневое множество хранится в памяти как список указателей на объекты `gc_ptr`, являющиеся корнями. Чтобы снизить издержки на синхронизацию, каждый поток имеет свой экземпляр списка указателей на корни. Корневое множество поддерживается в консистентном состоянии при помощи `gc_new` и конструктора `gc_ptr`. В конструкторе `gc_ptr` проверяется уровень вложенности вызовов `gc_new` и если он равен нулю, то создаваемый указатель является корнем. Новые корни добавляются в голову списка, при удалении корня список также просматривается начиная с головы. Это связано с тем, что в C++ порядок вызова деструкторов почти всегда обратен порядку вызова конструкторов (временные объекты могут нарушать это предположение), поэтому такая оптимизация позволяет в большинстве случаев быстро добавлять и удалять корни.

Метаинформация. Для того чтобы сборщик имел возможность построить граф достижимых объектов, каждый объект должен содержать дополнительную метаинформацию, позволяющую определить, указатели на какие объекты он содержит. В нашей библиотеке используется два типа метаинформации — метаинформация класса и метаинформация объекта. Метаинформация класса хранит размер экземпляра данного класса и список смещений `gc_ptr`, содержащихся в экземпляре класса, относительно начала объекта. Метаинформация класса создается один раз при первом выделении объекта данного класса в управляемой куче. С каждым объектом, выделенным в управляемой куче, связана метаинформация объекта. Она хранится в куче сразу после самого объекта. Метаинформация объекта содержит указатель на метаинформацию класса, указатель на начало объекта и количество экземпляров класса внутри управляемого объекта[3] (для поддержки массивов). Метаинформация создается и поддерживается с помощью специального протокола взаимодействия `gc_new` и конструктора `gc_ptr`, подробное описание которого может быть найдено в работе [1].

Куча. Для хранения управляемых объектов используется собственная реализация кучи. Куча представляет собой набор пулов (segregated storage[4]), каждый из которых обслуживает объекты определенного размера. Адресное пространство кучи разделено на страницы, по умолчанию размер страницы равен 4096 байт. Размеры объектов являются степенями двойки от 32 (минимальный размер управляемого объекта) до 2^{32} байт. Пул выделяет память блоками размера, кратного размеру страницы, с помощью системного вызова `mmap`. Затем этот блок делится на подблоки равного размера. С каждым блоком также связывается дескриптор, хранящий два битовых массива — массив битов маркировки (`mark_bits`) и массив битов закрепления (`pin_bits`), мьютекс, а также другую метаинформацию. Блоки индексируются с помощью двух-трёх уровненого сильноветвящегося дерева. Дерево индексации позволяет

[3]Под объектом в данном случае понимается не экземпляр какого-либо класса, а область памяти в управляемой куче

[4]`http://www.memorymanagement.org/glossary/s.html#term-simple-segregated-storage`

по указателю на управляемый объект за константное время определить связанный с ним дескриптор и соответствующие биты маркировки и закрепления, а также по произвольному указателю определить, является ли объект, на который он указывает, управляемым.

Подобная реализация кучи достаточно эффективна, но для объектов, размер которых существенно превышает размер страницы, выделение блока объектов может попросту закончится неудачей, так как у операционной системы не окажется непрерывной области виртуальной памяти такого размера. Из сказанного выше можно сделать вывод, что для выделения объектов большого размера следует использовать другую стратегию аллокации. Однако в данной работе подобная стратегия не предложена, её реализация является заделом для будущей работы.

Остановка мира. На стадии сжатия и освобождения кучи все потоки приложения приостанавливаются. Однако поток может быть приостановлен не в любой момент, а только когда соблюдаются определенные инварианты, необходимые для корректной работы сборщика. К примеру, предположим, что сборщик мусора решит приостановить поток мутатора, в то время как он добавляет новый корень в корневое множество. Поток может быть приостановлен в середине этой операции, и корневое множество окажется в неконсистентном состоянии.

Безопасной точкой называется такое состояние мутатора, в котором он может быть приостановлен сборщиком. Если сборщик мусора имеет поддержку со стороны компилятора, как правило, именно компилятор расставляет безопасные точки в пользовательском коде. То есть, компилятор вставляет последовательность инструкций, которая проверяет, была ли запрошена сборка мусора, и останавливает поток если необходимо. Как уже упоминалось, наша библиотека не имеет поддержки компилятора. Поэтому, безопасные точки в предыдущей версии были вручную расставлены в некоторых примитивах библиотеки, например, в функции `gc_new`. Такое решение имеет существенный недостаток — если какой-либо поток редко используют примитивы нашей библиотеки, он не сможет быть остановлен сборщиком. Сборка мусора не может

быть начата, пока все потоки мутатора не будут приостановлены. Таким образом, пользователь библиотеки мог столкнуться с ситуацией, при которой сборка мусора никогда бы не была вызвана. Для преодоления этой проблемы в данной работе был предложен новый алгоритм остановки мира, описанный в разделе 3.1.

Стоит также отметить, что для остановки мира, как в старой, так и в новой реализации, необходимо поддерживать список активных потоков. Для поддержки этого списка реализована обёртка над библиотекой pthreads, включающая функции `thread_create`, `thread_join`, `thread_exit` и `thread_cancel`. Пользователь библиотеки должен использовать эти функции для управления потоками.

Преобразование управляемого указателя в "сырой" указатель. Для хранения управляемых указателей пользователь библиотеки должен использовать объекты класса `gc_ptr`. Однако иногда может понадобиться получить "сырой"[5] указатель на управляемый объект (например, для передачи такого указателя в функцию сторонней библиотеки, не знающей о существовании сборки мусора). Отметим также, что некоторые указатели в программе не могут быть "умными" [10], например, указатель `this`. Проиллюстрируем это нижеследующим примером.

[5]"сырым" указателем называется обычный указатель C/C++ вида T*, где T — тип объекта

Листинг 3: Пример неявного разыменования управляемого указателя

```
1  struct A {
2      int x;
3      int y;
4      void f() {
5          x = 0;
6          y = 42;
7      }
8  }
9
10 gc_ptr<A> a = gc_new<A>();
11 a->f();
```

В данном примере при вызове метода *f()* "сырой" указатель на объект *a* (`this`) будет неявно сохранен на стеке. Если между строками 5 и 6 работа приложения будет остановлена сборщиком мусора, а объект *a* будет перемещен, то после возобновления исполнения на стеке окажется некорректный указатель, после чего поведения программы становится неопределенным (undefined behaviour).

Для предотвращения этой ситуации в предыдущей версии библиотеки разыменованные управляемые указатели заносились в хэш-таблицу разыменованных указателей. Перед запуском сборки мусора просматривались стеки потоков. Указатели на стеке, содержащиеся в хеш-таблице, считались корнями. Также у объектов, на которые они указывали, выставлялся бит закрепления. Наличие этого бита говорило сборщику, что указанный объект не может быть перемещён. Момент, когда закрепление может быть снято, не отслеживался точно. Все указатели из хэш-таблицы, которые не были найдены на стеке, удалялись из неё перед возобновлением работы мутатора.

Так как сборщик мусора использовал консервативный обход стека для идентификации разыменованных указателей, строго говоря он не являлся точным. Была возможна ситуация, при которой не оставалось указателей на объект, и тем не менее занимаемая им память не

освобождалась. Поэтому предыдущая версия сборщика мусора являлась *почти-точной* (*mostly-precise*). В данной работе предложен новый механизм преобразования управляемых указателей в “сырые” указатели, который возвращает сборщику свойство точности и позволяет точно отслеживать момент, когда закрепление может быть снято.

3. Реализация

В данном разделе приводится описание выполненных модификаций библиотеки.

3.1. Остановка мира

В разделе 2 были перечислены недостатки использовавшегося алгоритма остановки мутатора. Для их преодоления был предложен новый подход. Вместо расстановки безопасных точек в примитивах библиотеки предлагается помечать небезопасные участки кода, а другие участки считать безопасными. Преимущество данного подхода в том, что теперь мутатор может быть уведомлен об инициации сборки мусора в любой момент своей работы. Если в момент инициации сборки поток находится в небезопасном участке, то его приостановка откладывается до момента, когда он покинет этот участок. Заметим, что небезопасными являются только участки кода, выполняющие некоторые операции с глобальными структурами данных (корневым множеством, кучей, и.т.д) и расположеные в примитивах самой библиотеки.

Для реализации этого подхода необходимо иметь возможность уведомить поток об инициации сборки мусора в любой момент времени. После получения такого уведомления поток должен приостановить свою работу до момента, когда сборка будет окончена. Подобный функционал может быть реализован при помощи механизма сигналов операционной системы Linux и примитивов синхронизации, являющихся асинхронно-сигналобезопасными. Далее в данной главе будут рассмотрены сигналы и реализация механизма их временной блокировки, учитывающая специфику их использования в нашей библиотеке. Также будет описана реализация асинхронно-сигналобезопасных примитивов синхронизации. В последнем подразделе описан новый алгоритм остановки мира.

3.1.1. Сигналы ОС Linux

Сигналы в Linux — один из механизмов межпроцессорного взаимодействия [14]. Linux поддерживает несколько типов сигналов, каждый из которых имеет уникальный идентификатор и мнемоническое имя. Сигнал может быть отправлен процессу либо ядром операционной системы, либо другим процессом с помощью системного вызова `kill`. При получении сигнала вызывается соответствующий обработчик сигнала. Для некоторых типов сигналов разрешается переопределять обработчик сигнала (при помощи системного вызова `sigaction`). Также имеется возможность временно заблокировать доставку сигналов определенного типа (системный вызов `sigprocmask`)

Библиотека pthreads позволяет отправлять сигналы конкретному потоку процесса с помощью системного вызова `pthread_kill`, тем самым позволяя использовать сигналы для межпоточного взаимодействия.

3.1.2. Блокирование сигналов

В библиотеке pthreads имеется функция `pthread_sigmask` аналогичная `sigprocmask`, но блокирующая доставку сигналов только данному потоку. Однако при каждом вызове `pthread_sigmask` происходит переход в *режим ядра* (*kernel mode*), что вызывет длительные задержки. Временное блокирование доставки сигнала об инициации сборки мусора очень частая операция. Например, она необходима при каждой модификации корневого множества. Из-за её низкой скорости выполнения может пострадать производительность всего приложения, использующего сборку мусора.

Для того, чтобы ускорить блокирование доставки сигнала инициации сборки мусора, был реализован механизм, учитывающий специфику задачи. Идея заключается в том, чтобы отследить попытку доставки сигнала потоку, выполнившему его блокирование, и отложить выполнение обработчика до момента разблокировки. В каждом потоке объявляются две локальные для потока (`thread_local`) переменные

`depth` и `pending_flag`. `depth` отслеживает вложенность блокировок, `pending_flag` инициализируется значением `false`. В обработчике сигнала сначала проверяется значение `depth`. Если оно не равно нулю, значит в текущий момент доставка сигнала заблокирована, выставляется `pending_flag` и происходит выход из обработчика. Иначе происходит нормальное исполнение обработчика сигнала. Для того чтобы заблокировать доставку сигнала поток просто инкрементирует переменную `depth`. При разблокировании доставки сигнала значение `depth` декрементируется и если оно становится равным нулю и при этом установлен флаг `pending_flag`, вызывается код обработчика сигнала, но уже без дополнительных проверок. Код обработчика сигнала, блокирования и разблокированя доставки сигналов на C++ может быть найден в листинге 4.

В таблице 1 приводится сравнение времени работы механизма блокирования сигналов с помощью функции `pthread_sigmask` и механизма, предложенного в нашей работе. Измерялось суммарное время работы пары операций — блокирования и разблокирования сигнала. В таблице приведено среднее время работы операций $\bar{x}$ и стандартное отклонение s. Заметим, что ускорения удалось достичь за счёт того, что предложенный подход к блокированию сигналов является менее общим и учитывает специфику нашей задачи.

Таблица 1: Сравнение времени работы двух методов блокирования сигналов (среднее время $\bar{x}$ и стандартное отклонение s указано в наносекундах)

	$\bar{x}$	s
pthread_sigmask	143.661	6.741
siglock/sigunlock	3.965	0.033

3.1.3. Асинхронно-сигналобезопасные примитивы синхронизации

Стандарт POSIX[6] накладывает жёсткие ограничения на код, работающий внутри обработчика сигнала. Обработчик сигнала должен быть асинхронно-сигналобезопасной функцией (async-signal safe function)[7].

После получения сигнала об инициации сборки мусора поток должен приостановить свою работу до тех пор, пока он не получит уведомление об окончании сбоки. Код, выполняющий эту задачу, должен работать в контексте обработчика сигнала, так как иного способа уведомить поток о некотором событии в произвольный момент времени кроме использования сигналов нет. Так как никакие из примитивов синхронизации библиотеки pthreads не являются асинхронно-сигналобезопасными[8], необходимо использовать собственную реализацию примитивов синхронизации.

В нашей библиотеке было реализована два примитива: `signal_safe_barrier` и `signal_safe_event`.

- `signal_safe_barrier` имеет два метода:
    1. `notify()` для уведомления о том, что поток достиг некоторой точки в программе;
    2. `wait(size_t n)` для приостановки текущего потока до момента, когда `n` других потоков достигнут барьера.
- `signal_safe_event` также определяет два метода:
    1. `notify(size_t n)` для уведомления `n` потоков о наступлении события;
    2. `wait()` для ожидания события.

[6]Стандарт POSIX описывает интерфейс для доступа к функциям операционной системы. Обеспечивает переносимость прикладных программ между ОС, поддерживающими POSIX. `http://pubs.opengroup.org/onlinepubs/9699919799/`

[7]Определение асинхронно-сигналобезопасной функции может быть найдено по ссылке `https://www.securecoding.cert.org/confluence/display/c/BB.+Definitions#BB.Definitions-asynchronous-safefunction`

[8]Список async-signal safe функций ОС Linux может быть найден по ссылке `http://pubs.opengroup.org/onlinepubs/9699919799/functions/V2_chap02.html#tag_15_04_03`

Оба примитива реализованы с помощью одной и той же техники. Системные вызовы `read` и `write`, предназначенные для чтения и записи данных из потока ввода-ввывода, являются асинхронно-сигналобезопасными. Кроме того, `read` является блокирующим, то есть поток, вызвавший `read`, будет приостановлен, пока из потока не будет прочитано указанное количество байт. На основе этих двух системных вызовов, а также системного вызова `pipe`, открывающего безымянный двунаправленный поток ввода-ввывода, и построены наши примитивы.

Оба примитива реализованы как классы. В конструкторе они открывают безымянный поток ввода-вывода. Метод `notify()` класса `signal_safe_barrier` записывает один байт в этот поток. Метод `wait(size_t n)` читает из потока `n` байт. Аналогично для `signal_safe_event`, `notify(size_t n)` записывает `n` байт в поток, а `wait()` читает из потока один байт.

Листинг 4: Блокирование доставки сигналов

```
1  thread_local volatile sig_atomic_t depth = 0;
2  thread_local volatile sig_atomic_t pending_flag = false;
3
4  void check_gc_siglock(int signum) {
5      if (depth > 0) {
6          pending_flag = true;
7          return;
8      }
9      gc_signal_handler();
10 }
11
12 void siglock() {
13     if (depth == 0) {
14         std::atomic_signal_fence();
15         depth = 1;
16         std::atomic_signal_fence();
17     } else {
18         depth++;
19     }
20 }
21
22 void sigunlock() {
23     if (depth == 1) {
24         std::atomic_signal_fence();
25         depth = 0;
26         std::atomic_signal_fence();
27         bool pending = pending_flag;
28         pending_flag = false;
29         if (pending) {
30             gc_signal_handler();
31         }
32     } else {
33         depth--;
34     }
35 }
```

3.1.4. Алгоритм остановки мира

Опишем теперь сам алгоритм остановки мира. Рассмотрим отдельно часть, отвечающую за приостановку потока и выполняющуюся в обработчике сигнала, и часть, инициирующую сборку. В ходе выполнения обе части синхронизируют свою работу с помощью одного объекта класса `signal_safe_barrier` и одного объекта `signal_safe_event`. В обработчике сигнала поток сначала вызывает метод `notify()` барьера, чтобы уведомить поток, инициировавший остановку мира, что он достиг обработчика. Затем вызывается метод `wait()` события. Когда сборка мусора окончится, поток, выполнявший сборку, возобновит работу остальных потоков, вызвав метод `notify(size_t)` этого события. Затем в обработчике события будет вызван `notify()` барьера ещё раз, теперь чтобы уведомить поток инициатор, что обработка сигнала завершена и поток возвращается к нормальному исполнению. Для того, чтобы два потока одновременно не инициировали остановку мира, код инициирования защищён мьютексом. Псевдокод инициирования остановки мира приведён в листинге 5.

Листинг 5: Алгоритм остановки мира

```
1 lock(gc_mutex)
2 for (thread in threads)
3     send(gc_signal, thread)
4 signal_safe_barrier.wait(threads.count - 1)
5 gc()
6 signal_safe_event.notify(threads.count - 1)
7 signal_safe_barrier.wait(threads.count - 1)
8 unlock(gc_mutex)
```

3.2. Закрепление объектов

Механизм закрепления объектов, использовавшийся в предыдущей версии библиотеки (раздел 2), включал процедуру консервативного обхода стека потоков. Поэтому в некоторых ситуациях сборщик мог оши-

бочно не идентифицировать мусор. В новой версии реализован механизм закрепления объектов, лишённый этого недостатка, и кроме того, точно отслеживающий время, когда закрепление может быть снято.

Используется идиома RAII, упомянутая в разделе 1.3. Вводится новый примитив — шаблонный класс `gc_pin<T>`. Конструктор класса `gc_pin<T>` принимает в качестве аргумента ссылку на объект типа `gc_ptr<T>` и выполняет закрепление объекта, на который указывает этот `gc_ptr`. В деструкторе `gc_pin<T>` закрепление объекта снимается. Указатели на все закрепленные объекты хранятся в структуре данных, аналогичной той, что используется для хранения корневого множества (раздел 2). То есть используется список указателей, причём новые указатели вставляются в голову и при удалении список просматривается с головы. Также как и в случае корневого множества, каждый поток имеет собственный экземпляр списка закрепленных объектов. Таким образом, в конструкторе `gc_pin<T>` указатель на объект вносится в список закрепленных объектов, а в деструкторе удаляется из него.

На этапе сжатия/освобождения кучи, после остановки мира, сборщик просматривает списки закрепленных объектов всех потоков. У объектов из этих списков выставляется бит закрепления (наличие этого бита говорит сборщику о запрете перемещать данный объект), также они сканируются сборщиком (для того, чтобы пометить все объекты, достижимые из закрепленного, и предотвратить их удаление).

Как уже упоминалось, закрепление объектов необходимо в двух случаях:

1. для передачи сырого указателя на управляемый объект в функцию, принимающую сырой указатель как аргумент;

2. для закрепления указателя `this` при вызове оператора доступа к члену класса (`operator->()`);

В первом случае пользователь может самостоятельно создать объект типа `gc_pin` и получить сырой указатель с помощью метода `get()` этого объекта. Стоит однако отметить, что время жизни полученного

таким образом сырого указателя не должно превышать время жизни соответствующего объекта `gc_pin`. Во втором случае объект `gc_pin` создаётся неявно самой библиотекой. Используется следующее соглашения языка C++ [8]: если оператор доступа к члену класса (`operator->()`) возвращает объект типа, отличного от `T*`, вызывается оператор доступа к члену класса этого объекта. То есть, перегруженный оператор доступа к члену класса `gc_ptr` создаёт на стеке объект типа `gc_pin` и возвращает его, затем, согласно соглашению языка C++, вызывается оператор доступа к члену класса этого временного объекта, который уже возвращает сырой указатель. После этого вызывается деструктор временного объекта `gc_pin`.

3.3. Инкрементальная параллельная маркировка

Рассмотрим реализацию алгоритма инкрементальной параллельной маркировки. Как было сказано в разделе 1.1.4 инкрементальная маркировка позволяет чередовать исполнение кода мутатора и сборщика. На компьютере с многоядерным процессором потоки мутатора и сборщика могут даже работать параллельно, если обеспечивается должная синхронизация.

В нашей работе сборщик мусора работает в отдельном потоке. Любой поток приложения может запросить у потока-сборщика начать маркировку или сжатие. Для этого поток захватывает мьютекс, проверяет, что соответствующий запрос ещё не был отправлен другим потоком, присваивает переменной `event` типа перечисления одно из значений (`START_MARKING`, `START_COMPACTING`, `NULL` - последнее значение говорит об отсутсвии запросов). С переменной `event` также связана условная переменная (`pthread_cond_t`) и после присваивания поток уведомляет сборщик о запросе с помощью функции `pthread_cond_signal`. Поток-сборщик выполняет ожидание на условной переменной и, получив запрос, переходит в фазу маркировки или сжатия.

В фазе маркировки поток сборщик работает параллельно с мутатором, не приостанавливая его. Более того, одновременно маркировку

могут выполнять несколько потоков. Остановка мира происходит дважды за один цикл работы сборщика. Один раз перед началом маркировки для сканирования корневого множества и один раз для сжатия и освобождения кучи. После освобождения кучи начинается новый цикл работы сборщика.

На этапе маркировки сборщик производит обход графа достижимых объектов. Сборщик поддерживает глобальную свободную от блокировок (lock-free) очередь указателей на объекты, ожидающие сканирования ("серые" объекты). Используется реализация свободной от блокировок очереди из библиотеки Boost [5]. Изначально в очередь копируются указатели из корневого множества. Затем сборщик по очереди вынимает указатели из очереди и сканирует объекты, на которые они указывают. Если обнаружится, что объект содержит указатели, они заносятся в очередь. Чтобы снизить количество обращения к глобальной очереди и, следовательно, конкурентность доступа к ней, каждый поток, выполняющий маркировку, также поддерживает локальную очередь указателей. Поток работает со своей локальной очередью пока все указатели из неё не будут обработаны, а затем копирует указатели из глобальной очереди в локальную.

Как уже упоминалось в разделе 1.1.4, инкрементальный сборщик мусора должен перехватывать операции записи ссылок (так называемый "барьер записи"). В нашей библиотеке барьер записи реализован программно и вызывается из конструктора копирования и оператора присваивания `gc_ptr`. Мы используем барьер записи Дейкстры, его псевдокод приведён в листинге 1.

Барьер записи Дейкстры "перекрашивает" объект, ссылка на который записывается, в "серый" цвет, если он был "белым". В нашем случае это означает, что указатель на этот объект должен быть помещён в глобальную очередь указателей, ожидающих сканирования. Барьер записи перехватывает операции записи только если сборщик находится в фазе маркировки. Также стоит отметить, что в фазе маркировки новые объекты создаются "чёрными", то есть гарантированно переживают текущий цикл сборки мусора. Операции присваивания указателей и созда-

ния новых объектов очень часты, поэтому необходимо, чтобы проверка текущей фазы была достаточно быстрой. В нашей реализации для этого используется переменная `phase` типа перечисления (допустимые значения: `IDLE`, `MARKING`, `COMPACTING`), обёрнутая в класс `std::atomic` из стандартной библиотеки C++11. Шаблонный класс `std::atomic<T>` гарантирует, что все операции с экземпляром этого класса будут атомарны [8, 16]. В случае, если тип `T` имеет размер не больше, чем размер машинного слова, для обеспечения этой гарантии на большинстве современных платформ будут использованы не блокировки на основе мьютексов, а специальные инструкции процессора. Отсутствие блокировки для доступа к переменной `phase` очень важно, так как в этом случае потокам не придется конкурировать за доступ к ней и проводить время в ожидании на блокирующем системном вызове. Изменения значения переменной `phase` происходят только во время остановок мира, а операции присваивания указателей и создания нового объекта помечены как “небезопасные” (см. раздел 3.1), то есть во время их выполнения не может произойти stop-the-world паузы и, следовательно, не может измениться фаза сборки мусора.

4. Апробация

Функциональность сборщика мусора была протестирована на модульных тестах, написанных вручную с помощью фреймворка **googletest** [7]. Также было произведено тестирование производительности библиотеки. Время работы некоторых примитивов библиотеки (таких как конструкторы `gc_ptr`, операторы присваивания `gc_ptr` и прочие) было измерено при помощи фреймворка **nonious** [11], а затем сопоставлено с аналогичными примитивами для класса `std::shared_ptr` и "сырых" указателей. В тестах шаблоны `gc_ptr` и `shared_ptr` параметризировались примитивным типом `size_t`, тип сырого указателя — `size_t*`. Стоит отметить, что время работы примитивов может различаться в зависимости от некоторых условий. Например, конструктор `std::shared_ptr` по умолчанию и конструктор от нулевого указателя работают намного быстрее, чем конструктор, принимающий не нулевой указатель `T*`. С другой стороны, время конструирования `gc_ptr` зависит от того, является ли конструируемый объект корнем или нет. В замерах учитывались эти особенности и некоторые примитивы измерялись в различных условиях. Результаты измерений могут быть найдены в таблицах 2, 3, 4, 5, также они проиллюстрированы на рис. 1.

Из результатов измерений следует, что присваивание `gc_ptr` с барьером записи (то есть, в фазе маркировки) существенно медленее присваивания других типов указателей. Также операция разыменования `gc_ptr` во много раз медленее разыменования других типов указателей, что связано с тем, что для разыменования `gc_ptr` необходимо выполнить закрепление объекта. Для повышения производительности библиотеки необходимо оптимизировать эти операции.

Помимо измерения времени работы отдельных функций также было проведено измерение времени работы тестовых программ, активно работающих с динамической памятью. В частности, тестировалась производительность библиотеке на известном тесте Бёма. В этом тесте строятся двоичные деревья различной глубины с различным количе-

Таблица 2: Время конструирования указателей (среднее время $\bar{x}$ и стандартное отклонение s указано в наносекундах)

	$\bar{x}$	s
“сырой” указатель	0.2837	0.0048
std::shared_ptr(nullptr_t)	0.5862	0.0035
std::shared_ptr(T*)	24.4356	0.1567
gc_ptr (корень)	9.9342	0.3913
gc_ptr (не корень)	5.5482	0.0090

Таблица 3: Время присваивания указателей (среднее время $\bar{x}$ и стандартное отклонение s указано в наносекундах)

	$\bar{x}$	s
“сырой” указатель	0.2984	0.0667
std::shared_ptr	6.7130	0.0722
gc_ptr (без барьера записи)	7.5937	0.1314
gc_ptr (с барьером записи)	36.8900	13.9104

Таблица 4: Время разыменования[10] указателей (среднее время $\bar{x}$ и стандартное отклонение s указано в наносекундах)

	$\bar{x}$	s
“сырой” указатель	0.0038	0.0008
std::shared_ptr	0.0018	0.0009
gc_ptr	20.1288	0.3760

Таблица 5: Время создания объектов (среднее время $\bar{x}$ и стандартное отклонение s указано в наносекундах)

	$\bar{x}$	s
new	16.2064	1.1857
std::make_shared	36.9603	1.8717
gc_new	92.7400	13.4719

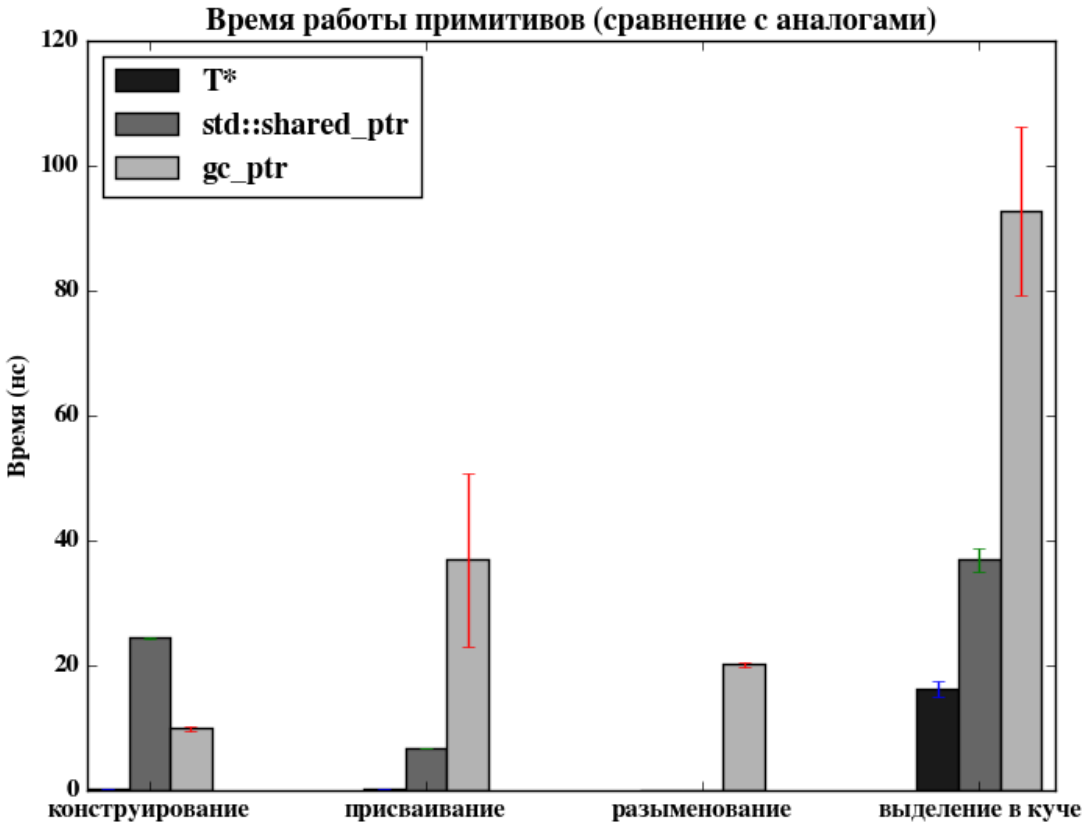

Рис. 1: Время работы примитивов

ством узлов. Деревья строятся двумя способами: от листов к корню и наоборот, от корня к листьям. Сперва было выполнено сравнение с предыдущей версией библиотеке, не использовавшей алгоритм инкрементальной параллельной маркировки. Измерялось суммарное время работы теста, среднее и максимальное время stop-the-world паузы (для текущей версии отдельно измерялось время паузы на сканирование корневого множества и время на сжатие/освобождение кучи), количество stop-the-world пауз и количество полных циклов сборки мусора. Все замеры были взяты как среднее и стандартное отклонение по 20 запускам теста. Результаты представлены в таблицах 6, 7 и на рис. 2. Хотя суммарное время работы и увеличилось, среднее и максимальное время паузы уменьшились почти в два раза. Стандарнтное отклонение также существенно уменьшилось, что свидетельствует о том, что время паузы стало более предсказуемым.

Таблица 6: Тест Бёма: старая и новая версия библиотеки (среднее время $\bar{x}$ и стандартное отклонение s указано в миллисекундах)

	Текущая версия		Предыдущая версия	
	$\bar{x}$	s	$\bar{x}$	s
суммарно время	5816.50	54.94	4972.40	11.92
количество пауз	131	—	30	—
количество сборок мусора	66	—	30	—

Таблица 7: Тест Бёма — время паузы: старая и новая версия библиотеки (среднее время $\bar{x}$ паузы, максимальное время max паузы и стандартное отклонение s указано в миллисекундах)

	Текущая версия			Предыдущая версия		
	$\bar{x}$	s	max	$\bar{x}$	s	max
Время паузы (сканирование корневого множества)	0.0669	0.0288	0.1650	—	—	—
Время паузы (сжатие/освобождение кучи)	29.7142	4.4798	52.2060	53.4216	33.1159	109.8400

Кроме того, сравнивалась производительность в сравнении с аналогами. Сравнение производилось на тесте Бёма, а также на тесте с многопоточной сортировкой слиянием. В этом тесте сначала создавался односвязный список из 4096 узлов, содержащих одно значение типа `int`, инициализировавшееся случайным числом. Данный список разделялся на несколько частей, каждая часть сортировалась в отдельном потоке сортировкой слиянием, затем части сливались в единый список. Измерялось время создание списка, время его сортировки и, в случае ручного управления памятью, время освобождения памяти, занимаемой списком. В экспериментах принимали участие “сырые” указатели с ручным управлением (`new`/`delete`), сборщик мусора Бёма-Демерса-Вайзера с

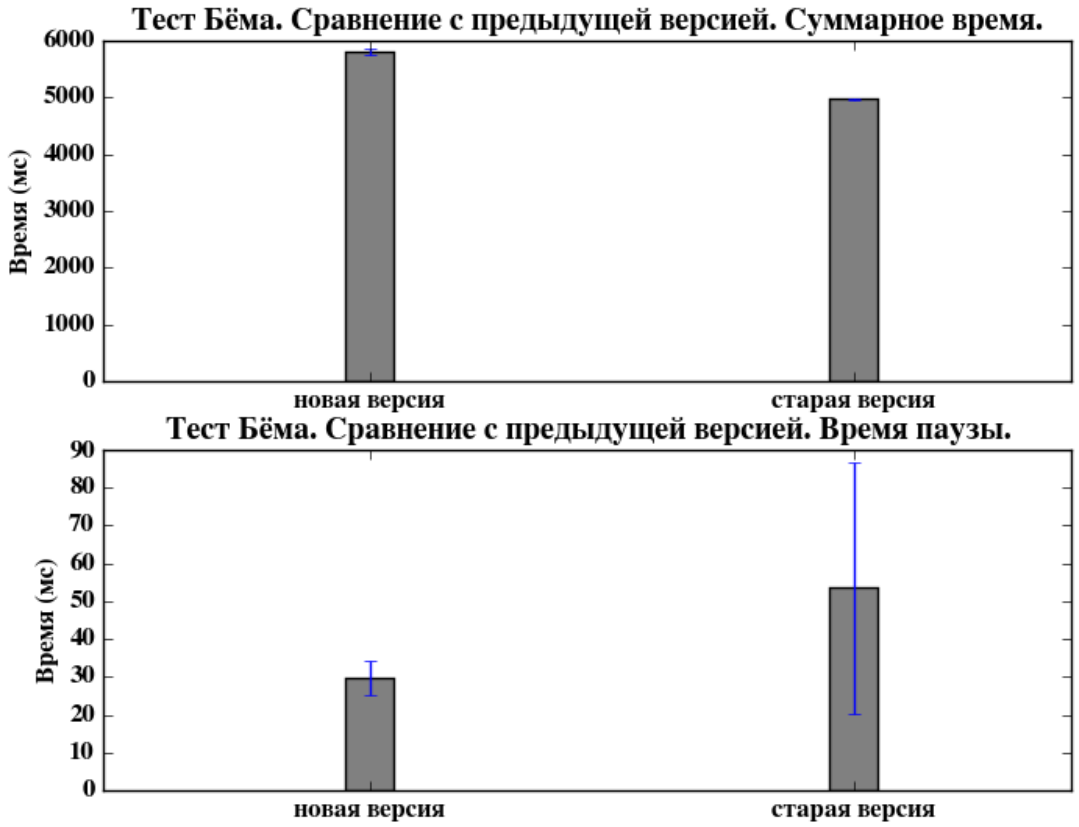

Рис. 2: Тест Бёма, старая и новая версия библиотеки

включенной опцией инкрементальной маркировки и `std::shared_ptr`. В этом тесте сжатие кучи в нашей библиотеке было отключено, так как ни один из аналогов также не выполняет его. Сравнивалось только суммарное время работы теста, а для сборщика Бёма-Демерса-Вайзера и нашего сборщика также количество сборок мусора. Все замеры были взяты как среднее и стандартное отклонение по 20 запускам теста. Результаты для теста Бёма представлены в таблице 8 и на рис. 3. Результаты для многопоточной сортировки слиянием представлены в таблице 9 и на рис. 4. Наша библиотека работает примерно в 5-10 раз медленее, чем другие решения, участвовавшие в эксперименте, что свидетельствует о необходимости дальнейшей оптимизации.

Таблица 8: Тест Бёма: сравнение с аналогами (среднее время $\bar{x}$ и стандартное отклонение s указано в миллисекундах)

	Суммарное время		Количество сборок мусора
	$\bar{x}$	s	$\bar{x}$
new/delete	347.1500	4.1143	—
std::shared_ptr	414.8500	11.1637	—
BoehmGC	351.8500	10.8087	234
gc_ptr	4144.2500	45.3661	66

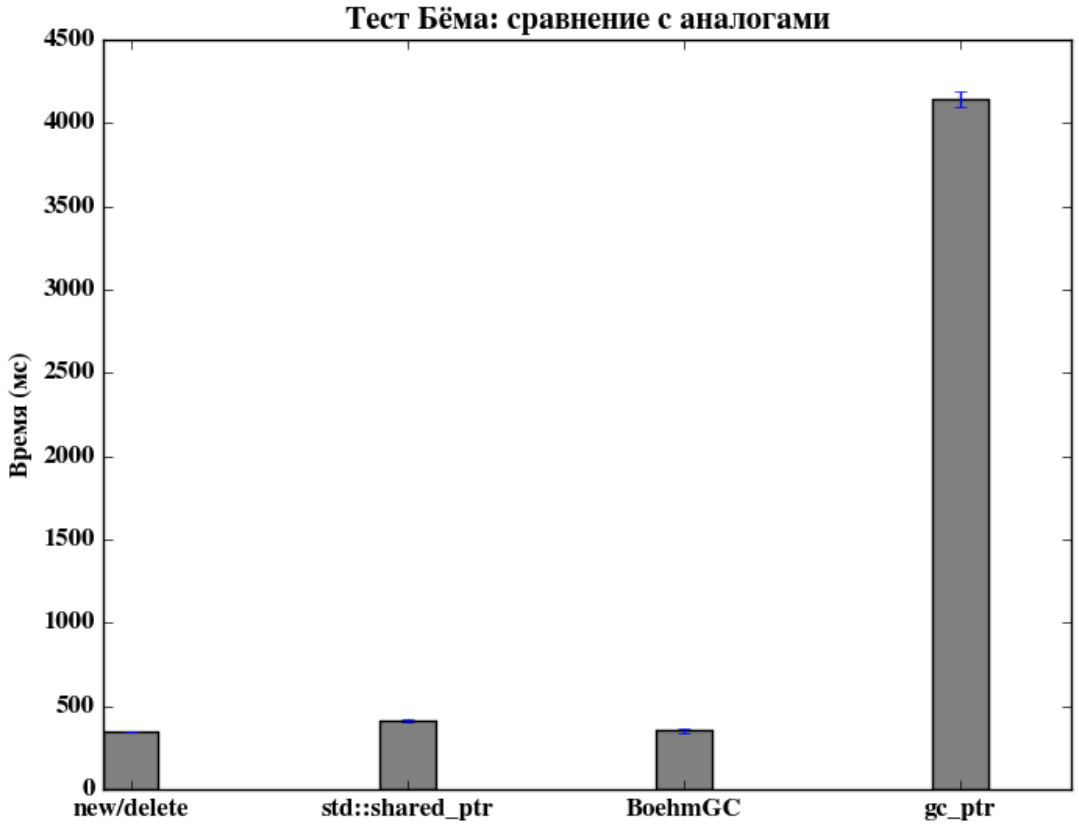

Рис. 3: Тест Бёма

Результаты

В ходе работы были достигнуты следующие результаты:

- проанализированы ограничения и недостатки предыдущей версии библиотеки;

Таблица 9: Многопоточная сортировка слиянием: сравнение с аналогами (среднее время $\bar{x}$ и стандартное отклонение s указано в миллисекундах)

	Суммарное время		Количество сборок мусора
	$\bar{x}$	s	$\bar{x}$
new/delete	7.4642	2.6347	—
std::shared_ptr	10.6421	2.3226	—
BoehmGC	7.2499	2.2163	33
gc_ptr	38.2628	9.4771	4

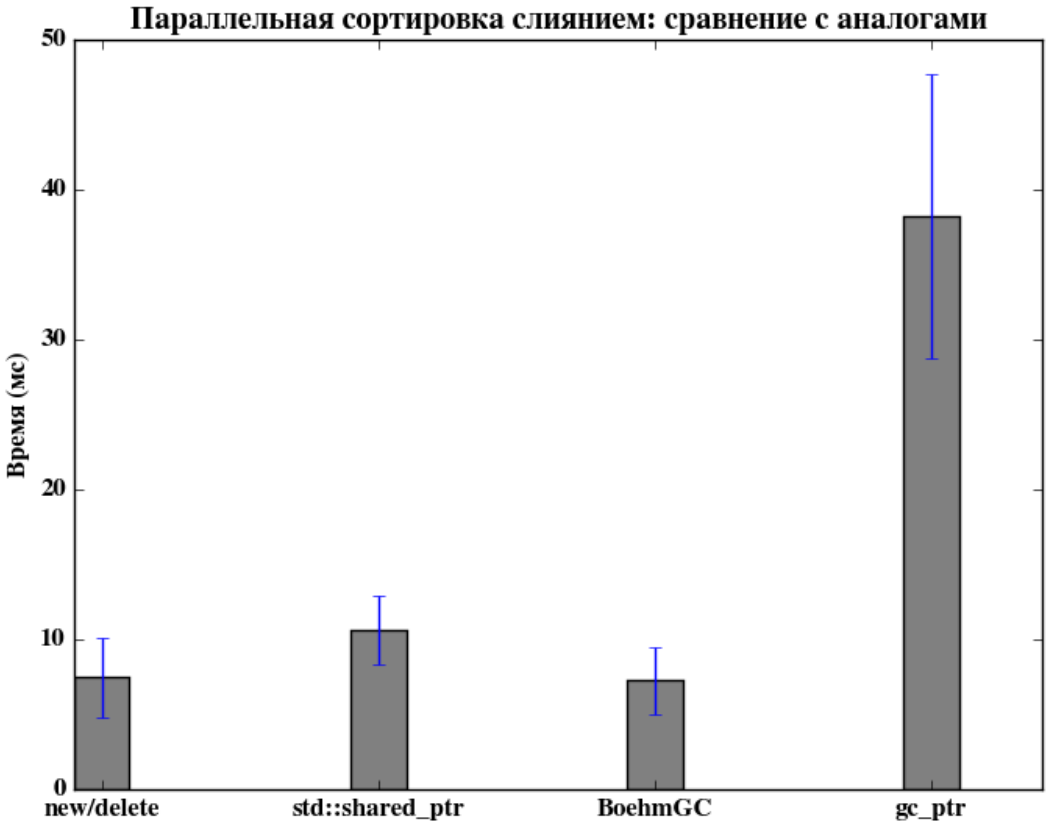

Рис. 4: Параллельная сортировка слиянием

- реализован новый алгоритм остановки мира;
- реализован новый алгоритм закрепления объектов;
- реализована модификация алгоритма инкрементальной парал-

лельной маркировки;

- выполнена апробация: протестирована функциональность библиотеки, измерена её производительность, произведено сравнение с аналогами.

Список литературы

[1] Berezun Daniil, Boulytchev Dmitri. Precise Garbage Collection for C++ with a Non-cooperative Compiler. Proceedings of the 10th Central and Eastern European Software Engineering Conference in Russia. — 2014.

[2] Boehm Hans-Juergen. Space efficient conservative garbage collection // ACM SIGPLAN Notices / ACM. — Vol. 28. — 1993. — P. 197–206.

[3] Boehm Hans-J, Spertus Michael. Transparent Programmer-Directed Garbage Collection for C+ // URL: http://www.open-std.org/jtc1/sc22/wg21/docs/papers. — 2007. — no. 2310.

[4] BoehmGC - A garbage collector for C and C++. — URL: `http://www.hboehm.info/gc/` (online; accessed: 17.12.2015).

[5] Boost — free peer-reviewed portable C++ source libraries. — URL: `http://www.boost.org/` (online; accessed: 15.05.2016).

[6] C++/CLI Language Specification. — URL: `http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-372.pdf` (online; accessed: 08.04.2016).

[7] Google Test, Google's C++ test framework. — URL: `https://github.com/google/googletest` (online; accessed: 15.05.2016).

[8] ISO/IEC. Working Draft, Standard for Programming Language C++. — 2014. — URL: `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4296.pdf` (online; accessed: 17.12.2015).

[9] Jones Richard, Hosking Antony, Moss Eliot. The garbage collection handbook: the art of automatic memory management. — Chapman & Hall/CRC, 2011.

[10] Jones Richard, Lins Rafael D. Garbage collection: algorithms for automatic dynamic memory management. — Wiley, 1996.

[11] Nonious, a C++ micro-benchmarking framework. — URL: `https://nonius.io/` (online; accessed: 15.05.2016).

[12] On-the-fly garbage collection: An exercise in cooperation / Edsger W Dijkstra, Leslie Lamport, Alain J Martin et al. // Communications of the ACM. — 1978. — Vol. 21, no. 11. — P. 966–975.

[13] Березун Даниил. Реализация основных примитивов библиотеки неконсервативной сборки мусора для C++. Труды лаборатории языковых инструментов JetBrains, выпуск 2.

[14] Робачевский Андрей Михайлович. Операционная система UNIX, 2 изд. — БХВ-Петербург, 2010.

[15] Самофалов Александр. Библиотека почти точной копирующей сборки мусора для C++. Труды лаборатории языковых инструментов JetBrains,, выпуск 3.

[16] Уильямс Энтони. Параллельное программирование на C++ в действии. Практика разработки многопоточных программ. — Litres, 2014.