Санкт-Петербургский государственный электротехнический университет
«ЛЭТИ» им. В. И. Ульянова (Ленина)
(СПбГЭТУ «ЛЭТИ»)

Направление подготовки: 09.03.01 «Информатика и вычислительная техника»
Профиль: «Вычислительные машины, комплексы, системы и сети»

Факультет компьютерных технологий и информатики
Кафедра вычислительной техники

К защите допустить:
Заведующий кафедрой
д. т. н., профессор _____ М. С. Куприянов

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА БАКАЛАВРА

Тема: «Микроархитектурная оптимизация обращений к динамически размещаемым объектам в памяти»

Студент _____ И. Г. Ткачев

Руководитель
к. т. н., доцент _____ А. А. Пазников

Консультант по экономическому
обоснованию _____ О. Г. Алексеева

Консультант от кафедры
к. т. н., доцент, с. н. с. _____ И. С. Зуев

Санкт-Петербург
2022 г.

Санкт-Петербургский государственный электротехнический университет «ЛЭТИ» им. В. И. Ульянова (Ленина) (СПбГЭТУ «ЛЭТИ»)

Направление: 09.03.01 «Информатика и вычислительная техника»
Профиль: «Вычислительные машины, комплексы, системы и сети»
Факультет компьютерных технологий и информатики
Кафедра вычислительной техники

УТВЕРЖДАЮ
Заведующий кафедрой ВТ
д. т. н., профессор
(М. С. Куприянов)

«___» _____ 2022 г.

ЗАДАНИЕ
на выпускную квалификационную работу

Студент Ткачев Игорь Геннадьевич Группа № **8307**

1. Тема: «Микроархитектурная оптимизация обращений к динамически размещаемым объектам в памяти»
(утверждена приказом № _____ от _____)

Место выполнения ВКР: кафедра вычислительной техники СПбГЭТУ «ЛЭТИ»

2. Объект и предмет исследования: динамическая память и алгоритмы динамического распределения памяти
3. Цель: Исследование алгоритмов оптимизации размещения данных в памяти на основе предварительного профилирования двоичных файлов программ
4. Исходные данные: Существующие стратегии динамического распределения памяти, а также научные статьи по теме работы
5. Содержание: Пояснительная записка, иллюстративные материалы
6. Дополнительные разделы: Экономическое обоснование ВКР
7. Результаты: Эксперименты с использованием алгоритмов оптимизации

Дата выдачи задания «_____» _____ 2022 г.

Дата представления ВКР к защите «_____» _____ 2022 г.

Руководитель к. т. н., доцент _____ А. А. Пазников

Студент _____ И. Г. Ткачев

Санкт-Петербургский государственный электротехнический университет «ЛЭТИ» им. В. И. Ульянова (Ленина) (СПбГЭТУ «ЛЭТИ»)

Направление: 09.03.01 «Информатика и вычислительная техника»
Профиль: «Вычислительные машины, комплексы, системы и сети»
Факультет компьютерных технологий и информатики
Кафедра вычислительной техники

УТВЕРЖДАЮ
Заведующий кафедрой ВТ
д. т. н., профессор
(М. С. Куприянов)

«___» _____ 2022 г.

КАЛЕНДАРНЫЙ ПЛАН
выполнения выпускной квалификационной работы

Тема: «Микроархитектурная оптимизация обращений к динамически размещаемым объектам в памяти»

Студент Ткачев Игорь Геннадьевич Группа № **8307**

№ этапа	Наименование работ	Срок выполнения
1	Обзор литературы по теме работы	24.02 – 10.03
2	Стратегии динамического распределения памяти	11.03 – 31.03
3	Проблема распределителей общего назначения	01.04 – 09.04
4	Подход профилирования двоичных файлов	10.04 – 02.05
5	Экономическое обоснование ВКР	03.05 – 07.05
6	Оформление пояснительной записки	08.05 – 31.05
7	Представление работы к защите	16.06.2022

Руководитель
к. т. н., доцент _____ А. А. Пазников

Студент _____ И. Г. Ткачев

РЕФЕРАТ

Пояснительная записка 61 стр., 28 рис., 08 табл., 14 ист., 01 прил.

РАСПРЕДИЛИТЕЛИ ПАМЯТИ (АЛЛОКАТОРЫ), ДИНАМИЧЕСКАЯ ПАМЯТЬ, ОПТИМИЗАЦИЯ, АЛГОРИТМЫ, КУЧА, ПУЛ ПАМЯТИ, КЭШ

Тема выпускной квалификационной работы: «Микроархитектурная оптимизация обращений к динамически размещаемым объектам в памяти».

Объектом исследования является динамическая память.

Предметом исследования – алгоритмы динамического распределения памяти.

Цель работы – Исследование алгоритмов оптимизации размещения данных в памяти на основе предварительного профилирования двоичных файлов программ.

В данной работе приведены существующие основные стратегии динамического выделения памяти, описана проблема распределителей общего назначения и детально описан один из подходов по оптимизации алгоритмов распределения памяти – метод профилирования двоичных файлов программ.

В ходе выполнения работы проведены эксперименты с использованием алгоритмов оптимизации в области динамического распределения памяти.

Результаты работы могут быть использованы разработчиками программного обеспечения и иными техническими специалистами при решении проблем с оптимизацией памяти в каждом конкретном случае (проекте).

ABSTRACT

Today, general-purpose memory allocators dominate the landscape of dynamic memory management. While they can provide reasonably good execution across a wide range of tasks, it is an unfortunate reality that their behaviour for any particular task can be highly suboptimal. By catering primarily to average and worst-case usage patterns, these allocators deny programs the advantages of domain-specific optimisations, and thus may inadvertently place data in an order that hinders performance, generating unnecessary cache misses.

Therefore, the purpose of this work is to study approaches to optimizing memory allocation algorithms with a detailed description of one of them.

СОДЕРЖАНИЕ

ОПРЕДЕЛЕНИЯ, ОБОЗНАЧЕНИЯ И СОКРАЩЕНИЯ

В настоящей пояснительной записке применяют следующие термины с соответствующими определениями, обозначения и сокращения:

ВКР – выпускная квалификационная работа

Аллокатор (англ. Allocator) или распределитель памяти в языке программирования C++ – специализированный класс, реализующий и инкапсулирующий малозначимые детали распределения и освобождения ресурсов компьютерной памяти

Метапрограммирование – вид программирования, который позволяет писать программы, которые создают другие программы

Эври́стика (от др.-греч. εὑρίσκω – «отыскиваю», «открываю») – научная область, изучающая специфику созидательной деятельности

HALO (Heap Allocation Layout Optimizer) – оптимизатор компоновки распределения кучи

Id (идентификатор) – базовая лексическая единица определенного языка программирования , созданная программистом в соответствии с синтаксисом данного языка программирования, используемая для идентификации и ссылки на определенный элемент исходного кода

TLB (англ. translation lookaside buffer) – буфер ассоциативной трансляции – специализированный кэш центрального процессора, используемый для ускорения трансляции адреса виртуальной памяти в адрес физической памяти

ВВЕДЕНИЕ

Поскольку разрыв между быстродействием памяти и процессора продолжает увеличиваться, эффективное использование кэша становится более важным, чем когда-либо. В то время как компиляторы уже давно используют такие методы, как переупорядочение базовых блоков, разделение циклов, а также интеллектуальное распределение регистров для улучшения поведения кэша программ, расположение динамически выделяемой памяти остается в значительной степени недоступным для статических инструментов.

Сегодня, когда программа *C++* вызывает оператор *new* или программа *C* – функцию *malloc*, ее запрос удовлетворяется распределителем общего назначения, не имеющим подробных знаний о том, что делает программа или как используются ее объекты данных. Естественно, это приводит к неэффективности и может породить программы, производительность которых зависит от прихотей общего алгоритма компоновки данных, генерируя ненужные промахи кэша, промахи *TLB* и сбои предварительной выборки. В то время как пользовательские распределители и тщательные методы программирования могут решить эти проблемы с поразительной эффективностью [4], за пределами высокопроизводительных ниш, таких как программирование игр, эти решения могут быть сложными и, в целом, легко ошибиться [5].

Всё это подталкивает к изучению различных оптимизаций алгоритмов распределения памяти. И основное внимание в данной работе будет уделено подходу профилирования двоичных файлов программ с целью разумного размещения данных в памяти во время выполнения, примером реализации которого является рассматриваемый конвейер оптимизации компоновки данных кучи *HALO* [9].

Цель работы: исследование алгоритмов оптимизации размещения данных в памяти на основе предварительного профилирования двоичных файлов программ.

Объектом исследования является динамическая память.

Предметом исследования являются алгоритмы динамического распределения памяти.

Результат ВКР (выпускная квалификационная работа) заключается в выполнении экспериментов с применением алгоритмов оптимизации в области динамического распределения памяти.

Для достижения поставленной цели и получения определенного результата в первом разделе ВКР освещаются основные стратегии динамического выделения памяти. Во втором разделе описана проблема распределителей общего назначения. В третьем разделе детально описан один из подходов по оптимизации алгоритмов распределения памяти – метод профилирования двоичных файлов программ и выполнены эксперименты. В четвертом (дополнительном) разделе выполнен расчет себестоимости ВКР.

1 Стратегии динамического распределения памяти

Достаточно часто разработчикам приходится решать задачи, связанные с особыми требованиями к проектам, и знание о стратегиях динамического распределения памяти будут полезны. Существует множество стратегий (методов), применимых в различных ситуациях и в данном разделе рассмотрены некоторые из них.

1.1 Линейное размещение

Linear (линейный) *Allocator* [13] линейным способом выделяет по очереди фрагменты памяти из фиксированного пула (рисунок 1.1).

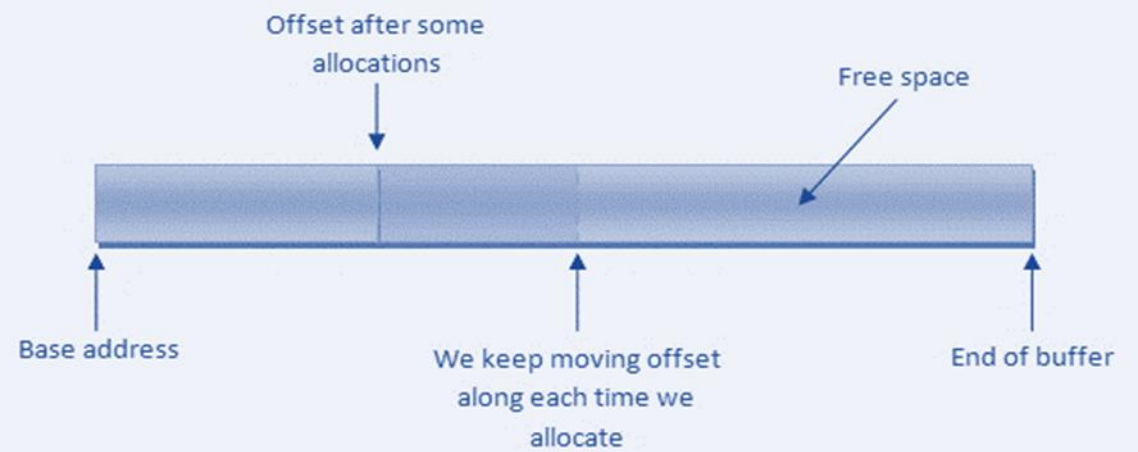

Рисунок 1.1 – Схема линейного аллокатора

Смысл в том, чтобы сохранить заданному распределителю указатель на начало блока памяти и использовать другой указатель, который будет перемещаться всякий раз, когда выделение из аллокатора завершено. В данном распределении внутренняя фрагментация минимальна, так как все элементы добавляются последовательно и заключается только в выравнивании между ними.

Рассмотрим на примере, как управляет данный аллокатор блоком памяти равным 14 байтам. На рисунке 1.2 изображена схема блока памяти линейного аллокатора, где обозначены:

а) указатель на начало памяти (*start*);

б) указатель на конец памяти (*end*);

в) указатель на конец зарезервированной участка (*used*).

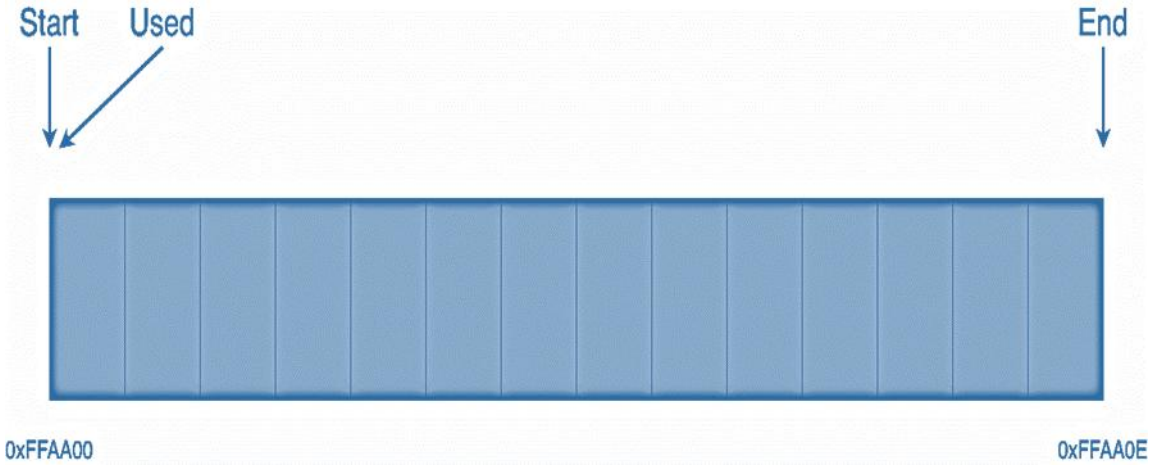

Рисунок 1.2 – Блок памяти линейного аллокатора равный 14 байтам

Предположим, что распределитель получил запрос на выделение 4 байт памяти. Как будет работать аллокатор? Алгоритм работы распределителя следующий:

а) определить имеется ли достаточный объем памяти для выделения;

б) сохранить указатель *used* для пользователя, как указатель на блок выделенной памяти из аллокатора;

в) сдвинуть указатель *used* на величину равную поступившему запросу, а именно, на 4 байта как изображено на рисунке 1.3.

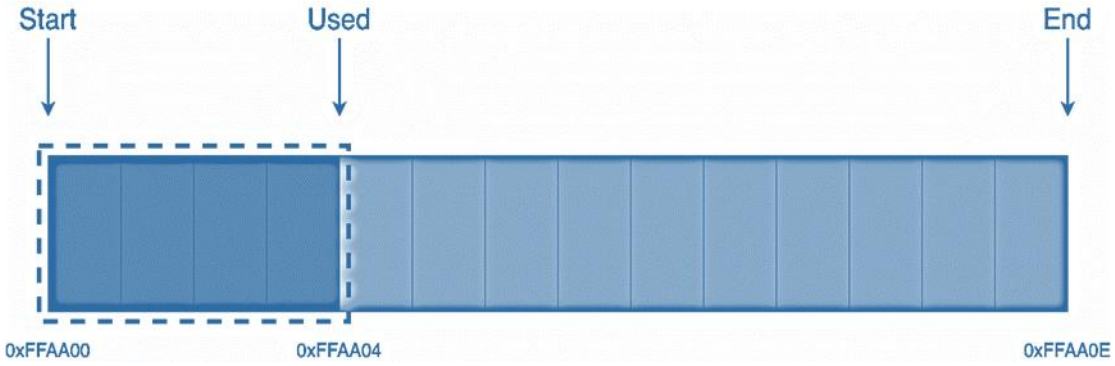

Рисунок 1.3 – Выделение 4 байт памяти

Если поступает новый запрос на выделение 8 байт, то аллокатор будет работать по такому же алгоритму, какой бы размер выделяемого блока памяти ни был, как показано на рисунке 1.4.

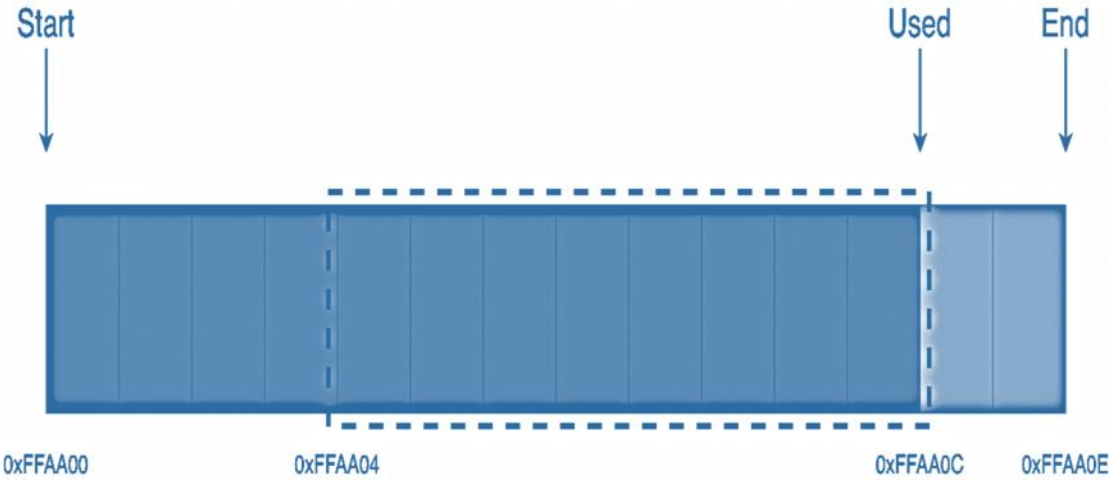

Рисунок 1.4 – Выделение 8 байт памяти

На рисунке 1.5 изображена работа линейного аллокатора, при запросе на выделение только 1 байта, которое не требует выравнивать блоки памяти. Здесь аллокатор действует точно также, как и в предыдущих случаях.

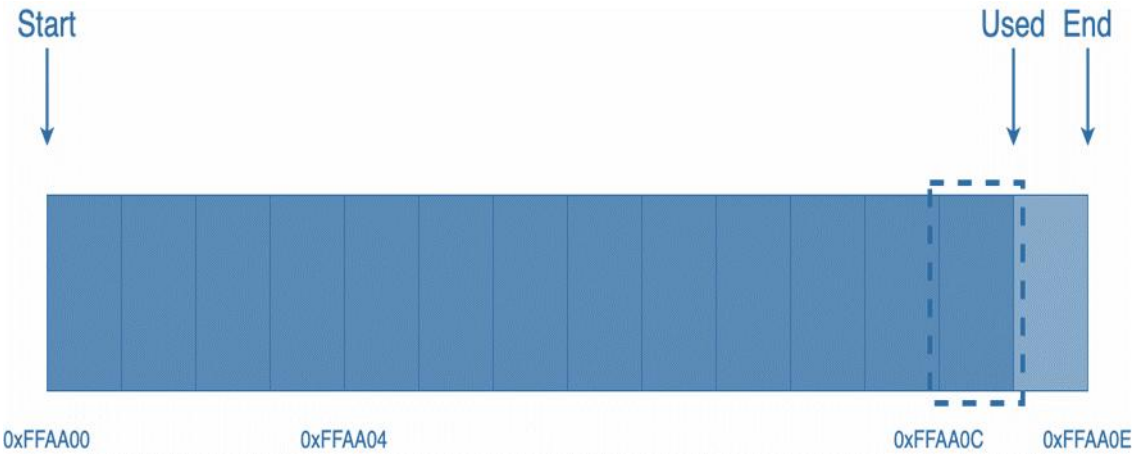

Рисунок 1.5 – Выделение 1 байта памяти

Далее рассмотрим, как действует распределитель при запросе на выделение блоков памяти с выравниванием. Например, выравнивание адресов кратных 2, как изображено на рисунке 1.6.

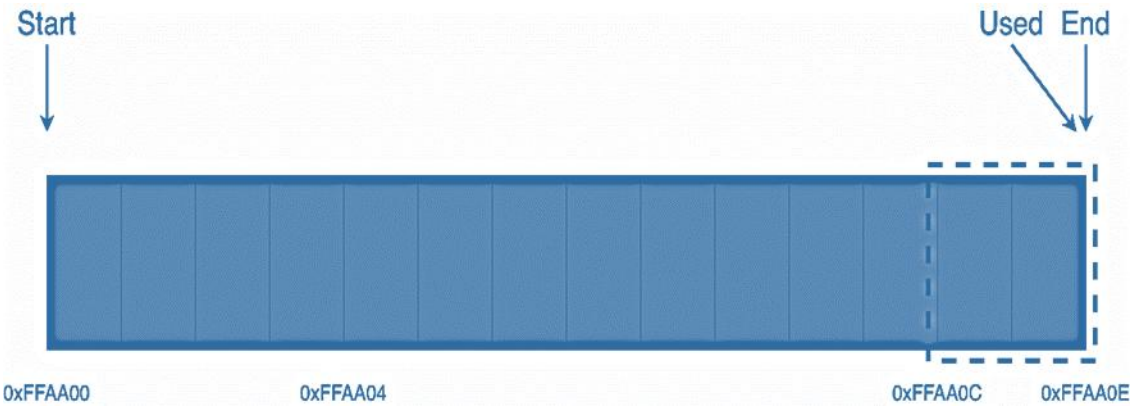

Рисунок 1.6 – Выравнивание блоков кратных 2

В данном случае алгоритм работы аллокатора будет немного другой. Кроме данных равных 1 байту, из памяти задействуется еще один дополнительный байт, который используется для выравнивания. В этом и заключается фрагментация памяти внутри данного аллокатора. Особенность данного вида аллокатора в том, что нельзя выборочно освободить некоторые блоки памяти, только всю память внутри аллокатора и затем работать, как с абсолютно пустым.

1.2 Стековое размещение

Stack Allocator [13]**,** в принципе, является усовершенствованной версией линейного распределителя. Как и в линейном распределителе фрагментация в данной модели минимальная.

Для того, чтобы понять, как работает данный алллокатор, используем для примера, такой же блок памяти, как и в предыдущем линейном аллокаторе, равный 14 байтам. Условные обозначения используем те же (*start*, *end*, *used*), как показано на рисунке 1.7.

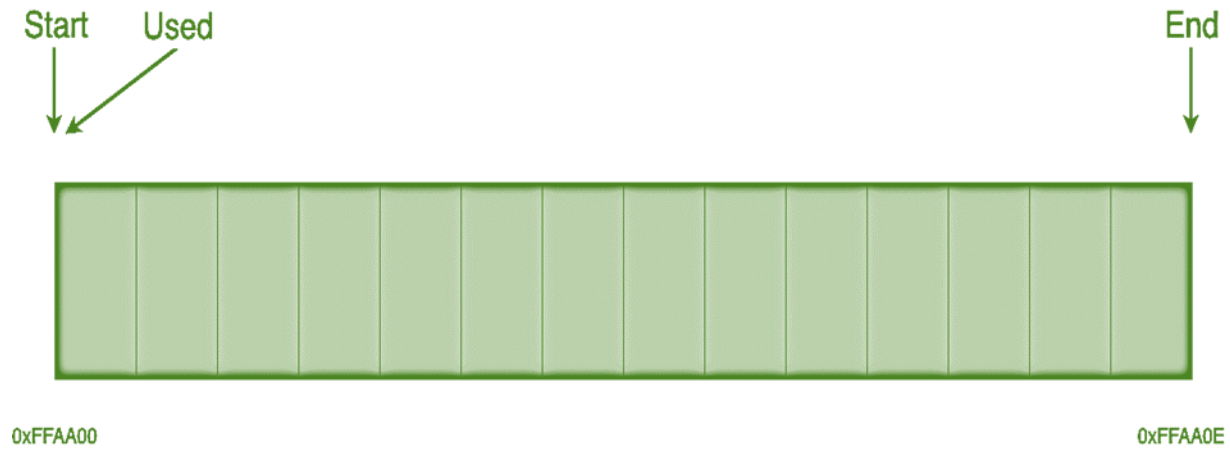

Рисунок 1.7 – Блок памяти в 14 байт

При поступлении от пользователя запроса на выделение памяти кроме запрашиваемого объема памяти, необходимо выделить заголовок, в котором будет информация о том, сколько байт выделено. В нашем примере размер заголовка равен 2 байтам. При запросе о выделении памяти размером 2 байта, пользователю будет отдан указатель не на заголовок (*header*), а на блок, следующий сразу за заголовком, который в нашем примере имеет адрес *0xFFAA02*, как показано на рисунке 1.8.

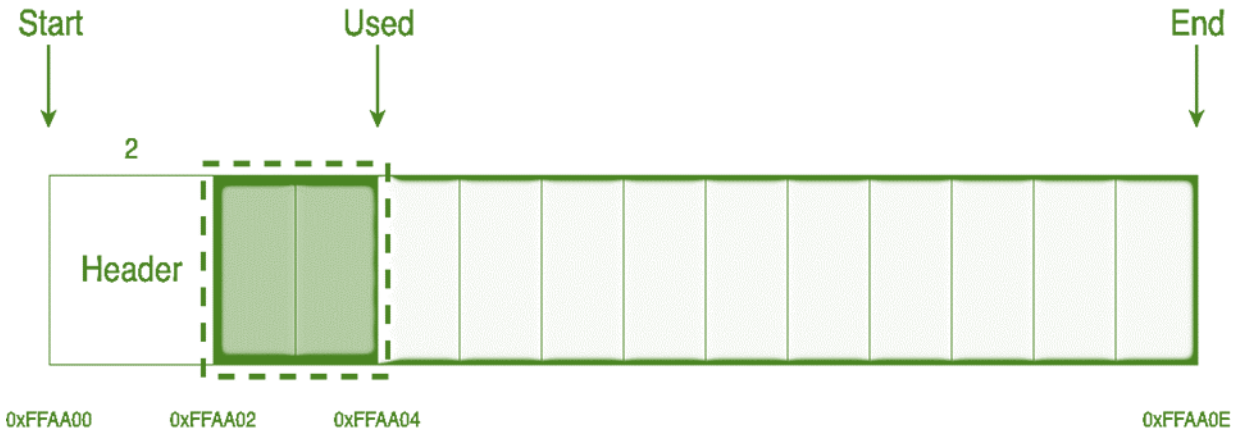

Рисунок 1.8 – Выделение 2 байт памяти

Точно так же будет происходить выделение памяти, например, объемом равным 6 байтам (рисунок 1.9).

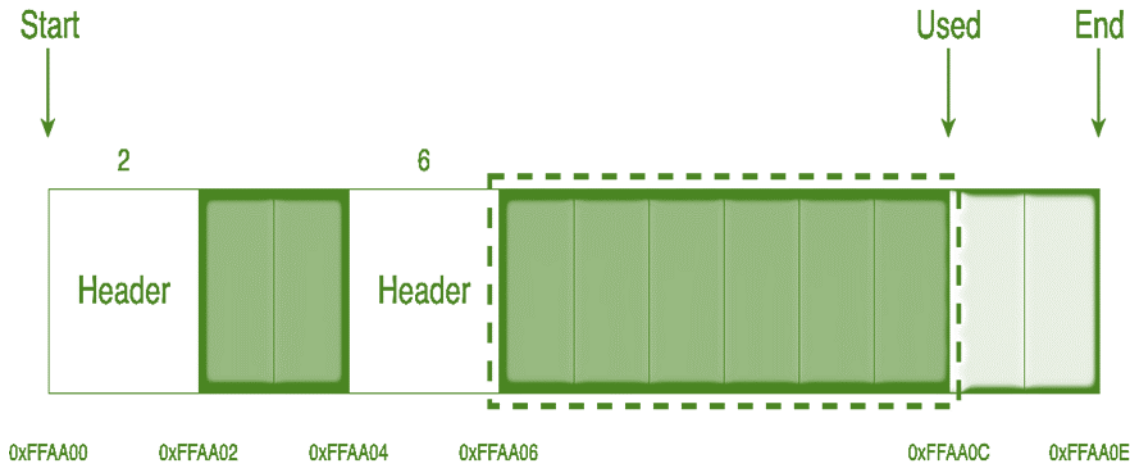

Рисунок 1.9 – Выделение 6 байт памяти

Блок освобождается следующим образом:

1) от указателя, на освобождение которого пришел запрос от пользователя, отнимается размер заголовка;

2) происходит разыменование значения;

3) сдвигается указатель *used* на размер заголовка вместе с размером блока, полученного из заголовка, как на рисунке 1.10.

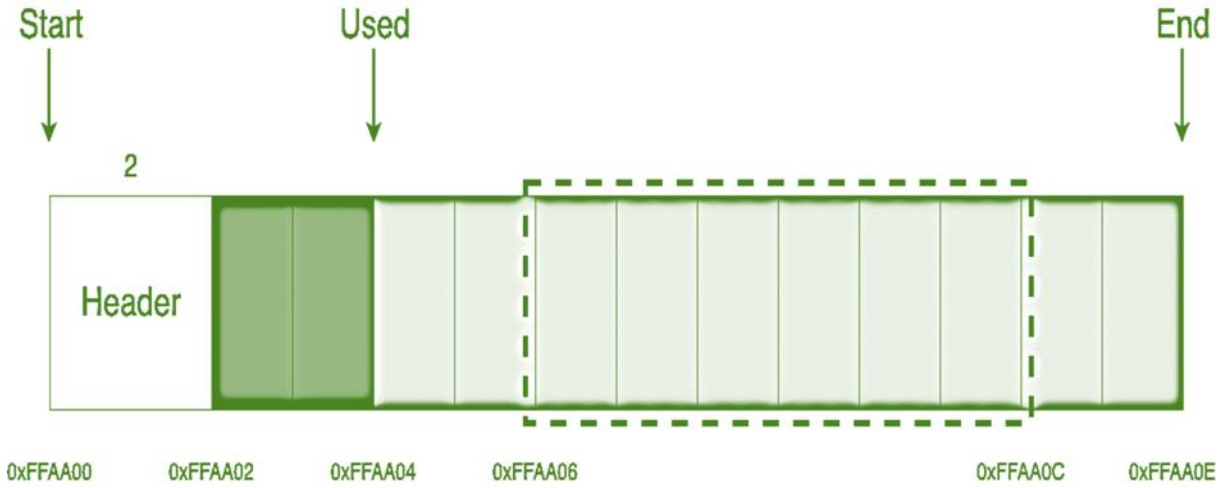

Рисунок 1.10 – Освобождение блока памяти

В процессе выделения блоков памяти возможно освобождение «случайных» блоков памяти, что является негативным фактором и может потребовать дополнительной проверки аллокатора.

1.3 Размещение блоков фиксированного размера в пуле

Данный вид аллокатора участок памяти большого размера разделяет на равные маленькие участки [13]. При получении запроса на выделение блока памяти аллокатор возвращает свободный участок памяти фиксированного размера, а при запросе на освобождение – сохраняет этот участок памяти для последующего использования. Поэтому распределение происходит достаточно быстро, а фрагментация, как и в предыдущих двух видах распределителей, незначительная.

Рассмотрим на примере, как управляет аллокатор блоком памяти размером 12 байт. В примере используем следующие условные обозначения:

а) *start* – указатель на начало памяти;

б) *end* – указатель на общий размер памяти;

в) *freeblocks* – список из адресов свободных блоков в аллокаторе.

Средством для хранения данных о том, что блок свободен или занят здесь используется односвязный список (рисунок 1.11). Звенья данного списка могут храниться в свободных блоках памяти, что не приведет к лишним расходам памяти.

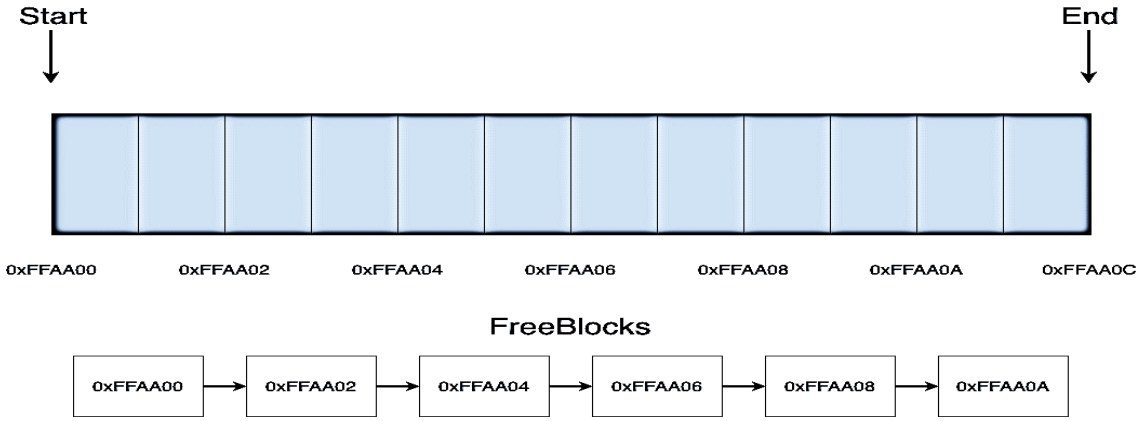

Рисунок 1.11 – Односвязный список из адресов свободных блоков

Принцип работы данного аллокатора так же, как и в предыдущих двух видах не сложный. При поступлении запроса на выделение одного блока памяти аллокатор проверяет, есть ли звенья в списке свободных блоков, если

звенья в списке отсутствуют, значит память в аллокаторе закончилась. При наличии хотя бы одного звена аллокатор изымает корневое или хвостовое звено и отдает его адрес пользователю. На рисунке 1.12 показано использование хвостового звена.

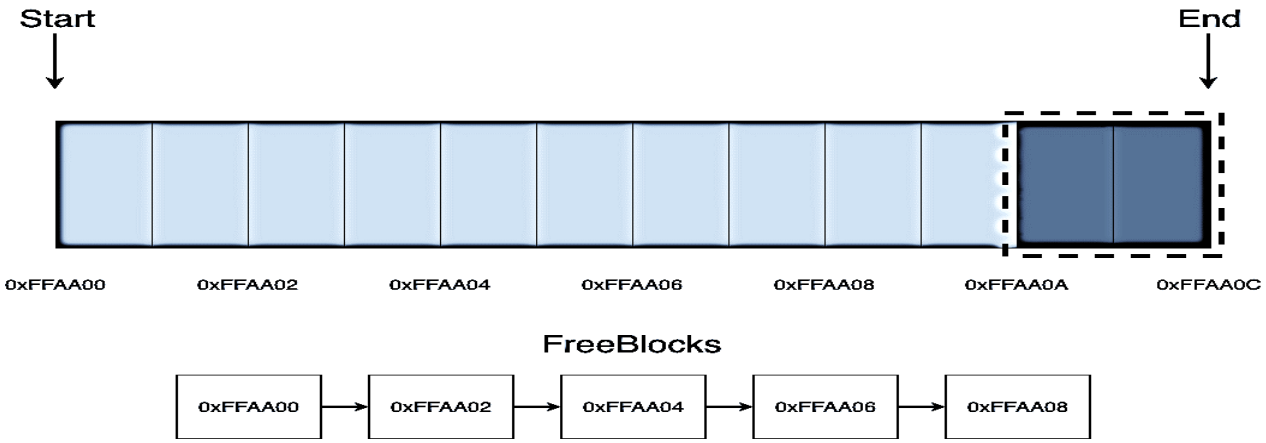

Рисунок 1.12 – Выделение одного блока памяти

На рисунке 1.13 изображены действия аллокатора при поступлении запроса на выделение нескольких блоков памяти. Распределитель работает точно так же, как и в предыдущем примере – по очереди проделывает те же действия.

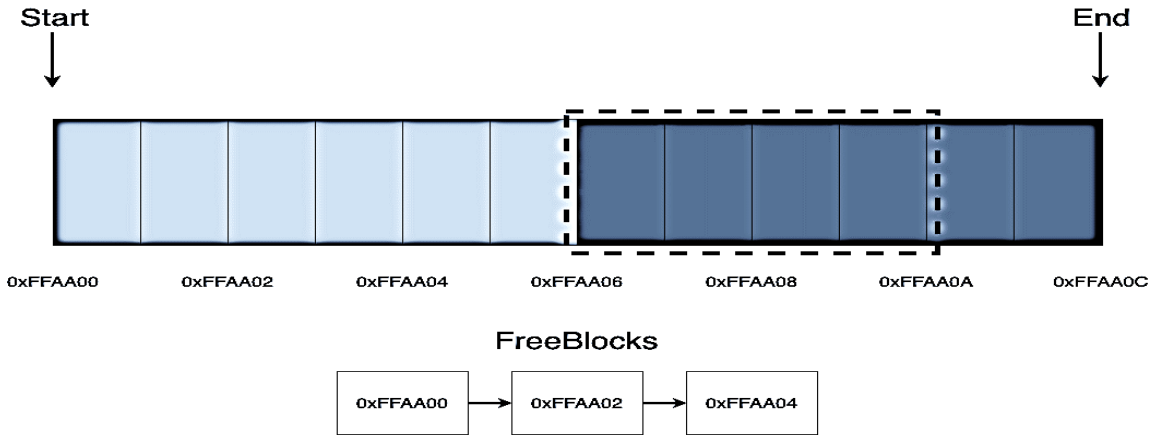

Рисунок 1.13 – Выделение нескольких блоков памяти

При поступлении запроса на освобождение блока памяти, аллокатор добавляет этот адрес в один из концов односвязного списка. Сложность может возникнуть, если поступит запрос с адресом блока, который не соответствует адресу памяти аллокатора, например, *0xEFAB12*, тогда придется отдать пользователю тот участок памяти, который нам не принадлежит, и это в итоге может привести к выходу за пределы памяти, которой не управляет аллокатор и в итоге к краху программы. Сложность может возникнуть и при запросе пользователя освободить адрес, находящийся в области памяти аллокатора, размер которого не равен адресу начала какого-либо из блоков, например, блока с адресом *0xFFAA07*. Эта операция может привести к неопределенному поведению [9] .

1.4 Размещение соседей

Размещение соседей (*Buddy*) [6, 7] представляет собой разделение на небольшие части большого блока памяти с целью удовлетворения поступившего запроса. Две маленькие части блока памяти одинакового размера называются приятелями. Точно также один из двух приятелей будет делиться на еще более мелкие участки, до тех пор, пока запрос не будет исполнен. Этот вид размещения имеет важное преимущество. Два приятеля способны объединиться в большой блок, чтобы соответствовать поступившему запросу.

Например, если поступил запрос на выделение блока памяти размером 25Kb, то выделяется блок размером 32Kb, как показано на рисунке 1.14.

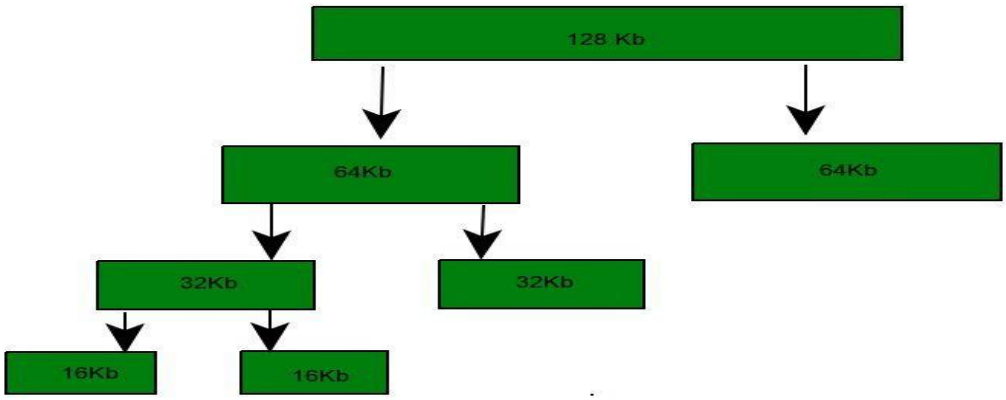

Рисунок 1.14 – Схема распределения Buddy

У данной системы распределения есть ряд существенных преимуществ:

- небольшая внешняя фрагментация по сравнению с более простыми методами;

- достаточно быстрое выделение памяти и освобождение памяти;

- коалесцирование, то есть быстрота объединения соседних приятелей в более крупные сегменты;

- вычисление адреса происходит легко.

Недостатком системы *buddy* считается внутренняя фрагментация, поскольку запрос памяти размером 36 КБ будет выполнен только сегментом 64 КБ, а оставшаяся память израсходуется впустую.

Вывод: Каждый из рассмотренных аллокаторов по-своему решает проблемы с фрагментацией, с нехваткой памяти, со скоростью получения и освобождения блоков необходимого размера, с временем жизни объектов и занимаемой ими памятью. Поэтому в каждой конкретной ситуации нужно подбирать самый оптимальный аллокатор или создавать композицию из нескольких аллокаторов.

2 Проблема распределителей общего назначения

Большинство приложений управляют динамической памятью с помощью распределителя памяти общего назначения, такого как *ptmalloc2 (glibc), tcmalloc* или *jemalloc*. Хотя они могут обеспечить достаточно хорошее выполнение в широком диапазоне задач, их поведение для любой конкретной задачи может быть крайне неоптимальным. Ориентируясь в первую очередь на средние и наихудшие модели использования, эти распределители не могут приспособиться к специфическому поведению отдельных программ и, таким образом, лишают их преимуществ оптимизации для конкретной предметной области [8].

Одна из таких упущенных возможностей, имеющая особое значение, касается размещения объектов данных с сильной временной локальностью. Поскольку распределители общего назначения традиционно не обладают какими-либо глубокими знаниями о том, как значения используются в программе, они могут полагаться только на простую эвристику для размещения данных и, таким образом, могут непреднамеренно разбросать сильно связанные объекты по всей куче.

В частности, для удовлетворения запросов на распределение с разумными затратами времени и пространства почти все современные распределители общего назначения, включая *ptmalloc2, jemalloc и tcmalloc*, основаны на схемах распределения с разделением по размеру. То есть они организуют свободные блоки и служебную информацию вокруг фиксированного количества классов размеров. В результате распределения располагаются совместно на основе их размера и порядка, в котором они выделяются, как показано на рисунке 2.1.

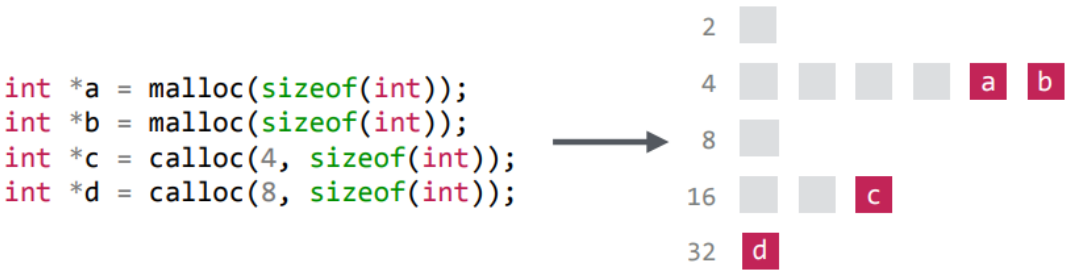

Рисунок 2.1 – Иллюстрация простого распределителя с разделением по классам размера степени двойки

В то время как стратегия размещения такого рода является разумным подходом в отсутствие какой-либо дополнительной информации, любая программа, в которой сильно связанные и часто используемые объекты данных распределяются непоследовательно или принадлежат к разным классам размеров, вероятно, будет работать относительно плохо в рамках этой модели. В таких случаях связанные объекты вряд ли будут помещены в одну и ту же строку кэша или, возможно, даже на одну и ту же страницу распределителем с разделением по размеру, и поэтому могут генерировать ненужные промахи кэша и *TLB* [14], а также сбои предварительной выборки [12]. Это может быть особенно проблематично для программ, написанных на таких языках, как *C++*, которые поощряют выделение множества относительно небольших объектов, размещение которых может иметь решающее значение для общей производительности [3].

3 Подход профилирования двоичных файлов

В данном разделе детально исследован метод оптимизации, основанный на предварительном профилировании двоичных файлов программ с целью разумного размещения данных в памяти во время выполнения, примером реализации которого является конвейер оптимизации компоновки данных кучи HALO [9].

3.1 Ожидаемый результат оптимизации

Для демонстрации ожидаемого результата оптимизации, применяемой в данном подходе, взята тестовая программа *povray* из *SPEC CPU* 2017 [32]. Её шаблоны доступов были сильно упрощены и полученная версия представлена ниже.

```
// Выделение
Object *list = NULL;
while (!eof) {
    Token token = get_token();
    if (token.type == A) {
        Object *obj = create_a();
        obj->sibling = list;
        list = obj;
    } else if (token.type == B) {
        Object *obj = create_b();
        obj->sibling = list;
        list = obj;
    } else {
        Object *obj = create_c();
        do_something(obj);
    }
}

// Доступ
Object *obj = list;
while (obj) {
    process(obj);
    obj = obj->sibling;
}
```

Программа считывает токены (метки) из входного потока, выделяя объект кучи для каждого с помощью процедуры, специфичной для типа токена. Когда весь ввод потребляется, программа затем проходит некоторое подмножество этих объектов, в этом случае получая доступ к объектам типов *A* и *B*, оставляя в стороне объекты типа *C*.

И типичный распределитель может размещать эти объекты, как показано на рисунке 3.1(a). Однако при этом объекты располагаются исключительно в соответствии с порядком, в котором они были выделены, в то время как способ фактического доступа к ним сильно зависит от их типа. Эта деталь, на которую традиционные распределители не обращают внимания, означает, что распределитель невольно создает макет с плохой пространственной локальностью, разбрасывая несвязанные объекты типа *C* между объектами типов *A* и *B*, снижая эффективность кэша и приводя к неоптимальной производительности.

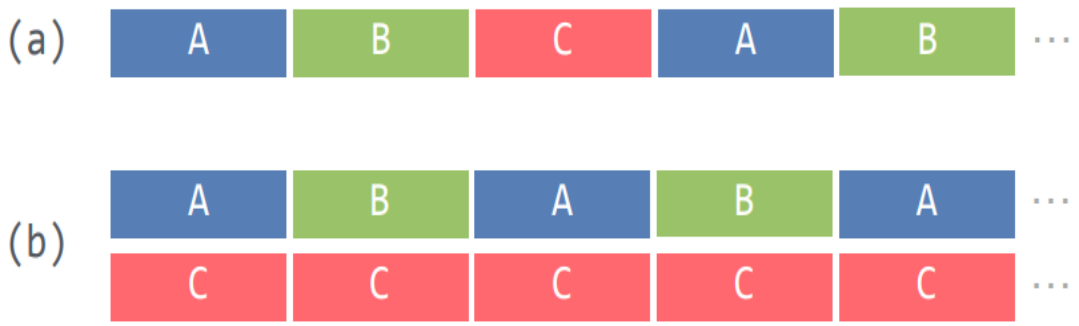

Рисунок 3.1 – Два возможных варианта размещения данных в куче для рассматриваемого примера кода

Распределители, повышающие локальность, напротив, стремятся использовать именно такие внеочередные закономерности для принятия улучшенных решений о размещении данных. В этом случае такой распределитель может заметить, что, поскольку каждый из трех типов распределений возникает в результате отдельного вызова *malloc* (в рамках

каждой из процедур *create_**) и каждый из них, вероятно, будет иметь разные характеристики доступа. Анализируя природу этих характеристик с помощью выполнения профилирования, распределитель затем может перенаправить выделения типов *A* и *B* в отдельный от *C* пул памяти, как показано на рисунке 3.1(b). Теоретически, это должно позволить циклу доступа работать без добавления каких-либо объектов типа *C* в кэш, повышая эффективность кэша и, следовательно, производительность.

К сожалению, в то время как существующие решения могут с легкостью устранять недостатки в простых примерах, подобных этому, реальные программы могут создавать дополнительные сложности, которые в конечном итоге могут привести к их сбою. Если рассмотреть пример в его исходном контексте в *povray,* например, где типы *A* и *B* соответствуют геометрическим объектам *CSG*, можно увидеть, что почти все данные кучи выделяются с помощью функций-оболочек *pov::pov_malloc,* что противоречит подходам, которые стремятся охарактеризовать распределения используя только точки вызова *malloc*. Кроме того, даже если эти функции-оболочки удалены, объекты геометрии, выделенные из разных мест в программе, используются по-разному, причем доступ к некоторым типам осуществляется гораздо реже, чем к другим. Таким образом, для наиболее эффективного разделения объектов требуется подробная контекстная информация, помимо той, которая может быть разумно извлечена путем обхода динамического стека вызовов во время выполнения.

Результатом оптимизации должно стать устранение одной из основных проблем, связанной с пространственной локализацией. Использование исследуемых алгоритмов группировки и идентификации, позволят создавать сплоченные группы распределения, которые используют весь стек вызовов, и идентифицируют их во время выполнения с чрезвычайно низкими накладными расходами.

3.2 Проектирование и реализация

Высокоуровневый обзор рассматриваемого проекта *HALO* [9] представлен на рисунке 3.2. В общих чертах, сначала целевая программа профилируется, чтобы получить представление о том, как связаны распределения, создаваемые в различных контекстах. Затем контексты распределения группируются в тесно связанные кластеры, которые могут извлечь выгоду из улучшенного пространственного расположения. После этого сформированные группы проходят идентификацию, а целевая программа переписывается с помощью пользовательского прохода *BOLT*, чтобы эти группы можно было эффективно распознавать во время выполнения. И наконец, генерируется специализированный распределитель, который должен быть связан с программой во время выполнения, чтобы разрешить применение новой политики размещения, ориентированной на группы.

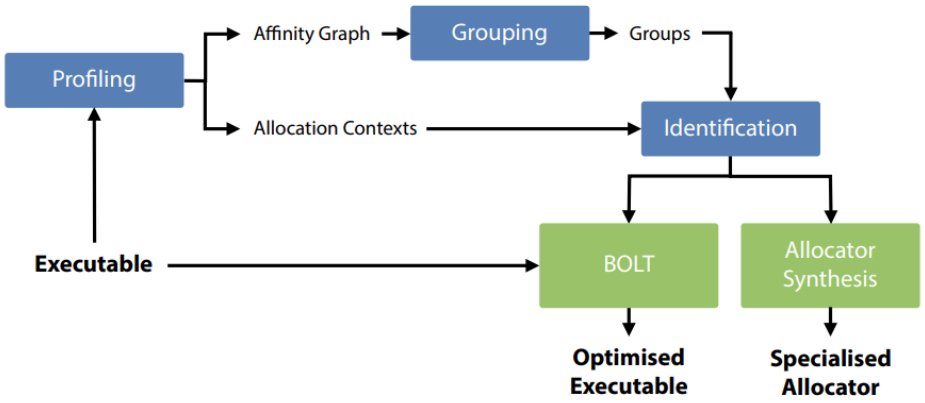

Рисунок 3.2 – Высокоуровневый обзор конвейера оптимизации *HALO*

3.2.1 Профилирование

Для получения информации о том, как целевая программа обращается к данным кучи, выполняется её запуск с помощью пользовательского инструмента анализа, написанного на платформе *Intel Pin*. Чтобы определить,

как связаны выделения, этот инструмент использует вызовы всех функций управления памятью *POSIX.1*, отслеживая текущие данные с детализацией на уровне объекта. Весь процесс анализа можно поделить на несколько параллельно выполняющихся подзадач.

3.2.1.1 Формирование контекста выделения

Каждое выделение имеет контекст (цепь предшествующих вызовов), в котором оно было произведено, и для его формирования используется стек вызовов. Для этого сперва заводится «теневой» стек, который несколько отличается от истинного. После чего каждая команда вызова (или другая межфункциональная передача управления) обрабатывается по следующему принципу:

- если это не инструкция возврата и цель вызова статически связана с основным двоичным файлом или является одной из нескольких отслеживаемых извне функций, таких как *malloc* или *free*, то в «теневой» стек следует добавить запись (функция, точка вызова);
- если это инструкция возврата, то сократить «теневой» стек до точки возврата.

Точки вызовов могут быть косвенными (в случае библиотечных процедур) и отслеживаются до их ближайших исходных точек в основном исполняемом файле. Также если «теневой» стек содержит рекурсивные вызовы, то он преобразуется в каноническую сокращенную форму, в которой сохраняются только самые последние из любой пары (функция, точка вызова). Таким образом, цепь вызовов, сформировавшаяся в «теневом» стеке к моменту выделения, покажет его контекст, пример которого продемонстрирован на рисунке 3.3.

```
CTX 0:
        __libc_malloc from 0x401064
        .plt.sec from 0x401420
        create_target from 0x40165d
        my_main from 0x401808
        main from 0
```

Рисунок 3.3 - Пример контекста выделения

3.2.1.2 Отслеживание выделений и привязка контекста

Совершается проход по стеку вызовов и при обнаружении функций выделения памяти, для них извлекаются контексты из «теневого» стека. Каждое выделение и контекст заносятся в соответствующие хэш-таблицы. Таблица выделений содержит такие параметры, как адрес, размер, *id* объекта и *id* контекста. А таблица контекстов — контекст и его *id*. Поскольку выделения могут происходить в одном и том же контексте, совпадающие контексты в таблицу повторно не заносятся, а выделениям ставится одинаковый *id* контекста. Кроме этого для установления связей между выделениями в рамках каждого контекста определяется порядок, в котором они происходили. Например, есть контексты *A, B, C,* и соответствующие им выделения создавались в таком порядке: *a1, a2, b1, b2, c1, c2*. Тогда будет сформирована таблица, как показано ниже (таблица 3.1).

Таблица 3.1. — Пример таблицы выделений

Id	Выделение	Id контекста	Id до	Id после
1	a1	0	0	3
2	b1	1	0	4
3	a2	0	1	0
4	b2	1	2	0
5	c1	2	0	6
6	c2	2	5	0

3.2.1.3 Формирование графа связей между контекстами

Установив, какие распределения производятся и в каком контексте, инструмент анализа затем пытается смоделировать взаимосвязи между контекстами, анализируя шаблоны доступа целевой программы. С этой целью генерируется попарный граф, узлами которого являются контексты, как описано выше, а ребра которого взвешены в соответствии с количеством одновременных обращений к объектам, выделенным из них в пределах некоторого фиксированного окна.

По мере выполнения целевой программы ее операции чтения и записи анализируются. Когда осуществляется доступ к объекту кучи, отслеживаемому инструментом, этот доступ добавляется в очередь близости. Она содержит идентификаторы объектов данных, к которым последний раз обращались. Элементы, для которых размер записей между ними в очереди близости составляет менее A байт, определяем как близкие, где A — расстояние в байтах, на котором неявно определена очередь. Пример такой очереди представлен на рисунке 3.4. Здесь программа перебирает 10 объектов, делая 4-байтовые обращения, каждый из которых изображен в виде прямоугольника. При $A = 32$ самый новый элемент (оранжевый) будет считаться близким к семи другим слева от него (черные).

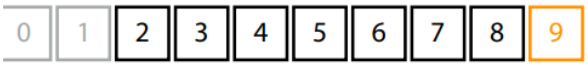

Рисунок 3.4 - Визуальное изображение очереди близости

Когда доступ a к объекту кучи u, выделенному из контекста x, добавляется в очередь близости в результате чтения или записи, очередь просматривается для идентификации всех родственных связей с этим новым доступом. В результате вес на ребре (x, y) увеличивается для каждого объекта v, выделенного из контекста y, при последних обращениях к нему на A байт в очереди, при условии соблюдения следующих четырех ограничений:

Устранение дубликатов: последовательные обращения на машинном уровне к одному объекту считаются частью одного и того же доступа на макроуровне и, следовательно, не вызывают повторного обхода очереди.

Отсутствие связи с собой: объекты не могут быть связаны сами с собой (u != v), поскольку они занимают только одну ячейку памяти.

Отсутствие двойного подсчета: каждый уникальный объект v может быть связан с u не более одного раза за один обход очереди.

Возможность совместного распределения: никакие распределения, сделанные между u и v в хронологическом порядке, не могут исходить ни из x, ни из y.

Из этого, последнее наименее интуитивно понятно. Это гарантирует, что было бы возможно фактически совместно размещать u и v во время выполнения, если бы все объекты, происходящие из контекстов x и y, были выделены последовательно из общего пула. Псевдокод, демонстрирующий данную проверку представлен ниже.

```
// Проверка на возможность совместного распределения
bool is_coallocatable(allocation u, allocation v) {
    // Проверка на соответсвие порядку выделения
    if (u.id < v .id) {
        allocation tmp = u;
        u = v;
        v = tmp;
    }

    // Извлечение id предыдущего выделения в рамках
    // контекста объекта u
    ObjectId u_pred = u.predecessor;
    // Извлечение id следующего выделения в рамках контекста
    //объекта v
    ObjectId v_succ = v.successor;
    // Проверка на попадание выделений в отрезок
    // [u_pred, v_succ]
    return (!u_pred || v.id >= u_pred) &&
              (!v_succ || u.id <= v_succ);
}
```

Как только целевая программа завершает выполнение, узлы графа перебираются от наиболее доступных к наименее доступным. При этом их количество обращений добавляется к текущему итогу, и после учета 90% всех наблюдаемых обращений любые оставшиеся узлы отбрасываются и не вносятся в сгенерированный граф. Это помогает уменьшить шум за счет исключения ненужных контекстов из графа. На рисунке 3.5 приведен пример сформированного графа, в котором до знака решетки обозначены узлы (*id* контекста, количество доступов), а после — ребра (контекст − контекст, количество совместных доступов).

```
2 1026
0 1024
3 1022
#
0 0 1022
2 0 2048
2 2 1024
3 0 1021
3 2 1021
3 3 1020
```

Рисунок 3.5 - Пример графа связей

3.2.2 Группирование

Получив представление допустимых временных взаимосвязей в целевой программе, разрабатывается схема, с помощью которой эти взаимосвязи могут быть использованы для повышения производительности. Для этой цели набор контекстов выделения разделяется на группы таким образом, чтобы члены каждой группы могли быть выделены из общей области памяти для улучшения локальности кэша. Чтобы создать такие группы связанных распределений, применяется простой жадный алгоритм, который генерирует кластеры, более поддающиеся совместному

распределению на основе регионов, чем при стандартной модульности, *HCS* или методах кластеризации на основе разрезов. Код описывающий данный алгоритм приведен ниже.

```
# Процесс образования групп
def group(graph, args):
    groups = []
    # Удаление коротких ребер из графа
    remove_short_edges(graph, args.min_edge_weight)
    # Сортирование вершин по наибольшему весу инцидентных
    # ребер и добавление их в список доступных
    # (негруппированных) вершин
    available = rank_available_nodes(graph, graph.nodes)
    while available:
        # Извлечение первой вершины из доступных и
        # добавление в группу
        group = [available.pop(0)]

        # Наращивание группы
        # Совершается пока не достигнут предел размера
        # группы и есть кандидаты на добавление
        while len(group) < args.max_group_size:
            best_match = None
            best_score = 0.0
            # Извлечение подграфа для текущей группы
            group_graph = nx.Graph(graph.subgraph(group))
            # Просмотр доступных для слияния вершин
            for stranger in available:
                # Получение значения улучшения после слияния
                # группы и вершины
                improvement = merge_benefit(group, [stranger],
                                            graph, args.tolerance)
                # Если выгода от слияния с данной вершиной
                # больше чем от предыдущих, то сохранить её
                if improvement > best_score:
                    best_match = stranger
                    best_score = improvement
            # Если нет выгоды от слияния с какой-либо
            # вершиной, то завершить формирование группы
            if best_match is None:
                break
            # Удаление «лучшей» вершины из списка доступных
            # и её добавление в группу
            available.remove(best_match)
            group.append(best_match)
        # Пересортирование вершин с учётом оставшихся
        # негруппированных
        available = rank_available_nodes(graph, available)
```

```
            # Получение суммы весов инцидентных ребер для каждой
            # вершины
            node_degrees, _ = degree(group_graph)
            # Получение веса группы
            group_accesses = sum(node_degrees.values())
            # Добавление группы в список
            groups.append((group, group_accesses))
    return groups
```

Алгоритм работает путем многократного увеличения сплоченных кластеров вокруг наиболее перспективных узлов в графе. Начиная с одного из двух узлов, которые участвуют в самом сильном негруппированном ребре, формируется одноэлементная группа. Затем эта группа обрабатывается путем рассмотрения каждого оставшегося негруппированного узла по очереди и вычисления «выгоды слияния» от добавления этого кандидата в группу. Узел с наибольшим значением выбирается для добавления, и этот процесс продолжается до тех пор, пока преимущество от слияния не станет меньше или равно нулю, после чего группа считается завершенной и формируется другая группа, начиная со следующего негруппированного узла с самым сильным ребром. В то время как асимптотическая сложность этого процесса квадратична по количеству узлов в графе, это значение вряд ли станет большим и может быть сделано сколь угодно малым с помощью фильтрации.

Для создания высококачественных групп метрика выгоды от слияния тщательно разработана таким образом, что, если узел-кандидат недостаточно хорошо связан с другими узлами в группе или ему лучше находиться в отдельной группе, операция слияния не будет выполняться. С этой целью функция выгоды от слияния *merge_benefit* количественно оценивает качество данной группы с помощью функции оценки *score*.

```
# Вычисление выгоды от слияния
def merge_benefit(group_a, group_b, graph, merge_tolerance):
    # Извлечение подграфов для групп
    group_a_alone = nx.Graph(graph.subgraph(group_a))
    group_b_alone = nx.Graph(graph.subgraph(group_b))
```

```
    # Получение максимальной взвешенной плотности среди групп
    separated = max(score(group_a_alone), score(group_b_alone))
    # Получение взвешенной плотности графа после объединения
    together = score(nx.Graph(graph.subgraph(group_a+group_b)))
    # Расчёт улучшения после слияния с учётом параметра
    # допуска
    return together - (separated * (1.0 - merge_tolerance))

# Вычисление взвешенной плотности графа
def score(graph):
    # Получение числа вершин
    num_nodes = float(graph.number_of_nodes())
    # Получение суммы весов инцидентных ребер для каждой
    # вершины и количества петель
    node_degrees, self_edges = degree(graph)
    # Максимальное возможное число ребер для данного графа
    max_edges = self_edges + ((num_nodes * (num_nodes - 1)) / 2)
    # Расчет плотности
    return (float(sum(node_degrees.values())) / max_edges) if
                                              max_edges else 0
```

Это вариант взвешенной плотности графа. Стандартная формулировка взвешенной плотности не учитывает ребра-петли, поэтому они были добавлены в неё, и теперь полученная метрика оценки распределяет вес между петлями только тогда, когда они присутствуют в графе. В сочетании с пороговым значением границ ребер, которое применяется для уменьшения шума, получена эффективная целевая функция, с помощью которой принимаются решения о группировке.

С учетом этой оценки, выгода от слияния рассчитывается, так, чтобы дать положительное значение только в том случае, если слияние выгодно обеим сторонам. Для того, чтобы узел-кандидат считался выгодным, граф, сформированный путем его добавления к существующей группе, должен давать более высокую оценку, чем группа, либо узел-кандидат в отдельности. Единственным исключением из этого правила является случай, когда оценка объединенного графа лишь незначительно ниже, чем у разделенных графов, и в этом случае допускается объединение для стимулирования формирования группы. Без этого условия поведение слияния было бы слишком строгим, и

большинство групп состояло бы только из одного или двух узлов вокруг самых
сильных ребер. Этот провал в расчете выгоды от слияния можно контролировать с помощью параметра допуска *merge_tolerance*, который, хорошо работает на уровне около 5%.

На рисунке 3.6 показаны группы, сформированные путем применения этого алгоритма к тестовой программе povray из *SPEC CPU* 2017. В этом случае каждый узел соответствует одному контексту распределения, окрашенному в соответствии с его группой, а толщина ребер между узлами обозначает вес. Узлы, отмеченные серым цветом, не сгруппированы из-за недостаточной выгоды от слияния. Несмотря на сложность шаблонов доступа в этом случае, алгоритм создает группы с высокой ценностью — группирование, например, сильно связанных распределений из *Copy_CSG* и *Copy_Plane*, что, как было замечено в разделе 3.1, является полезным.

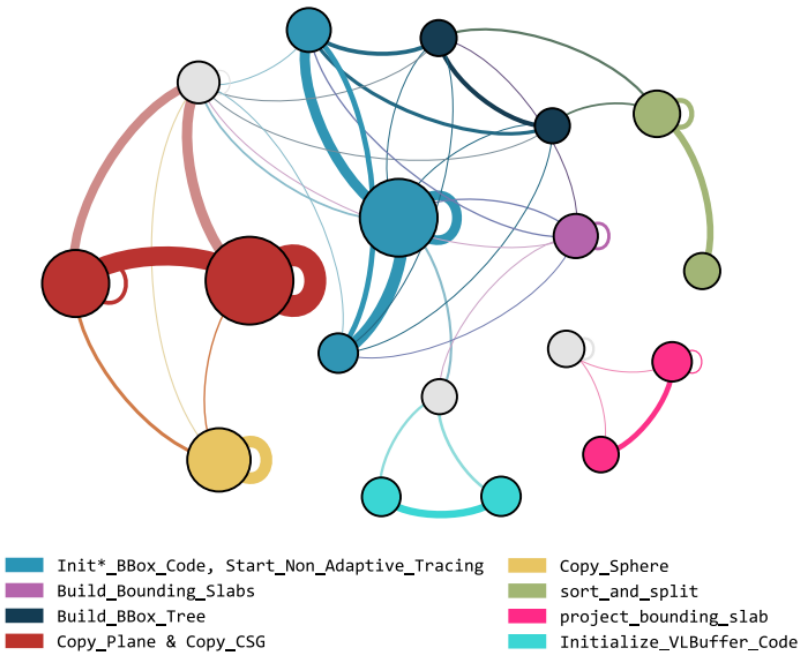

Рисунок 3.6 - Группы распределений, созданные путем анализа тестовой программы *povray*

3.2.3 Идентификация

Установив ряд групп, вокруг которых могут приниматься решения о размещении данных, теперь разрабатывается схема, с помощью которой эти группы могут быть идентифицированы во время выполнения. В то время как большая часть существующей работы в этой области для этой цели опирается на динамический стек вызовов, в данной реализации используется другой подход, использующий двоичную перезапись для определения принадлежности к группе на основе поведения потока управления вокруг нескольких важных точек в целевой программе.

Более конкретно, отношение распределения к группе устанавливается на основе селекторов: логических выражений, которые определяют, принадлежит ли конкретное распределение определенной группе или нет, на основе того, что прошел ли поток управления через определенный набор точек вызовов. Чтобы сгенерировать их, применяется простой жадный алгоритм, код которого приведен ниже.

```
# Процесс образования селекторов
def identify(groups, contexts):
    ignore = []
    selectors = {}
    # Проход по группам, отсортированным в порядке возрастания
    # их веса
    for group in sorted(groups, key=lambda (g, w): -w):
        selectors[group.id] = []
        # Добавление группы в список игнорирующихся – селекторы
        # строятся относительно групп ещё не имеющих селекторов
        # (кроме себя) и контекстов вне групп
        ignore.append(group.id)
        # Построение селектора для каждого члена группы
        for context in group.contexts:
            # Поиск цепочки вызовов, которая уникально
            # идентифицирует этот контекст
            selector = Set()
            conflicts = np.inf
            while conflicts and len(selector) <
                                    max_selector_length:
                # Поиск совпадений точек вызовов данного
```

```
                # контекста среди других, которые содержат точки
                # текущего селектора (или всех, если селектор
                # пуст)
                matches = count_matches(selector, contexts)
                # Подсчёт количества обнаруженных совпадений для
                # каждой точки вызова
                ext_matches = [(loc, sum(conflicts
                        for group_id,conflicts
                            in matches.get(loc, {}).items()
                                if group_id not in ignore))
                    for loc in context.chain if loc[0] != 0]
                # Точка вызова с минимальным числом совпадений
                minimum, count = min(ext_matches,
                                     key=lambda x: x[1])
                # Добавить новое значение только в том случае,
                # если оно уменьшает количество совпадений
                if count == conflicts:
                    break

                # Изменение числа совпадений и добавление точки
                # вызова в селектор
                conflicts = count
                selector.add(minimum)
            # Добавление селектора контекста в общий селектор
            selectors[group.id].append(list(selector))
            selectors = sorted(selectors .items())
    # Назначение уникальных значений точкам вызовов в общем
    # селекторе для их дальнейшего использования в побитовых
    # операциях с вектором «группового состояния» при
    # идентификации во время выполнения
    loc_ids = {}
    next_site_id = 0
    for group_id, group in selectors:
        for selector in group:
            for location in selector:
                if location not in loc_ids:
                    loc_ids[location] = next_site_id
                    next_site_id += 1
     return selectors, loc_ids
```

По своей сути, чтобы отличать каждого из членов группы от несвязанных контекстов, селекторы создаются в дизъюнктивной нормальной форме путем объединения конъюнктивных выражений. Несмотря на простоту и неоптимальность данного подхода, которая может возникнуть в результате рассмотрения конъюнктивных выражений для каждого члена

группы независимо от его результатов более чем достаточно для текущего прототипа.

Также после формирования селектора на его основе в результате метапрограммирования создаётся файл с кодом для определения принадлежности выделения к какой-либо группе. Пример данного файла приводится ниже. В данном случае идентифицируются выделения для двух групп, первая из которых содержит 2 контекста, а вторая – 1.

```
// Проверка бита в векторе
#define BIT_SET(val, bit) ((val) & (1ULL << (bit)))
// Проверка битов вектора на принадлежность к группам
// в соответствии с их селекторами
#define IN_GROUP_1(state) ((BIT_SET(state, 0) /* 0x40A055 */) ||
                           (BIT_SET(state, 1) /* 0x40A035 */))
#define IN_GROUP_2(state) ((BIT_SET(state, 2) /* 0x40A06D */))

// Текущее состояние вектора групп
static uint64_t *group_state;

// Функция получения идентификатора группы
static int get_group_id(size_t size)
{
    int id = -1;
    // Проверка размера выделения на превышение предела
    if (size > MAX_SIZE) return id;
    // Получение вектора состояния групп
    else if (unlikely(group_state == NULL))
        group_state = get_group_state();

    // Проверка на членство в группах
    uint64_t state = *group_state;
    if (IN_GROUP_1(state))
        id = 0;
    else if (IN_GROUP_2(state))
        id = 1;
    # Выделение не из группы вернёт значение -1

    return id;
}
```

Чтобы зафиксировать поведение потока управления вокруг точек вызовов и воздействовать на него, целевой двоичный файл переписывается с ис-

пользованием платформы оптимизации *BOLT post-link*. Специально созданный пользовательский проход для оптимизации компоновки кучи, вокруг каждой точки, входящей в селекторы (точки вызовов идентифицирующи е группы) вставляет инструкции, которые устанавливают, а затем сбрасывают один бит в общем битовом векторе «группового состояния», чтобы указать, прошел ли через них поток управления.

3.2.4 Распределение

И наконец, установив группы связанных контекстов выделений, а также вектор состояния, с помощью которого они могут быть идентифицированы, создается специализированный распределитель, который может использовать эту информацию для совместного размещения данных во время выполнения. Высокоуровневый дизайн этого распределителя показан на рисунке 3.7.

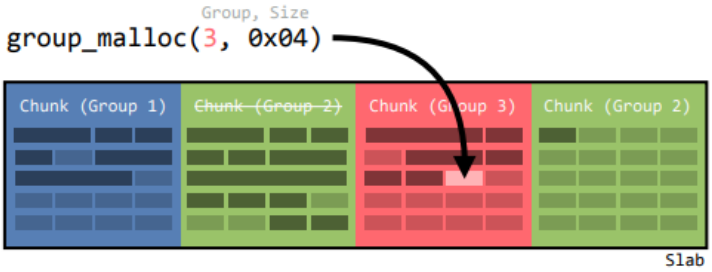

Рисунок 3.7 - Наглядное описание стратегии, используемой в специализированном распределителе групп, который сочетает в себе гарантии эффективности и непрерывности линейного распределения с моделью повторного использования на основе блоков

После того, как распределитель загружен в память, его первой задачей является определение адреса вектора состояния группы, описанного в предыдущем разделе. Когда делается запрос на выделение, это состояние используется для определения того, принадлежит ли выделение группе или нет. Для этого распределитель сравнивает размер выделения с максимальным размером сгруппированного объекта и проверяет содержимое вектора состояния группы на соответствие набору селекторов, полученных при

идентификации группы. Если выделение не принадлежит группе и, следовательно, не должно быть удовлетворено специализированным распределителем, оно перенаправляется следующему доступному распределителю через *dlsym*. Однако, если размер выделения меньше размера блока, а вектор состояния группы соответствует селектору группы, выделение выполняется с помощью функции *group_malloc*.

Память резервируется в операционной системе в виде больших слябов, разбитых на страницы для амортизации затрат на *mmap,* и управляется в виде небольших блоков, специфичных для группы, из которых могут быть выделены участки. Всякий раз, когда выполняется группированное выделение, распределитель сначала пытается зарезервировать участок запрошенного размера и выравнить из "текущего" блока, связанного с группой. Это происходит за счет простого линейного выделения без заголовков для каждого объекта и, таким образом, гарантируется непрерывность между подавляющим большинством последовательных сгруппированных выделений. Если в блоке недостаточно оставшегося места или если это первое выделение из данной группы, новый блок извлекается из текущего сляба и назначается в качестве текущего блока для целевой группы. В свою очередь, если на текущем слябе недостаточно свободного места, новый сляб резервируется из операционной системы и назначается в качестве текущего. Все выделения выполняются с минимальным выравниванием в 8 байт.

Всякий раз, когда участок освобождается или перераспределяется, распределитель должен определить, был ли этот участок первоначально выделен группой, или запрос на освобождение должен быть перенаправлен распределителю по умолчанию. В случае, если освобождаемый участок был выделен группой, его резервирование освобождается функцией *group_free* из соответствующего блока посредством заголовка. Поскольку блоки всегда

выровнены по их размеру в памяти, заголовок любого конкретного блока может быть тривиально вычислен из адреса участка с помощью простых побитовых операций. В случае операции освобождения, поле заголовка *live_objects*, которое увеличивается после каждого выделения из данного блока, уменьшается. Если после этого его значение равно нулю, блок пуст и, следовательно, может быть использован повторно.

Поскольку участки в блоках выделяются линейно, поведение фрагментации в распределителе зависит исключительно от размера блока и времени жизни последовательно сгруппированных объектов. В худшем случае, при котором блок будет переведен в состояние, в котором он пуст, за исключением одного участка, почти весь блок останется неиспользуемым в качестве внутренней фрагментации. Это может оказаться проблематичным в некоторых случаях использования, однако не является фундаментальным ограничением. Есть много областей, в которых данный прототип мог бы быть более проработанным, и поведение фрагментации - лишь одна из них. В равной степени можно было бы представить расширение описанной здесь модели для поддержки многопоточных распределений или для использования более сложных методов очистки грязных страниц [8].

3.3 Эксперименты

В данном разделе проводится 2 эксперимента, которые наглядно демонстрируют результат работы алгоритмов оптимизации. Оба примера моделируют ситуацию схожую с той, что описана в разделе 3.1 – последовательно выделяются элементы для трех списков, после чего совершаются попарные доступы к элементам первых двух списков и отдельно перебираются элементы третьего. Отличие между экспериментами заключается в способе добавления нового элемента в список.

Эксперимент 1 — добавление элементов в начало списка

```
#include <stdio.h>
#include <stdlib.h>

// Структура — элемент списка (значение, указатель на следующий
// элемент)
typedef struct Node {
    int data;
    struct Node *next;
} Node;

// Функция добавления нового элемента в начало списка
void push(Node **head, int data) {
    Node *tmp = (Node*)malloc(sizeof(Node));
    tmp->data = data;
    tmp->next = (*head);
    (*head) = tmp;
}

// Функция-оболочка для визуальной идентификации контекстов в
файле групп
void add_A(Node **head, int data)
{
    push(head,data);
}
void add_B(Node **head, int data)
{
    push(head,data);
}
void add_C(Node **head, int data)
{
    push(head,data);
}

int my_main() {
    // Размер списка
    int list_size = 1000;

    Node *A = NULL, *B = NULL, *C = NULL;
    // Последовательное добавление элементов в списки
    for(int i = 0; i < list_size; i++)
    {
        add_A(&A,i);
        add_B(&B,i+1);
        add_C(&C,i+2);
    }

    int sum[list_size];
```

```
    Node *A_copy = A, *B_copy = B;
    // Попарный доступ к элементам списков A и B
    for(int i = 0; i < list_size; i++)
    {
        sum[i] = A_copy->data + B_copy->data;
        A_copy = A_copy->next;
        B_copy = B_copy->next;
    }

    int res = 0;
    // Последовательные доступы к элементам списка C
    for(int i = 0; i < list_size; i++)
    {
        res += C->data;
        C = C->next;
    }

    return 0;
}

int main() {
    return my_main();
}
```

В результате работы программы, приведённой выше, создаётся граф связей (рисунок 3.8) и на его основе формируются вполне очевидные группы (рисунок 3.9), поскольку контекст C (id=1) недостаточно связан с остальными двумя, но его элементы связаны между собой.

```
0 2999
2 2997
1 2000
#
0 0 3002
1 0 2014
1 1 1998
2 0 7003
2 1 2013
2 2 3000
```

Рисунок 3.8 - Граф, сформированные в эксперименте 1

```
GRP 1 20008:
        CTX 0:
                __libc_malloc from 0x401084
                .plt.sec from 0x401418
                push from 0x40181F
                add_B from 0x401C75
                main from 0x0
        CTX 2:
                __libc_malloc from 0x401084
                .plt.sec from 0x401418
                push from 0x40161F
                add_A from 0x401C5E
                main from 0x0
GRP 2 1998:
        CTX 1:
                __libc_malloc from 0x401084
                .plt.sec from 0x401418
                push from 0x401A1F
                add_C from 0x401C8C
                main from 0x0
```

Рисунок 3.9 - Группы, сформированные в эксперименте 1

Пользовательский распределитель размещает элементы контекстов A и B вместе, а элементы контекста C – в отдельный блок как показано на рисунке 3.10.

```
Allocating chunk for group 0...
Allocating slab...
[Group 0] Allocated 16 bytes: 0x7f8210300040
[Group 0] Allocated 16 bytes: 0x7f8210300050
Allocating chunk for group 1...
[Group 1] Allocated 16 bytes: 0x7f8210400040
[Group 0] Allocated 16 bytes: 0x7f8210300060
[Group 0] Allocated 16 bytes: 0x7f8210300070
[Group 1] Allocated 16 bytes: 0x7f8210400050
[Group 0] Allocated 16 bytes: 0x7f8210300080
[Group 0] Allocated 16 bytes: 0x7f8210300090
[Group 1] Allocated 16 bytes: 0x7f8210400060
...
```

Рисунок 3.10 - Фрагмент, демонстрирующий работу распределителя в эксперименте 1

Эксперимент проведён для разных значений размера списка (таблица 3.2). И установлено, что при увеличении размера списка, уменьшается количество кэш-промахов и время работы программы. Однако, также стоит отметить, что улучшения наступают только с определенного момента, поскольку накладные расходы на размещение поначалу превышают выигрыш от совместного распределения.

Таблица 3.2 - Результат работы программы в эксперименте 1 при различных значениях размера списка

Размер списка	Уменьшение кэш-промахов, %	Ускорение программы
1000	2	1,02
10000	17	1,094
100000	36	1,29

Эксперимент 2 - добавление элементов в конец списка

```
#include <stdio.h>
#include <stdlib.h>

// Структура — элемент списка (значение, указатель на следующий
// элемент)
typedef struct Node {
    int data;
    struct Node *next;
} Node;

// Функция поиска последнего элемента в списке
Node* getLast(Node *head) {
    if (head == NULL) {
        return NULL;
    }
    while (head->next) {
        head = head->next;
    }
    return head;
}

// Функция добавления нового элемента в конец списка
void pushBack(Node **head, int data) {
```

```
    Node *last = getLast(*head);
    Node *tmp = (Node*)malloc(sizeof(Node));
    tmp->data = data;
    tmp->next = NULL;
    if (last == NULL) {
        *head = tmp;
    }
    else {
        last->next = tmp;
    }
}

// Функция-оболочка для визуальной идентификации контекстов в
// файле групп
void add_A(Node **head, int data)
{
    pushBack(head,data);
}
void add_B(Node **head, int data)
{
    pushBack(head,data);
}
void add_C(Node **head, int data)
{
    pushBack(head,data);
}

int my_main() {
    // Размер списка
    int list_size = 1000;

    Node *A = NULL, *B = NULL, *C = NULL;
    // Последовательное добавление элементов в списки
    for(int i = 0; i < list_size; i++)
    {
        add_A(&A,i);
        add_B(&B,i+1);
        add_C(&C,i+2);
    }

    int sum[list_size];
    // Попарный доступ к элементам списков A и B
    for(int i = 0; i < list_size; i++)
    {
        sum[i] = A->data + B->data;
        A = A->next;
        B = B->next;
    }

    int res = 0;
```

```
    // Последовательные доступы к элементам списка C
    for(int i = 0; i < list_size; i++)
    {
        res += C->data;
        C = C->next;
    }

    return 0;
}

int main() {
    return my_main();
}
```

Аналогично эксперименту 2 создан граф связей (рисунок 3.11) и группы распределений (рисунок 3.12), но в данном случае сформированных групп 3 и каждый контекст является единственным членом. Это объясняется способом добавления новых элементов в список, а именно – добавлением в конец, ведь при этом совершается полный проход списка. Поэтому каждый контекст в графе имеет огромный вес у ребра-петли, что и способствует формированию таких групп.

```
0 503498
1 502499
2 502496
#
0 0 503505
1 0 371
1 1 502505
2 0 5349
2 1 363
2 2 502499
```

Рисунок 3.11 - Граф, сформированный в эксперименте 2

```
GRP 1 503505:
        CTX 0:
                __libc_malloc from 0x401084
                .plt.sec from 0x40162B
                pushBack from 0x401A1F
                add_B from 0x401E6C
                my_main from 0x40220D
                main from 0x0
GRP 2 502505:
        CTX 1:
                __libc_malloc from 0x401084
                .plt.sec from 0x40162B
                pushBack from 0x401C1F
                add_C from 0x401E80
                my_main from 0x40220D
                main from 0x0
GRP 3 502499:
        CTX 2:
                __libc_malloc from 0x401084
                .plt.sec from 0x40162B
                pushBack from 0x40181F
                add_A from 0x401E58
                my_main from 0x40220D
                main from 0x0
```

Рисунок 3.12 - Группы, сформированные в эксперименте 2

Как видно на рисунке 3.13, элементы списков располагаются последовательно каждый в соответствующем блоке.

```
Allocating chunk for group 2...
Allocating slab...
[Group 2] Allocated 16 bytes: 0x7f218ab00040
Allocating chunk for group 0...
[Group 0] Allocated 16 bytes: 0x7f218ac00040
Allocating chunk for group 1...
[Group 1] Allocated 16 bytes: 0x7f218ad00040
[Group 2] Allocated 16 bytes: 0x7f218ab00050
[Group 0] Allocated 16 bytes: 0x7f218ac00050
[Group 1] Allocated 16 bytes: 0x7f218ad00050
[Group 2] Allocated 16 bytes: 0x7f218ab00060
[Group 0] Allocated 16 bytes: 0x7f218ac00060
[Group 1] Allocated 16 bytes: 0x7f218ad00060
[Group 2] Allocated 16 bytes: 0x7f218ab00070
[Group 0] Allocated 16 bytes: 0x7f218ac00070
[Group 1] Allocated 16 bytes: 0x7f218ad00070
...
```

Рисунок 3.13 - Фрагмент, демонстрирующий работу распределителя в эксперименте 2

Данному эксперименту соответствуют такие же выводы, как и первому, однако ключевой можно сделать из таблицы 3.3 и рисунка 3.11, и заключается он в том, что чем сильнее связаны члены групп (больше одновременных доступов), тем больший прирост производительности даст такая политика размещения программам.

Таблица 3.3 - Результат работы программы 2 при различных значениях размера списка

Размер списка	Уменьшение кэш-промахов, %	Ускорение программы
100	-14	0,9
1000	78	1,28
10000	67	1,49

В ходе проведения экспериментов можно сделать следующие выводы:

1) Реализация алгоритмов на основе предварительного профилирования двоичных файлов программ приводит к увеличению производительности программ с сильносвязанными объектами данных;

2) У данного алгоритма имеется большой запас прочности, что позволяет работать над его дальнейшим совершенствованием;

3) Алгоритм работает на двоичном уровне без какой-либо необходимости в высокоуровневом исходном коде.

4 Экономическое обоснование ВКР

В данном разделе выполнен расчет затрат на выполнение ВКР, который включает расходы на оплату труда исполнителей, отчисления на социальные нужды, расходы на материалы, оборудование необходимые для выполнения работы и расходы на услуги сторонних организаций.

4.1 Концепция экономического обоснования

Выпускная квалификационная работа «Микроархитектурная оптимизация обращений к динамически размещаемым объектам в памяти» посвящена изучению алгоритмов оптимизации в области динамического распределения памяти. Для некоторых отраслей экономики производительность вычислительных систем имеет огромное значение. Поэтому важно понимать, какой объем расходов, может быть связан с поиском необходимых инструментов, позволяющих значительно увеличить производительность некоторых программ.

4.2 Определение трудоемкости выполняемой работы

На основе трудоемкости выполнения ВКР рассчитываются издержки на оплату труда ее исполнителей, являющиеся одной из основных статей калькуляции себестоимости ВКР.

В процессе разработки организационного плана работ определяется перечень мероприятий, необходимых для выполнения всего объема работ, которые приведены в таблице 4.1.

Себестоимость работ будет определяться по фактическим затратам. Расчет себестоимости осуществляется по следующим статьям:

а) основная и дополнительная заработная плата исполнителей;

б) отчисления на социальные нужды;

в) материалы и оборудование, необходимые для выполнения работы;

г) издержки на амортизацию оборудования;

д) накладные расходы.

Таблица 4.1 – Исполнители и трудоемкость работ

Наименование работ	Трудоемкость (T_i), чел./дни	
	Руководитель (T_1)	Исполнитель (T_2)
Разработка технического задания	1	1
Изучение литературы	1	15
Стратегии динамического распределения памяти	1	21
Проблема распределителей общего назначения	1	9
Подход профилирования двоичных файлов	1	22
Сдача ВКР	1	1
Итого	6	69

4.3 Затраты на основную заработную плату непосредственных исполнителей ВКР и отчисления на социальные нужды

Для расчета заработной платы непосредственных исполнителей используются данные трудоемкости работ (таблица 4.1) и данных, приведенных в таблице 4.2:

Таблица 4.2 – Заработная плата непосредственных исполнителей ВКР, отчисления на социальные нужды

Основная заработная плата непосредственных исполнителей разработки, отчисления на социальные нужды, накладные расходы	Обозначение	Рубли, %
Дневная ставка руководителя	C_1	4048 руб.
Дневная ставка исполнителя	C_2	2381 руб.
Процент дополнительной заработной платы	$Н_{доп}$	14%
Процент отчислений на социальные нужды	$З_{соц}$	30,2 %

Дневная ставка руководителя рассчитана исходя из заработной платы за месяц равной 85000 руб. Дневная ставка исполнителя рассчитана исходя из медианной заработной платы инженера-программиста за месяц, которая по данным официального сайта *gorodrabot.ru* в апреле 2022 года в Санкт-Петербурге составила 50000 руб. (Приложение А). При расчете использовано количество рабочих дней в месяце равное 21.

Основную заработную плату исполнителей рассчитываем по формуле 4.1:

$$З_{осн.з/пл} = \sum_{i=1}^{k} T_i \cdot C_i , \quad (4.1)$$

где, З $_{осн.з/пл}$ – расходы на основную заработную плату исполнителей, руб.; k – количество исполнителей; T_i – время, затраченное i-м исполнителем на проведение исследования, дни; C_i – ставка i-го исполнителя, руб./день.

$$З\,осн.з/пл = (k \cdot T1 \cdot C_1) + (k \cdot T2 \cdot C_2)$$

$$З_{осн.з/пл} = 1 \cdot 6 \cdot 4048 + 1 \cdot 2381 \cdot 69 = 24288 + 164289 = 188577 \text{ (руб.)}$$

Дополнительную заработную плату рассчитываем по формуле 4.2:

$$З_{доп.з/пл} = З_{осн.з/пл} \cdot Н_{доп} : 100, \quad (4.2)$$

где, $З_{доп.з/пл}$ - расходы на дополнительную заработную плату исполнителей, руб.; $З_{осн.з/пл}$ - расходы на основную заработную плату исполнителей, руб.; $Н_{доп}$ – норматив дополнительной заработной платы, %. При выполнении расчетов в ВКР норматив дополнительной заработной платы принимаем равным 14 %.

$$З_{доп.з/пл} = 188577 \cdot 14 : 100 = 26400{,}78 \text{ (руб.)}$$

К статье «Отчисления на социальные нужды» относятся:

- отчисления на социальное страхование;
- отчисления на пенсионное обеспечение;
- отчисления на медицинское страхование;
- страхование от несчастных случаев на производстве и профессиональных заболеваний.

Отчисления на социальные нужды рассчитываем по формуле 4.3:

$$З_{соц} = (З_{осн.з/пл} + З_{доп.\ з/пл}) \times Н_{соц.} : 100, \quad (4.3)$$

где, $З_{соц}$ – отчисления на социальные нужды с заработной платы, руб.;
$З_{осн.з/пл}$ – расходы на основную заработную плату исполнителей, руб.;
$З_{доп.з/пл}$ – расходы на дополнительную заработную плату исполнителей, руб.;
$Н_{соц}$ – норматив отчислений страховых взносов на обязательное социальное, пенсионное, медицинское страхование, и страхование от несчастных случаев на производстве и профессиональных заболеваний, %.

$$З_{соц} = (188577 + 26400{,}78) \cdot 30{,}2 :100 = 64923{,}29 \text{ (руб.)}$$

4.4 Затраты на материалы, необходимые для выполнения работы

На статью «Материалы» относятся расходы на основные и вспомогательные материалы и комплектующие изделия, которые могут понадобиться при выполнении работ (таблица 4.3).

Таблица 4.3 – Калькуляция расходов по статье «Материалы»

Материалы	Кол-во	Цена, руб.	Сумма, руб.
Бумага для печатного экземпляра ВКР и рабочих записей, пачка 500 л. А4	1	250	250
Накопитель информации (флеш-накопитель)	1	700	700
Папка скоросшиватель пластиковая А4	2	30	60
ИТОГО:			**1040**

4.5 Затраты по статье «Оборудование»

Для изучения принципов действия существующих альтернативных распределителей динамической памяти, проведения экспериментов необходим персональный компьютер, имеющий следующие характеристики:

- процессор *Intel(R) Core(TM) i3*-7020U @ 2.30 ГГц;
- кэш-память: L1d - 64 КБ, L2 - 512 КБ, L3 - 3 МБ;
- оперативная память: 16 ГБ;
- дисковое пространство: 960 ГБ.

Калькуляция затрат по статье «Оборудование» приведена в таблица 4.4

Таблица 4.4 – Калькуляция расходов по статье «Оборудование»

Оборудование	Кол-во	Цена, руб. (без НДС)	Сумма, руб.
ПК на базе Intel Core i3	1	57200	57200
Транспортные расходы (10%)			5720
ВСЕГО:			**62920**

Поскольку программное обеспечение, применяемое в данной работе, находится в свободном доступе в интернете, расходы на его приобретение не будут включаться в затраты на выполнение ВКР.

В соответствии с пунктом 1 статьи 256 НК РФ основные средства менее 100000 рублей в налоговом учете не являются амортизируемым имуществом, поэтому в расчет себестоимости амортизационные отчисления не включаются.

4.6 Накладные расходы

К статье «Накладные расходы» относятся расходы на управление и хозяйственное обслуживание. Поскольку ВКР выполняется на кафедре универ-

ситета, расчет накладных расходов имеет некоторые особенности. СПбГЭТУ «ЛЭТИ» является организацией бюджетной сферы, поэтому состав накладных расходов и порядок их распределения утверждается учреждением, как правило, самостоятельно. В качестве базы для распределения накладных расходов могут быть:

1) прямые затраты по оплате труда;
2) материальные затраты;
3) иные прямые затраты;
4) объем выручки от реализации продукции (работ, услуг);
5) иной показатель, характеризующий результаты деятельности учреждения.

В данной работе к накладным расходам можно отнести услуги сторонней организации по обеспечению доступа к сети интернет (услуги связи). Средняя стоимость безлимитного мобильного интернета в период выполнения ВКР составила 542 руб. в месяц без НДС. За три месяца работы стоимость услуг связи составила 1626 руб.

4.7 Себестоимость работ

В таблице 4.5 приведена калькуляция себестоимости выпускной квалификационной работы, на основании полученных данных по отдельным статьям калькуляции.

Таблица 4.5 – Калькуляция себестоимости работы

№	Статья затрат	Сумма, руб.
1	Основная заработная плата	188577
2	Дополнительная заработная плата	26400,78
3	Отчисления на социальные нужды	64923,29
4	Материалы	1040
5	Оборудование	62920
6	Накладные расходы	1626
	Итого себестоимость	**345487,07**

4.8 Выводы

В данном разделе рассчитана себестоимость выполнения ВКР на тему: «Микроархитектурная оптимизация обращений к динамически размещаемым объектам в памяти». При анализе основных затрат становится очевидным, что максимальные расходы приходится на основную заработную плату и оборудование.

Исходя из расчета, себестоимость выполнения работы составила 345487,07 руб. Учитывая, что средняя заработная плата работника в сфере науки и техники по данным Росстата за октябрь 2021 года составила 80791 руб. можно сказать, что затраты на поиск инструментов, оптимизирующих процессы управления памятью для предприятий, крайне заинтересованных в высокопроизводительной компьютерной технике могут быть оправданы.

ЗАКЛЮЧЕНИЕ

В процессе выполнения работы, были, изучены основные стратегии динамического размещения данных. Каждый из рассмотренных распределителей по-своему способен решить проблемы с фрагментацией, с нехваткой памяти, со скоростью получения и освобождения блоков необходимого размера, с временем жизни объектов и занимаемой ими памятью. Поэтому в каждой конкретной ситуации требуется подбирать самый оптимальный распределитель или создавать композицию из нескольких.

В ходе выполнения следующего этапа работы, выявлена основная проблема распределителей памяти общего назначения – плохая пространственная локализация связанных объектов данных.

На следующем этапе выполнения работы детально изучен метод оптимизации, позволяющий решить обозначенную проблему. Применение алгоритмов оптимизации размещения данных в памяти на основе предварительного профилирования двоичных файлов программ позволяет сделать вывод, что использование исследуемых алгоритмов группировки и идентификации, приводит к созданию сплоченных групп распределения, которые используют весь стек вызовов, и идентифицируют их во время выполнения.

Проведенные эксперименты показали, что, *во-первых*, реализация алгоритма на основе предварительного профилирования двоичных файлов программ приводит к увеличению производительности, *во-вторых*, у данного алгоритма имеется большой запас прочности, что позволяет работать над его дальнейшим совершенствованием; *в-третьих*, алгоритм работает на двоичном уровне без какой-либо необходимости в высокоуровневом исходном коде.

В дополнительном разделе рассчитана себестоимость выполнения ВКР. Анализ основных затрат показал, что максимальные расходы приходится на основную заработную плату и оборудование. Исходя из расчета, себестои-

мость выполнения работы составила 345487,07 руб. Учитывая, что средняя заработная плата работника в сфере науки и техники по данным Росстата за октябрь 2021 года составила 80791 руб. можно сказать, что затраты на поиск инструментов, оптимизирующих процессы управления памятью для предприятий, крайне заинтересованных в высокопроизводительной компьютерной технике могут быть оправданы.

Результаты данной работы могут быть использованы разработчиками программ при выборе метода оптимизации, необходимого в ходе реализации конкретного проекта.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. ГОСТ 7.32–2001. Система стандартов по информации, библиотечному и издательскому делу. Отчет о научно-исследовательской работе. Структура и правила оформления [Электронный ресурс].URL: http://www.opengost.ru/download/1863/GOST_7_322001_SIBID_Otchet_o_nauchno-issledovatel_skoy_rabote_Struktura_i_pravila_oformleniya.html (дата обращения: 24.05.2022).

2. Пользовательские распределители памяти в C++. [Электронный ресурс], 2020. URL: https://github.com/mtrebi/memory-allocators (дата обращения: 09.04.2022).

3. Brad Calder, Dirk Grunwald, and Benjamin Zorn. 1994. Quantifying behavioral differences between C and C++ programs = Количественная оценка поведенческих различий между программами на C и C++. [Электронный ресурс], Journal of Programming Languages 2, 4 (1994), 313ś351. URL: https://cseweb.ucsd.edu/~calder/papers/JplVersion.pdf (дата обращения: 12.05.2022).

4. Emery D. Berger, Benjamin G. Zorn, and Kathryn S. McKinley. 2001. Composing High performance Memory Allocators = составление высокопроизводительных распределителей памяти. [Электронный ресурс], In Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation (PLDI ’01). ACM, New York, NY, USA, 114ś124. URL: https://dl.acm.org/doi/10.1145/378795.378821 (дата обращения: 12.05.2022).

5. Emery D. Berger, Benjamin G. Zorn, and Kathryn S. McKinley. 2002. Reconsidering Custom Memory Allocation = Пересмотр пользовательского распределения памяти. [Электронный ресурс], In Proceedings of the 17th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and

Applications (OOPSLA ’02). ACM, New York, NY, USA, 1ś12. URL: https://dl.acm.org/doi/10.1145/582419.582421 (дата обращения: 24.04.2022).

6. GeeksforGeeks. Выделение памяти ядра (buddy system и slab system). [Электронный ресурс], 2021. URL: https://www.geeksforgeeks.org/operating-system-allocating-kernel-memory-buddy-system-slab-system/ (дата обращения 09.04.2022).

7. Guilherme Kunigami. Распределение памяти Buddy. [Электронный ресурс], 2020. URL: https://www.kuniga.me/blog/2020/07/31/buddy-memory-allocation.html (дата обращения 25.04.2022).

8. Jason Evans. 2015. Tick Tock, Malloc Needs a Clock = Тик - так, Маллоку нужны часы. [Электронный ресурс], In Applicative 2015 (Applicative 2015). ACM, New York, NY, USA. URL: https://dl.acm.org/doi/10.1145/2742580.2742807 (дата обращения: 12.05.2022).

9. Joe Savage, Timothy M. Jones. 2020. HALO: post-link heap-layout optimization = Halo: служба оптимизации компоновки кучи после линковки. [Электронный ресурс], CGO 2020: Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization.
URL: https://dl.acm.org/doi/abs/10.1145/3368826.3377914 (дата обращения: 24.04.2022).

10. Niklas. Allocation Adventures 3: The Buddy Allocator = приключения с распределителем 3: распределитель приятелей. [Электронный ресурс], 2015. URL: http://bitsquid.blogspot.com/2015/08/allocation-adventures-3-buddy-allocator.html (дата обращения 25.04.2022).

11. Robert Love. Разработка ядра Linux. Устройство слябового распределителя памяти. ВикиЧтение. [Электронный ресурс], URL: https://it.wikireading.ru/1878 (дата обращения: 25.04.2022).

12. Trishul M. Chilimbi and Ran Shaham. 2006. Cache-conscious Coallocation of Hot Data Streams = Размещение горячих потоков данных с учетом

кэша. [Электронный ресурс], In Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'06). ACM, New York, NY, USA, 252ś262. URL: https://dl.acm.org/doi/abs/10.1145/1133981.1134011 (дата обращения: 12.05.2022).

13. Vladimir Balum. Аллокаторы памяти. [Электронный ресурс], 2020. URL: https://habr.com/ru/post/505632/ (дата обращения: 09.04.2022).

14. Zhenjiang Wang, Chenggang Wu, and Pen-Chung Yew. 2010. On Improving Heap Memory Layout by Dynamic PoolAllocation. = Об улучшении ком поновки кучи памяти путем динамического выделения пула. [Электронный ресурс], In Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '10). ACM, New York, NY, USA, 92 ś100 URL: https://dl.acm.org/doi/10.1145/1772954.1772969 (дата обращения: 12.05.2022).

Приложение А

Информация с официального сайта gorodrabot.ru

Данные официального сайта gorodrabot.ru о медианной заработной плате инженера-программиста в апреле 2022 года в г. Санкт-Петербурге представлены на рисунке А.1

Рисунок А.1 - Снимок экрана официального сайта gorodrabot.ru