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: “Средства динамической инструментации программ для микроархитектурной оптимизации”

Студент _____ И. С. Смеюха

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

Консультант
к. э. н. _____ Н. И. Олехова

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)
"___" _____ 202__г.

ЗАДАНИЕ
на выпускную квалификационную работу

Студент И. С. Смеюха Группа № **8305**

1. Тема: Средства динамической инструментации программ для микроархитектурной оптимизации

(утверждена приказом № _____ от _____)

Место выполнения ВКР: Санкт-Петербургский государственный электротехнический университет «ЛЭТИ» им. В. И. Ульянова (Ленина)

2. Объект исследования: методы анализа программного обеспечения.

3. Предмет исследования: динамическая инструментация программ.

4. Цель: программа–профилировщик, использующая динамическую инструментацию.

5. Исходные данные: документация по использованию фреймворка DynamoRIO.

6. Содержание

- Динамическая инструментация.
- Исследование инструментов DynamoRIO для динамической двоичной инструментации.
- Разработка собственного профилировщика.
- Демонстрация итогов разработки.

7. Технические требования

Разрабатываемая программа должна выполнять следующие требования:

- Находить в профилируемой программе все аллокации памяти.
- Получить информацию, об адресе выделения и количестве выделенной памяти.
- Получить имя перменной и тип операции обращения к памяти.
- Информация должна выводится в отдельный трассировочный файл.

8. Дополнительный раздел: экономическое обоснование.

9. Результаты: пояснительная записка, презентация, реферат.

Дата выдачи задания	Дата представления ВКР к защите
«1» апреля 2022 г.	«22» июня 2022 г.

Руководитель к. т. н., доцент	_____	А. А. Пазников
Студент	_____	И. С. Смеюха

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)
“___” _____ 202__г.

КАЛЕНДАРНЫЙ ПЛАН
выполнения выпускной квалификационной работы

Тема Средства динамической инструментации программ для микро-архитектурной оптимизации

Студент И. С. Смеюха Группа № **8305**

№ этапа	Наименование работ	Срок выполнения
1	Обзор литературы по теме работы	01.04 – 09.04
2	Изучение документации DynamoRIO	10.04 – 18.04
3	Обучение работе с DynamoRIO	18.04 – 28.04
4	Разработка программы	29.04 – 15.05
5	Тестирование программы	16.05 – 22.05
6	Оформление пояснительной записки	23.05 – 31.05
7	Предварительное рассмотрение работы	07.06
8	Представление работы к защите	10.06

Руководитель
к. т. н., доцент _____ А. А. Пазников
Студент _____ И. С. Смеюха

РЕФЕРАТ

Пояснительная записка содержит: … страниц, … рисунков, … таблиц, … приложений, … источников литературы.

Целью выпускной квалификационной работе ставится исследования методов анализа программного обеспечения, а также написание собственного профилировщика, использующего динамическую инструментацию.

В данной работе проводится анализ предметной области, рассматриваются виды профилирования, инструменты, специализированные на проведении динамической инструментации, процесс создания своего профилировщика, а также обосновывается выбор тех или иных средств и инструментов.

Результатом работы является программа–профилировщик, проводящий поиск функций выделения памяти в пользовательских программах, представленных в виде исполняемого файла. С помощью данного профилировщика можно анализировать работу с памятью даже в тех случаях, когда нет доступа к исходному коду программы, что делает его полезным инструментом для контроля потребляемой памяти.

ABSTRACT

The purpose of the final qualifying work is to study the methods of software analysis, as well as writing a profiler using dynamic instrumentation.

The result of the work is a profiler program that searches for memory allocation functions in user programs presented as an executable file. This profiler provides the ability to analyze memory usage even in cases where there is no access to the source code of the program, which makes it a useful tool for monitoring memory consumption.

СОДЕРЖАНИЕ

ОПРЕДЕЛЕНИЯ, ОБОЗНАЧЕНИЯ И СОКРАЩЕНИЯ

Фреймворк – программное обеспечение, облегчающее разработку и объединение разных компонентов большого программного проекта

Аллокатор – распределитель памяти, занимающийся выделением и отчисткой памяти в процессе работы программы.

Базовый блок – последовательность команд, имеющая одну точку входа и заканчивающаяся одно или несколькими точками выхода.

Функция обратного вызова – функция, которая начинает свое выполнение после того, как другая функция закончит свое выполнение. Обычно функция обратного вызова передается в качестве параметра другого кода.

DBI - Dynamic Binary Instrumentation – динамическая двоичная инструментация. Вид инструментации, проводящийся непосредственно во время выполнения программы.

IR - Intermediate Representation – промежуточное представление. Это код, который образуется в процессе перевода исходного кода, написанным на одном из высокоуровневых языков программирования, в машинный язык.

API - Application Programming Interface – программный интерфейс приложения. Это набор способов и правил, по которым одна программа взаимодействует с другой.

AST - Abstract Syntax Tree – абстрактное синтаксическое дерево. Это древовидное представление программы, где каждый узел представляет конструкцию из исходного кода программы.

ВВЕДЕНИЕ

Сложность программного обеспечения постоянно растёт, а значит растут и расходы, которые компании тратят на его создание, поэтому вопрос уменьшения затрат встает особенно остро. Профилирование – процесс сбора информации, характеризующей работу программы. Это могут быть, например, объём занимаемой памяти, время выполнения отдельных участков кода, количество вызовов функций. Профилирование призвано упростить процесс разработки, поддержки и оптимизации программного обеспечения. С его помощью можно значительно уменьшить время на отлавливание ошибок, утечек памяти и устранение других трудностей, возникающих в процессе разработки.

В этой работе подробно рассмотрен один из видов профилирования, а именно динамическая двоичная инструментация. Также была написана программа-профилировщик, использующая фреймворк DynamoRIO.

Для достижения поставленной цели были выделены следующие задачи:

- анализ предметной области;
- анализ существующих аналогов;
- описание инструментов, с которыми предстоит работать;
- создание программы–профилировщика на основе выбранного фреймворка;
- тестирование полученной программы.

Целью данной работы является разработка программы–профилировщика, использующую динамическую инструментацию.

Объектом исследования выпускной квалификационной работы является изучение методов анализа программного обеспечения. Предмет исследования – динамическая инструментация программ.

В первом разделе работы рассматриваются конкретные методы анализа программного обеспечения, а также описываются фреймворки, подходящие для дальнейшей работы.

Во втором разделе подробно описывается используемые инструменты.

В третьем разделе представлено описание процесса разработки программы-профилировщика.

В четвертом разделе представлен дополнительный раздел по теме «Экономическое обоснование выпускной квалификационной работы».

1 Динамическая инструментация

Данный раздел посвящен обзору существующих методов анализа программного обеспечения, а также фреймворков, предоставляющих пользователю функционал для создания программы использующую динамическую двоичную инструментацию.

1.1 Виды анализа программного обеспечения

Анализ программного обеспечения [1] делится на две группы, в зависимости от того, когда он производится:

- Статический анализ, проводящийся без запуска программы [1, 2]. Этот метод применяют как независимые инструменты, так и, например, компиляторы, анализирующие код на ошибки, или на возможность оптимизации. К этому методу анализа относится в том числе и ручной просмотр кода, то есть тот, который проводит сам разработчик, построчно проверяя код на наличие ошибок.
- Динамический анализ, проводящийся в программе по мере её выполнения [1, 2, 3, 4]. Анализирующий код может встраиваться в код программы и исполняться, как часть обычного выполнения программы, осуществляя сбор информации о работе программы. Такой код никак не влияет на результат работы, кроме, вероятно, увеличения времени выполнения.

Эти два подхода дополняют, а не исключают друг друга. Однако динамический анализ работает с реальными значениями, а следовательно, более точный и простой, чем статический.

Также стоит отметить, что анализ может проводится по-разному в зависимости от того, в каком виде представлен анализируемый код. Это может быть исходный код программы или двоичный. Анализ исходного кода происходит, на уровне исходного кода программы, написанного на одном из высокоуровневых языках программирования. Анализ двоичного кода происходит

на уровне машинного кода, хранящегося либо в виде объектного, либо в виде исполняемого кода.

Рассмотрим основные подходы к анализу, а именно сэмплирование (статический анализ) и инструментация (динамический анализ).

1.1.1 Сэмплирование

Суть сэмплирования [2] заключается в том, что работа программы время от времени приостанавливается, после чего производятся замеры некоторых характеристик исполняемой программы. Безусловным преимуществом такого метода является то, что нет необходимости модифицировать код профилируемой программы. Накладные расходы при таком подходе минимальны. В то же время серьезной проблемой является то, что остановка программы и последующие замеры могут происходить в неподходящее время, что ведёт к потере части важной информации и сильно уменьшает точность проводимого анализа.

1.1.2 Инструментация

Инструментация – модификация исполняемого или исходного кода программы с целью её дальнейшего анализа [1, 2]. Существует два методы инструментации:

- Статическая инструментация – метод, при котором код программы изменяется до её запуска. Удобство в том, что инструментация происходит единожды, после чего полученный исполняемый файл можно выполнять неограниченное количество раз. Но есть и недостаток, при изменении исходной программы, весь процесс инструментации приходится производить заново. Накладные расходы при таком подходе включают в себя только затраты на выполнение добавленных процедур [1, 3]. Программы для статической инструментации: Dyninst, EEL, ATOM.
- Динамическая инструментация – метод, при котором в ходе выполнения программы управление перехватывается, после чего происходит замена

блоков кода на модифицированные. При таком подходе помимо перехвата управления необходимо выполнить дизассемблирование, модификацию и замену кода. Все это ведет к сильному повышению накладных расходов [1, 3, 4]. Программы для динамической инструментации: DynamoRIO, PIN, Dyninst.

Серьёзная проблема статической инструментации заключается в том, что исходный код программы имеется не всегда. В большинстве случаев есть доступ только к исполняемому файлу, а значит от количества информации, которую можно получить из него будет зависеть качество проводимой инструментации. Это ведет к тому, что с ростом сложности программ, все сложнее становится получить полное покрытие кода с помощью статической инструментации. Динамическая же инструментация такой проблемы лишена, что выделяет её на фоне статического подхода.

Эта работа посвящена динамической инструментации, поэтому дальше рассмотрим ее особенно подробно.

1.2 Динамическая двоичная инструментация

Как уже говорилось выше, динамическая инструментация – способ профилирования, при котором в ходе выполнения программы управление перехватывается, после чего происходит замена блоков кода на модифицированные. Вставленный таким образом анализирующий код исполняется, как часть обычного выполнения программы, но дополнительно выводит информацию о ходе работы [1, 4].

Динамическая инструментация имеет два явных преимущества перед статической. Во-первых, при таком методе анализа пользователю не надо как-либо подготавливать анализируемый код. Во-вторых, статическая инструментация, может быть затруднена, если код и данные смешаны, или вообще невозможна, если используется динамически генерируемый код. В то же время процесс динамической инструментации естественным образом охватывает

весь код. Но стоит выделить два основных недостатка, а именно большие накладные расходы, и то, что динамическую двоичную инструментацию сложно реализовать.

Проблемы, связанные с использованием динамической двоичной инструментации смягчаются тем, что существуют специальные фреймворки, предоставляющие необходимый инструментарий, а значит пользователю нет необходимости создавать их с нуля. Код анализа сконцентрирован в фреймворке и может быть использован снова в будущих задачах анализа. Также стоит упомянуть, что фреймворки постоянно усовершенствуются, и накладные расходы становятся все меньше и меньше.

1.3 Обзор фреймворков, использующихся для динамической двоичной инструментации

В данный момент существует множество фреймворков, для проведения динамической двоичной инструментации. Но мы остановимся на самых популярных:

- PIN;
- DynInst;
- DynamoRIO;
- Valgrind;
- BAP;
- KEDR;
- ERESI.

Для рассмотрения были выбраны три самых популярных фреймворка, а именно, PIN, DynInst, DynamoRIO.

В общем случае процесс взаимодействия фреймворка с исходным кодом представлен на рисунке 1.1.

Рисунок 1.1 – Взаимодействия фреймворка с исходным кодом

Рассмотрим каждый из выбранных фреймворков подробнее.

1.3.1 PIN

PIN – средство динамической двоичной инструментации, предоставляемое бесплатно компанией Intel. PIN позволяет вставлять анализирующий код в произвольное место исполняемого файла [5, 6].

PIN предоставляет широкие возможности инструментации. Например, передавать контекстную информацию, такую как содержимое регистра, в вставляемый код в качестве параметров. PIN автоматически сохраняет и восстанавливает регистры, которые были перезаписаны вставленным кодом. Также PIN предоставляет доступ к символьной и отладочной информации.

Концептуально процесс инструментации состоит из двух компонентов:

- механизм, решающий, где и какой код вставлять;
- код, выполняемый в точках вставки.

Оба этих компонента являются инструментирующим и анализирующим кодом, и оба находятся в одном исполняемом файле – Pintool. Pintool регистрирует функции обратного вызова, которые вызываются каждый раз, когда нужно сгенерировать новый код. Эти функции обратного вызова являются компонентом инструментирующего кода. Они проверяют генерируемый код, и решают, следует ли вызывать функции анализа, которые собирают информацию о приложении.

1.3.2 DynInst

DynInst отличается от других средств динамического инструментирования своим абстрактным, машинно-независимым интерфейсом [6, 7, 8]. Его ключевыми особенностями являются:

- инструментация уже запущенных программ;
- вставка инструментирующего кода в созданную копию исполняющего файла;
- выполнение статического и динамического анализа двоичных файлов и процессов.

Библиотека DynInstAPI предоставляет интерфейс для инструментации и работы с двоичными файлами. Пользователь пишет *мутатор,* который использует DynInstAPI для работы с приложением. Процесс, содержащий мутатор и DynInstAPI называется *процесс-мутатор*. Двоичные файлы и процессы, с которыми работает процесс-мутатор называются *мутирующие*. Также важными понятиями в DynInstAPI являются точка и сниппет. Точка – это место в программе, куда может быть вставлен анализирующий код. Сниппет – это исполняемый код, который должен быть вставлен в программу в определенный момент.

1.3.3 DynamoRIO

DynamoRIO предоставляет пользователю богатую библиотеку, с помощью которой он может создать *клиент.* Клиент – это библиотека, связанная с

DynamoRIO для совместной работы с двоичным кодом. Для взаимодействия с клиентом DynamoRIO предоставляет *события*, которые клиент может отлавливать. События перехватываются клиентом с помощью специальных функций, которые DynamoRIO вызывает в определенный момент [10, 11].

Вывод

В данном разделе были рассмотрены методы анализа программного обеспечения, такие как статический и динамический. А также подвиды динамического анализа, а именно сэмплирование и инструментация. Далее были рассмотрены основные фреймворки для динамической бинарной инструментации. Благодаря богатой библиотеке, удобству и простоте использования DynamoRIO был выбран, как подходящий для выполнения данной работы фреймворк. Поэтому более подробное его описание будет представлено в следующем разделе.

2 Исследование инструментов DynamoRIO для динамической двоичной инструментации

В этом разделе подробно описывается фреймворк DynamoRIO, выбранный нами в прошлом разделе, как наиболее подходящий для выполнения поставленной задачи. Также рассматривается Clang AST, как дополнительный инструмент для разработки профилировщика. В конце будут описан основные стратегии размещения памяти.

2.1 Обзор DynamoRIO

DynamoRIO – это система обработки кода, позволяющая модифицировать исходный код в любой части программы во время выполнения. DynamoRIO [10, 11] предоставляет инструментарий, позволяющий выполнять ряд задач: программный анализ, профилирование, инструментация, оптимизация и т.д.

Главная проблема динамической двоичной инструментации – большие накладные расходы, связанные в первую очередь с постоянным переключении контекста между работающей программой и фреймворком. Здесь и раскрывается одна из особенностей DynamoRIO.

2.1.1 Оптимизация накладных расходов

Цель DynamoRIO в последовательном выполнении блоков кода с возможностью в любой момент проанализировать отдельную инструкцию до ее выполнения. Самый простой способ сделать это – использовать механизм интерпретации. На рисунке 2.1 показан процесс взаимодействия программы с интерпретатором.

На рисунке 2.1 код программы представлен в виде последовательности базовых блоков, перемещение между которыми осуществляется с помощью прямых или косвенных переходов, обозначающихся соответственно прямой и пунктирной линиями.

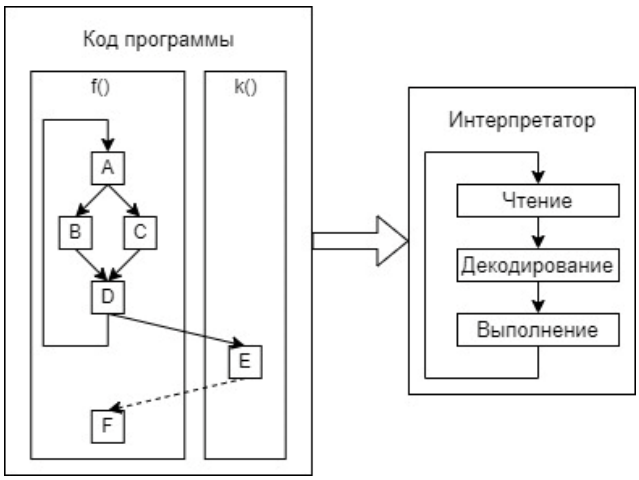

Рисунок 2.1 – Использование механизма интерпретации

Прямой переход – это переход, в команде которого указывается метка следующей команды, к которой необходимо перейти. Косвенный переход отличается от прямого тем, что в команде указывается не метка следующей команды, а регистр, в котором эта команда находится. Это означает, что адрес перехода может быть модифицирован в ходе выполнения программы.

Проблема механизма интерпретации в том, что этот способ является далеко не самым эффективным. Поэтому DynamoRIO использует кэширование базовых блоков (последовательность инструкций, заканчивающаяся инструкцией передачи управления), чтобы использовать их в дальнейшем без необходимости декодирования. Схема такого взаимодействия представлена на рисунке 2.2.

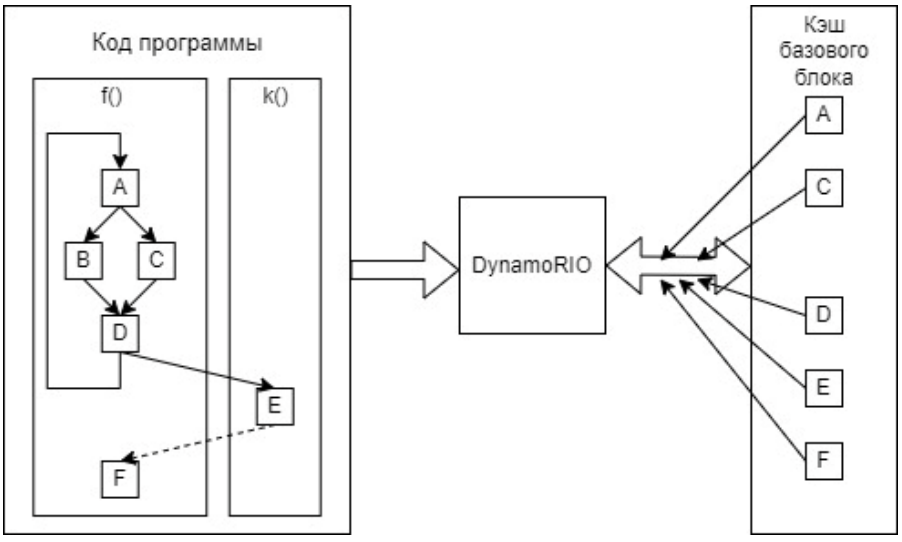

Рисунок 2.2 – Использование кэширования базовых блоков

В конце каждого кэшированного блока состояние приложения должно быть сохранено, а управление возвращено DynamoRIO, который является в данном случае переключателем контекста. Данный прием уже позволяет ускорить работу приложения в 10 раз, но на этом оптимизация не заканчивается.

Если базовый блок заканчивается прямым переходом, и следующий блок уже находится в кэше, то DynamoRIO соединяет их вместе (рисунок 2.3). Назовем этот процесс линковкой.

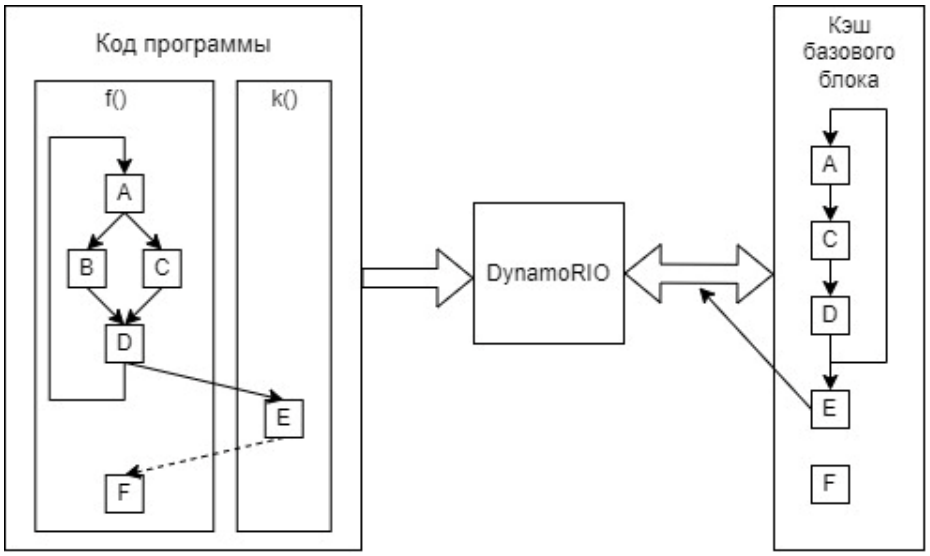

Рисунок 2.3 – Использование линковки базовых блоков

Это означает, что в дальнейшем для перехода между этими базовыми блоками не надо будет передавать управление DynamoRIO. Блоки будут выполняться друг за другом в кэше. Это позволяет сильно уменьшить накладные расходы на переключение контекста, что ускоряет работу приложения еще в 10 раз.

Косвенные переходы нельзя связать таким же образом, так как их цель не может быть явно определена. Проблема решается созданием хеш-таблицы, в которой будут хранится адреса косвенных переходов, преобразованные в соответствующие адреса кэш кода (рисунок 2.4). Тогда при выполнении перехода, будет произведен быстрый поиск в хэш-таблице и переход по найденному адресу. Это также уменьшает накладные расходы на переключение контекста, и ускоряет работу приложения еще в 2.5 раза.

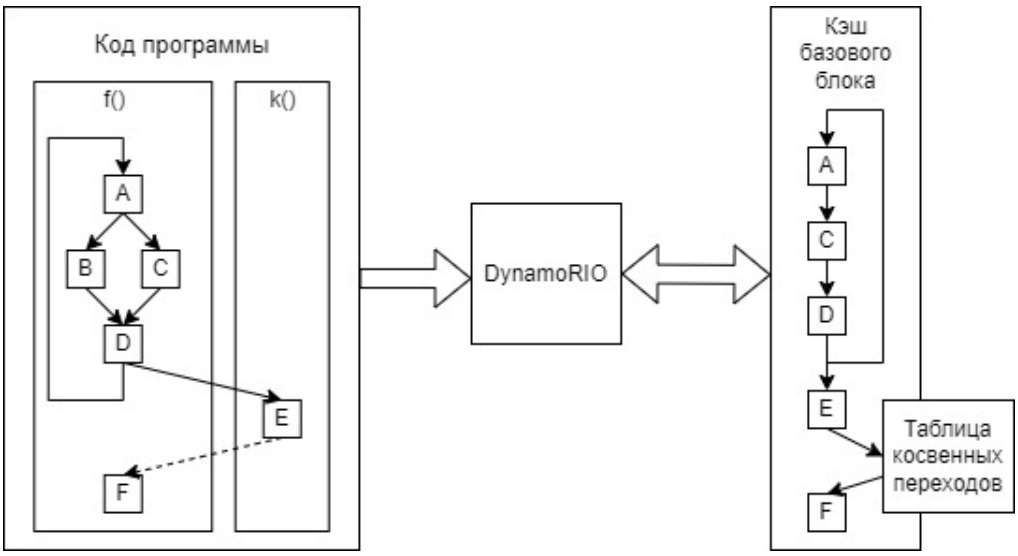

Рисунок 2.4 – Использование хеш-таблицы косвенных переходов

Чтобы повысить эффективность использования косвенных переходов и добиться лучшей компоновки кода, базовые блоки, которые часто выполняются последовательно, объединяют в один блок, называемый трассировкой. При этом в месте, где переход за пределы базового блока производится по косвенному переходу, вставляется проверка, указывает ли косвенный переход на следующий элемент трассировки (рисунок 2.5).

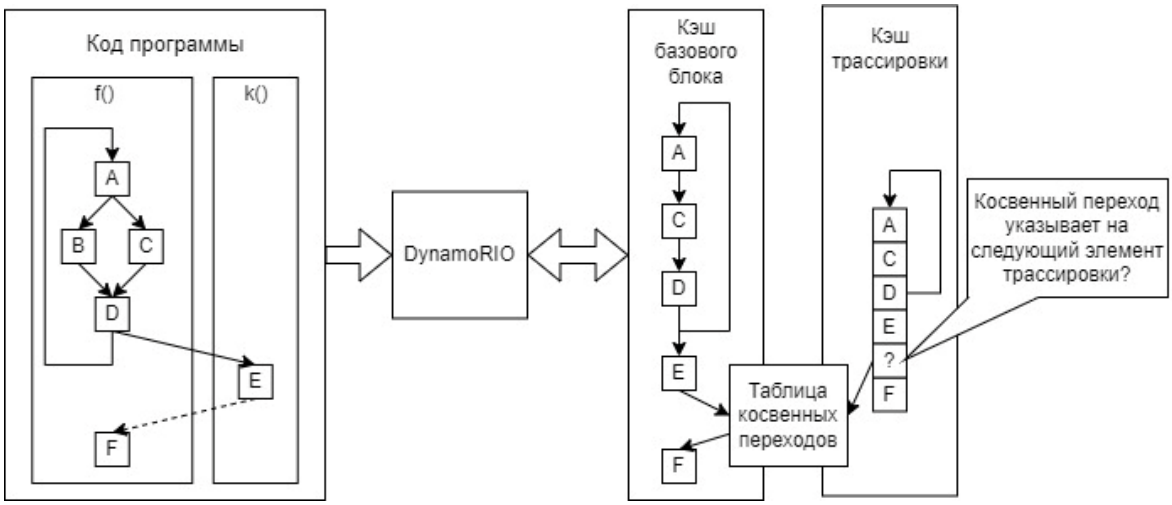

Рисунок 2.5 – Использование кэша трассировки

Такая проверка происходит ощутимо быстрее, чем поиск по хэш-таблице. В случае же, если проверка завершилась неудачей, придется выполнить полноценный поиск следующего блока. Очень важна хорошая компоновка кода, так как необходимо перебить затраты на создание трассировок. Но как правило использование кэша трассировок ускоряют работу программы.

Таким образом накладные расходы на переключение контекста при работе с DynamoRIO сведены к минимуму и оказывают слабое влияние на время работы программы. Основные накладные расходы связаны с выполнением тех участков кода, которые добавил пользователь на этапе инструментации.

2.1.2 Клиентский интерфейс

DynamoRIO берет на себя работу с низкоуровневыми деталями системы, такие как управление кэшем, построение трассировки и переключение контекста, предоставляя пользователю богатую библиотеку для написания собственного клиента. Как уже говорилось в предыдущем разделе, клиент – это библиотека, связанная с DynamoRIO для совместной работы с двоичным кодом. Клиент и DynamoRIO вместе являются прослойкой между работающим приложением и аппаратной частью, что представлено на рисунке 2.6.

Рисунок 2.6 – Взаимодействие DynamoRIO с программой

2.1.3 Система событий

Для взаимодействия с клиентом DynamoRIO предоставляет систему событий, которые клиент может отлавливать. Функции перехвата событий, если они объявляются клиентом, вызываются DynamoRIO в определенный момент. Это могут быть следующие события:

- создание или удаление базового блока и трассировки;
- инициализация и завершение процесса;
- инициализация и завершение потока;
- загрузка и выгрузка библиотек приложения;

- ошибки и исключения приложения;
- системные вызовы;
- сигналы (только в Linux).

Регистрация и отмена регистрации обратного вызова выглядит в общем случае следующим образом:

- dr_register_*_event;
- dr_unregister_*_event.

Клиент регистрирует нужные события при инициализации в своей функции dr_client_main(). Это основная функция, с которой начинается выполнение кода клиента. Как правило, именно в ней регистрируется большинство функций обратного вызова для определенных событий. Затем DynamoRIO вызывает зарегистрированные функции в соответствующее время. Каждое событие имеет определенную процедуру регистрации и связанную с ней процедуру отмены регистрации. Например, функция dr_register_bb_event(bb_event_cb) используется для регистрации события базового блока, в которую передается название функции обратного вызова bb_event_cb. DynamoRIO будет вызывать её перед добавлением нового базового блока в кэш или в трассировку. После завершения программы будет произведена отмена регистрации, осуществляемая с помощью функции dr_unregister_bb_event(bb_event_cb). Клиентам разрешено регистрировать несколько обратных вызовов для одного и того же события. Важно, что для каждой регистрации необходимо выполнить отмену регистрации.

Стоит обратить внимание на то, что событие возникает не при каждом выполнении базового блока, а только тогда, когда он впервые добавляется в кэш. В обычной ситуации событие сработает только один раз, хотя базовый блок будет выполняться еще много раз в кэше.

Тут появляется еще пара понятий, которые важно различать. Точка, в которой возникает событие, при котором код приложения копируется в кэш, называется *время трансформации* (transformation time). Здесь клиент

модифицирует код приложения, добавляя функции мониторинга или изменяя непосредственно сам код. Повторное выполнение этого модифицированного участка кода в кэше называется *время выполнения* (execution time).

DynamoRIO также поддерживает несколько клиентов, причем все они могут зарегистрировать функцию обратного вызова для одного и того же события. В этом случае DynamoRIO упорядочивает обратные вызовы событий в обратном порядке, в зависимости от того, в каком порядке они были зарегистрированы. Это означает, что первый зарегистрированный обратный вызов выполнится последним. Такой подход необходим, поскольку клиент зарегистрированный раньше может влиять на действия клиентов, зарегистрированных позже. При всем этом, DynamoRIO никак не влияет на совместную работу клиентов. Каждый клиент должен самостоятельно обеспечивать совместимость с другими клиентами.

2.1.4 Представление инструкций

Инструкции в DynamoRIO API представлены в виде структуры instr_t, которые объединяются в связный список instrlist_t. Существует ряд библиотек, предоставляющие возможности для манипуляции с инструкциями:

- создание новых инструкций;
- перебор операндов инструкций;
- перебор списка инструкций instrlist_t;
- вставка и удаление инструкций из списка instrlist_t.

Обычно клиент взаимодействует со списком инструкций, формирующих базовые блоки или трассировки, которые в совокупности принято называть *фрагментами*. Фрагменты представляют собой линейный поток управления, что означает, что они имеют один вход и один или несколько выходов.

Представление инструкции содержит операнды, как явные, так и неявные, но промежуточное представление (IR) DynamoRIO по большей части непрозрачно для клиентов.

Изменения в потоке команд, вносимые клиентом, делятся на две категории:

- Изменения или дополнения, рассматриваемые, как часть поведения приложения.
- Дополнения, которые носят наблюдательный характер, и не выполняются, как часть приложения. Это называется *мета-инструкции*.

Клиент должен помечать любые инструкции, которые не должны рассматриваться, как инструкции приложения, как мета-инструкции. Таким образом DynamoRIO поймет, что нужно сперва выполнить эти инструкции и не должны попасть в новые фрагменты базового блока. Для пометки таких инструкций существует несколько функций:

- instr_set_meta(), которая указывает, что переданная в аргументах инструкция является мета-инструкцией;
- instrlist_meta_*, где на место звездочки ставится preinsert, postinsert или append, в зависимости от того, куда необходимо вставить передаваемую в аргументах инструкцию.

2.2 Обзор основных расширений DynamoRIO

DynamoRIO имеет богатую библиотеку расширений, каждое из которых нужно для решения большого количество задач. Но не все из них были использованы для написания программы-профилировщика, поэтому их описание будет опущено в данной работе. Теперь перейдем непосредственно к рассмотрению использованных расширений.

2.2.1 Multi-Instrumentation Manager

Рекомендуется разделять анализ кода приложения от вставки в него инструментария. Специально для этой цели в библиотеке DynamoRIO есть Multi-Instrumentation Manager.

Multi-Instrumentation Manager выступает в роли посредника для координации и объединения нескольких проходов инструменации. Он заменяет обычные события DynamoRIO своими собственными, которые расширяют функционал событий. Это также позволяет иметь несколько компонентов, например одного клиента и нескольких библиотек, или разделять клиент на модули.

Всего представлено четыре отдельных типа проходов, выполняющихся последовательно в перечисленном порядке:

- app-to-app transformations;
- instrumentation insertion;
- instrumentation-to-instrumentation transformations;
- meta-instrumentation transformations.

Первый этап направлен на изменения в самом коде приложения, которые могут повлиять на поведение или производительность приложения. Важно, что вносить изменения в код приложения можно только на этом этапе, но нельзя на последующих.

Второй этап делится на два этапа: анализ кода приложения и вставка инструментации. Проход анализа приложения на втором этапе не может добавлять или изменять список команд и предназначен для анализа кода приложения либо для непосредственного использования, либо для использования на третьем этапе. Проход вставки инструментации, в отличие от других этапов, передает только одну инструкцию обратному вызову, позволяя обрабатывать все инструкции по очереди. На этом этапе позволяется вставлять только мета-инструкции, и только перед текущей инструкцией.

Четвертый этап направлен на оптимизацию всех остальных проходов инструментации.

Пятый этап направлен на проведение отладки всех остальных проходов инструментации.

Каждый компонент, такой как клиент или библиотек, использующий Multi-Instrumentation Manager может зарегистрировать функцию обратного вызова для нескольких или всех этапов. На каждом этапе вызывается функция обратного вызова для всех зарегистрированных компонентов. Это объединяет одинаковые этапы вместе, что гарантирует, что последующие этапы не аннулируют их действия.

2.2.2 Symbol Access Library

Symbol Access Library или библиотека доступа к символам предоставляет информацию о символах программы. С помощью этой библиотеки можно получить такие данные, как имена, адреса модулей и функций программы, номера строк и т.п. Большая часть этих данных хранится в структуре _drsym_info_t, доступ к которой можно получить с помощью разных функций. Например, drsym_lookup_address() возвращает эту структуру для функции находящейся по передаваемому в качестве аргумента адресу. А drsym_enumerate_lines() создает функцию обратного вызова, перечисляющую все строки модуля, для каждой из которых можно получить ряд полезной информации.

2.2.3 Function Wrapping and Replacing Extension

Function wrapping или как будем называть его дальше механизм обертывания позволяет находить и оборачивать функцию по имени. Это значит, что при каждом последующем вызове функции будут вызываться также и две другие функции, пре–функция, вызываемая до основной функции, пост–функция, вызываемая после основной функции. Более того можно даже заменить найденную функцию на другую, либо же вообще отменить ее вызов. Главное учитывать, что это скажется на результате работы программы и может привести в том числе к появлению ряда ошибок.

Сам поиск и обертывание функции происходит в момент загрузки нового модуля (обычно это библиотека или исполняемый файл). Значит для этого нужно использовать drmgr_register_module_load_event(). Обертывание

функции происходит по адресу вызываемой функции. Для библиотечных функций используется dr_get_proc_address(), а для внутренних используется другое расширение DynamoRIO, о котором речь пойдет дальше. В аргументы функции dr_get_proc_address() передаётся имя целевой функции, после чего возвращается адрес целевой функции. Если текущий модуль, в котором происходит поиск, не содержит функции с таким именем, то будет возвращен NULL.

Далее вызывается drwrap_wrap(), в аргументы которой передаются адрес целевой функции и две функции обратного вызова, то есть пре–функция и пост–функция. Одна из этих функций может передаваться, как NULL, что означает, что вызываться она не будет, но обе так передать нельзя.

Применение этих функций несколько отличается. В пре-функции возможно получить или изменить аргументы основной функции, или даже отменить ее вызов вовсе. В пост-функции можно получить или изменить возвращаемое основной функцией значение. Обращение к основной функции в них происходит по передаваемому в каждую void указателя wrapcxt. Также можно передавать необходимые данные между пре и пост-функцией. Например, можно получить значения аргументов основной функции в пре-функции, после чего передать их в пост-функцию, для выведения сразу всей информации о основной функции.

С помощью функции drwrap_replace() можно заменить исходную функцию. Новая функция будет передаваться клиенту через события базового блока в числе прочих инструкций, и будет выполняться, как обычная функция приложения. При этом замененная функция останется в базовом блоке, но будет представлять из себя инструкцию перехода, и код этой функции выполнен не будет.

2.3 Обзор Clang AST

Одна из проблем, с которой пришлось столкнуться это то, что DynamoRIO не владеет информацией о типах и именах переменных. Это значит, что в ходе инструментации нет возможности получить эту информацию. Поэтому был использован дополнительный инструмент – Clang AST [12]. Clang – это фронтенд, переводящий код на языке семейства C в промежуточное представление. При этом он параллельно может анализировать и оптимизировать код.

2.3.1 Абстрактное синтаксическое дерево

Основными классами в Clang являются:

- Decl – объявление;
- Stmt – оператор;
- Type – тип.

В основе работы Clang лежит абстрактное синтаксическое дерево или же AST. AST представляет собой иерархию узлов. Все они наследуются от одного из трех основных классов. На рисунке 2.7 представлено AST дерево простой программы.

```
-FunctionDecl 0x564f1454eb70 <test.c:3:1, line:19:1> line:3:5 main 'int ()'
 `-CompoundStmt 0x564f1454eed8 <col:12, line:19:1>
   |-DeclStmt 0x564f1454ecb0 <line:4:5, col:14>
   | `-VarDecl 0x564f1454ec28 <col:5, col:13> col:9 used a 'int' cinit
   |   `-IntegerLiteral 0x564f1454ec90 <col:13> 'int' 5
   |-DeclStmt 0x564f1454ed68 <line:5:5, col:14>
   | `-VarDecl 0x564f1454ece0 <col:5, col:13> col:9 used b 'int' cinit
   |   `-IntegerLiteral 0x564f1454ed48 <col:13> 'int' 2
   |-DeclStmt 0x564f1454ee90 <line:6:5, col:18>
   | `-VarDecl 0x564f1454ed98 <col:5, col:17> col:9 c 'int' cinit
   |   `-BinaryOperator 0x564f1454ee70 <col:13, col:17> 'int' '+'
   |     |-ImplicitCastExpr 0x564f1454ee40 <col:13> 'int' <LValueToRValue>
   |     | `-DeclRefExpr 0x564f1454ee00 <col:13> 'int' lvalue Var 0x564f1454ec28 'a' 'int'
   |     `-ImplicitCastExpr 0x564f1454ee58 <col:17> 'int' <LValueToRValue>
   |       `-DeclRefExpr 0x564f1454ee20 <col:17> 'int' lvalue Var 0x564f1454ece0 'b' 'int'
   `-ReturnStmt 0x564f1454eec8 <line:18:5, col:12>
     `-IntegerLiteral 0x564f1454eea8 <col:12> 'int' 0
```

Рисунок 2.7 – Абстрактное синтаксическое дерево

Важно, что каждый узел содержит информацию о своем месте в исходном коде программы. Например, на приведенном выше рисунке видно, что в пятой строке происходит объявление переменной a, имеющей тип int, и присваемое значение 5.

2.3.2 Матчеры

Весь анализ и трансформация кода в Clang AST происходит с помощью взаимодействия с узлами AST дерева. Это происходит с помощью матчеров [13] – выражений, которые сопоставляются с узлами дерева. Собираются такие выражения из разных компонентов, из чего получаются вполне осмысленные предложения, что очень облегчает написание матчеров. Например, выражение callExpr(callee(functionDecl(hasName("malloc")))).bind("functionCall") буквально означает «узел типа callExpr, вызывающий функцию, имеющую имя malloc». В данном случае functionCall – имя, под которым Clang дальше будет возвращать найденный узел.

В одной программе может быть множество матчеров, каждый из которых добавляется в специальную структуру данных MatchFinder, из которой потом можно будет получить все найденные узлы, соответствующие заданным матчерам.

2.4 Стратегии размещения памяти

Стандартные аллокаторы памяти [14, 15] имеют ряд существенных проблем. Не известно, где находится тот участок памяти, который вы только что выделили относительно других участков памяти, использующихся вашей программой. Это приводит к постоянным промахам кэша, что сильно сказывается на времени работ программ. Также будет возникать фрагментация памяти, что ведет к еще большим накладным расходам и раздутию количества памяти, используемой программой. Эти проблемы обусловлены тем, что стандартные аллокаторы должны подходить для решения любых задач. Поэтому для решения конкретных задач были придуманы различные стратегии размещения памяти. Рассмотрим основные из них.

2.4.1 Линейный аллокатор

Суть линейного аллокатора заключается в том, что он хранит только указатель на начало блока выделенной под него памяти и смещение относительно начала блока. Смещение указывает на начало свободной памяти. Соответственно, при каждом выделении памяти смещение будет двигаться все дальше от начала. Схема линейного аллокатора представлена на рисунке 2.8.

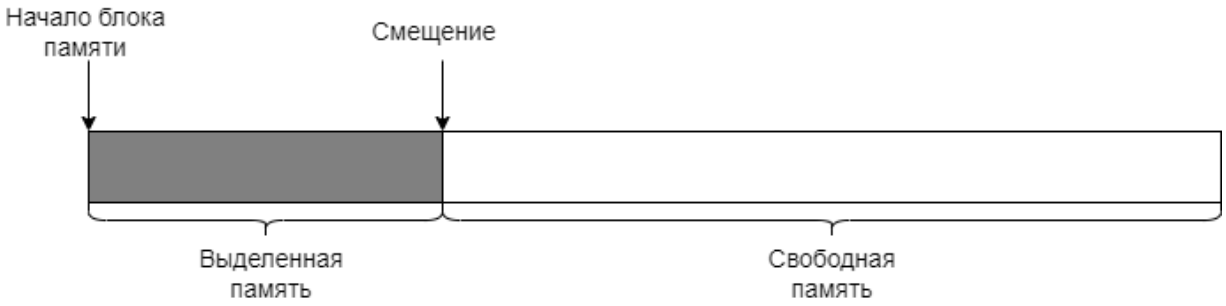

Рисунок 2.8 – Линейный аллокатор

Преимущество такого аллокатора в простоте реализации и практически отсутствующей фрагментации, так как блоки выделенной памяти расположены друг за другом. Единственная причина фрагментации – выравнивание, размер которого можно менять в зависимости от нужд конкретной задачи.

Главный недостаток заключается в невозможности освобождения отдельного участка памяти. Можно освободить только всю память целиком.

2.4.2 Пуловый аллокатор

Пуловый аллокатор разбивает выделенную под него память на блоки одинакового размера. При запросе на выделение памяти аллокатор возвращает указатель на один из свободных блоков. Обычно свободные блоки памяти хранятся в виде связного списка, содержащего адрес каждого блока. На рисунке 2.9 представлен пуловый аллокатор до и после выделения памяти. Как видно, свободные блоки удаляются из списка, в момент выделения.

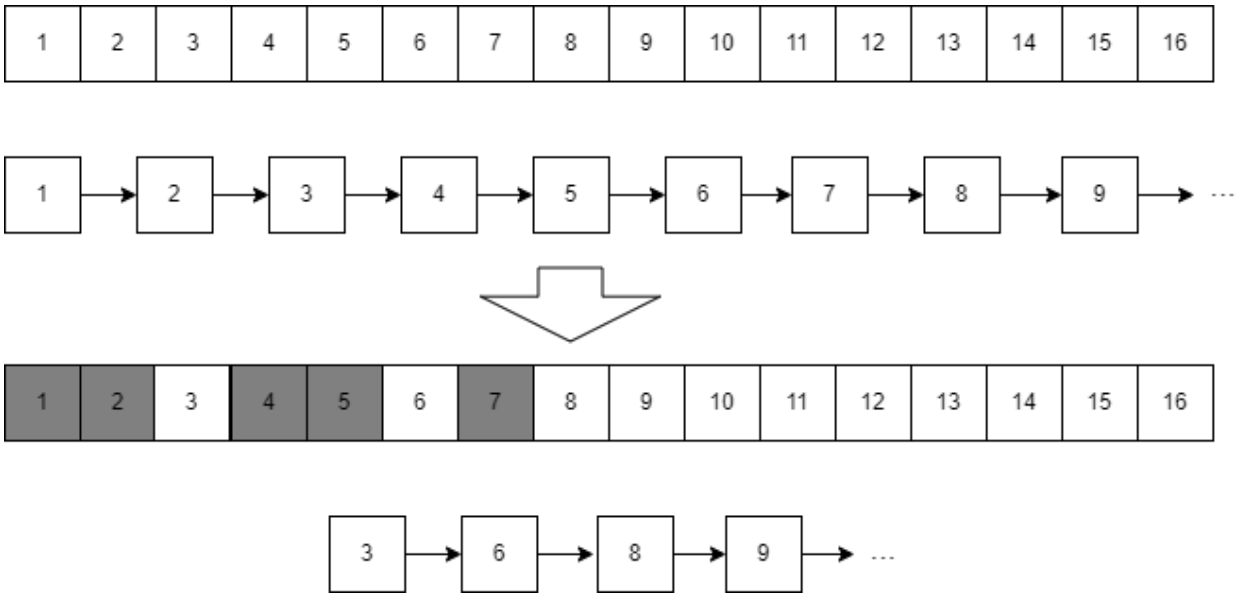

Рисунок 2.9 – Пуловый аллокатор

В отличие от линейного аллокатора, пуловый аллокатор позволяет очищать любые участки памяти. Для этого нужный блок просто добавляется в список свободных блоков.

2.3.3 Стековый аллокатор

Стеков аллокатор вместе с указателем на начало блока и смещением хранит заголовок, прикрепляемый к каждому блоку выделяемой памяти. Заголовок имеет фиксированный размер и хранит объем выделенной памяти. При этом пользователь получит указатель не на заголовок, а на начало блока памяти. Схема стекового аллокатора представлена на рисунке 2.10.

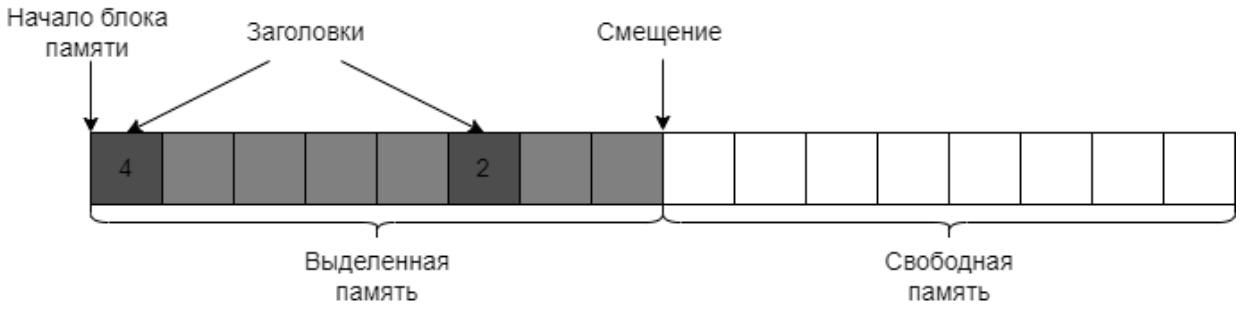

Рисунок 2.10 – Стековый аллокатор

Стековый позволяет очищать конкретные участки памяти. Правда делать это можно только по принципу стека, постоянно сдвигая смещение на размер блока, взятый из заголовка, и размер этого заголовка.

Описанные выше стратегии размещения памяти достаточно просты в своей реализации и при желании их можно спокойно реализовать самостоятельно. Они являются основой, на которой построены многие другие аллокаторы, использующиеся повсеместно, каждый для своей задачи. Например, MiMalloc, разрабатываемый компанией Майкрософт, или JeMalloc, использующийся компанией Facebook и многими другими крупными проектами.

2.3.4 Тестирование аллокаторов

В ходе работы были протестированы существующие аллокаторы: стандартный, JeMalloc [16] и MiMalloc [17]. Тестирование проводилось на простой программе, выполняющей цикличное выделение памяти. На рисунке 2.11 представлены результаты тестирования.

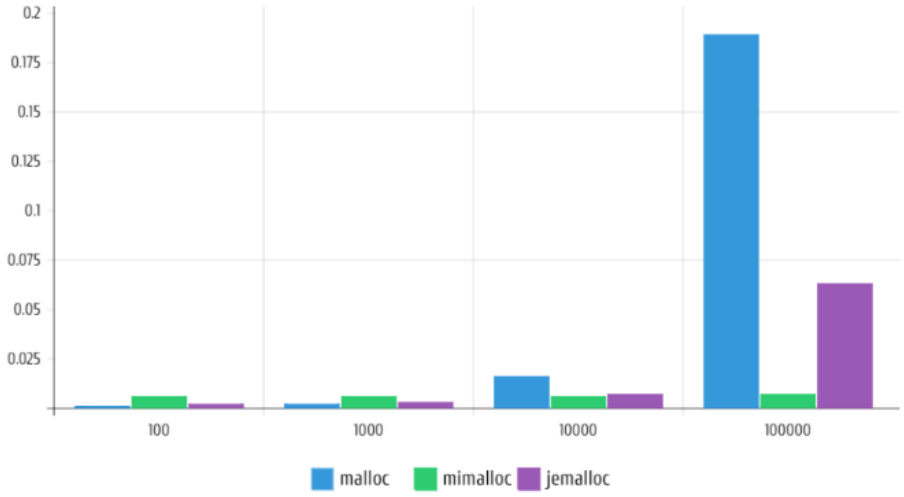

Рисунок 2.11 – Тестирование аллокаторов

На рисунке представлена зависимость времени выполнения от количества итераций. Видно, что стандартный malloc в среднем показывает себя хуже остальных, и его эффективность резко падает при большом количестве выделений и очистки памяти. Это связано с сильной фрагментацией памяти. У jemalloc ситуация похожая, хотя и падение эффективности не такое сильное,

как у стандартного malloc. В свою очередь mimalloc работает одинаково эффективно при разном количестве итераций. Таким образом, можно сделать вывод, что mimalloc на данный момент является лучшим вариантом для замены стандартного аллокатора.

Вывод

В данном разделе были рассмотрены инструменты, используемые в работе над профилировщиком. Был подробно описан DynamoRIO, его главное преимущество, а именно минимизация накладных расходов и то, какими методами это достигается. Также был рассмотрен принцип работы DynamoRIO и основные понятия этого фреймворка. Для разработки профилировщика понадобится еще один инструмент – Clang AST, который тоже был рассмотрен в разделе. В конце были описаны стратегии размещения памяти, а также протестированы наиболее распространенные аллокаторы.

3 Разработка собственного профилировщика

В данном разделе будет подробно описан процесс создания своего собственного профилировщика на основе фреймворка DynamoRIO. Будут использованы все описанные выше технологии. Также будет продемонстрированы результаты профилирования тестовых программ.

3.1 Написание клиента

Написание любого клиента начинается с описания dr_client_main() функции. В ней объявляются основные глобальные переменные, инициализируются расширения API и регистрируются функции обратного вызова для событий. Рассмотрим подробнее получившуюся main–функцию (рисунок 3.1).

```
DR_EXPORT void
dr_client_main(client_id_t id, int argc, const char *argv[])
{
    // 1
    file = log_file_open();
    log_file = log_stream_from_file(file);
    num_ref = 0;
    // 2
    if (!drmgr_init() || !drwrap_init() ||
        drsym_init(0) != DRSYM_SUCCESS)
        DR_ASSERT(false);
    // 3
    dr_register_exit_event(event_exit);
    if (!drmgr_register_module_load_event(module_load_event))
        DR_ASSERT(false);
}
```

Рисунок 3.1 – Реализация main–функции

Для удобства описания комментариями были выделены части кода, которые разберем в отдельности. Первая часть – объявление глобальных переменных. Эти переменные будут видны из любой части программы. В данном случае открывается лог–файл, в который будет выведена информация о результатах профилирования. Вторая часть – инициализация расширений API, которые будут использованы. Все они были описаны в предыдущем разделе.

Здесь появляется функция DR_ASSERT(), которая еще неоднократно будет использоваться в программе. Она получает в качестве аргумента булевое выражение и проверяет, верно ли оно. Если ложно, то выводится ошибка, и процесс мгновенно завершается, без вызова каких–либо событий завершения. Третья часть – регистрация функций обратного вызова для событий. Здесь мы регистрируем событие завершения клиента и событие загрузки модуля. Это событие будет срабатывать каждый раз, когда загружается новый модуль, например библиотека или исполняемый файл.

Далее напишем функцию обратного вызова, для события загрузки модуля (рисунок 3.2).

```
//Функция обратного вызова для события загрузки модуля
static void
module_load_event(void *drcontext, const module_data_t *mod, bool loaded)
{
    app_pc towrap = (app_pc)dr_get_proc_address(mod->handle, "malloc");
    if(towrap != NULL){
        drwrap_wrap(towrap, wrap_pre_malloc, wrap_post);
    }
}
```

Рисунок 3.2 – Функция обратного вызова

В аргументах функции обратного вызова передается указатель на структуру, содержащую информацию о текущем модуле. Там содержится такая информация, как адрес начала и конца модуля, имя модуля и полный путь до него. Функция dr_get_proc_address() находит в модуле функцию с именем malloc и возвращает адрес ее вызова. Если полученное значение не NULL, значит такая функция найдена. С помощью drwrap_wrap() мы оборачиваем функцию по найденному адресу с помощью двух функций, передаваемых в аргументах. Они будут вызываться каждый раз, когда будет вызываться обернутая функция.

Теперь рассмотрим пре–функцию и пост–функцию обратного вызова, которыми был обернута наша изначальная функция, изображенные соответственно на рисунке 3.3 и рисунке 3.4.

```
//Пре-функция обертки целевой функции
static void
wrap_pre_malloc(void *wrapcxt, OUT void **user_data)
{
    /*Получаем первый аргумент malloc,
      который хранит в себе количество выделенной памяти
      и добавляем его в разделяемую между оборачивающими
      функциями область памяти*/
    size_t sz = (size_t)drwrap_get_arg(wrapcxt, 0);
    *user_data = (void *)sz;
    time_start = dr_get_microseconds();
}
```

Рисунок 3.3 – Пре–функция

```
wrap_post(void *wrapcxt, void *user_data)
{
    uint64 time_end = dr_get_microseconds();
    module_data_t *mod;
    //Получаем адрес, где вызывается malloc и откуда он вызывается
    app_pc ret_addr = (app_pc)drwrap_get_retaddr(wrapcxt);
    app_pc ret_val = drwrap_get_retval(wrapcxt);
    //Получаем указатель на модуль по адресу
    mod = dr_lookup_module(ret_addr);
    //Объявляем структуру, хранящую информацию о символах
    //и заполняем ее базовыми значениями
    drsym_error_t symres;
    drsym_info_t sym;
    char name[256];
    char file[256];
    sym.struct_size = sizeof(sym);
    sym.name = name;
    sym.name_size = 256;
    sym.file = file;
    sym.file_size = 256;
    //Получаем структуру с информацией о символах для malloc
    symres = drsym_lookup_address(mod->full_path, ret_addr - mod->start,
                                  &sym, DRSYM_DEFAULT_FLAGS);
    //Выводим всю необходимую информацию в лог-файл
    if(sym.line > 0 ){
        fprintf(f, "%s %s %d %ld %d %ld %d\n", sym.file, sym.name,
                sym.line, ret_val, (size_t)user_data, ret_addr, time_end - time_start);
    }
    dr_free_module_data(mod);
}
```

Рисунок 3.4 – Пост–функция

Первой вызовется пре–функция wrap_pre(). В ней по указателю wrapcxt из функции drwrap_get_arg() получим первый аргумент обертываемой

функции. В данном случае это malloc, а значит аргумент у него всего один и означает количество выделяемой памяти. Дальше с помощью user_data мы можем передать эти данные в пост–функцию. Также тут замеряется время начала выполнения функции.

После того, как целевая функция была выполнена, вызывается пост–функция wrap_post(). Сперва замеряем время окончания выполнения функции. Далее с помощью drwrap_get_retaddr() получаем адрес, откуда целевая функция вызывалась внутри программы, а с помощью drwrap_get_retval() получаем адрес, куда malloc выделил память. Затем используем dr_lookup_module(), чтобы узнать, из какого модуля программы вызывается целевая функция. Это определяется по адресу вызова функции. Поскольку мы сами нашли этот модуль, важно в конце функции очистить module_data_t с помощью dr_free_module_data().

Чтобы получить оставшуюся информацию о функции, необходимо создать структуру drsym_info_t, которая будет содержать все, необходимые нам данные. Данную структуру необходимо заполнить вручную: выделить память под некоторые поля структуры, задать некоторые параметры, которые потом будут изменяться. Теперь используем drsym_lookup_address(). Эта функция заполнит подготовленную нами структуру. Используем несложную формулу ret_addr – mod–>star. Здесь ret_addr – абсолютный адрес вызываемой функции, а mod–>start – абсолютный адрес начала модуля. Соответственно их разница будет смещением целевой функции от начала модуля. Таким образом мы получим информацию именно о искомой функции malloc.

Полученная информация выводится в лог–файл. Данные выводятся в виде таблицы в формате, представленном в таблице 3.1.

Таблица 3.1 – Формат вывода информации

Путь до файла	Имя функции	Номер строки	Адрес, куда выделилась память	Размер выделенной памяти	Адрес, откуда был вызван malloc

Далее запускается клиент, создающий трассировку (этот клиент был взят из базовых примеров для работы с DynamoRIO, поэтому описан не будет), то есть выводящий в файл информацию о всех обращениях к памяти, чтение или запись. Используя полученные с помощью предыдущего клиента данные об адресах, куда выделялась память, можно выбрать все необходимые обращения к памяти.

3.2 Написание матчера

Поскольку, как писалось выше, DinamoRIO не сохраняет информацию о типах и именах переменных, был использован Clang AST. Прежде чем писать матчер, надо учесть, что есть две разные ситуации, когда вызывается malloc. В момент объявления переменной, и когда переменная уже объявлена, и выделение памяти происходит дальше по ходу программы. Чтоб наглядно продемонстрировать разницу, рассмотрим пример, представленный на рисунке 3.4.

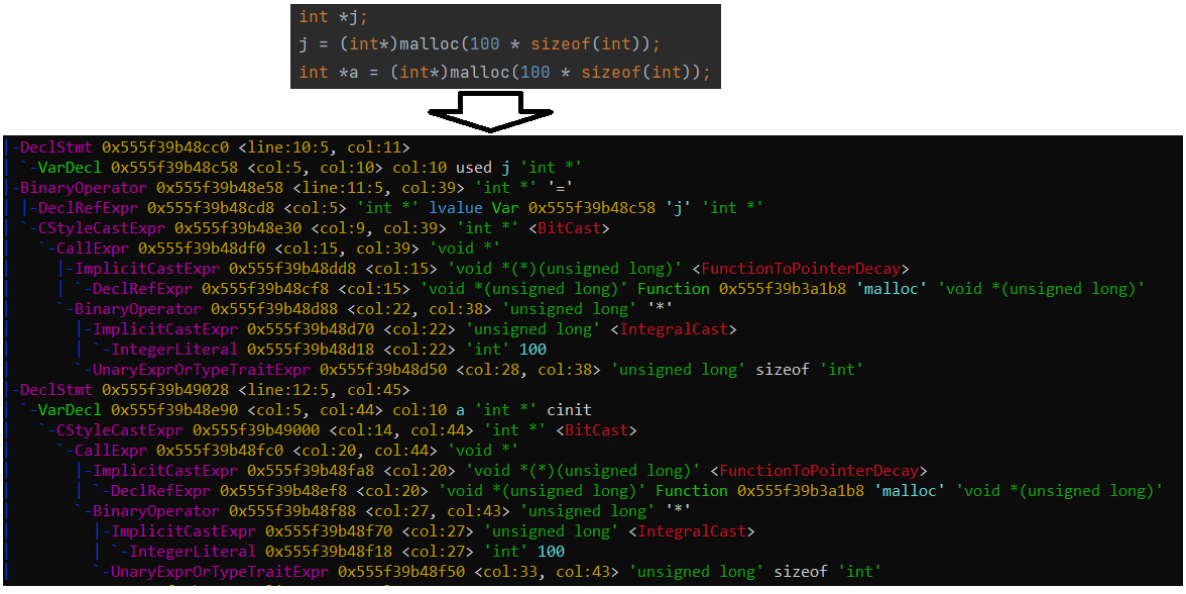

Рисунок 3.4 – Пример AST дерева

Можно заметить, что в каждом случае функция malloc является потомком разных узлов. В первом она потомок BinaryOperator, в другом VarDecl. Это значит, что и матчер для этих случаев должен выглядеть по–разному.

Теперь можно переходить к написанию матчеров, которые ищут все функции malloc, встречающиеся в программе (рисунок 3.5).

```
const auto mallocDeclaration = callExpr(
        callee(functionDecl(hasName("malloc"))),
        hasAncestor(
                varDecl().bind("declaration"))
).bind("functionCallDeclaration");

Finder.addMatcher(mallocDeclaration, &MallocFinderHandler);

const auto mallocAssignment = callExpr(
        callee(functionDecl(hasName("malloc"))),
        hasAncestor(
                binaryOperator(hasOperatorName("=")).bind("assignment1")
        )
).bind("functionCallAssigment");

Finder.addMatcher(mallocAssignment, &MallocFinderHandler);
```

Рисунок 3.5 – Матчер

Первый матчер находит вызов функции malloc в момент объявления переменной. Этот матчер означает, что надо найти все узлы типа CallExpr, которые вызывают функцию, имеющую имя malloc, и имеющие предка в виде узла VarDecl.

Второй матчер создавался по аналогии и находит все узлы типа CallExpr, которые вызывают функцию, имеющую имя malloc, и имеющие предка в виде узла BinaryOperator «=».

Информация о найденных с помощью этих матчеров вызовах функции malloc записывается в файл. Таким образом мы получили номер строки, в котором в оригинальной программе происходило динамическое выделение памяти, тип и имя переменной.

Теперь проанализировав оба полученных файла, мы можем создать итоговую трассировку. Написанный нами клиент DynamoRIO сопоставляет номера строк в каждом из файлов, заменяя номер строки на имя переменной. Получаем готовую трассировку, пример которой представлен на рисунке 3.6.

```
139915696700844 4 w j
139915696698024 8 w p
139915696698024 8 r p
139915696700488 4 w j
139915696698016 1 w p
```

Рисунок 3.6 – Пример трассировки

Здесь сперва идет адрес, к которому обращается программа. Затем количество записанных или прочитанных байт. Далее тип операции чтение или запись. И наконец имя переменной.

Теперь протестируем получившийся профилировщик.

3.3 Тестирование

Сперва протестируем на простой программе (рисунок 3.6).

1
```c
#include <stdlib.h>

int main() {
    int *j;
    int a;
    j = (int*)malloc(100 * sizeof(int));
    for(int i = 0; i < 100; i++){
        j[i] = i;
        a = j[i];
    }
    free(j);
    return 0;
}
```

2
```
139964980581024 4 w j
139964980581024 4 r j
139964980581028 4 w j
139964980581028 4 r j
139964980581032 4 w j
```

3 /home/ivan/projects/TraceDump/test.c main 6 140223581487776 400 140223581467007

Рисунок 3.6 – Тестирование простой программы

Здесь под номером один исходная программа, под номером два часть трассировки, на которой видно, что происходило последовательное обращение к массиву j, каждый раз читая или записывая по 4 байта. Под номером три идет список всех вызовов функции malloc.

Теперь проведем тестирование программы, реализующей работу кучи (рисунок 3.7).

```
139775705089712 8 r h
139775705089720 4 r h
139775705089720 4 w h
139775705089720 4 r h
139775705089752 8 w h
139775705089720 4 r h
139775705089712 8 r h
```

Рисунок 3.7 – Тестирование реализации кучи

Проведенные тесты показывают работоспособность разработанного профилировщика

Вывод

В данном разделе был описан процесс работы над профилировщиком, который выводит информацию о всех найденных аллокаторах, и затем строит трассировку исходной программы. Профилировщик состоит из двух частей: клиента DynamoRIO и анализатора Clang AST. Так как DynamoRIO не владеет информацией о типах и именах переменных, они были получены с помощью Clang AST. Также были проведен тесты профилировщика.

4 Экономическое обоснование

В данном разделе производится расчёт себестоимости исследования.

4.1 Обоснование целесообразности исследования

Сложность программного обеспечения постоянно растет, а значит растет и сложность оценки его безопасности. Динамическая бинарная инструментация позволяет получать информацию о работе программы в ходе ее выполнения. Эта особенность гарантирует, что мы исследуем все участки программы и получим более точную информацию, чем при статическом анализе. Таким образом, динамическая инструментация является актуальной темой для изучения, так как это мощный способ профилирования программного обеспечения.

4.2 Длительность этапа разработки и трудоемкость

Первым этапом при расчете полных затрат на выполнение разработки является составление детализированного плана работ, которые необходимо выполнить на каждом этапе проектирования [18]. Продолжительность выполняемых работ определяется по факту, то есть учитывать время, фактически затраченное исполнителями на выполнение каждого этапа и работы. В данном обосновании единица измерения трудоемкости работ – человеко-дни. Месячный оклад руководителя возьмем равным 85000 рублей, студента – 3600 рублей. Ставка вычисляется путем деления месячного оклада исполнителя на количество рабочих дней в месяце, в данной работе принято равным 21.

Таким образом, стоимость человеко-дня составляет:

– для руководителя: $\text{СЧД}_{\text{рук}} = \frac{85000}{21} = 4\,048$ руб/день;

– для студента: $\text{СЧД}_{\text{студ}} = \frac{2200}{21} = 105$ руб/день;

План с указанием исполнителей, трудоемкости и ставкой представлен в таблице 4.1.

Таблица 4.1 Исполнители и трудоемкость этапов разработки

№	Наименование работы	Исполнитель	Трудоемкость, человеко-дни	Ставка, руб./день
1	Разработка и выдача технического задания	Руководитель	1	4048
2	Получение и изучение задания	Студент	2	105
3	Изучение теоретического материала по теме	Студент	8	105
4	Изучение документации DynamoRIO	Студент	8	105
5	Обучение работе с DynamoRIO	Студент	10	105
6	Разработка программы	Студент	18	105
7	Тестирование программы	Студент	6	105
8	Консультации с научным руководителем	Руководитель	1	4048
		Студент	1	105
9	Оформление пояснительной записки	Студент	10	105

Таким образом, на выполнение всех работ было затрачено:

– студентом: 63 ч. дн.

– руководителем: 2 ч. дн.

4.3 Расчет затрат на оплату труда исполнителей

На основе данных о трудоемкости выполняемых работ и ставки соответствующих исполнителей определим расходы на заработную плату исполнителей и отчислений на страховые взносы на обязательное социальное, пенсионное и медицинское страхование.

Расходы на основную заработную плату исполнителей:

$$З_{\text{осн.зп.ст}} = \sum_{i=1}^{k} T_{\text{ст},i} * C_{\text{ст},i} = 63 * 105 = 6615, \quad (4.1)$$

$$З_{\text{осн.зп.рук}} = \sum_{i=1}^{k} T_{\text{рук},i} * C_{\text{рук},i} = 2 * 4048 = 8096. \quad (4.2)$$

где $З_{осн.зп.ст/рук}$ – расходы на основную заработную плату студента или руководителя, руб.; $T_{ст/рук}$ – время, затраченное студентом или руководителем на работу, дней; $C_{ст/рук}$ – ставка студента или руководителя, руб./день.

Расходы на дополнительную заработную плату исполнителей:

$$З_{доп.зп.ст} = З_{осн.зп.ст} * \frac{Н_{доп}}{100} = 6615 * \frac{8,3}{100} = 549, \quad (4.3)$$

$$З_{доп.зп.рук} = З_{осн.зп.рук} * \frac{Н_{доп}}{100} = 8096 * \frac{8,3}{100} = 672. \quad (4.4)$$

где $З_{доп.зп.ст/рук}$ – расходы на дополнительную заработную плату студента или руководителя, руб.; $З_{осн.зп.ст/рук}$ – расходы на основную заработную плату студента или руководителя, руб.; $Н_{доп}$ – норматив дополнительной заработной платы, равный 8,3%.

Отчисления на страховые взносы на обязательное социальное, пенсионное и медицинское страхование с основной и дополнительной заработной платы исполнителей:

$$З_{соц.ст} = (З_{осн.ст} + З_{доп.ст}) * \frac{Н_{соц}}{100} = (6615 + 549)\frac{30}{100} = 2149, \quad (4.5)$$

$$З_{соц.рук} = (З_{осн.рук} + З_{доп.рук}) * \frac{Н_{соц}}{100} = (8096 + 672)\frac{30}{100} = 2630. \quad (4.6)$$

где $З_{соц.ст/рук}$ – отчисления на социальные нужды с заработной платы, руб.; $Н_{соц}$ – норматив отчислений на страховые взносы на обязательное страхование, равный 30%; $З_{доп.ст/рук}$ – дополнительная заработная плата студента или руководителя, руб.; $З_{осн.ст/рук}$ – основная заработная плата студента или руководителя, руб.

Итого затраты по статье «Расходы на оплату труда» составляют 15932 руб., а по статье «Отчисления на социальные нужды» 4779 руб.

4.4 Расчет затрат на сырье и материалы

Затраты на сырье и материалы рассчитываются по следующей формуле и учитывают транспортные нужды считаются по формуле:

$$З_М = \sum_{l=1}^{L} G_l Ц_l (1 + \frac{Н_{т.з.}}{100}). \quad (4.7)$$

где $З_М$ – затраты на сырье и материалы, руб.; l – индекс вида сырья или материала; G_l – норма расхода l-го материала на единицу продукции, ед.; $Ц_l$ – цена приобретения l-го материала, руб./ед.; $Н_{т.з.}$ – норма транспортно-заготовительных расходов, равная 10%.

Расчёт затрат на сырье и материалов представлен в таблице 4.2.

Таблица 4.2 – Расходы на сырье и материалы

Изделие	Тип	Норма расходов на изделие, ед.	Цена за единицу, руб.	Сумма на изделие, руб.
Бумага офисная	Формат А4, 500 листов	1	585	585
Картридж для принтера	Черный	1	3900	3900
Папка офисная	Формат А4	1	200	200
Ручка	Черная	1	15	15
Итого:				4700

Общие расходы по статье «Сырье и материалы» составили 4700 руб.

4.5 Амортизационные отчисления

Амортизационные отчисления по основному средству i рассчитываются по формуле:

$$A_i = Ц_{п.н.i} \cdot \frac{H_{ai.}}{100}, \quad (4.8)$$

где A_i – амортизационные отчисления за год по i-му основному средству, руб.; $Ц_{п.н.i}$ – первоначальная стоимость i-го основного средства, руб.; H_{ai} – годовая норма амортизации i-го основного средства, %.

Норма амортизации рассчитывается как:

$$H_{ai} = \frac{1}{T_{п.и.}} \cdot 100\%, \quad (4.9)$$

где $T_{\text{п.и.}}$ – срок полезного использования основного средства.

Величина амортизационных отчислений по i-му основному средству, используемому при работе над ВКР, определяются по формуле:

$$A_{i\text{ВКР}} = A_i \cdot \frac{T_{i\text{ВКР}}}{12}, \quad (4.10)$$

где $A_{i\text{ВКР}}$ – амортизационные отчисления по i-му основному средству, используемому при работе над ВКР, руб.; A_i – амортизационные отчисления по i-му основному средству за год, руб.; $T_{i\text{ВКР}}$ – время, в течение которого используется i-е основное средство, мес.

Используемое в роботе над ВКР основное средство представлено в таблице 4.3.

Таблица 4.3 – Основное средство

Основное средство	Первоначальная стоимость, руб.	Срок полезного использования, лет
Ноутбук Acer Aspire 7 A715-71G-50LS	58990	4

Расчет нормы амортизации:

$$H_{ai} = \frac{1}{4} \cdot 100\% = 25\% \quad (4.11)$$

Расчет амортизационных отчислений за год:

$$A_{\text{ноутбук}} = 58990 \cdot \frac{25}{100} = 14748 \text{ руб.} \quad (4.12)$$

Расчет амортизационных отчисления при работе над проектом. Время, в течение которого используется основное средство возьмем равным 2 месяцам.

$$A_{\text{ноутбук}} = 14748 \cdot \frac{2}{12} = 2458 \text{ руб.} \quad (4.13)$$

Итого общие затраты по статье «Амортизационные отчисления» составляют 2458 рублей.

4.6 Расчет накладных расходов

Накладные расходы рассчитываются по формуле:

$$З_{\text{накл.расх.}} = \left(З_{\text{осн.зп}} + З_{\text{доп.зп}}\right) * \frac{Н_{\text{накл.расх.}}}{100}, \quad (4.14)$$

где $З_{накл.расх}$ – накладные расходы, руб.; $З_{осн.зп}$ – расходы на основную заработную плату исполнителей, руб.; $З_{доп.зп}$ – расходы на дополнительную заработную плату исполнителей, руб.; $Н_{накл.расх}$ – процент накладных расходов, равный 20%.

$$З_{накл.расх.} = (8096 + 6615 + 672 + 549) * \frac{20}{100} = 3186 \quad (4.15)$$

Таким образом, затраты по статье «Накладные расходы» составляют 3186 рублей.

4.7 Себестоимость ВКР

В таблице 4.4 указана себестоимость ВКР.

Таблица 4.4 – Себестоимость ВКР

№ п/п	Наименование статьи	Сумма, руб.
1	Расходы на оплату труда	15932
2	Отчисления на социальные нужды	4779
3	Материалы	4700
4	Амортизационные отчисления	2458
5	Накладные расходы	3186
ИТОГО затрат		31055

Полные затраты на разработку составляют 31 тысяча 55 рублей. Основными статьями расхода являются расходы на оплату труда (51%), отчисления на социальные нужды (15%) и на материалы (15%).

Вывод

Динамическая бинарная инструментация – актуальная тема для изучения, так как профилирование с ее помощью позволяет упростить поиск неисправностей приложения, найти участки кода, которые можно оптимизировать. У нее есть ряд преимуществ, такие как, простота использования и полное покрытие анализируемого кода. Эти преимущества выделяют ее на фоне других

способов профилирования программного обеспечения. Благодаря динамической бинарной инструментации можно добиться снижения стоимости разработки и поддержки программного обеспечения, ведь она позволяет легко и быстро найти ошибки и места для оптимизации, на поиск которых без ее использования могло уйти в разы больше времени и ресурсов.

ЗАКЛЮЧЕНИЕ

Результатом выпускной квалификационной работы является программа–профилировщик, находящий все аллокации памяти и строящий трассировку работы с динамической памятью. В ходе написания работы были подробно рассмотрены методы анализа программного обеспечения, и особое внимание было уделено динамической инструментации. Были проанализированы существующие фреймворки, подходящие для выполнения динамической двоичной инструментации. По результатам данного анализа был выбран DynamoRIO, решающий главную проблему динамической инструментации, а именно большие накладные расходы. Был описан принцип работы с фреймворком, его особенности и преимущества. Также был рассмотрен Clang AST, необходимый нам для анализа исходного кода профилируемого приложения. Были перечислены стратегии размещения памяти, и протестированы основные аллокаторы, представляющие альтернативу стандартному.

Помимо прочего в работе был подробно описан сам процесс разработки профилировщика. Все поставленные в начале работы задачи были выполнены.

Программное обеспечение становится все сложнее, а значит усложняется и процесс его отладки. Поэтому развитие программных анализаторов является актуальной задачей. Разработанный в данной работе профилировщик может стать хорошей основой для написания более сложных и совершенных анализирующих программ.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Nethercote N. Dynamic binary analysis and instrumentation = Динамический бинарный анализ и инструментация / N. Nethercote // University of Cambridge, Computer Laboratory, 2004. – №. UCAM-CL-TR-606. – 177 с.

2. Дерюгин Д. Е. Профилирование операционных систем реального времени / Д. Е. Дерюгин. – СПб., 2014. – 15 с.

3. Ермаков М.К. Применение статической бинарной инструментации с целью проведения динамического анализа программ для платформы ARM / М.К. Ермаков, С.П. Вартанов // Труды ИСП РАН, том 27, вып. 1. – 2015 г. – с. 5-24.

4. Uh G. R. Analyzing dynamic binary instrumentation overhead = Анализ накладных расходов динамической двоичной инструментации / Gang-Ryung Uh, Robert Cohn, Bharadwaj Yadavalli, Ramesh Peri, Ravi Ayyagari // WBIA workshop at ASPLOS. – 2006. – 8 с.

5. Intel. Pin 3.23 User Guide = Pin 3.23 Руководство Пользователя / https://software.intel.com/sites/landingpage/pintool/docs/98579/Pin/doc/html/index.html

6. Reddi V. J. Pin: a binary instrumentation tool for computer architecture research and education = Pin: инструмент двоичной инструментации для исследований и обучения архитектуре компьютера / V. J. Reddi, A. Settle [и др.] // Proceedings of the 2004 workshop on Computer architecture education: held in conjunction with the 31st International Symposium on Computer Architecture. – 2004. – 8 с.

7. Dyninst Programmer’s Guide = Dyninst Руководство программиста / https://github.com/dyninst/dyninst/blob/v10.1.0/dyninstAPI/doc/dyninstAPI.pdf

8. Bryan B. An API for Runtime Code Patching = API для исправления кода во время выполнения / B. Bryan, J. K. Hollingsworth // Computer Science Department, University of Maryland, College Park, MD 20742 USA. – 12 с.

9. Bernat A. R. Anywhere, any-time binary instrumentation = Двоичная инструментация в любом месте и в любое время / A. R. Bernat, B. P. Miller // Proceedings of the 10th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools. – 2011. – с. 9-16.

10. DynamoRIO documentation = DynamoRIO документация. – https://dynamorio.org/

11. Bruening D. An Infrastructure for Adaptive Dynamic Optimization = Инфраструктура для адаптивной динамической оптимизации / D. Bruening, T. Garnett, S. Amarasinghe // Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA 02139. – 11

12. Clang 15.0.0git documentation = Clang 15.0.0git документация. – https://clang.llvm.org/docs/index.html

13. AST Matcher Reference = Справочник по AST Matcher. – https://clang.llvm.org/docs/LibASTMatchersReference.html

14. Habr. Альтернативные аллокаторы памяти. – https://habr.com/ru/post/274827/

15. Habr. Аллокаторы памяти. – https://habr.com/ru/post/505632/

16. Habr. Inside The JeMalloc. Базовые Структуры Данных: Pairing Heap & Bitmap Tree. – https://habr.com/ru/post/470674/

17. Microsoft. Mi-malloc Documentation = Mi-malloc документация. – https://microsoft.github.io/mimalloc/

18. Алексеева О. Г. Выполнение дополнительного раздела ВКР бакалаврами технических направлений: Учебно-методическое пособие / О.Г. Алексеева // СПб.: Изд-во СПбГЭТУ “ЛЭТИ”, 2018 – 15 с.

ПРИЛОЖЕНИЕ А

Исходный код клиента DynamoRIO

```
#include "dr_api.h"
#include "drmgr.h"
#include "drreg.h"
#include "drutil.h"
#include "drx.h"
#include "drwrap.h"
#include "drsyms.h"

file_t file;
FILE* log_file;
int num_ref;

//Функции открытия и закрытия лог-файла
file_t
log_file_open()
{
    file_t file = dr_open_file("trace.txt", DR_FILE_CLOSE_ON_FORK |
                                             DR_FILE_ALLOW_LARGE |
                                             DR_FILE_WRITE_OVERWRITE);
    return file;
}

void
log_file_close(file_t file)
{
    dr_close_file(file);
}

FILE *
log_stream_from_file(file_t file)
{
    return fdopen(file, "w");
}

void
log_stream_close(FILE *file)
{
    fclose(file);
}

//Пре-функция обертки целевой функции
static void
wrap_pre(void *wrapcxt, OUT void **user_data)
{
    /*Получаем первый аргумент malloc,
      который хранит в себе количество выделенной памяти
      и добавляем его в разделяемую между оборачивающими
      функциями область памяти*/
    size_t sz = (size_t)drwrap_get_arg(wrapcxt, 0);
    *user_data = (void *)sz;
}
//Пост-функция обертки целевой функции
static void
wrap_post(void *wrapcxt, void *user_data)
{
    module_data_t *mod;
    //Получаем адрес, где вызывается malloc и откуда он вызывается
```

```
    app_pc ret_addr = drwrap_get_retaddr(wrapcxt);
    app_pc ret_val = (app_pc)drwrap_get_retval(wrapcxt);
    //Получаем указатель на модуль по адресу
    mod = dr_lookup_module(ret_addr);
    //Объявляем структуру, хранящую информацию о символах
    //и заполняем ее базовыми значениями
    drsym_error_t symres;
    drsym_info_t sym;
    char name[256];
    char file[256];
    sym.struct_size = sizeof(sym);
    sym.name = name;
    sym.name_size = 256;
    sym.file = file;
    sym.file_size = 256;
    //Получаем структуру с информацией о символах для malloc
    symres = drsym_lookup_address(mod->full_path, ret_addr - mod->start,
                                  &sym, DRSYM_DEFAULT_FLAGS);
    //Выводим всю необходимую информацию в лог-файл
    fprintf(log_file, "%s %s %d %d %d %d\n", sym.file, sym.name,
            sym.line, ret_val, (size_t)user_data, ret_addr);
    dr_free_module_data(mod);
}

//Функция обратного вызова для события загрузки модуля
static void
module_load_event(void *drcontext, const module_data_t *mod, bool loaded)
{
    app_pc towrap = (app_pc)dr_get_proc_address(mod->handle, "malloc");
    if(towrap != NULL){
            drwrap_wrap(towrap, wrap_pre, wrap_post);
    }
}

//Функция обратного вызова для события завершения клиента
static void
event_exit(void)
{
    //Отмена регистраций событий
    if (!drmgr_unregister_module_load_event(module_load_event))
        DR_ASSERT(false);
    log_stream_close(log_file);
    //Завершение расширений
    drsym_exit();
    drwrap_exit();
    drmgr_exit();
}
//main-функция клиента
DR_EXPORT void
dr_client_main(client_id_t id, int argc, const char *argv[])
{
    file = log_file_open();
    log_file = log_stream_from_file(file);
    num_ref = 0;
    //Инициализируем расширения
    if (!drmgr_init() || !drwrap_init() ||
        drsym_init(0) != DRSYM_SUCCESS)
        DR_ASSERT(false);
    //Регистрируем события
    dr_register_exit_event(event_exit);
    if (!drmgr_register_module_load_event(module_load_event))
        DR_ASSERT(false);
}
```

ПРИЛОЖЕНИЕ Б

Матчер Clang AST

```
MallocFinderASTConsumer::MallocFinderASTConsumer(Rewriter &R)
    : MallocFinderHandler(R)
    {
        const auto mallocDeclaration = callExpr(
                callee(functionDecl(hasName("malloc"))),
                hasAncestor(varDecl().bind("declaration"))
        ).bind("functionCallDeclaration");

        Finder.addMatcher(mallocDeclaration, &MallocFinderHandler);

        const auto mallocAssignment = callExpr(
                callee(functionDecl(hasName("malloc"))),
                hasAncestor(
                        binaryOperator(hasOperatorName("="))
                        .bind("assignment")
                )
        ).bind("functionCallAssigment");

        Finder.addMatcher(mallocAssignment, &MallocFinderHandler);

}
```