МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «ОДЕСЬКА ЮРИДИЧНА АКАДЕМІЯ»

ФАКУЛЬТЕТ КІБЕРБЕЗПЕКИ ТА ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

Кафедра інформаційних технологій

КВАЛІФІКАЦІЙНА РОБОТА

на тему:

«ПРОГРАМНА РЕАЛІЗАЦІЯ ПРОЦЕСУ ОПТИМІЗАЦІЇ ЧАСУ ВИКОНАННЯ ПРОГРАМ НА МОВІ С++»

Рівень «бакалавр»

Галузь знань 12 «Інформаційні технології»,

спеціальність 122 «Комп'ютерні науки»

Виконав здобувач 4 курсу

Сидоренко Олег Ігорович

Керівник к.т.н., доцент

Чикунов Павло Олександрович

Рецензент к.т.н., доцент

Манаков Сергій Юрійович

Одеса – 2024

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «ОДЕСЬКА ЮРИДИЧНА АКАДЕМІЯ»

Факультет кібербезпеки та інформаційних технологій

Кафедра інформаційних технологій

Галузь знань 12 Інформаційні технології

Спеціальність 122 «Комп'ютерні науки»

Назва освітньої програми «Комп’ютерні науки»

ЗАТВЕРДЖУЮ

Завідувач кафедри інформаційних технологій

доцент Наталія Логінова _____

«____ »_____ 2024 року

З А В Д А Н Н Я

на кваліфікаційну (бакалаврську) роботу

здобувачу Сидоренко Олегу Ігоровичу

1. Тема роботи: «Програмна реалізація процесу оптимізації часу виконання програм на мові С++».

   Керівник роботи: к.т.н, доцент Чикунов Павло Олександрович
   затверджені наказом НУ «ОЮА» № 809-06 від «02» квітня 2024 року
2. Дата видачі завдання «15» вересня 2023 року.
3. Строк подання здобувачем роботи «20» травня 2024 року.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів бакалаврської роботи	Строк виконання етапів роботи	Примітка
1	Аналіз предметної області та наявних аналогів	10.10.2023	
2	Формулювання цілей та завдань дослідження	05.11.2023	
3	Проєктування оптимізаційного компілятора	07.12.2023	
4	Програмна реалізація та тестування оптимізаційного компілятора	10.02.2024	
5	Оформлення звітної частини	19.05.2024	

Здобувач _____ Олег Сидоренко
(підпис) (ім’я та прізвище)

Керівник роботи _____ Павло Чикунов
(підпис) (ім’я та прізвище)

АНОТАЦІЯ

Сидоренко О. І. Програмна реалізація процесу оптимізації часу виконання програм на мові C++. – Кваліфікаційна робота бакалавра за спеціальністю 122 «Комп'ютерні науки», освітньою програмою «Комп’ютерні науки».

Кваліфікаційна робота викладена на 48 сторінках основного тексту і 54 сторінках загального тексту, містить 3 розділи, 15 рисунків, 1 таблицю, 2 додатки, 26 використаних джерела.

Метою дослідження є проєктування, програмна реалізація та тестування оптимізаційного компілятора для забезпечення додаткової оптимізації часу виконання програм на мові C++ засобами регульованого агресивного вбудовування функцій.

Об’єктом дослідження є процес зниження затримок в навантажених секціях коду C++.

Предметом дослідження є техніки оптимізації часу виконання програм на мовах високого рівня, зокрема засобами регульованого агресивного вбудовування функції.

У кваліфікаційній роботі виконана програмна реалізація оптимізаційного компілятора, який дозволяє оптимізувати середній час виконання програм на мові програмування C++ за допомогою вбудовування функцій з широким спектром параметрів для регулювання рівня агресивності. Програмна реалізація позиціонується не як незалежна оптимізація, що блокує застосування інші критерії, а як додаткова оптимізація, яка може проводитися поряд з іншими.

Ключові слова: оптимізація часу виконання, компіляція, мова C++, вбудовування функцій, Low Level Virtual Machine, компілятор Clang, генератор CMake.

ABSTRACT

Sydorenko O. I. Software implementation of execution time optimization process for programs in C++. – Bachelor's work in the specialty 122 "Computer Science", educational program "Computer Science".

The thesis consists of 48 pages of main text and 54 pages of total text, including 3 chapters, 15 figures, 1 table, 2 appendices, and references to 26 sources.

The purpose of the research is the design, software implementation and testing of an optimizing compiler to provide additional optimization of the execution time of programs in the C++ language by means of adjustable aggressive embedding of functions.

The object of research is the process of reducing delays in loaded sections of C++ code.

The subject of the research is techniques for optimizing the execution time of programs in high-level languages, in particular, by means of adjustable aggressive function embedding.

The work includes the implementation of an optimizing compiler that allows optimizing the average execution time of programs in the C++ programming language by embedding functions with a wide range of parameters to adjust the level of aggressiveness. The software implementation is positioned not as an independent optimization that blocks the application of other criteria but as additional optimization that can be carried out alongside others.

Keywords: execution time optimization, compilation, C++, function inlining, Low Level Virtual Machine, Clang, CMake generator.

ЗМІСТ

ВСТУП

У світі є безліч програмного забезпечення, написаного засобами C++. Вважається, що ця мова програмування є дуже швидкою. Тим не менш, є сфери, що вимагають від програми високої продуктивності, яка недосяжна лише вдалим вибором мови.

Написати код із мінімальною асимптотичною складністю – не означає написати оптимальний з погляду затримки код: нерідко вирішальним фактором є компілятор, а саме його оптимізації, що виробляються над програмою.

Актуальність дослідження обумовлена тим фактом, що при розробці високопродуктивного коду, є потреба у застосуванні додаткової оптимізації вихідного коду. З цим завданням є добре справляється компілятор, використовуючи оптимізаційні прапори компіляції, при чому часто досягається результат, що задовольняє користувача, проте далеко не завжди цього достатньо.

Відомо, що компілятор розглядає вбудовані параметри розширення та ключові слова як пропозиції. Однією з найважливіших оптимізації компілятора є вбудовування функцій – заміна виклику функції безпосередньо тілом функції.

У межах дослідження середній час роботи програми мовою C++ розглядається як затримка.

Low Level Virtual Machine (LLVM) – це універсальна система аналізу, трансформації і оптимізації програм, що реалізує віртуальну машину з RISC-подібними інструкціями для оптимізації компілятор байт-коду в машинний код програмно-апаратної платформи. Прогнозується, що використання LLVM має скоротити затримку у високонавантажених секціях коду C++.

Метою дослідження є проєктування, програмна реалізація та тестування оптимізаційного компілятора для забезпечення додаткової оптимізації часу виконання програм на мові C++ засобами регульованого агресивного вбудовування функцій.

Для досягнення мети сформульовано такі завдання:

– провести огляд предметної галузі та наявних підходів до оптимізації;

– обрати засоби та інструменти для програмної реалізації оптимізаційного компілятора;

– виконати програмну реалізації та тестування оптимізаційного компілятора на наявних проєктах;

– оцінити середній час виконання отриманих проєктів у порівнянні з базовою компіляцією в однакових умовах компіляції.

Об’єктом дослідження є процес зниження затримок в навантажених секціях коду C++.

Предметом дослідження є техніки оптимізації часу виконання програм на мовах високого рівня, зокрема засобами регульованого агресивного вбудовування функції.

Методи досліджень: методи системного аналізу для дослідження предметної області, методи математичного моделювання об’єктів пошуку, методи програмної реалізації алгоритмів оптимізації коду програм, методи управління процесами життєвого циклу програмної системи, методи тестування програмного продукту.

Практичним результатом дослідження є програмна реалізація компілятора, який дозволяє оптимізувати середній час виконання програм мовою програмування C++ за допомогою вбудовування функцій із широким спектром параметрів для регулювання рівня агресивності. Програмна реалізація позиціонується не як незалежна оптимізація, що блокує застосування інші критерії, а як додаткова оптимізація, яка може проводитися поряд з іншими.

Публікації та апробація кваліфікаційної роботи: публікація тези доповіді на тему «Розробка оптимізаційного компілятора для мови C++» у IX Всеукраїнській науково-практичній конференції «Інформаційне суспільство: проблеми та перспективи» (Одеса, 24 травня 2024 р.).

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ВИБІР ЗАСОБІВ РОЗРОБКИ

1.1 Основні техніки оптимізації програм

Основним завданням програмування є створення правильних, а не ефективних програм. Ефективна програма не потрібна, якщо вона не забезпечує правильних результатів. Правильну, але не ефективну програму можна оптимізувати та зробити ефективною.

Для великих програм на стадії проєктування визначаються необхідні параметри, що включають виконання і ємність пам'яті. Оптимізація програми – це процес побудови за вихідною програмою еквівалентної програми, що має кращі характеристики часу роботи та/або обсягу займаної оперативної пам’яті.

Основні типи програм:

1. програми, що часто використовуються (ОС, компілятори, постпроцесори); вони повинні бути ефективними;

2. програми промислового призначення; їх ефективність повинна бути суттєвою;

3. програми, написані не програмістами; їх ефективність потрібна, якщо є обмеження пам'яті або часу.

Програму роблять ефективнішою лише у випадках:

– програма не міститься у пам'яті;

– програма надто довго виконується;

– програма має бути включена до бібліотеки та часто використовуватись.

Для виконання оптимізації необхідно, по-перше, використати оптимізаційний компілятор [12]. По-друге, необхідно визначити критичні сфери, що підлягають оптимізації та застосувати локальну оптимізацію в критичних областях (часті програми, що найчастіше використовуються).

Транслятор – це програма, призначена для перекладу (трансляції) опису алгоритмів з однієї формальної мови на іншу. Перша з цих мов називається

вхідною, друга – вихідною або об'єктною. Найбільш поширені транслятори з мов процедурно-орієнтованих у мови машинні та мови машинно-орієнтовані.

Найпопулярнішою схемою для традиційного статичного компілятора є трифазна конструкція, основними компонентами якої є інтерфейс, оптимізатор і back-end [14]. Інтерфейс аналізує вихідний код, перевіряє його на наявність помилок і створює специфічне для мови абстрактне синтаксичне дерево для представлення вхідного коду.

Рисунок 1.1 – Основні компоненти оптимізаційного компілятора

Оптимізатор відповідає за виконання широкого спектру перетворень, щоб спробувати покращити час виконання коду, наприклад, усунути зайві обчислення, і зазвичай більш-менш не залежить від мови та цілі. Потім серверна частина (також відома як генератор коду) відображає код у цільовому наборі інструкцій. Окрім створення правильного коду, він відповідає за створення хорошого коду, який використовує переваги незвичайних функцій підтримуваної архітектури. Загальні частини задньої частини компілятора включають вибір інструкцій, розподіл регістрів і планування інструкцій.

Ця модель однаково добре підходить для інтерпретаторів і JIT-компіляторів. Віртуальна машина Java (JVM) також є реалізацією цієї моделі, яка використовує байт-код Java як інтерфейс між інтерфейсом і оптимізатором.

Транслятор є одним із основних засобів автоматизації програмування.

Зазвичай транслятор складається з низки блоків, що виконують незалежні функції:

– синтаксичний аналіз програм;

– аналіз описів даних та перетворення їх у зручну для зберігання та подальшого використання форму;

– розподіл пам'яті для об'єктів, що використовуються в програмі;

– встановлення відповідності між конструкціями вхідної мови та еквівалентними нею конструкціями вихідної мови;

– друк у редагованому вигляді тексту вихідної програми;

– повідомлення про виявлені в процес трансляції помилки у вихідній програмі;

– еквівалентні перетворення на рівні вхідної мови з метою оптимізації програми.

Для однієї й тієї ж вхідної мови доцільно мати кілька трансляторів, одне із яких дозволяє здійснити швидку трансляцію, видаючи менш ефективні програми. Такий транслятор можна використовувати для налагодження програм. Компілятори, що створюють ефективну об'єктну програму, зазвичай бувають великими і працюють повільно.

Оптимізаційний транслятор, застосовуючи різні методи оптимізації може отримати якіснішу об'єктну програму, що використовується для багаторазових розрахунків.

Зазвичай більшість часу витрачається виконання дуже невеликий частини програми, але найчастіше використовуваної, званої, критичної областю.

Зазвичай, лише критична область програми оптимізується програмістом вручну. Для виявлення цих критичних областей використовують спеціальні засоби, названі профільниками.

Підпрограми спочатку з'явилися як засіб оптимізації програм за обсягом пам'яті, що займається – вони дозволили не повторювати в програмі ідентичні блоки коду, а описувати їх одноразово і викликати в міру необхідності. До теперішнього часу ця функція підпрограм стала допоміжною, головне їхнє призначення – структуризація програми з метою зручності її розуміння та супроводу.

Виділення набору дій у підпрограму та виклик її за необхідності дозволяє логічно виділити цілісну підзавдання, що має типове рішення. Така дія має ще одну (крім економії пам'яті) перевагу перед повторенням однотипних дій: будь-яка зміна (виправлення помилки, оптимізація, розширення функціональності),

зроблена в підпрограмі, автоматично відбивається на всіх її викликах, у той час як при дублюванні кожну зміну необхідно вносити в кожне входження коду, що змінюється.

Навіть у тих випадках, коли в підпрограму виділяється набір дій, що одноразово виробляється, це виправдано, тому що дозволяє скоротити розміри цілісних блоків коду, що становлять програму, тобто зробити програму більш зрозумілою і доступною для огляду.

Під оверлейністю розуміють можливість перенесення підпрограм під час роботи програми в швидкодіючу пам'ять з іншого типу пам'яті таким чином, що кілька підпрограм в різний час займають одну і ту ж область пам'яті. Можливість оверлейності реалізується при використанні віртуальної пам'яті.

Якщо машина має віртуальну пам'ять, то ОС автоматично ділить програму частини фіксованої довжини, звані сторінками. З іншого боку, ОС, у разі потреби, пересилає сторінки в ОП. Таким чином, проблема організації оверлейності перестає бути обов'язком програміста і покладається на машину.

1.2 Оптимізація компілятора засобами вбудовування функції

Процес вбудовування функції – це оптимізація компілятора, що спрямована на заміну виклику функції безпосередньо тілом функції [22]. Ця дія може покращити швидкість роботи програми, оскільки після вбудовування відсутні накладні витрати на виклик функції. Важливо зазначити, що порядок застосування оптимізації компілятором також має значення, тому оптимізації, застосовані після вбудовування функцій можуть мати кращий ефект.

Вбудовування функцій у мові програмування C++ дозволяє оптимізувати роботу компілятора, зменшуючи витрати на виклики функцій та покращуючи швидкодію програми. У середовищі Visual Studio це може бути досягнуто за допомогою ключового слова *_forceinline*, яке підказує компілятору вбудувати вміст функції безпосередньо в місце виклику, не створюючи окремого виклику функції. Це особливо корисно для невеликих функцій, які викликаються дуже

часто, таких як прості гетери та сетери або короткі функції обробки даних. Компілятор може ігнорувати цей підказ, якщо він вважає, що вбудовування функції не вплине на результат.

Однак варто враховувати, що вбудовування функцій може призвести до збільшення розміру коду, що може підвищити навантаження на кеші пам'ять, що може призвести до погіршення продуктивності. Також уповільнюється компіляція програми. Тому вбудовування функцій слід застосовувати обережно та з урахуванням специфіки конкретної програми.

Проблема полягає в тому, що навіть наявність знання про специфіку не дозволяє легко впливати на вбудовування функцій. Наприклад, є специфікатор *inline* [4] у реалізації LLVM, який додає деяке чисельне значення до вартості вбудовування, яке використовується для перевірки доцільності вбудовування. Тобто використання даного специфікатора не гарантує вбудовування, а лише є деякою підказкою компілятору. Також є атрибут noinline, що забороняє вбудовування функції (виключенням є вбудовування внаслідок згортки констант) [6]. Параметри та ключові слова підстановки компілятор опрацьовує як рекомендації. Немає жодної гарантії, що будь-яка функція буде розгорнута. Можна вимкнути вбудовані розширення, але ви не можете примусово ввімкнути певну функцію, навіть якщо використовується ключове слово _forceinline. Щоб виключити функції з розгляду як кандидатів на вбудоване розширення, можна використовувати _declspec(noinline) або регіон, позначений директивами #pragma auto_inline(off) та #pragma auto_inline(on).

Профільна оптимізація (Profile Guided Optimization, PGO) – це техніка оптимізації програмного коду, яка базується на аналізі профілю виконання програми [7]. Основна ідея полягає в тому, щоб зібрати інформацію про те, як саме програма виконується у різних сценаріях і використовувати ці дані для покращення швидкодії та ефективності програми. PGO дозволяє компілятору краще адаптувати оптимізації до конкретного використання програми, що може призвести до помітного покращення її продуктивності.

Завдяки цьому можна зробити більш точні висновки про ті чи інші оптимізації. Якщо повернутися до вбудовування функцій, PGO може допомогти компілятору, наприклад, у ситуації, коли одна функція викликається частіше за іншу. У такому випадку компілятор може з більшою впевненістю зробити висновок про вбудовування першої функції.

1.3 Огляд наявних рішень оптимізації часу виконання програм

В роботі [9] запропоновано нову техніку розбиття функції на основі ранніх повернень функції для досягнення обох цілей: знайдено новий підхід до опису функцій, який дозволяє розбивати їх на основі підграфу викликів. Також особливу увагу було приділено мінімізації дублювання коду. Запропоновано новий аналіз витрат та вигоди при оцінці вартості вбудовування, що дозволило гарантувати контроль за накладними витратами на виклик виділеної функції.

Часткове вбудовування функцій – це ефективний спосіб, який вбудовує лише частину викликаної функції, таким чином зменшуючи розширення коду. Ключова проблема полягає в тому, як ефективно розділити функцію, щоб можна було зменшити як накладні витрати на виклик, так і розширення коду. Попередні методи або призводять до розбиття функцій, занадто великих для вбудовування, або не в змозі ефективно зменшити накладні витрати виклику.

Спосіб використовує модель витрат і вигоди, щоб забезпечити ефективний розподіл функцій. Експериментальні результати показують, що запропонована техніка може зменшити затримку на 33%, при цьому розмір коду збільшиться лише на 5,8%. Автори реалізували часткове вбудовування поверх LLVM. У результаті дана робота справді вирішила завдання реалізації часткового вбудовування функцій з мінімальним дублюванням коду, але час роботи в половині випадків побільшав, а в іншій половині – збільшився до 3%.

У статті [10] описується проблема вибору евристик вбудовування функцій у процесі компіляції, яка може сильно впливати на продуктивність застосунку.

Автори пропонують метод автоматичного налаштування евристик вбудовування функцій на основі генетичних алгоритмів.

Були описані експерименти, в яких метод автоматичного налаштування евристик був застосований для вбудовування функцій набору тестових програм. Результати показали, що автоматичне налаштування евристик вбудовування функцій може значно покращити продуктивність застосунків порівняно з налаштуваннями, встановленими за умовчанням.

Метод автоматичного налаштування евристик вбудовування функцій може бути застосований у різних галузях, де потрібна оптимізація продуктивності коду.

У роботі [11] запропоновано алгоритм для вбудовування функцій у програми, які виконуються на мікроконтролерах з обмеженим об'ємом пам'яті. Запропоновано метод, який дозволяє визначити, які функції слід вбудовувати, щоб мінімізувати розмір коду, зберігаючи при цьому продуктивність програми. Для цього використовується алгоритм, який оцінює вплив вбудовування кожної функції на розмір коду та на час виконання програми. Також враховано залежність між функціями, щоб уникнути помилок під час виконання.

В результаті експериментів встановлено, що метод дозволяє досягти суттєвого зменшення розміру коду та покращення продуктивності програми на мікроконтролерах з обмеженим об'ємом пам'яті. Таким чином, цей метод може бути корисним для розробників, які працюють із вбудованими системами, де обмежений обсяг доступної пам'яті, і потрібне максимально ефективне використання ресурсів.

Переглянуті дослідження на тему оптимізації шляхом вбудовування функцій не ставили перед собою завдання оптимізувати час роботи програми, а більше були зосереджені на зменшенні розміру коду після вбудовування функцій. Але встановлено, що мета оптимізації затримки коду може бути вельми корисною для будь-якої програмної системі, яка додає можливість провести експеримент для успішнішої компіляції програми.

1.4 Обґрунтування вибору засобів розробки

LLVM – це програмна система, що є набором інструментів для розробки компіляторів, асемблерів та інших систем, пов'язаних з обробкою програмного коду [3]. LLVM надає універсальний, низькорівневий та модульний інтерфейс для компіляції програм, який може бути використаний для розробки компіляторів для різних мов програмування.

Процес компіляції в LLVM включає кілька етапів. Під час першого етапу, вихідний код програми перетворюється на проміжне уявлення (intermediate representation) LLVM IR за допомогою компілятора Clang [8] за допомогою виклику програми clang++ або іншого сумісного компілятора. Потім IR передається оптимізатор LLVM, який застосовує різні оптимізації, такі як вбудовування функцій, видалення недосяжного коду та інші, для поліпшення продуктивності та якості згенерованого коду.

У компіляторі на основі LLVM інтерфейс відповідає за розбір, перевірку та діагностику помилок у вхідному коді, а потім переклад проаналізованого коду в LLVM IR [14]. Цей IR необов’язково проходить через серію проходів аналізу та оптимізації, які покращують код, а потім надсилається в генератор коду для створення рідного машинного коду. Це дуже проста реалізація трифазної конструкції, але цей простий опис приховує частину потужності та гнучкості, які архітектура LLVM отримує від LLVM IR.

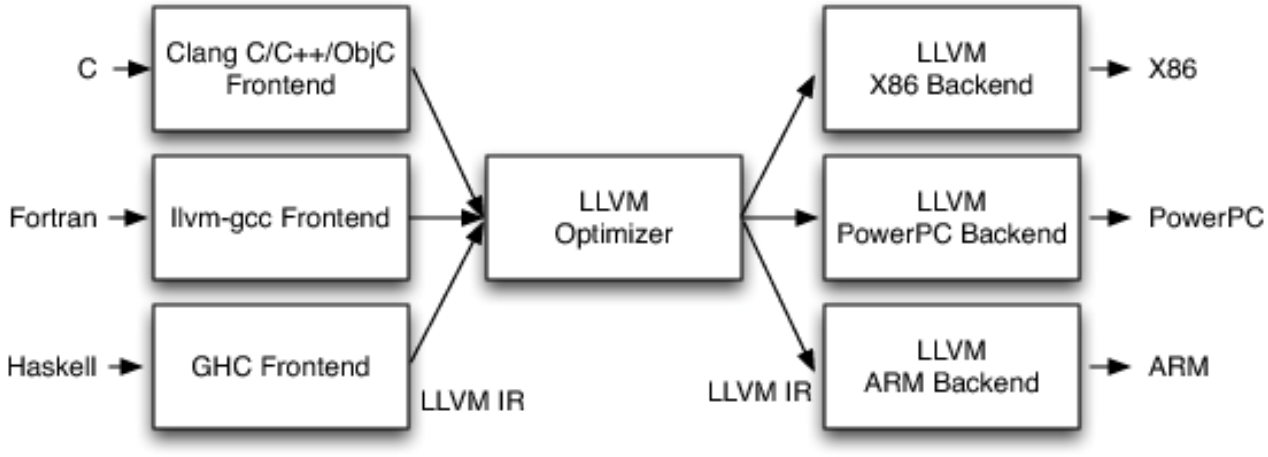

Рисунок 1.2 – Реалізація трифазної конструкції LLVM

Починаючи з версії 2019, MS Visual Studio включає підтримку редагування, створення та налагодження за допомогою Clang/LLVM у проєктах CMake, націлених на Windows. Після налаштування конфігурації Clang, розробник може створювати та налагодити проєкт. Середовище Visual Studio самостійно виявляє компілятор Clang, і надає можливості IntelliSense для підсвічування, навігації та інших функції редагування [8].

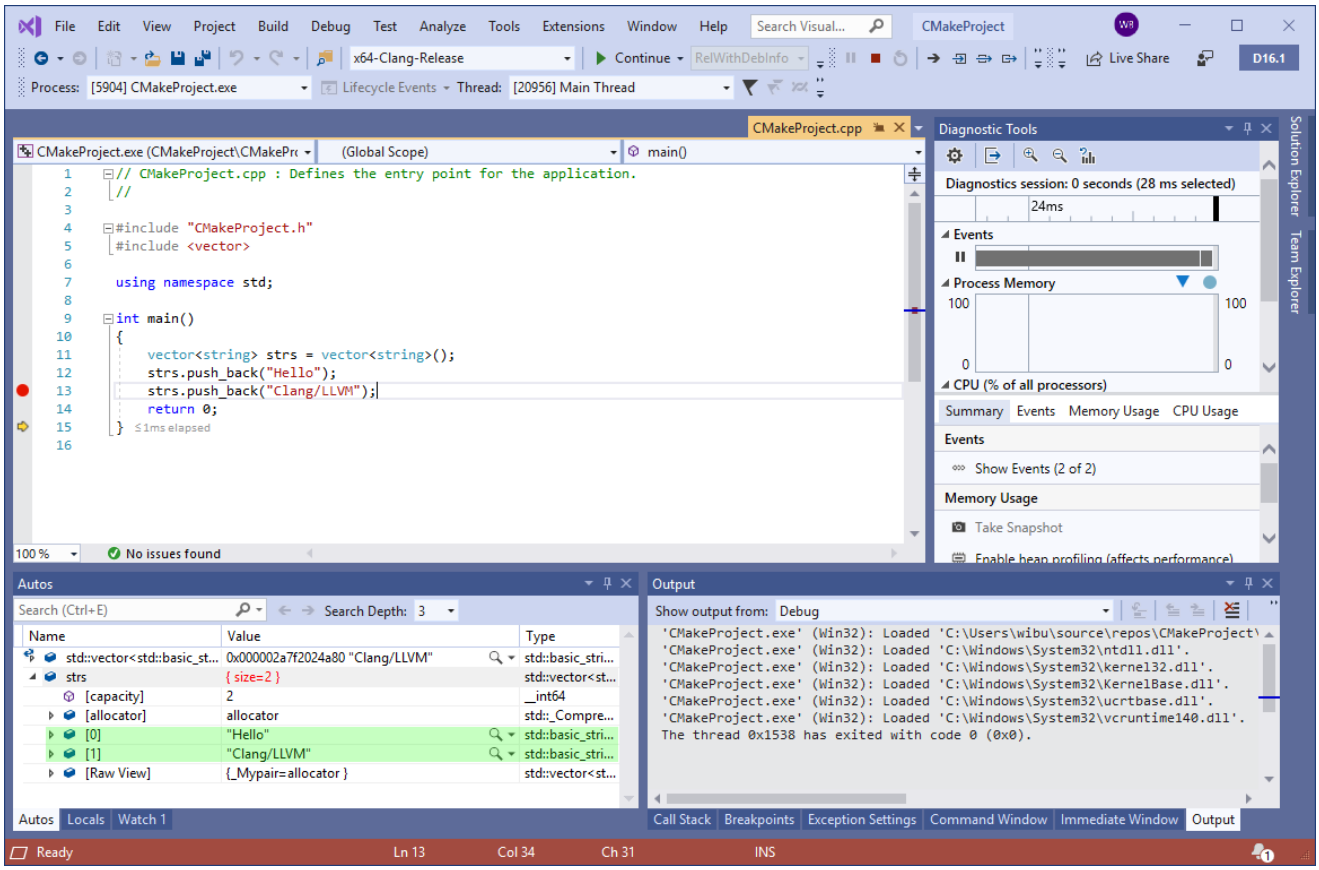

Рисунок 1.3 – Робота з компілятором Clang у середовищі Visual Studio

Далі оптимізоване проміжне уявлення IR перетворюється на машинний код цільової архітектури, використовуючи генератор коду LLVM. При генерації машинного коду LLVM використовує безліч оптимізації та техніки генерації коду для створення швидкого та ефективного виконуваного файлу.

У результаті LLVM забезпечує високу ефективність та гнучкість у компіляції програмного коду, що робить його популярним інструментом для розробки компіляторів, асемблерів та інших систем, пов'язаних з обробкою програмного коду.

У рамках дослідження необхідно модифікувати процес компіляції, що робить вибір LLVM виправданим.

Кросплатформовий відкритий генератор сценаріїв складання CMake є стандартом де-факто для створення коду на мові C++. Це потужне комплексне рішення для керування процесом створення програмного забезпечення. CMake дозволяє розробникам описувати, як створювати прості та дуже складні програмні системи за допомогою єдиного набору вхідних файлів. Систему можна використовувати для створення програмного забезпечення на багатьох платформах, від Android до iOS і високопродуктивних обчислювальних систем.

Кожне дерево збірки CMake містить файл кешу, який містить змінні, встановлені як частина цієї збірки (рис. 1.4). Сюди входить усе, знайдене під час самоаналізу системи, шляхи до встановленого програмного забезпечення та прапорці, які використовуються для збірки. Cmake містить кілька GUI, які дозволяють редагувати цей кеш-файл.

CMake підтримує кілька цільових систем збірки, включаючи Visual Studio, Xcode, ninja, make та VSCode, а також інструменти збірки командного рядка, тому розробники можуть вибрати інструмент, який буде для них найбільш продуктивним.

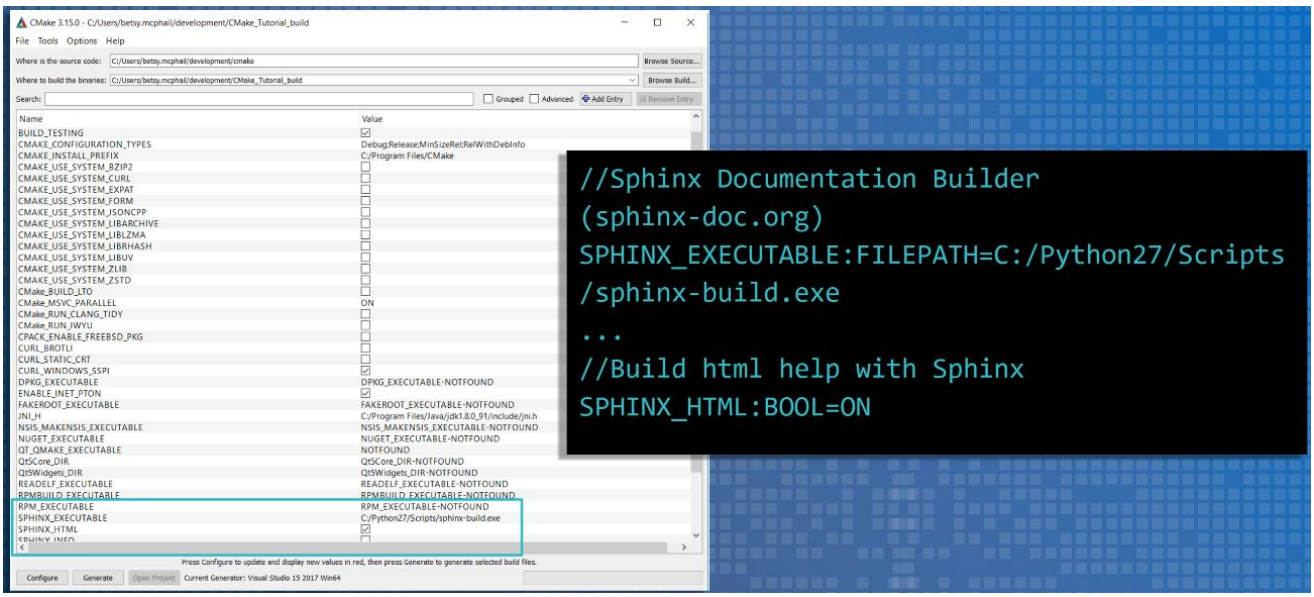

Рисунок 1.4 – Робота з кеш-файлом збірки у CMake

1.5 Висновки за розділом

В результаті аналізу предметної галузі:

– розглянуто основні техніки оптимізації програм;

– встановлено основні компоненти оптимізаційного компілятора;

– виконано огляд наявних рішень оптимізації часу виконання програм, зокрема процесу вбудовування функції;

– для проєктування оптимізаційного компілятора вибрано програмну систему LLVM, компілятор Clang для середовища Visual Studio та відкритий генератор сценаріїв складання CMake.

2 ПРОЄКТУВАННЯ ОПТИМІЗАЦІЙНОГО КОМПІЛЯТОРА

2.1 Особливості агресивності вбудовування функцій

Програмна реалізація оптимізаційного компілятора буде базуватися на мовах C++ і Python, наборі компіляторів LLVM, генераторі сценаріїв складання CMake.

В рамках даної роботи необхідно реалізувати агресивне вбудовування функцій, що регулюється. Тобто з одного боку вбудовування функцій повинно призводити до збільшення розміру коду, що виконується, з іншого боку хочеться надати користувачеві можливість регулювати цей процес.

Хочеться виділити два параметри, які мають регулювати процес вбудовування. Перший параметр – агресивне вбудовування використовується лише для функцій, які користувач помітив спеціальним новим атрибутом *aggressiveInline*. Це дозволить користувачеві проводити безліч експериментів у пошуках кращої для специфічного випадку оптимізації. Другий – обмеження розмір файлу, який виходить після застосування оптимізації. Це дозволить користувачеві не дозволяти оптимізації вбудовувати надто багато коду, адже це може бути недозволено в деяких випадках.

Назвемо оптимізаційним компілятором програмну систему, що дозволяє запускати оптимізацію LLVM таким чином, щоб на помічених спеціальним атрибутом *aggressiveInline* функціях спрацьовувало вбудовування функцій.

Для контролю агресивності вбудовування використовуються фіксовані параметри.

Назвемо оптимізатором програму, яка дозволяє знаходити в певному просторі пошуку параметри локального оптимізатора, які збільшують розмір файлу в байтах, після застосування локального оптимізатора. У оптимізаторі також потрібно передбачити обмеження зверху на розмір вихідного файлу.

Назвемо модифікатором програму, що дозволяє модифікувати команди компіляції проєкту із системою складання CMake у ланцюжок команд, серед яких з'являється звернення до оптимізатору.

Назвемо виконавцем програму, яка дозволяє запускати команди компіляції. Реалізація кожної компоненти надає рішення першої задачі, сформульованої у постановці задачі: для запуску оптимізації необхідно скористатися модифікатором і виконавцем.

2.2 Особливості процесу оптимізації LLVM

Є багато різних видів оптимізації компілятора, тому важко дати алгоритм, як вирішити будь-яку проблему оптимізації. Проте більшість оптимізацій мають просту структуру з трьох частин:

– зразок коду, який потрібно трансформувати;

– перевірка, чи перетворення безпечне/правильне для відповідного зразку коду;

– перетворення та оновлення коду.

Найбільш тривіальною оптимізацією є зіставлення шаблонів на арифметичних тотожностях: для будь-якого цілого числа X дорівнює X-0, X-X, (X*2)-X. Перше питання полягає в тому, як вони виглядають у LLVM IR. Деякі приклади:

```
⋮⋮⋮
%example1 = sub i32 %a, %a
⋮⋮⋮
%example2 = sub i32 %b, 0
⋮⋮⋮
%tmp = mul i32 %c, 2
%example3 = sub i32 %tmp, %c
⋮⋮⋮
```

Для такого роду перетворень LLVM надає інтерфейс спрощення інструкцій, який використовується як утиліти різними іншими перетвореннями

вищого рівня. Ці конкретні перетворення знаходяться у SimplifySubInst функції та виглядають так:

```
// X - 0 -> X
if (match(Op1, m_Zero()))
return Op0;
// X - X -> 0
if (Op0 == Op1)
return Constant::getNullValue(Op0->getType());
// (X*2) - X -> X
if (match(Op0, m_Mul(m_Specific(Op1), m_ConstantInt<2>())))
return Op1;
…
// Якщо нічого не збігається, поверніть null,
// щоб вказати відсутність перетворення
return 0;
```

У цьому коді *Op0* та *Op1* прив'язані до лівого та правого операндів цілочисельної інструкції віднімання. LLVM реалізовано в C++, який не надто відомий своїми можливостями зіставлення шаблонів (порівняно з функціональними мовами, такими як Objective Caml), але він пропонує дуже загальну систему шаблонів, яка дозволяє реалізувати щось подібне. Функція *match* та *m_функції* дозволяють виконувати декларативні операції зіставлення шаблону з кодом LLVM IR. Наприклад, у *m_Specific* предикат збігається, лише якщо ліва частина множення збігається з параметром *Op1*.

Разом ці три випадки відповідають шаблону і функція повертає заміну, якщо вона може, або нульовий покажчик, якщо заміна неможлива. Виклик функції *SimplifyInstruction* є диспетчером, який перемикає код операції інструкції, надсилаючи допоміжні функції кожного коду операції. Він викликається з різних оптимізацій. Простий драйвер виглядає так:

```
for (BasicBlock::iterator I = BB->begin(), E = BB->end(); I != E; ++I)
if (var* V = SimplifyInstruction(I))
I->replaceAllUsesWith(V);
```

Цей код просто перебирає кожну інструкцію в блоці, перевіряючи, чи спрощується якась із них. Якщо так (оскільки *SimplifyInstruction* повертає not-null), він використовує метод *replaceAllUsesWith* для оновлення будь-чого в коді за допомогою спрощеної операції з простішою формою. На рис. 2.1 показано приклад передачі управління при виконанні циклів.

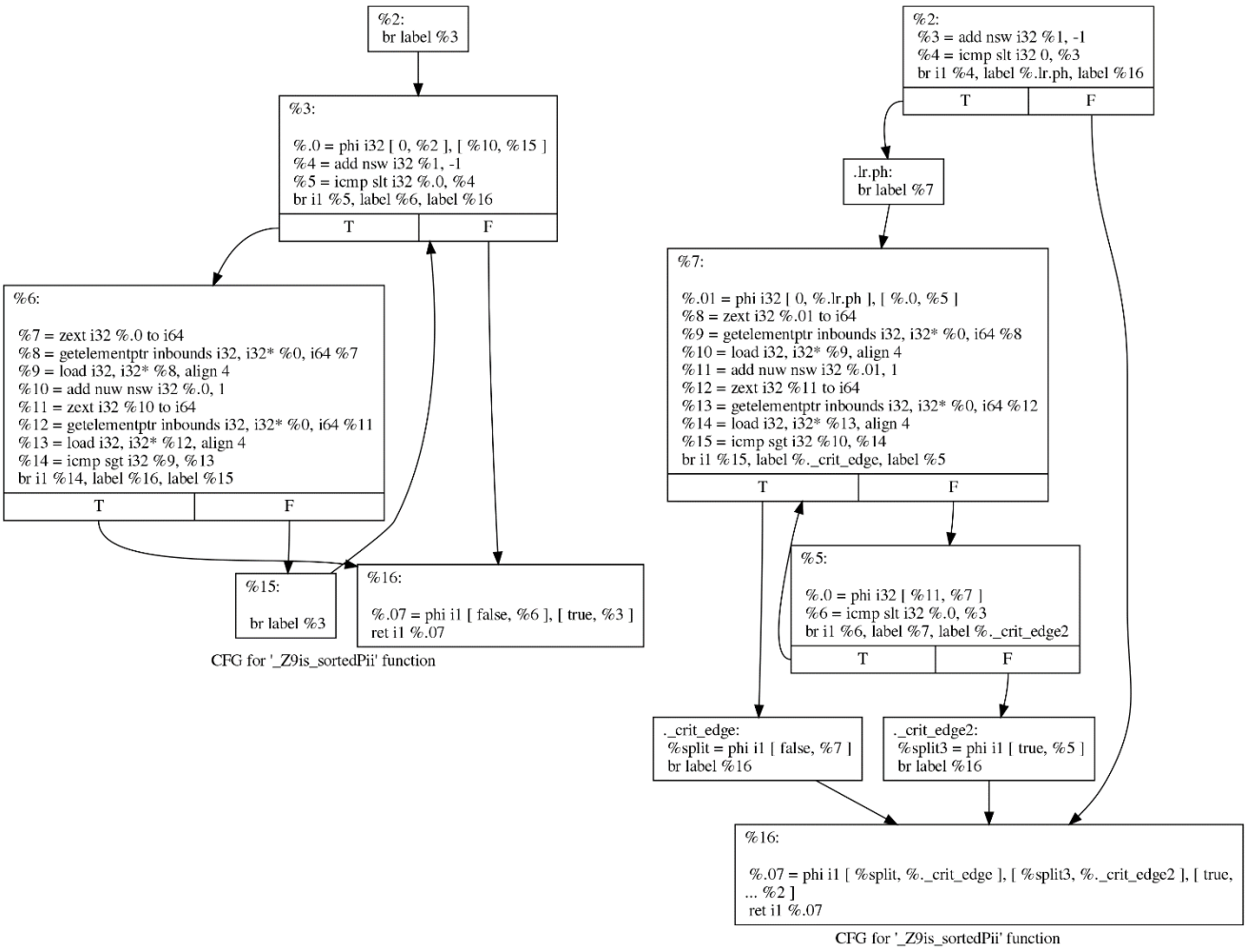

Рисунок 2.1 – Графік передачі управління до і після проходу ротації циклів

2.3 Проєктування локального оптимізатору для вбудовування функцій

Написання швидкого коду вимагає розуміння всіх аспектів програми та її взаємодії із системою. Щоб зібрати відомості про продуктивність коду, можна використовувати монітор продуктивності [24]. Промахи кешу (як внутрішнього, так і зовнішнього) та помилки сторінок (при зверненні до додаткового сховища за програмними інструкціями та даними) знижують продуктивність програми.

Потрапляння до кешу ЦП дозволяє програмі виграти 10–20 тактів. Потрапляння у зовнішньому кеші дозволяє виграти 20-40 тактів. Помилка сторінки може коштувати мільйон циклів годин (якщо процесор, що обробляє 500 мільйонів інструкцій/секунду та час 2 мілісекунди для збою сторінки). Тому

для оптимального виконання програми необхідно написати код, який знизить кількість промахів кешу та помилок сторінок.

Однією з причин повільної роботи програм є занадто часті помилки сторінок та промахи кешу. Щоб уникнути цієї проблеми, важливо використовувати структури даних із гарною локалізацією посилань, що означає збереження пов'язаних речей разом. Іноді зовні чудова структура даних виявляється жахливою через неправильне розташування посилань, і навпаки.

Бібліотека Microsoft Foundation Classes (MFC) може значно спростити написання коду. При написанні термінового коду слід знати про накладні витрати, пов'язані з деякими класами. Перевірте код MFC, який використовується терміновим кодом, щоб визначити, чи він відповідає вимогам до продуктивності.

Повторне використання коду бажане. Однак, якщо розробник має намір використовувати код іншого користувача, необхідно повинні точно знати, що це робить у тих випадках, коли продуктивність критично важлива. Найкращий спосіб зрозуміти його полягає у покроковому переході за вихідним кодом або вимірюванням за допомогою таких засобів, як PView або Монітор продуктивності.

Локальний оптимізатор повинен надати програмну систему, яка дозволяє запускати вбудовування функцій на функціях, помічених атрибутом aggressiveInline. Агресивність вбудовування має налаштовуватися зовні за допомогою аргументів командного рядка, оскільки далі потрібно оптимізувати агресивність вбудовування.

Пропонується декомпозувати рішення на такі завдання:

– додавання атрибуту *aggressiveInline*;

– додавання компонента, що відповідає за вбудовування функцій;

– додавання параметрів команди *opt*, які відповідають за агресивність вбудовування функцій.

Необхідно з’ясувати, які частини компілятора можуть використовувати атрибут. Зрозуміло, що використання користувачами атрибуту спричиняє необхідність обробляти вихідний код програми, таким чином необхідно

модифікувати перший етап компіляції. Атрибути функцій можуть впливати і оптимізації (другий етап компіляції), і вибір інструкцій під час перетворення на машинний код (третій етап компіляції). Тобто для додавання атрибуту необхідно модифікувати всі три частини компіляції.

Було проведено модифікацію кількох файлів компілятора Clang та програмної системи LLVM:

– файли clang/include/clang/Basic/Attr.td та clang/include/clang/Basic/ AttrDocs.td, що відповідають за визначення, опис та документування всіх атрибутів Clang. Вони необхідно було додати стиль написання атрибута: в C++ дозволяється використовувати різні стилі, наприклад, attribute (noinline) чи [noinline]. У разі використовується перший варіант. Також потрібно уточнити сутність, яку можна пов'язати з атрибутом;

– clang/lib/CodeGen/CodeGenModule.cpp та clang/lib/Sema/SemaDeclAttr.cpp; перший файл відповідає за генерацію коду, другий – за обробку атрибутів та перевірку застосовності атрибутів до відповідних декларацій;

– файли LLVMBitCodes. BitcodeReader.cpp та BitcodeWriter.cpp були змінені для додавання нового атрибуту aggressiveInline в обробку біткоду; який використовується при роботі з бінарним представленням файлів LLVM;

– файл Attributes.td відповідає за визначення атрибутів всередині LLVM IR;

– файли LLLexer.cpp, LLParser.cpp та LLToken.h були змінені для підтримки нового *aggressiveInline* атрибута в парсері LLVM IR;

– файл Attributes.cpp відповідає за роботу з атрибутами всередині системи LLVM;

– файл Verifier.cpp займається перевіркою коректності та узгодженості LLVM IR.

Таким чином, було проведено комплексну модифікацію компілятора Clang та системи LLVM, що робить використання атрибуту *aggressiveInline* можливим.

LLVM Pass – це одиниця обробки програмного коду у фреймворку LLVM, тобто невеликі програмні модулі, які виконують різноманітні операції над проміжним представленням програми у форматі LLVM IR (Intermediate

Representation), такі як аналіз, трансформація та оптимізація. Завдяки LLVM Pass можна впроваджувати різноманітні оптимізації коду або здійснювати аналіз програмного коду для різних цілей, таких як покращення продуктивності, виявлення помилок або автоматичне генерування коду [5].

З метою тестування працездатності запропонованого підходу було реалізовано LLVM Pass, який друкує в консоль LLVM будь-яку функцію, позначену новим атрибутом при використанні команди *llc* (третій етап компіляції) у режимі налагодження.

```
int fib(int n) attribute((aggressiveInline))
{
      if (n == 0)
      {
            return 1;
      }
      if (n == 1)
      {
            return 2;
      }
      return fib(n - 1) + fib(n - 2);
}
```

Для компіляції коду будемо використовувати генератор cmake разом з LLVM. Далі необхідно використати цей Pass, запустивши LLVM операцію компіляції із прапором *-basicblockcounter*.

У результаті на консолі отримуємо такі дані:

```
Contents of BasicBlock corresponding to MachineBasicBlock : 0x5613102bb850
Contents of MachineBasicBlock : bb.1.cond.true :;
predecessors:% bb.0 successors : % bb.3 JMP_1 % bb.3
```

Вбудовування функцій у термінах LLVM є міжпроцедурною оптимізацією (interprocedural optimization, IPO). llvm/lib/Transforms/IPO – директорія, що містить реалізації кількох LLVM Pass, що виробляють модифікації оптимізованого LLVM IR.

Наприклад, файл ArgumentPromotion.cpp оптимізує виклики функцій, замінюючи аргументи константи, якщо це можливо. Дані оптимізації запускаються під час використання команди *opt* (за замовчуванням запускається лише деяка частина, для використання певних IPO існують певні аргументи запуску).

Для реалізації необхідної компоненти необхідно створити трансформуючий LLVM Pass, що реалізує клас *LegacyInlinerBase*, що містить логіку визначення доцільності вбудовування функцій.

Доцільність вбудовування функції визначається шляхом оцінювання вартості вбудовування функції InlineCost – це метрика або оцінка, яка використовується компіляторами для прийняття рішення про те, чи варто вбудовувати функцію (викликати її код безпосередньо там, де вона використовується) в місце її виклику [20].

Вбудовування функцій може призвести до оптимізації швидкодії програми за рахунок уникнення накладних витрат на виклик функції (наприклад, на збереження та відновлення регістрів, передачу параметрів, тощо). InlineCost вимірює вартість вбудовування функції, враховуючи такі фактори, як розмір коду функції, кількість її викликів, вплив на розмір кеш-пам'яті процесора та інші параметри. Більш низьке значення InlineCost зазвичай вказує на те, що вбудовування функції може бути корисним з точки зору оптимізації швидкодії програми.

Є верхня межа вартості: якщо вбудовування функції має вартість більшу за верхню межу, то вважається, що це вбудовування збиткове. Повний спектр всіх правил визначення доцільності вбудовування функцій міститься в директорії llvm/lib/Analysis.

Таким чином, для того, щоб реалізувати контрольоване агресивне вбудовування функцій, необхідно реалізувати клас, що успадковується від *LegacyInlinerBase* та перевизначальний метод, відповідальний за обчислення вартості вбудовування.

У рамках дослідження достатньо визначити обчислення вартості та верхньої межі вартості константними для функцій, позначених атрибутом *aggressiveInline*. У коді це виглядає наступним чином:

```
InlineCost getInlineCost(CallBase& CB) override
{
	Function* Callee = CB.getCalledFunction();
	if (!Callee || Callee->isDeclaration() || !Callee ->
		hasFnAttribute(Attribute::AttrKind::AggressiveInline))
	{
		return InlineCost::getNever("Not suitable");
	}
	return InlineCost::get(Cost, Threshold);
}
```

Змінні Cost та Threshold – чисельні константи, визначені під час створення класу. Саме вони є параметрами, які регулюють агресивність вбудовування функції.

Для того, щоб реалізований вище компонент можна було контролювати при запусках *opt*, необхідно надати можливість фіксувати вартість та верхню межу вартості в момент запуску *opt*. Було ухвалено рішення передавати вищезазначені параметри, використовуючи параметри командного рядка під час виклику *opt*. LLVM надає зручний інтерфейс для цього. Достатньо лише створити статичну змінну з типом, для якого вже реалізовано взаємодію з параметрами виклику:

```
static cl::opt<int>
MyInlinerCost("my-inliner-cost", cl::init(0),
	cl::desc("My inliner cost for optimization"));
```

Тепер вищезгадані змінні *Cost* та *Threshold* визначаються при створенні класу значеннями статичних змінних *MyInlinerCost* та *MyInlinerThreshold*, що дозволяє налаштовувати агресивність вбудовування викликом команди *opt* з певними аргументами командного рядка.

Локальний оптимізатор – це модифікована версія команди *opt*. Нею можна скористатися так:

```
opt -O2 -my-inliner -my-inliner-cost=30
-my-inliner-threshold=400 main.bc -o main.opt.bc
```

Тут *-my-inliner* – назва реалізованого LLVM Pass, яку необхідно запустити під час оптимізації LLVM IR коду. Константи my-inliner-cost і my-inlinerthreshold відповідають за агресивність вбудовування позначених атрибутом aggressiveInline функцій.

Оптимізатор повинен знаходити оптимальні параметри агресивності локального оптимізатора в рамках задачі максимізації вихідного розміру файлу. Також для додаткової регуляризації процесу необхідно надати можливість задавати простір пошуку вищеописаних параметрів.

Можна було б реалізувати пошук оптимальних параметрів безпосередньо всередині *opt*, але в C++ бібліотеки для вирішення задач оптимізації зазвичай мають набагато складніший інтерфейс. У зв'язку з цим для реалізації оптимізатора вибрано мову Python, всередині якого необхідно розташувати виклики *opt*.

Під максимально агресивним вбудовуванням функцій мається на увазі вбудовування якомога більшої кількості коду, що дорівнює максимізації розміру вихідного файлу. Таким чином, необхідно провести оптимізацію багатовимірної функції, аргументи якої – параметри, що передаються локальному оптимізатору, а значення, що повертається – розмір файлу після виклику команди *opt* із даними аргументами.

2.4. Висновки за розділом

У результаті огляду особливостей процесу агресивного вбудовування функцій вибрано алгоритм виконання процесу оптимізації у LLVM на рівні проміжного уявлення.

Виконано проєктування локального оптимізатору для вбудовування функцій, за допомогою якого можна знаходити оптимальні параметри агресивності в рамках задачі максимізації вихідного розміру файлу.

Запропоновано декомпозувати рішення задачі оптимізації на такі завдання:

– додавання атрибуту aggressiveInline;

– додавання компонента, що відповідає за вбудовування функцій;

– додавання параметрів команди opt, які відповідають за агресивність вбудовування функцій.

3 ПРОГРАМНА РЕАЛІЗАЦІЯ ОПТИМІЗАЦІЙНОГО КОМПІЛЯТОРА

3.1 Реалізація вбудовування функцій у терміналі LLVM

Для оптимізації цієї функції використовується SciPy – відкрита бібліотека високоякісних наукових інструментів для мови програмування Python [16]. У рамках цієї бібліотеки є безліч проєктів, що дозволяють оптимізувати багатовимірну функцію з обмеженням простору пошуку.

В рамках дослідження не потрібна висока точність, тому при виборі рішення увага акцентована лише на можливості використання та швидкості виконання. Саме тому вибрано метод диференційної еволюції [17], який є методом багатовимірної математичної оптимізації з класу стохастичних алгоритмів оптимізації, з використанням ідеї генетичних алгоритмів, але без необхідності роботи зі змінними у бінарному коді. Цей метод прямий, тобто він вимагає лише можливості обчислювати значення цільової функції, але не її похідних. Диференційна еволюція призначена для знаходження глобального екстремуму недиференційних, нелінійних, мультимодальних функцій від багатьох змінних.

Програма має такі обов'язкові параметри:

– inliner_input_file – шлях файлу бінарного формату LLVM IR (біткод), який потрібно оптимізувати;

– inliner_output_file – шлях для збереження виведення алгоритму оптимізації, також є бінарним форматом LLVM IR;

– inliner_inline_lines_upper_bound – обмеження зверху на розмір вбудованого коду (в байтах).

Також передбачено низку необов'язкових параметрів:

– inliner_opt_path – шлях до оптимізації інструменту LLVM. Необхідно, щоб ця команда була зібрана на проєкті LLVM, в якому є реалізація локального оптимізатора;

– inliner_arguments – додаткові аргументи для внутрішнього запуску opt. Можливо корисно скористатися, наприклад, аргументом O2;

– inliner_cost_left_bound, inliner_cost_right_bound, inliner_threshold_left_bound, inliner_threshold_right_bound – параметри, що задають межі простору пошуку параметрів my-inliner-cost та my- inliner-threshold для викликулокального оптимізатора;

– inliner_cores_to_use – число, яке необхідно підставити в аргумент workers при виклику диференціальної еволюції. Цей аргумент відповідає за розпаралелювання оптимізації;

– inliner_tmp_folder_name – назва підпапки у директорії викликуоптимізатора, в якому будуть розміщуватись тимчасові файли, необхідні для роботи.

Реалізація оптимізатора полягає в тому, щоб оптимізувати функцію, що повертає розмір файлу, що виходить при використанні локального оптимізатора з параметрами my-inliner-cost та my-inlinerthreshold, що оптимізуються за допомогою диференціальної еволюції.

В результаті реалізовано оптимізатор – застосунок «optimizer». Далі наведено приклад використання оптимізатора:

```
python3.8 optimizer --inliner_input_file = tinyxml2.cpp.bc
- inliner_output_file = tinyxml2.cpp.bc.out
- inliner_inline_lines_upper_bound = 1600000
- inliner_cost_right_bound = 100
- inliner_threshold_right_bound = 500
- inliner_cores_to_use = 16 --inliner_arguments = -O2
```

3.2 Реалізація модифікатора команд компіляції

Розроблено модифікатор команд компіляції проєкту із системою складання CMake у вигляді ланцюжка команд звернення до оптимізатору.

Оскільки робота проводиться на рівні LLVM IR, тому необхідно було модифікувати процес збірки проєкту таким чином, щоб автоматично викликати

оптимізатор. Це необхідно хоча б для того, щоб провести вимір затримки оптимізованих проєктів.

При використанні Clang для компіляції до LLVM IR, необхідно додати опцію *-O* для автоматичного включення оптимізацій. Важливо враховувати, що рівень оптимізації безпосередньо впливає на час збірки та якість згенерованого коду. Тому слід експериментувати з різними рівнями оптимізації та оцінювати їхні наслідки на продуктивність та поведінку програми.

Так як складання проєктів сама по собі є дуже складною і великою темою, в рамках дослідження пропонується зробити низку припущень. При використанні генератору складання CMake команда CMAKE_EXPORT_COMPILE_COMMANDS генерує коректний файл compile_commands.json.

Структура файлу compile_commands.json (шляхи до файлів для стислості скорочені):

```
[
{
     "directory": "Diplom/RedBlackTree",
     "command" : "Diplom/llvm-project/build/bin/clang++
          - I / Diplom / RedBlackTree / include - std = gnu++17
          - o CMakeFiles / example_RBT.dir / examples /
          RBT.cpp.o - c Diplom / RedBlackTree / examples / RBT.cpp
          ", "file": "Diplom / RedBlackTree / examples / RBT.cpp",
          "output" : "CMakeFiles/example_RBT.dir/examples/RBT.cpp.o"
}
]
```

Модифікатор розбиває одну команду компіляції на кілька етапів.

1. Спочатку компільований вихідний файл перетворюється не на об'єктний файл, а на біткод у форматі LLVM IR. Це досягається за допомогою прапора *emit-llvm*. Необхідно використовувати модифіковане складання LLVM, в якому реалізовано локальний оптимізатор.

2. Далі над отриманим файлом у бінарному форматі LLVM IR викликається оптимізатор.

3. Отриманий оптимізований файл перетворюється на машинний код для цільової архітектури. Для цього необхідно використати команду *llc*.

4. Зрештою, використовується команда *as* для отримання об'єктного файлу з машинного коду.

Для структури наведеної вище отримуємо такі результати:

```
[
{
      "command": "Diplom/llvm-project/build/bin/clang++ -
            I/ Diplom/RedBlackTree/include -std=gnu++17 -emit-llvm
            -o CMakeFiles/example_RBT.dir/examples/RBT.cpp.ll
            -c Diplom/ RedBlackTree/examples/RBT.cpp",
            "Directory" : "Diplom/RedBlackTree",
            "file" : "Diplom/RedBlackTree/examples/RBT.cpp"
},
{
"command": "python3.8 optimizer
--inliner_input_file=
Diplom/RedBlackTree/CMakeFiles/example_RBT.dir/examples/ RBT.cpp.ll
--inliner_output_file=Diplom/RedBlackTree/
CMakeFiles / example_RBT.dir / examples / RBT.cpp.ll.out
-inliner_inline_lines_upper_bound = 50000
-inliner_cores_to_use = 16", "directory": "Diplom / llvm - project /
scripts",
"file": "Diplom / RedBlackTree / examples / RBT.cpp"
},
{
"command": "Diplom/llvm-project/build/bin/llc Diplom/
RedBlackTree/CMakeFiles/example_RBT.dir/ examples/ RBT.cpp.ll.out
- o Diplom / RedBlackTree / CMakeFiles / example_RBT.dir / examples /
RBT.cpp.ll.s",
"directory" : "Diplom/llvm-project/scripts",
"file" : "Diplom/RedBlackTree/examples/RBT.cpp"
},
{
"command": "as Diplom/RedBlackTree/CMakeFiles/
example_RBT.dir/examples/RBT.cpp.ll.s
-o CMakeFiles/ example_RBT.dir/examples/RBT.cpp.o", "directory" :
"Diplom/RedBlackTree ",
"file" : "Diplom/RedBlackTree/examples/RBT.cpp"
}
]
```

В підсумку модифікатор створює JSON-файл, який можна модифікувати для експериментів оптимізації. Зокрема, якщо потрібно, можна передати додаткові аргументи оптимізації.

Таким чином, реалізована програма *modify_compile_commands*, яка приймає шлях до файлу, що містить команди компіляції. Також вона приймає шлях, яким повинен бути розташований модифікований файл і прапори оптимізації, що використовуються у викликах команд компіляції.

3.3 Реалізація виконавця команд компіляції

Виконавцем команд оптимізаційного компілятора є застосунок run_compile_commands, який запускає команди компіляції в порядку, заданому у файлі з командами компіляції, паралельно виводячи на екран поточну команду, що виконується. Виконавець приймає єдиний аргумент – шлях до файлу з командами компіляції. Реалізація цієї компоненти зводиться до аналізу JSON-файлу та послідовного застосування команд, розміщених у файлі.

Під час виконання дослідження реалізовано програмну систему, користуватися якою можна, якщо виконати такі кроки:

1. зібрати модифіковану версію llvm (це необхідно зробити один раз);
2. форсувати генерацію compile_commands.json;
3. викликати команду cmake;
4. модифікувати файл compile_commands.json;
5. запустити команди з модифікованого compile_commands.json;
6. викликати команду make.

Розглянемо використання цієї роботи на прикладі тестового проєкту [29], присвяченому парсингу XML [18] файлів. Парсинг XML – це процес аналізу структури та вмісту XML-файлів для витягування потрібних даних. Для цього зазвичай використовують спеціалізовані бібліотеки у мовах програмування, таких як Python, Java або C#. В Python, наприклад, є бібліотеки, такі як xml.etree.ElementTree та lxml, які надають зручні засоби для роботи з XML.

Модифікуємо файл CodeGuida/CodeGuida.cpp, розставивши attribute (aggressiveInline) на всі функції та методи. Вказуємо в CMakeLists.txt шлях до зібраної модифікованої версії Clang++ та форсуємо генерацію списку команд компіляції compile_commands.json:

```
set(CMAKE_CXX_COMPILER Diplom / llvm - project / build / bin / clang++)
set(CMAKE_EXPORT_COMPILE_COMMANDS ON)
```

Далі викликаємо команду CMakeLists.txt, потім необхідно модифікувати список команд компіляції за допомогою модифікатора:

```
python3.8 modify_compile_commands.py --build_input =
Diplom / tinyxml2 / compile_commands.json --build_output =
Diplom / tinyxml2 / compile_commands_cmake.opt.json
--build_opt_flag = -O2
```

Залишилось запустити виконавець команд компіляції та зібрати проєкт викликом команди make:

```
python3.8 run_compile_commands.py --compile_command_path =
Diplom / tinyxml2 / compile_commands_cmake.opt.json
```

В результаті файли, що виконуються, будуть складені з використанням техніки агресивного вбудовування функцій.

3.4 Тестування та результати

Для вимірювання середнього часу роботи програми були написані додаткові програми, що симулюють навантаження на проєкти, що тестуються. Також у випадку декартового дерева (див. дод. А) необхідно було зафіксувати ініціалізацію генератора випадкових чисел, тому що в іншому випадку вимірювання проводилися б на деревах різної структури.

Важливим аспектом тестування є забезпечення однакових умов виконання тестових вимірювань. Середовище виконання може впливати на результати вимірювань та їх точність.

Віртуальна машина є одним із способів забезпечення однакових умов тестування. Віртуальний сервер дозволяє створити ізольоване середовище, яке може бути налаштоване для максимальної продуктивності та стійкості до зовнішніх впливів. Для тестування було вибрано хостинг Google Compute Engine [15]. Google Compute Engine є сервісом обчислення у хмарній платформі Google Cloud, який дозволяє створювати віртуальні машини за допомогою клієнтів MobaXterm, Termius та Termux. Нові користувачі Google Cloud можуть безкоштовно використовувати одну віртуальну машину протягом місяця.

Розробник може використовувати розгортання програм, контейнерів Docker, кластерів Kubernetes, баз даних, хостингу веб-сайтів.

Для тестування проєктів створено віртуальну машину з такими характеристиками: HDD-диск 2ГБ, CPU Intel Xeon Gold 6338, тактова частота 2 ГГц, оперативна пам'ять 8 ГБ.

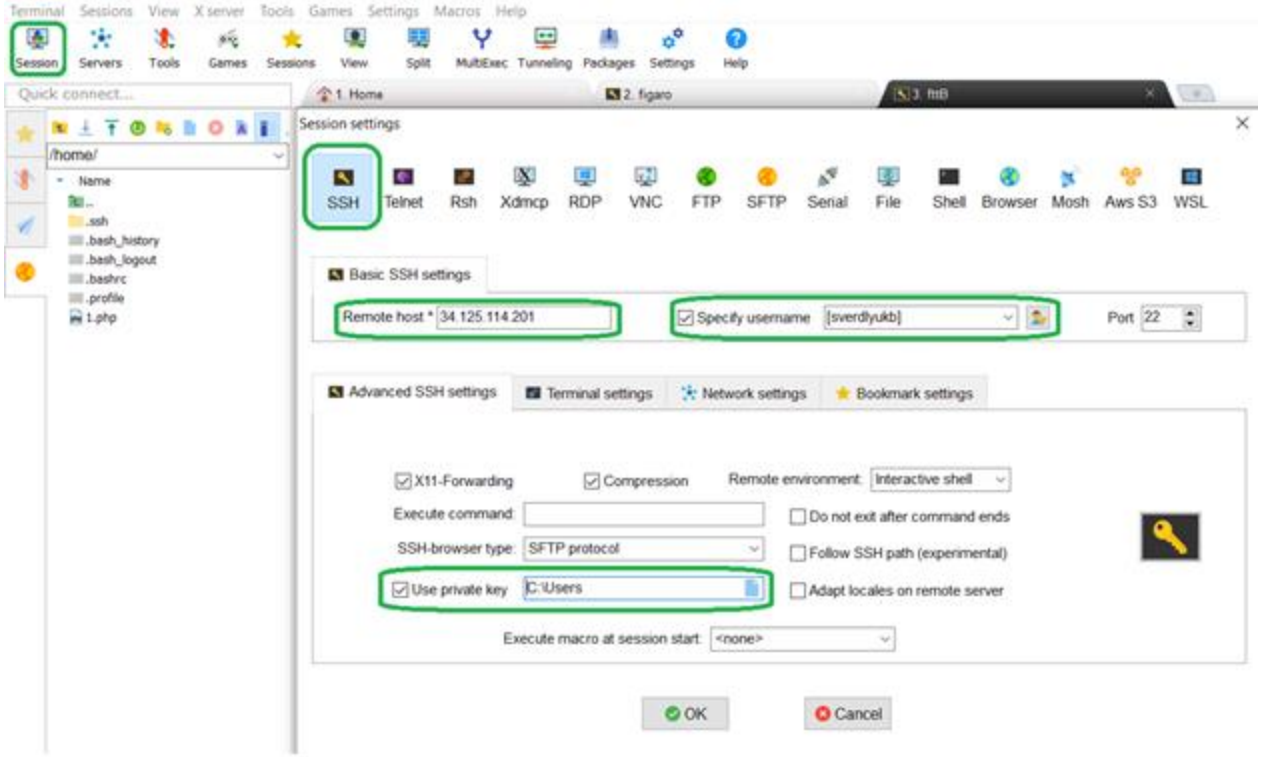

Рисунок 3.1 – Підключення до віртуальної машини з терміналу MobaXterm

Ізолювання ядра також є важливим аспектом тестування. Це дозволяє уникнути впливу інших процесів на результати вимірів. Зокрема, ізолювання ядра дозволяє зменшити вплив операційної системи на продуктивність програми.

Для тестування та вимірювання затримок було вибрано три проєкти, розташовані у вільному доступі: CodeGuida [18], RedBlackTrees [19] та Cartesian Tree [20]:

– проєкт CodeGuida призначений для ефективного парсингу файлів XML;

– проєкт RedBlackTrees містить реалізацію червоно-чорного дерева;

– проєкт Cartesian Tree містить у собі реалізацію декартового дерева.

Хоча завдання поставлене для затримки в сенсі середнього часу роботи (throughput), у цьому розділі наводяться також і персентилі. Таким чином, можна використовувати, наприклад, дев'яносто дев'ятий персентиль для вимірювання затримки (в сенсі latency).

Всі тестові виміри затримок були зроблені в ізольованому ядрі. Для мінімізації шуму при вимірах програми, що симулюють навантаження, було запущено 2000 разів.

Вимірювання затримок проводилися для різних оптимізаційних прапорів компіляції: O0, O2 та O3. Ці прапори передавалися clang++, opt і llc. Прапори компіляції O0, O2 та O3 - це різні рівні оптимізації коду, які використовуються компіляторами при перетворенні вихідного коду програми в машинний код. Рівень оптимізації O3 застосовує більш агресивні оптимізації, які можуть покращити продуктивність програми значно, але при цьому можуть збільшити ризик неправильної роботи програми або збільшити час компіляції.

Варіантів розміщення атрибуту *aggressiveInline* у цих проєктах дуже багато. Весь процес складання, вимірювання часу роботи та підбиття результатів займає значну кількість часу, тому було прийнято рішення проводити вимірювання при розстановці атрибуту на всіх функціях та методах.

Вимірювання виконуються без обмеження на розмір файлу, що виводиться оптимізатором. Усі відсотки скорочуються до двох знаків після коми стандартним математичним правилом скорочення.

Відомо, що однією з найбільш фундаментальних структур даних є бінарне дерево пошуку. Однак продуктивність бінарного дерева пошуку сильно залежить від його форми, і в гіршому випадку воно може виродитися в лінійну структуру з часовою складністю O(n). Червоно-чорні дерева є типом збалансованого бінарного дерева пошуку, яке використовує певний набір правил, щоб гарантувати, що дерево завжди збалансоване [19].

На рис. 3.2 зображено значення персентилів часу роботи реалізації RedBlackTrees з різними способами компіляції. Модифікація – розроблений оптимізаційний компілятор, базовий компілятор – компілятор на базі LLVM

версії 12. У підпису вказано назву проєкту і оптимізаційний прапор, що використовується під час запуску. На вертикальній осі вказано час виконання в секундах.

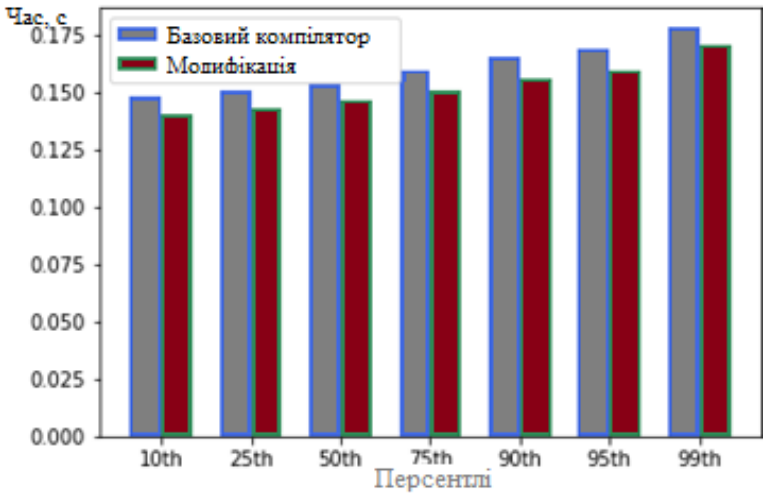

Рисунок 3.2 – Результати тестування RedBlackTrees, прапор компілятора O0

У разі прапора *O0* середній час роботи базового компілятора становив приблизно 0.155 секунд, модифікація показала приблизно 0.147 секунд. Таким чином отримано прискорення на 4.94%.

На рис. 3.3 зображені самі виміри, але за використанні прапора *O2*. Зауважимо, що середній час роботи базового компілятора значно прискорився, таким чином, за наявності прапорів оптимізації, отримувати прискорення стає складніше. Модифікація показала прискорення середнього часу роботи на 1.22%.

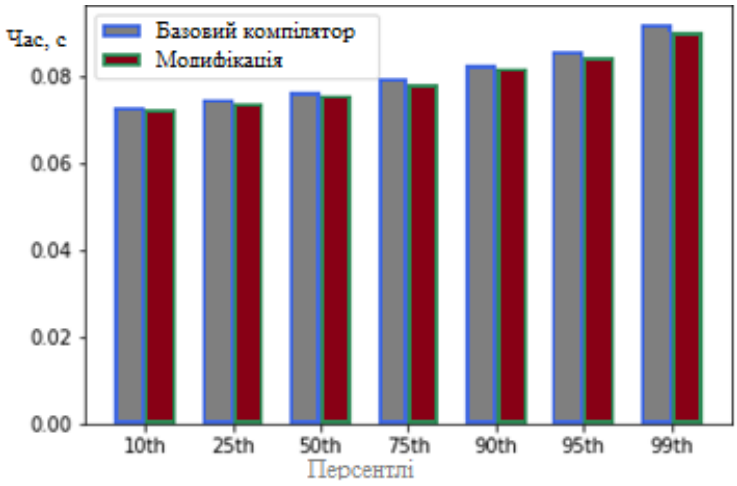

Рисунок 3.3 – Результати тестування RedBlackTrees, прапор компілятора O2

Перейдемо до прапора оптимізації *O3*. На рис. 3.4 зображено графік, що показує, що агресивне вбудовування на деяких персентилях погіршило час роботи. Середній час роботи збільшився на 0.39%.

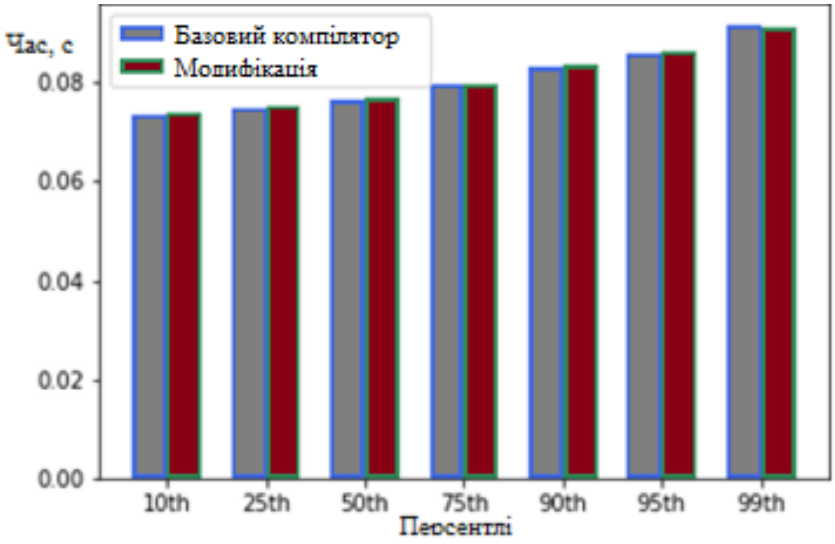

Рисунок 3.4 – Результати тестування RedBlackTrees, прапор компілятора O3

На рис. 3.5-3.7 зображено вимірювання часу роботи проєкту CodeGuida з прапорами компіляції O0, O2, O3 відповідно.

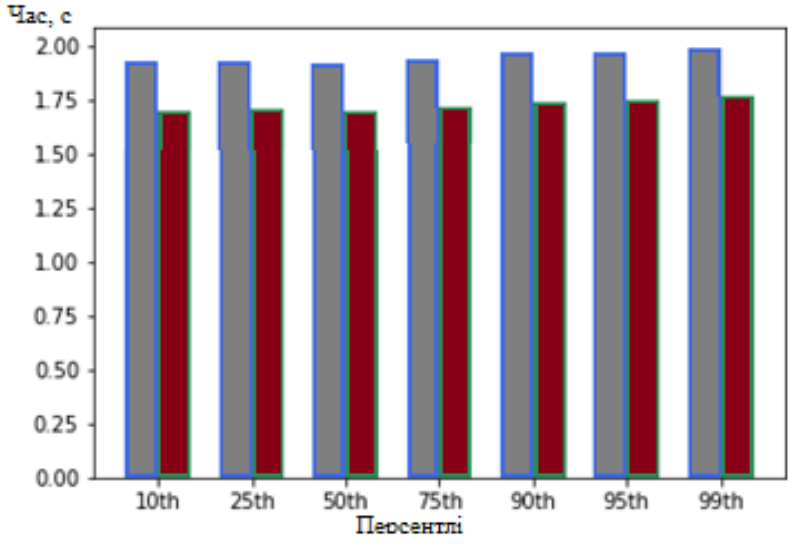

Рисунок 3.5 – Результати тестування CodeGuida, прапор компілятора O0

Агресивне вбудовування функцій добре виявляє себе на прапорі компіляції O0, прискоривши середню швидкість роботи на 11.2%.

При використанні O2 тенденція зменшення часу роботи зберігається на всіх персентилях. Середній час роботи пришвидшується на 1,8%.

Прапор O3 також зберігає тенденцію зменшення часу роботи під час використання розробленої програмної системи. Прискорення середнього часу роботи вийшло 1.7%.

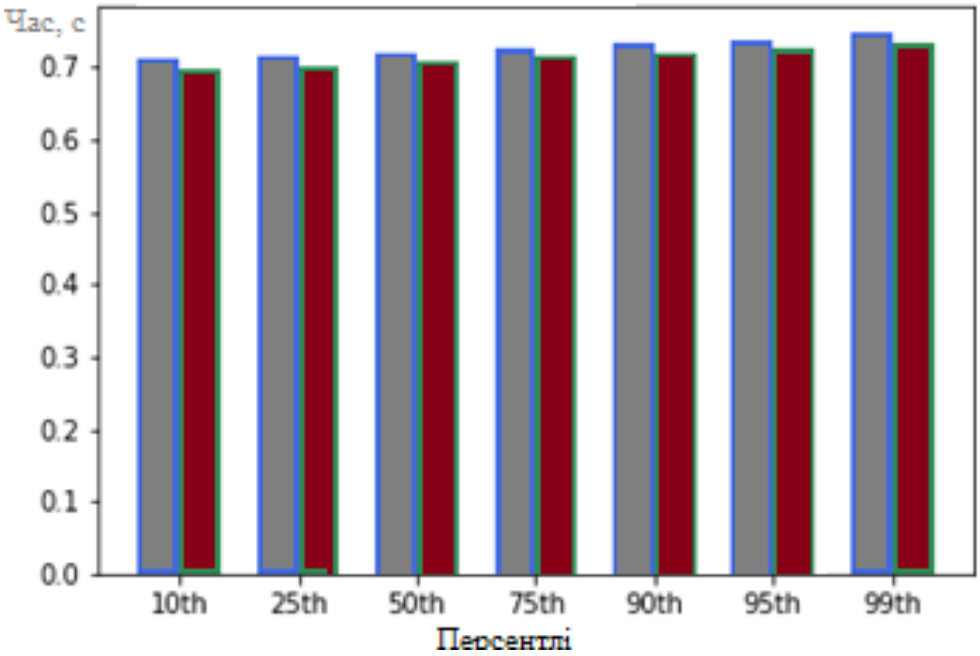

Рисунок 3.6 – Результати тестування CodeGuida, прапор компілятора O2

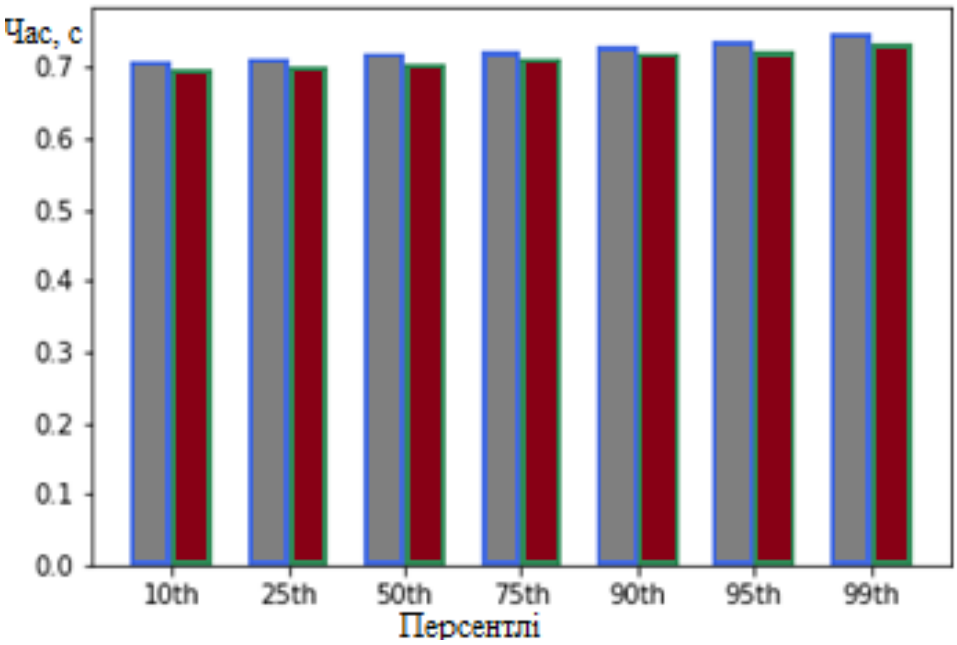

Рисунок 3.7 – Результати тестування CodeGuida, прапор компілятора O3

На рис. 3.8-3.10 зображено вимірювання часу роботи проєкту Cartesian Tree з прапорами компіляції O0, O2, O3 відповідно. Вимірювання демонструють значне прискорення: наприклад, при використанні прапора оптимізації O2 сформовано наступний звіт для базового компілятора:

```
Total time taken: 319.025875 seconds
Average time taken: 0.159513 seconds
Variance: 0.0000529622295807
Standard deviation: 0.0072775153439022
10th percentile: 0.151603 seconds
25th percentile: 0.154321 seconds
50th percentile: 0.158299 seconds
75th percentile: 0.163166 seconds
90th percentile: 0.169203 seconds
95th percentile: 0.173528 seconds
99th percentile: 0.181843 seconds
```

Отримано наступний звіт для модифікації:

```
Total time taken: 269.129571 seconds
Average time taken: 0.134565 seconds
Variance: 0.0000399415451146
Standard deviation: 0.0063199323662995
10th percentile: 0.128123 seconds
25th percentile: 0.130002 seconds
50th percentile: 0.133393 seconds
75th percentile: 0.137574 seconds
90th percentile: 0.142931 seconds
95th percentile: 0.146749 seconds
99th percentile: 0.155217 seconds
```

Отримано такі прискорення середнього часу роботи:

– для прапора *O0* прискорення на 12,6%;

– для прапора *O2* прискорення на 16,1%;

– для прапора *O3* прискорення на 14,5%.

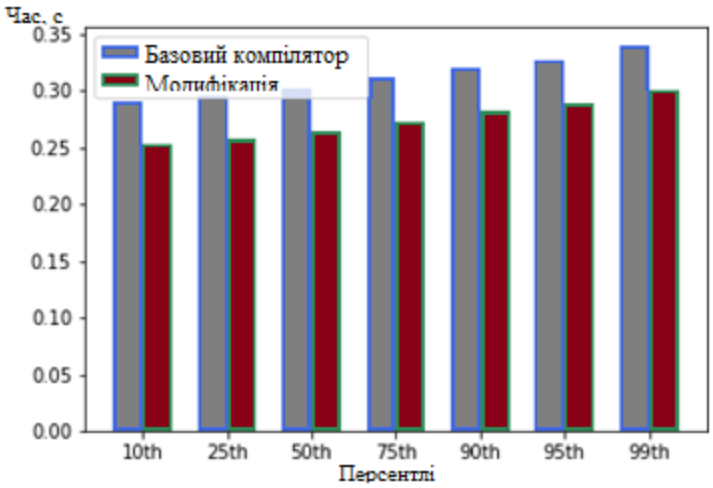

Рисунок 3.8 – Результати тестування Cartesian Tree, прапор компілятора O0

Отримані дані демонструють статистично значуще прискорення середнього часу роботи всіх прапорах компіляції, де статистична значимість визначається t-критерієм Уелча з рівнем значимості 0.001. Цей критерій є модифікацією t-критерію Стьюдента і дозволяє робити висновки, навіть коли дисперсії двох груп неоднакові або коли вибірки мають різний розмір.

Також видно, що прискорення характерне для всіх зазначених персентлей.

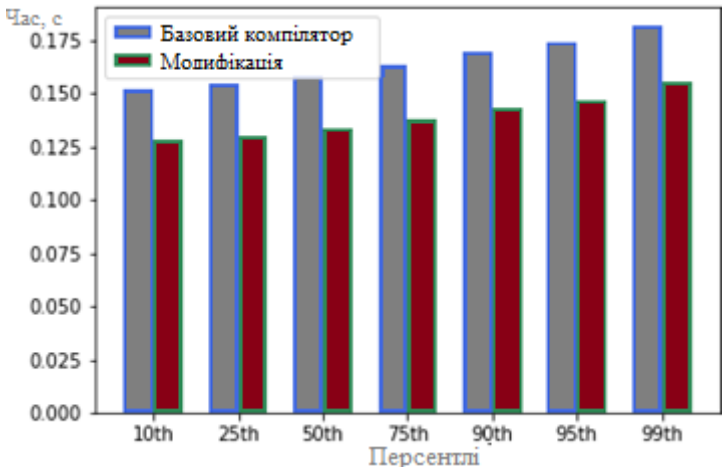

Рисунок 3.9 – Результати тестування Cartesian Tree, прапор компілятора O2

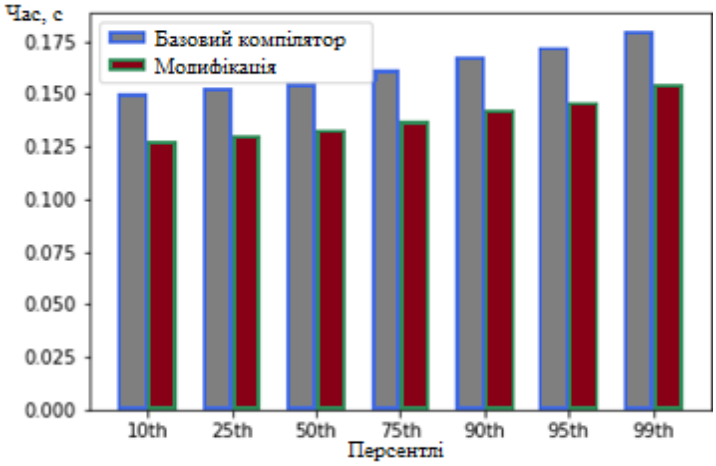

Рисунок 3.10 – Результати тестування Cartesian Tree, прапор компілятора O3

Всі виміри проводилися без обмеження розміру виведення оптимізатора. Розглянемо, наскільки збільшилися розміри файлів, що виконуються після використання реалізованої програмної системи.

З таблиці 3.1 видно, що при прапорі *O0* розмір файлу може збільшитися у сотні разів. Для двох із трьох проєктів відбулося збільшення у 2-3 рази.

Таблиця 3.1 – Порівняння розміру виконуваних файлів

Прапор оптимізації	**Тестований проєкт**	**Базовий компілятор**	**Модифікація**
O0	CodeGuida	170 КБ	450 КБ
	RedBlackTrees	23 КБ	65 КБ
	Cartesian Tree	28 КБ	1.1 МБ
O2	CodeGuida	177 КБ	345 КБ
	RedBlackTrees	22 КБ	40 КБ
	Cartesian Tree	22 КБ	70 КБ
O3	CodeGuida	185 КБ	365 КБ
	RedBlackTrees	21 КБ	37 КБ
	Cartesian Tree	22 КБ	66 КБ

При прапорі O2 розмір виконуваних файлів збільшується в 1.7-3 рази. Увімкнення прапора O3 не сильно відрізняється від O2 у плані розміру файлів. Таким чином, розроблений оптимізаційний компілятор збільшує розмір файлів за відсутності обмежень. Але на тестових проєктах значне збільшення (на два порядки) вийшло лише у разі відсутності оптимізаційних прапорів компіляції.

3.5 Висновки за розділом

Виконана програмна реалізація оптимізаційного компілятора, а саме: вбудовування функцій диференційної еволюції у терміналі LLVM, реалізація модифікатора команд компіляції проєкту у вигляді ланцюжка команд звернення до оптимізатору, реалізація виконавця команд компіляції.

Виконано порівняльний аналіз середнього часу роботи тестових проєктів CodeGuida, RedBlackTrees, Cartesian Tree для встановлення відмінностей між оптимізаційним компілятором та базовим компілятором, у ході якого виявлено, що навіть за використання оптимізаційних прапорів компіляції вдається досягти прискорення. Встановлено прискорення часу виконання у середньому до 15%, що підтверджує практичну значущість дослідження.

ВИСНОВКИ

У дослідженні виконано проєктування, програмну реалізацію та тестування оптимізаційного компілятора, який забезпечує додаткову оптимізацію часу виконання програм на мові C++ засобами агресивного вбудовування функцій з широким спектром параметрів регулювання рівня агресивності та контролем розміру вихідного файлу.

У результаті огляду особливостей процесу агресивного вбудовування функцій вибрано алгоритм виконання процесу оптимізації у програмні системі LLVM на рівні проміжного уявлення. Це дозволяє додавати нові мови програмування для здійснення оптимізації.

Для реалізації оптимізаційного компілятора вибрано програмну систему LLVM, компілятор Clang для середовища Visual Studio та відкритий генератор сценаріїв складання CMake.

Програмна реалізація позиціонується не як незалежна оптимізація, що блокує застосування інші критерії, а як додаткова оптимізація, яка може проводитися поряд з іншими.

Урегульованість рівня агресивності вбудовування функцій підтримується такими параметрами:

– розробник сам вибирає, які функції можна агресивно вбудовувати;

– розробник може використовувати прапори оптимізації, що реалізовані в clang++, opt та llc;

– вибір простору пошуку параметрів, які визначають структуру класу InlineCost для визначення доцільності вбудовування функцій, помічених атрибутом *aggressiveInline*;

– можливість обмежувати розмір вихідного файлу, якщо великий розмір або більш триваліша компіляція становлять розробнику суттєві незручності.

Виконано порівняльний аналіз середнього часу виконання тестових проєктів CodeGuida, RedBlackTrees, Cartesian Tree для встановлення відмінностей між оптимізаційним компілятором та базовим компілятором, у ході якого

виявлено, що навіть за використання оптимізаційних прапорів компіляції вдається досягти прискорення програм. Встановлено прискорення часу виконання проєкту RedBlackTrees у середньому на 2%, проєкту CodeGuida у середньому на 4,9%, проєкту Cartesian Tree у середньому на 14,4%. Найбільше прискорення зафіксовано для проєкту Cartesian Tree до 16,1%, що підтверджує практичну значущість роботи.

Також встановлено, що розроблений оптимізаційний компілятор збільшує розмір файлів за відсутності обмежень.

У напрямку подальших досліджень можна додатково досліджувати структуру функцій, щоб точніше обчислювати вартості вбудовування функцій, використовувати більш нові LLVM, додавати мови програмування високого рівня для здійснення оптимізації.

Виконана апробація кваліфікаційної роботи: публікація тези доповіді на тему «Розробка оптимізаційного компілятора для мови C++» у IX Всеукраїнській науково-практичній конференції «Інформаційне суспільство: проблеми та перспективи» (Одеса, 24 травня 2024 р.).

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Налаштування компілятора: Попередження та помилки. URL: https://acode.com.ua/urok-9-nalashtuvannya-kompilyatora-poperedzhennya-ta-pomylky.

2. Microsoft Learn. /Ob (Inline Function Expansion). URL: https://learn.microsoft.com/en-us/cpp/build/reference/ob-inline-function-expansion?view=msvc-170.

3.Wiki: Low Level Virtual Machine. URL: https://uk.wikipedia.org/wiki/Low_Level_Virtual_Machine.

4. What is FORCEINLINE macro? URL: https://forums.unrealengine.com/t/what-is-forceinline-macro/438373.

5. Writing an LLVM Pass (legacy PM version). URL: https://llvm.org/docs/WritingAnLLVMPass.html.

6. Microsoft Learn. Параметр /Ob (розширення функцій, що вбудовуються) URL: https://learn.microsoft.com/cpp/build/reference/ob-inline-function-expansion?view=msvc-170.

7. Відбувся реліз .NET 8. URL: https://dou.ua/forums/topic/46226.

8. Clang/LLVM support in Visual Studio projects. URL: https://learn.microsoft.com/en-us/cpp/build/clang-support-cmake?view=msvc-170&source=recommendations.

9. Jun-Pyo L. Aggressive Function Splitting for Partial Inlining. *15th Workshop on Interaction between Compilers and Computer Architectures.* IEEE, 2011. Pp. 80-86.

10. Kulkarni S., Cavazos J., Wimmer C., Simon D. (2013, February). Automatic construction of inlining heuristics using machine learning. *In Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2013. Pp. 1-12.

11. Zhou X., Lilius J., Yan L. Function Inlining with Code Size Limitation in Embedded Systems. *B: Int. Arab J. Inf. Technol*. 2.3, 2005. Pp. 214-218.

12. Wiki: Оптимізаційний компілятор. URL: https://uk.wikipedia.org.

13. CMake: The Standard Build System. URL: https://cmake.org/features.

14. Chris Lattner. The Architecture of Open Source Applications (Volume 1) LLVM. URL: https://aosabook.org/en/v1/llvm.html.

15. Як скористатись безкоштовним хостингом Google — покрокова інструкція. URL: https://dou.ua/forums/topic/41251.

16. Fundamental algorithms for scientific computing in Python. URL: https://scipy.org.

17. Диференціальна еволюція. URL: https://www.wikidata.uk-ua.nina.az/Диференціальна_еволюція.html.

18. DevZone. XML в Python. URL: https://devzone.org.ua/post/xml-v-python.

19. Introduction to Red-Black Tree. URL: https://www.geeksforgeeks.org/introduction-to-red-black-tree.

20. Cartesian Tree. URL: https://www.geeksforgeeks.org/cartesian-tree.

21. llvm::InlineCost Class Reference URL: https://llvm.org/doxygen/classllvm_1_1InlineCost.html.

22. Чикунов П.О. Відстеження поширення по програмі неперевірених зовнішніх даних. Матеріали Міжнародної науково-практичної конференції (Одеса, 26 квітня 2024 р.): *Європейські орієнтири розвитку України: науково-практичний вимір в умовах воєнних викликів*. Одеса, вид-во «Гельветика», 2024. С. 100-105.

23. Чикунов П.О. Рефакторинг коду при конструюванні програмного забезпечення. Матеріали VI Міжнародної науково-практичної конференції: *Актуальні тенденції розвитку освіти, науки та технологій*. Бахмут – Харків: ННППІ УІПА, 2023. Ч 1. – С. 98-100.

24. Chykunov P. Profiling multithreaded programs in the Visualizer Concurrency / P. Chykunov, A. Kotov // Матеріали VI Всеукраїнської науково-практичної інтернет-конференції: *Сучасні технології в енергетиці, електромеханіці, системах управління та машинобудуванні*. Харків: ННППІ УІПА, 2023. С. 36-38.

25. Сидоренко О.І. Розробка оптимізаційного компілятора для мови С++ / Матеріали IX Всеукраїнської науково-практичній конференції: *Інформаційне суспільство: проблеми та перспективи»*. Одеса, ОНЮА, 2024 – депоновано.

26. Трофименко О. Г., Логінова Н. І., Манаков С. Ю. Методичні вказівки до виконання кваліфікаційної роботи здобувачами першого (бакалаврського) рівня вищої освіти за спеціальностями 121 «Інженерія програмного забезпечення» та 122 «Комп'ютерні науки» [Електронне видання] / О. Г. Трофименко, Н. І. Логінова, С. Ю. Манаков. – Одеса, 2024. – 39 с. Режим доступу: https://hdl.handle.net/11300/27211.

ДОДАТКИ

Додаток А. Приклад коду побудови декартового дерева для аналізу

```
// A O(n) C++ program to construct cartesian tree
// from a given array
#include<bits/stdc++.h>

/* A binary tree node has data, pointer to left
   child and a pointer to right child */
struct Node
{
    int data;
    Node *left, *right;
};

/* This function is here just to test buildTree() */
void printInorder (Node* node)
{
    if (node == NULL)
        return;
    printInorder (node->left);
    cout << node->data << " ";
    printInorder (node->right);
}

// Recursively construct subtree under given root using
// leftChild[] and rightchild
Node * buildCartesianTreeUtil (int root, int arr[],
          int parent[], int leftchild[], int rightchild[])
{
    if (root == -1)
        return NULL;

    // Create a new node with root's data
    Node *temp = new Node;
    temp->data = arr[root] ;

    // Recursively construct left and right subtrees
    temp->left = buildCartesianTreeUtil( leftchild[root],
                    arr, parent, leftchild, rightchild );
    temp->right = buildCartesianTreeUtil( rightchild[root],
                    arr, parent, leftchild, rightchild );

    return temp ;
}

// A function to create the Cartesian Tree in O(N) time
Node * buildCartesianTree (int arr[], int n)
{
    // Arrays to hold the index of parent, left-child,
    // right-child of each number in the input array
    int parent[n],leftchild[n],rightchild[n];

    // Initialize all array values as -1
    memset(parent, -1, sizeof(parent));
    memset(leftchild, -1, sizeof(leftchild));
    memset(rightchild, -1, sizeof(rightchild));
```

```
    // last processed of the Cartesian Tree.
    // Initially we take root of the Cartesian Tree as the
    // first element of the input array. This can change
    // according to the algorithm
    int root = 0, last;

    // Starting from the second element of the input array
    // to the last on scan across the elements, adding them
    // one at a time.
    for (int i=1; i<=n-1; i++)
    {
        last = i-1;
        rightchild[i] = -1;

        // Scan upward from the node's parent up to
        // the root of the tree until a node is found
        // whose value is greater than the current one
        // This is the same as Step 2 mentioned in the
        // algorithm
        while (arr[last] <= arr[i] && last != root)
            last = parent[last];

        // arr[i] is the largest element yet; make it
        // new root
        if (arr[last] <= arr[i])
        {
            parent[root] = i;
            leftchild[i] = root;
            root = i;
        }

        // Just insert it
        else if (rightchild[last] == -1)
        {
            rightchild[last] = i;
            parent[i] = last;
            leftchild[i] = -1;
        }

        // Reconfigure links
        else
        {
            parent[rightchild[last]] = i;
            leftchild[i] = rightchild[last];
            rightchild[last] = i;
            parent[i] = last;
        }

    }

    // Since the root of the Cartesian Tree has no
    // parent, so we assign it -1
    parent[root] = -1;

    return (buildCartesianTreeUtil (root, arr, parent,
                                    leftchild, rightchild));
}

/* Driver program to test above functions */
int main()
{
    /* Assume that inorder traversal of following tree
```

```
        is given
          40
         /   \
        10     30
       /         \
      5           28 */

    int arr[] = {5, 10, 40, 30, 28};
    int n = sizeof(arr)/sizeof(arr[0]);

    Node *root = buildCartesianTree(arr, n);

    /* Let us test the built tree by printing Inorder
       traversal */
    printf("Inorder traversal of the constructed tree : \n");
    printInorder(root);

    return(0);
}
```

Додаток Б. Апробація результатів роботи

РОЗРОБКА ОПТИМІЗУВАЛЬНОГО КОМПІЛЯТОРА ДЛЯ МОВИ C++

Сидоренко Олег

студент 4-го курсу факультету кібербезпеки та інформаційних технологій
Національного університету «Одеська юридична академія»

Основним завданням програмування є створення правильних, а не ефективних програм. Правильну, але не ефективну програму можна оптимізувати та зробити ефективною. У світі існує безліч програмного забезпечення, написаного на мові C++. Вважається, що ця мова є дуже швидкою, але існують сфери, що вимагають від програми високої продуктивності, яка недосяжна лише вдалим вибором мови. Для великих програм ще на стадії проєктування визначаються параметри, що включають середній час їх виконання, потрібний обсяг пам'яті та розмір файлу.

Актуальність дослідження обумовлена тим фактом, що при розробці високопродуктивного коду, існує потреба у застосуванні додаткової оптимізації вихідного коду. З цим завданням досить добре справляється компілятор, використовуючи оптимізаційні прапори компіляції [1], при чому часто досягається результат, що задовольняє користувача, проте далеко не завжди цього достатньо.

Метою дослідження є програмна реалізацію процесу оптимізації часу виконання програм на мові C++ засобами регульованого агресивного вбудовування функцій.

Для досягнення мети було сформульовано такі завдання:

– провести огляд предметної галузі та підходів;

– протестувати розроблений статичний аналізатор для визначення ефективності знаходження помилок та вразливостей у різних програмах;

– отримати покращення затримки прикладного проєкту у порівнянні з компіляцією без використання розробленої системи в однакових умовах.

Відомо, що компілятор розглядає вбудовані параметри розширення та ключові слова як пропозиції. Однією з найважливіших оптимізації компілятора є вбудовування функцій – заміна виклику функції безпосередньо тілом функції [5].

Процес вбудовування функції – це оптимізація компілятора, що спрямована на заміну виклику функції безпосередньо тілом функції. Ця дія може покращити швидкість роботи програми, оскільки після вбудовування відсутні накладні витрати на виклик функції. Вбудовування функцій у мові C++ дозволяє оптимізувати роботу компілятора, зменшуючи витрати на виклики функцій та покращуючи швидкодію програми. У середовищі Visual Studio це може бути досягнуто за допомогою ключового слова _forceinline, яке підказує компілятору вбудувати вміст функції безпосередньо в місце виклику, не створюючи окремого виклику функції.

LLVM – це програмна система, що є набором інструментів для розробки компіляторів, асемблерів та інших систем, пов'язаних з обробкою програмного коду [2]. LLVM надає універсальний, низькорівневий та модульний інтерфейс для компіляції програм, який може бути використаний для розробки компіляторів для різних мов програмування.

Процес компіляції в LLVM включає кілька етапів. Під час першого етапу, вихідний код програми перетворюється на проміжне уявлення (intermediate representation) LLVM IR за допомогою компілятора Clang [3] за допомогою виклику програми clang++ або іншого сумісного компілятора. Потім IR

передається оптимізатор LLVM, який застосовує різні оптимізації, такі як вбудовування функцій, видалення недосяжного коду та інші, для поліпшення продуктивності та якості згенерованого коду.

У компіляторі на основі LLVM інтерфейс відповідає за розбір, перевірку та діагностику помилок у вхідному коді, а потім переклад проаналізованого коду в LLVM IR. Це дуже проста реалізація трифазної конструкції (рис. 1), але цей простий опис приховує частину потужності та гнучкості, які архітектура LLVM отримує від LLVM IR.

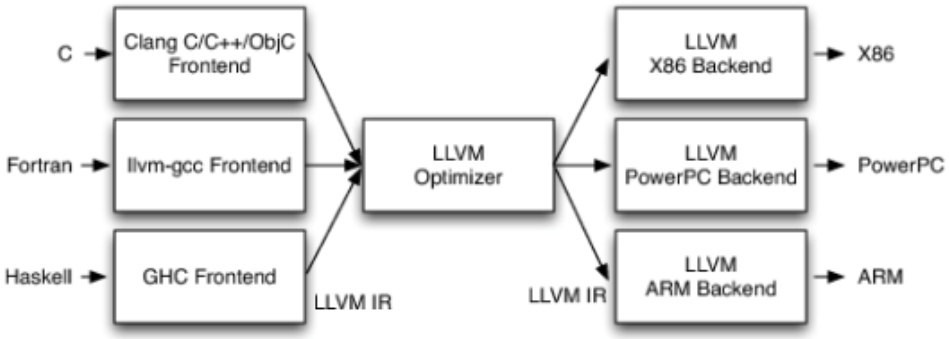

Рисунок 1 – Реалізація трифазної конструкції LLVM

Починаючи з версії 2019, MS Visual Studio включає підтримку редагування, створення та налагодження за допомогою Clang/LLVM у проєктах CMake [4], націлених на Windows. Після налаштування конфігурації Clang, розробник може створювати та налагодити проєкт. Середовище Visual Studio самостійно виявляє компілятор Clang, і надає можливості IntelliSense для підсвічування, навігації та інших функції редагування.

Далі оптимізоване проміжне уявлення IR перетворюється на машинний код цільової архітектури, використовуючи генератор коду LLVM. При генерації машинного коду LLVM використовує безліч оптимізації та техніки генерації коду для створення швидкого та ефективного виконуваного файлу.

Крос-платформовий відкритий генератор сценаріїв складання CMake є стандартом де-факто для створення коду на мові C++. Це потужне комплексне рішення для керування процесом створення програмного забезпечення. CMake дозволяє розробникам описувати, як створювати прості та дуже складні програмні системи за допомогою єдиного набору вхідних файлів. Систему можна використовувати для створення застосунків на багатьох платформах, від Android до iOS і високопродуктивних систем.

Програмна реалізація оптимізувального компілятору буде базуватися на мовах C++ і Python, наборі компіляторів LLVM, генераторі сценаріїв складання CMake.

Виконавцем команд оптимізувального компілятора є додаток run_compile_commands, який запускає команди компіляції в порядку, заданому у файлі з командами компіляції, паралельно виводячи на екран поточну команду, що виконується. Виконавець приймає єдиний аргумент – шлях до файлу з командами компіляції. Реалізація цієї компоненти зводиться до аналізу JSON-файлу та послідовного застосування команд, розміщених у файлі.

Під час виконання дослідження реалізовано програмну систему, користуватися якою можна, якщо виконати наступні кроки:

1. зібрати модифіковану версію llvm (це необхідно зробити один раз);
2. форсувати генерацію compile_commands.json;
3. викликати команду cmake;
4. модифікувати файл compile_commands.json;
5. запустити команди з модифікованого compile_commands.json;
6. викликати команду make.

Для вимірювання середнього часу роботи програми були написані додаткові програми, що симулюють навантаження на проєкти, що тестуються. Також у випадку декартового дерева необхідно було зафіксувати ініціалізацію генератора випадкових чисел, тому що в іншому випадку вимірювання проводилися б на деревах різної структури.

Віртуальний сервер дозволяє створити ізольоване середовище, яке може бути налаштоване для забезпечення однакових умов тестування, максимальної продуктивності та стійкості до зовнішніх впливів. Для створення віртуального серверу обрано хостинг Google Compute Engine.

Для тестування та вимірювання затримок було вибрано три проєкти, що розташовані у вільному доступі: CodeGuida для парсингу XML-файлів, RedBlackTrees з реалізацією червоно-чорного дерева, Cartesian Tree з реалізацією декартового дерева. Виконано порівняльний аналіз середнього часу роботи тестових проєктів для встановлення відмінностей між оптимізувальним компілятором та базовим компілятором, у ході якого виявлено, що навіть за використання оптимізаційних прапорів компіляції вдається досягти прискорення часу виконання до 15%, що підтверджує значущість дослідження.

Практичним результатом дослідження є реалізація прикладного інструменту, який дозволяє C++ розробникам застосовувати регульовану оптимізацію та впливати на обмеження розміру виконуваного файлу.

Список використаних джерел:

1. Налаштування компілятора: Попередження та помилки. URL: https://acode.com.ua/urok-9-nalashtuvannya-kompilyatora-poperedzhennya-ta-pomylky.
2. Writing an LLVM Pass (legacy PM version). URL: https://llvm.org/docs/WritingAnLLVMPass.html.
3. Clang/LLVM support in Visual Studio projects. URL: https://learn.microsoft.com/en-us/cpp/build/clang-support-cmake?view=msvc-170&source=recommendations.
4. CMake: The Standard Build System. URL: https://cmake.org/features.
5. Чикунов П.О. Відстеження поширення по програмі неперевірених зовнішніх даних / Європейські орієнтири розвитку України: науково-практичний вимір в умовах воєнних викликів : Матеріали Міжнародної науково-практичної конференції (Одеса, 26 квітня 2024 р.). Одеса, вид-во «Гельветика», 2024. С. 100-105.

Ключові слова: оптимізація часу виконання, компіляція, C++, вбудовування функцій, Low Level Virtual Machine

Keywords: runtime optimization, compilation, C++, function embedding, Low Level Virtual Machine

Науковий керівник: доцент П. Чикунов