Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет _____Комп'ютерних наук_____
(повна назва)

Кафедра _____Програмної інженерії_____
(повна назва)

АТЕСТАЦІЙНА РОБОТА
Пояснювальна записка

рівень вищої освіти – другий (магістерський)

Дослідження методів оптимізації, які використовуються у компіляторах коду
(тема)

Виконав: студент 2 курсу, групи ІПЗм-18-3___
_____Нємцов М.В._____
(прізвище, ініціали)

спеціальності 121- Інженерія програмного забезпечення
(код і повна назва спеціальності)

Освітньо-наукової програми
Інженерія програмного забезпечення

Керівник _____доц. каф. ПІ Каук В.І._____
(посада, прізвище, ініціали)

Допускається до захисту

Зав. кафедри, проф. _____ З.В.Дудар

2020 р.

ХАРКІВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ РАДІОЕЛЕКТРОНІКИ

Факультет комп'ютерних наук

Кафедра програмної інженерії

Рівень вищої освіти другий (магістерський)

Спеціальність 121– Інженерія програмного забезпечення
(код і повна назва)

Освітньо-професійна програма Інженерія програмного забезпечення
(повна назва)

ЗАТВЕРДЖУЮ:
Зав. кафедри
_____
(підпис)
«_____»_____ 20 ___ р.

ЗАВДАННЯ
НА АТЕСТАЦІЙНУ РОБОТУ

Студентові Нємцову Микиті В'ячеславовичу
(прізвище, ім'я, по батькові)

1. Тема роботи Дослідження методів оптимізації, які використовуються у компіляторах коду
затверджена наказом по університету від "27" березня 2020 р № 473 СТ
2. Термін подання студентом роботи до екзаменаційної комісії 13 травня 2020 р.
3. Вихідні дані до роботи *ОС Ubuntu, генетичний алгоритм, алгоритм пошуку сходженням до вершини, жадібний алгоритм, C++, Баєсові нейронні мережі, адаптивна компіляції, ітеративна компіляція*

4. Перелік питань, що потрібно опрацювати в роботі *мета роботи, аналіз проблемної галузі і постановка задачі, опис існуючих інструментів оптимізації параметрів компілятора, використовувані методи та алгоритми, опис розробленої програмної системи, аналіз можливих застосувань*

5 Консультанти розділів роботи

Найменування розділу	Консультант (посада, прізвище, ім’я, по батькові)	Позначка консультанта про виконання розділу	
		підпис	дата
Спецчастина	доц. каф. ПІ Каук В.І.		

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Терміни виконання етапів роботи	Примітка *
1.	Аналіз предметної галузі	29 березня 2020 р.	
2.	Огляд існуючих методів	10 квітня 2020 р.	
3.	Методи підвищення ефективності оптимізації компілятора	20 квітня 2020 р.	
4.	Підготовка пояснювальної записки	25 квітня 2020 р.	
5.	Спецчастина	26 квітня 2020 р.	
6.	Підготовка презентації та доповіді	30 квітня 2020 р.	
7.	Попередній захист	30 квітня 2020 р.	
8.	Нормоконтроль, рецензування	05 травня 2020 р.	
9.	Занесення диплома в електронний архів	08 травня 2020 р.	
10.	Допуск до захисту у зав. кафедри	12 травня 2020 р.	
* заповнюється вручну після виконання чергового пункту			

Дата видачі завдання __29___ _____березня_____2020 р.

Студент _____
(підпис)

Керівник роботи _____доц. каф. ПІ Каук В.І._____
(підпис) (посада, прізвище, ініціали)

РЕФЕРАТ / ABSTRACT

Атестаційна робота магістра містить: 94 с., 52 рис., 27 джерел, 3 додатка, 1 таблицю.

МЕТОДИ ОПТИМІЗАЦІЇ ПРОГРАМНОГО КОДУ, КОМПІЛЯТОР, ОПЦІЇ КОМПІЛЯЦІЇ, ГЕНЕТИЧНИЙ АЛГОРИТМ, АЛГОРИТМ СХОДЖЕННЯ ДО ВЕРШИНИ, ІТЕРАТИВНЕ ПЕРЕТВОРЕННЯ

Метою роботи є дослідження методів оптимізації, які використовуються у компіляторах коду, а також аналіз і оптимізація методів вибору опцій компіляції.

Методи розробки базуються на програмному додатку для ітеративної компіляції програмного коду ACOVEA та його похідному інструменті для збору метрик про процес компіляції.

В результаті було розглянуто методи оптимізації, які використовуються у компіляторах коду та спроектовано підхід до оптимізації алгоритму вибору опцій компіляції.

SOFTWARE OPTIMIZATION METHOD, COMPILER, COMPILATION OPTIONS, GENETIC ALGORITHM, HILL-CLIMBING ALGORITHM, ITERATIC TRANSFORMATION

The purpose of the study is to investigate the optimization methods used in code compilers, as well as to analyze and optimize methods for selecting compile options.

Development methods are based on the software application for iteratively compiling code ACOVEA and its derivative tool for collecting metrics about the compilation process.

As a result, the optimization methods used in the code compilers were considered and a plan for optimizing the algorithm for selecting the compilation options was designed.

ЗМІСТ

ВСТУП

Тема підвищення продуктивності програміста є дуже важливою в наш час. Для того, щоб вони могли бути повністю зосереджені на реалізації бізнес-правил, які цікавлять замовника більш за все, створюється безліч інструментів та практик. Так наприклад, Visual Studio [1], яка була розроблена компанією Microsoft, звільняє розробника від необхідності створювати шаблони класів та проектів. Вона сама може це створити за нього, а йому буде потрібно лише наповнити їх відповідними правилами. Чи автоматично створення файлів розгортки в доповненні редактору коду Rider [2] від компанії Jetbrains [3].

Такі тенденції оптимізації продуктивності торкнулись навість мов програмування. З появою високорівневих мов, розробники перестали піклуватись про звільнення пам’яті при видаленні об’єкту, оскільки за нас це зробить компілятор мови. Ми можемо не використовувати деякі мікро-оптимізації, оскільки компілятори коду можуть їх зробити самі. Так наприклад, при невірному порядку блоків коду, компілятор сам може все перенести для того, щоб ваша програма одразу потрапляла в необхідний блок і виконувалась швидше. Те, що раніше необхідно було робити самим, тепер автоматично виконується без відома програміста.

Авжеж, такий потужний механізм не завжди потрібен при розробці програмного забезпечення. Інколи буває так, що оптимізацією жертвують на користь зручності чи читаності. Саме тому, при виконанні перетворення, в нас є можливість налаштовувати поведінку компілятора. Ми можемо задавати опції, які будуть вказувати на те, чи необхідно виконувати деякі оптимізації чи ні.

На жаль, такий інструмент не завжди працює так як того хоче програміст. Наприклад, деякі команди не рекомендовані, оскільки призводять до збільшення об’єму займаної пам’яті, хоча загалом прискорюють продукт. Як результат, користувач має більш швидкий додаток, але заявлений розмір не є дійсним.

Іншим прикладом є команди, які полегшують процес розробки, оскільки дозволяють змінювати код динамічно, але при цьому негативно впливають на продуктивність. Окрім небажаного результату виконання оптимізації, команди налаштування компілятора можуть давати позитивний результат приросту швидкості на новітніх системах, але при цьому не працювати для систем, які були створені давно. Це пов’язано із тим, що деякі оптимізації компілятора є специфічними для процесору.

І таких опцій налаштування в різних типах компіляторів безліч. Через це, необхідно мати чітке уявлення про те, які методи оптимізації компіляторів слід використовувати в тих чи інших випадках та як вони можуть бути налаштовані для досягнення найбільшої можливої оптимізації програмного продукту. Оскільки використання опцій оптимізації без розуміння їх взаємодії та неявних дій може призвести до втрат у продуктивності, а як результат до витрат на дослідження та усунення проблем. Саме тому, можна зробити висновок, що тема магістерської наукової роботи є актуальною.

Метою даної роботи є дослідження інформації про методи оптимізації використаних при компіляції програмного коду, способи їх налаштування, їх недоліки, переваги та оптимізація роботи обраного способу.

Зважаючи на все вищесказане, можна затвердити, що об’єктом дослідження даної роботи є процес оптимізації вибору параметрів використаних при перетворенні програмного коду в набір машинних команд. Предметом дослідження є методи налаштування параметрів оптимізації, які використовуються у компіляторах коду.

В якості методів, використаних для досягнення поставленої мети, було використано дослідження та порівняння існуючих методів вибору параметрів оптимізації, тестування їх спільного використання на предмет отримання поліпшення продуктивності.

Областю застосування результатів даної роботи є будь-яка сфера в якій використовують засоби для компілювання та виконання програмного коду.

1 АНАЛІЗ ПРЕДМЕТНОЇ ГАЛУЗІ ТА СТАНУ РОЗВ'ЯЗАННЯ ПРОБЛЕМИ

1.1 Актуальність роботи

У сучасному світі невід’ємною частиною кожної людини є електронні пристрої. Вони займають таку велику частину нашого життя, що тепер уявити світ без смартфонів, ноутбуків чи високо-швидкісних мереж зв’язку майже неможливо. Ці пристрої завжди поряд з людиною і для задоволення усіх її потреб, вони повинні відповідати усім її вимогам. Для задоволення якомога більшої кількості вимог, пристрої сильно еволюціонують, додаючи у складності програмного та апаратного забезпечення.

Такий темп зростання складності залишив дуже великий відбиток не тільки на самих девайсах, а і на тому, як працюють сучасні розробники програмного забезпечення. Тепер навіть текстовий редактор для створення маленької програми, це складний інструмент, який має безліч компонентів. Такі речі не є обов’язковими, тобто все ще можливо написати програму, використовуючи блокнот Windows, але використовують їх для підвищення продуктивності. Так, наприклад, Visual Studio Code, розроблена компанією Microsoft, може бути використана не тільки для редагування коду, а ще і для управління доставкою програмного забезпечення на сервери, чи перенесення коду до центрального сховища [4], де з ним зможуть працювати інші.

Окрім змін в інструментах розробки, дуже сильно стали змінюватись самі мови програмування та способи їх виконання. Розробники не пишуть на мовах програмування, таких як Assembler, які дозволяють працювати з регістрами напряму, а навіть ті, які ще залишились починають мінятися. Зараз використовують високорівневі мови програмування, які мають додатковий рівень абстракції над командами процесору. Прикладами таких мов є C, Pascal та інші. Такий код транслюється з мови програмування в проміжний, а далі в команди процесору. Такий прошарок звільняє нас від необхідності постійно

тримати в голові додаткові операції, які потрібні при роботі. Як приклад, можна навести процес завершення використання непідконтрольних ресурсів, таких як файл чи мережеве з'єднання. Через таку просту помилку, безліч компаній потрапляли у ситуацію витоку пам'яті та втрачали велику кількість грошей. Окрім цього, код на таких мовах програмування виглядає набагато краще, адже зосереджений тільки на реалізації бізнес-логіки.

Крім підвищення продуктивності програмістів, високорівневі мови програмування мають ще одну перевагу. Код на зазначених мовах не можна одразу виконувати на процесорі, а спочатку необхідно перевести його в набір машинних команд за допомогою компілятора. Це дозволило розробникам мов програмування, вбудувати механізми перетворення коду на етапі компіляції. Таким чином, при процесі перенесення коду на машинні команди, компілятор може перевірити вхідний текст програми та провести її оптимізацію, замінив деякі конструкції.

Загалом, компілятор зі здатністю оптимізації не є новиною в світі програмування. Перші компілятори, які додавали корективи у програмний код датуються ще 1970 роками, а саме компілятор для системної мови програмування «BLISS» [5]. Він мав деякі проблеми з правильним транслюванням коду, але вже використовував методи оптимізації. Цей спосіб поліпшення працездатності програми настільки сподобався співтовариству, що запустив новий тренд на оптимізацію при транслюванні, яка тримається навіть зараз.

За декілька останніх десятиліть, кількість мов з використанням такого підходу дуже сильно зросла, а з появою мікро-сервісних архітектур та інтернету речей, питання оптимізації стало ще важливішим. Саме тому розвиток оптимізуючих компіляторів відбувається постійно. Окрім, звичайного привнесення оптимізації в процес компіляції, розробникам стали доступні методи налаштування тих самих компіляторів. Таким чином, залежно від параметрів робочої станції, можна налаштовувати окремі правила, які компілятор буде використовувати при оптимізації програмного коду.

В більшості випадків, для налаштування компіляторів з використанням оптимізації достатньо використовувати рівні оптимізації. Вони наявні в майже всіх новітніх компіляторах, та представляють собою набір налаштувань, які були визначені розробниками заздалегідь. Чим більший рівень оптимізації, тим сильніше вона змінює кодову базу для отримання вищої продуктивності. Окрім того, що користувач компілятора може використовувати рівні, йому ще доступні окремі опції, які він може налаштовувати і які мають вплив на окремі прийоми оптимізації.

Нажаль, така гнучкість використання, може обернутись проти самого розробника, оскільки не завжди застосування певного правила призводить до оптимізації. Так наприклад, в мові програмування C++ є можливість налаштувати компілятор таким чином, що він буде використовувати динамічну компоновку. Компоновкою називають процес об’єднання декількох модулів з бібліотек в один файл, який потім буде виконуватись. Динамічна компоновка дозволяє компілятору додавати модулі не перед етапом компіляції, а у момент виконання. Це повинно призвести до скорочення розміру результативної програми. Однак існує такий спосіб оптимізації простору, який призведе до зниження продуктивності за рахунок необхідності включення бібліотеки перед виконанням функції.

Крім того, що компілятори з використанням оптимізації програмного коду інколи призводять до неявного результату, існує проблема пов’язана з типом системи під яку проходить транслювання. В залежності від типу цільової системи, оптимізації, які використовують компілятори можуть працювати з різним результатом. Саме тому, програмісти часто зустрічаються з ситуацією коли на цільовій і альтернативній платформах, приріст продуктивності програмного коду різний. В таких ситуаціях починають використовувати різні методи динамічного налаштування компілятора коду. Завдяки цьому, розробники програмного забезпечення можуть змінювати налаштування оптимізації компілятора, базуючись на цільовій платформі автоматично. На жаль, такий підхід має низку проблем. Однією з найважливіших є те, що зазвичай не

відомо який набір налаштувань необхідно виконувати на платформі. Тому пошук перетворюється на процес спроб та помилок. Така поведінка може завдати шкоди великому бізнесу, оскільки при переході на інші платформі та операційні системи, може бути зміна в продуктивності, а для знаходження набору підходящих налаштувань, необхідний час та гроші.

Через все зазначене вище, можна стверджувати, що питання вибору методів оптимізації компіляторів, які дають максимальний приріст до продуктивності та можуть бути використані на різних системах досі є відкритим та актуальним.

1.2 Аналіз роботи оптимізуючого компілятора

Як було зазначено вище, компілятор з використанням оптимізації, може досить сильно перетворити та прискорити наш програмний код. Однак, для того щоб це все здійснити, йому потрібно пройти через ряд важливих кроків.

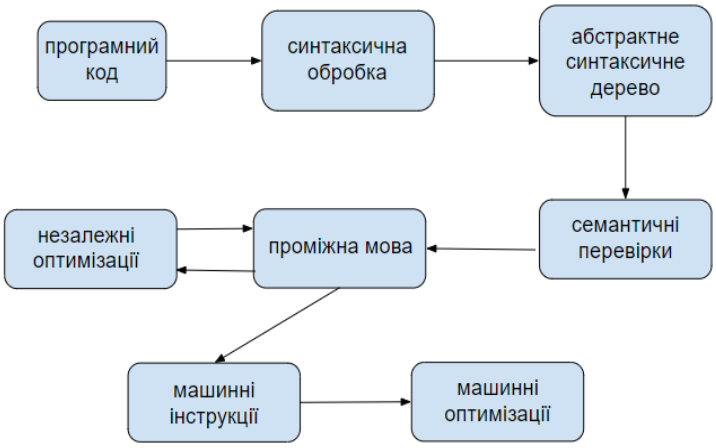


Рисунок 1.1 – Принцип роботи компілятора з використання оптимізації

Даними для вводу, чи старту компілятора є програмний код [6]. В залежності від мови програмування чи фреймворку, код буде упакований в різні структури та може мати різний вихідний тип. Першою стадією, яку проходить

програмний код називають синтаксична обробка. На цьому кроці, код перетворюється на набір спеціальних вираз, об’єктів, з якими компілятор може працювати. Це зроблено тому що, вхідний код завжди представлений у вигляді строки. Після того, як ми отримали набір токенів, ми можемо перевести його в абстрактне синтаксичне дерево. На цьому етапі, ми визначаємо чи є синтаксичні помилки. Якщо вони присутні, компіляція завершується з помилкою.

Після того, як ми провели синтаксичну обробку, починається процес семантичних перевірок. Цей процес впевнюється, що вхідний код відповідає всім обмеженням мови програмування. Для перевірки він використовує створене раніше абстрактне синтаксичне дерево та спеціальну таблицю символів. Якщо який-небудь тип невірний, компіляція завершується з помилкою перетворення типів даних.

Після того, як компілятор впевнився у тому, що код відповідає усім семантичним нормам, він починає процес перетворення, який тісно перетинається з процесом оптимізації. Річ у тому, що до того як перетворити код на мові програмування високого рівня на машинний, компілятор спочатку генерує код на проміжній мові. Для різних компіляторів та фреймворків цей код є різним. Так для Microsoft .Net Framework, цей код називається «Microsoft intermediate language» [7]. Після генерації програмного коду, компілятор проводить його аналіз та використовує результати для оптимізації. На цьому етапі він використовує набір незалежних від архітектури оптимізацій. До таких можуть відноситись перестанови чи заміни виклику функції на код функції. Оптимізації відбуваються на цьому кроці, оскільки наявна гарантія того, що вони можуть бути використані на будь-якому процесорі та будуть однаковими для будь-якого коду, оскільки проміжна мова є єдиною для всіх типів мов програмування, які підтримує компілятор.

Змінюючи вхідний код програми, компілятор повинен завжди піклуватись про те, щоб у результаті не відбулося зміни в роботі додатку. Це, є однією з найскладніших проблем компілятора з використання оптимізації.

Провівши необхідну оптимізацію, але ще не виконавши оптимізацію, яка залежить від кінцевої архітектури машини на які буде працювати додаток, компілятор починає генерацію машинного коду або байтного коду. Саме набір інструкцій буде виконуватись на цільовій машині. Однак, перед тим як почати виконання, компілятор проводить набір оптимізацій, які специфічні для конкретної системи. Якщо б компілятор виконував всі оптимізації без розподілення на системи, він міг би порушити один з найважливіших своїх законів, а саме – прискорювати виконання програмного коду. Пов’язано це з тим, що деякі системи можуть по-різному виконувати операції та якщо на один системах, оптимізація надасть приріст у продуктивності, не виключно що на іншій системі ця сама команда призведе до деградації продуктивності.

Враховуючи вищесказане, можна зазначити що на рівні з проблемою вибору опцій оптимізації які не залежать від архітектури кінцевої системи, існує проблема вибору налаштувань, залежних від системи, оптимізацій. Шляхи вирішення даної проблеми будуть розглянуті в даній роботі.

1.3 Аналіз існуючих типів оптимізації компілятора

Процес оптимізації програмного коду компілятором це дуже складна операція. Зазвичай, при використанні цієї техніки, час затрачений на збірку проекту досить сильно зростає. Пов’язано це з тим, що оптимізувати можна як строки коду так і цілі модулі.

Зараз компілятори використовують наступні типи методів оптимізації [8]:

– оптимізація «вічка». Даний тип оптимізації використовується для поліпшення продуктивності малих об’ємів коду, таких як метод, чи блок «if». Він оперує такими правилами, які дозволяють йому змінювати арифметичні операції, та видаляти непотрібний код. Такий тип оптимізації дає невеликий приріст оптимізації.

– локальні оптимізації. Цей тип оптимізації використовують для блоків коду. Він не дає великого приросту продуктивності, проте обчислення цього типу при компіляції досить швидке, за рахунок чого, він майже не має впливу на час трансляції програми.

– глобальні оптимізації. Даний тип оптимізації відноситься до «тяжких», оскільки виконується над цілим методом програми. Це дає змогу мати більший контекст інформації і більш вдало змінювати код програми з ціллю оптимізації. Такий підхід досить сильно впливає на час перетворення коду, тому має використовуватись вибірково.

– циклічна оптимізація. Такий тип оптимізації використовується для циклічних конструкцій коду і означає цілу низку змін. Наприклад, якщо в коді присутні два цикли, причому один знаходиться всередині другого, компілятор може їх об’єднати, тим самим зменшив складність алгоритму. Окрім об’єднання, циклічна оптимізація може використовувати векторизацію та паралельність для обробки масивів даних.

– оптимізація магазину. Даний тип оптимізації при компіляції програмного коду, використовують як службовий для інших типів. Він дозволяє присвоєння змінних до того моменту, коли це дозволяють блокування інших потоків.

– між-процедурна оптимізація. Цей тип оптимізації програмного коду є одним з найдійовіших. Пов’язано це з тим, що в даному сценарії відбувається оптимізації всієї програми, а не окремої частини коду. Такий підхід дозволяє замінювати цілі функції, оскільки контекст є досить великим. Через це, в результативному коді даного метода можна зустріти заміни викликів функцій на копію коду цих самих методів.

– оптимізація машинного коду. Такий тип оптимізації, використовують після отримання остаточного машинного коду програми. На цьому етапі, виконують ті оптимізації, для яких потребується увесь код програми та його контекст. Дані типи оптимізації є одними з найбільш ефективних. До таких операцій можна віднести стиснення макросів.

Кожен з вказаних типів оптимізації представляє собою набір опцій компілятора, які використовують оптимізації цього типу. Оскільки кількість опцій досить велика і у більшості випадків не потрібні всі опції одразу, розробники компіляторів додали два способи налаштування використаних оптимізацій.

Першим способом є використання опцій, які представляють одну окрему оптимізацію. Цей спосіб дає можливість більш тонкого налаштування, однак оскільки кількість таких налаштувань велика, розробники зазвичай не використовують описаний спосіб. Все ж таки цей спосіб знайшов своє використання при підході, коли вибором параметрів оптимізації займається комп’ютер.

Іншим, більш популярним, серед розробників підходом до вибору методів оптимізації стало використання профілів. Кожен профіль це набір опцій оптимізації, які не груповані за типом оптимізації. Зазвичай, вони груповані за тим наскільки сильно компілятор може змінити вхідний програмний код. Чим вищий рівень, тим більшу кількість перетворень компілятор використає і тим швидшим, в теорії, буде результативний код.

1.4 Аналіз причин які впливають на оптимізацію

Зважаючи на всі типи оптимізації які присутні при компіляції програмного коду, існує безліч причин, які впливають на результативний приріст продуктивності.

Перш за все, найбільший вплив на результат оптимізації привносить сама цільова машина. Зазвичай, код пишуть таким чином, щоб він виконувався на різних процесорах та операційних системах, це призводить до того, що деякі блоки коду трансформовані з одними й тими самими правилами оптимізації, виконуються за різний час та мають різну завантаженість процесу та пам’яті.

Саме через цю причину з’явились дві різні групи оптимізації: машино-незалежні та залежні оптимізації компілятора. Перший тип був розглянутий у минулих розділах та включає в себе перетворення коду через його розподілення та перенос.

Машино-залежні оптимізації виконуються вже після генерації коду. Зазвичай, саме цей тип оптимізації дає найбільший приріст у продуктивності. Він включає в себе такі зміни як оптимізації розподілень регістрів, яка дозволяє більш оптимально вибирати де саме розміщувати значення змінних при виконанні програмного коду. Окрім цього, цей тип, включає в себе такі оптимізації як вибір виклику одного з двох методів, які вирішують одну й ту саму задачу. Як можна помітити з усього вищезазначеного, машино-залежні оптимізації більш націлені на зміну та конфігурування цільової машини чи вибору методів з вже-генерованого програмного коду.

При стверджені, що цільова машина впливає на результат продуктивності оптимізації компілятора, варто мати на увазі одразу декілька речей, а саме архітектура процесору та архітектура пам’яті машини.

Окрім залежних від машини, впливати на результат проведення оптимізації можуть самі набори використаних опцій. Деякі опції оптимізації компілятора залежать одна від одної, тому ми не можемо їх використати одночасно. Це може призвести до погіршення продуктивності вихідного коду.

Також, фактором, який впливає на результат, може стати послідовність оптимізації в фазах, виконаних при процесі перетворення. До кодогенерації, виникає цикл з перетворення та перевірки. Оптимізуючий компілятор, отримує аналіз коду, проводить деякі оптимізації та запускає генерацію звіту заново. Таким чином перший прогін оптимізації, може виконувати такі операції, які будуть перешкоджати виконанню операцій оптимізації іншого прогону. Окрім цього, якщо порядок двох фаз перемикається, одна фаза виконує оптимізацію, яка може позбавити іншу фазу можливостей на оптимізацію.

1.5 Аналіз інструментів для налаштування оптимізації компілятору

1.5.1 Оптимізація з використанням машинного навчання

За останні декілька років, машинне навчання почало відігравати дуже велику роль у будь-які сфері нашого життя. Його використовують в комп’ютерних іграх, в камерах смартфонів при обробці зображень та іншому. Оптимізації компілятора не стали винятком.

Розробники почали роздивлятись способи налаштування компіляції програмного коду, але спочатку технології дозволяли проводити налаштування параметрів лише уже оптимізованого коду. З появою машинного навчання їм відкрилась можливість почати, оптимізувати не результативний набір параметрів, а сам процес вибору. Таким чином, почали з’являтись методи, які автоматично налаштовували моделі [9].

Починалось все з того, що модель вчили на якійсь тестовій вибірці опцій та додатків. У результаті, отримували навчену модель, яка могла передбачати які опції необхідно використовувати в залежності від додатку та функцій, які він використовує.

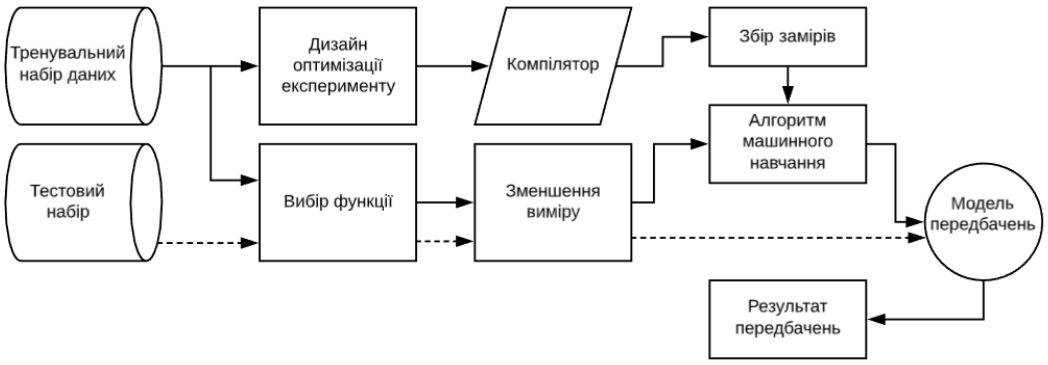


Рисунок 1.2 – Принцип роботи компілятора з використанням машинного навчання

Протягом останніх років, даний принцип роботи широко вивчається та покращується. Було використано та запропоновано велику кількість опцій, які покращили модель, її здатність передбачати необхідні опції та надавали приріст у продуктивності результативного програмного коду.

Зараз, останньою розробкою в цьому напрямі є використання Байєсовської нейронної мережі для визначення функцій. Цей підхід широко використовують в таких інструментах як Cobayn [10]. Порівнюючи даний підхід з його попередником, методом турнірних прогнозів, приріст продуктивності складає 11 відсотків, згідно з дослідженнями.

Однак, даний спосіб налаштування оптимізації, має деякі недоліки. Перш за все, складність такого підходу в декілька разів вища за інші. Це пов’язано з необхідністю тренування та налаштування моделей, необхідністю мати знання в цій області. Також, зазначений підхід має недолік в тому, що час на навчання моделей є дуже високим. Загалом, тренувальні набори даних складаються з великої кількості опцій оптимізації, які ніяк не пов’язані між собою. Це призводить до того, що для навчання моделі, яка буде давати більш реальні та корисні дані про вибір оптимізації, необхідна велика кількість часу. Окрім цього, модель потрібно буде тренувати для кожної цільової архітектури, щоб вона могла налаштовувати машино-залежні опції.

1.5.2 Оптимізація з використанням профілів

Описані проблеми компіляторів з використанням оптимізації з’явились ще досить давно та були добре вивчені. Один з найпопулярніших способів боротьби з проблемами при виборі необхідних опцій компілятора є налаштування за допомогою профілів. Цей спосіб також називають ітеративним.

Даний спосіб полягає у тому, що розробник вбудовує в свій додаток спеціальний код для збору інформації про виконання програми. Такий код може збирати різну інформацію про час виконання програми.

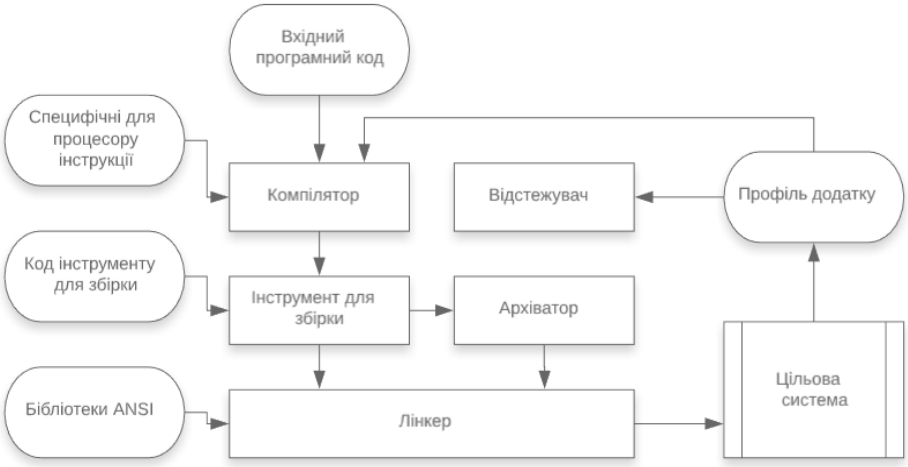


Рисунок 1.3 – Принцип роботи компілятора з використання профілів

Для того, щоб використати зазначений спосіб налаштування оптимізації компілятора, нам необхідно зробити збірку з мінімальним набором інструментів для профілювання та витримати деякий час на те, щоб він міг зібрати усі необхідні метрики. Як тільки вони будуть зібрані чи при наступній збірці додатку, компілятор зможе використати ці дані, для того, щоб оптимізувати набір опцій. Варто зазначити, що чим більший набір метрик, тим вища вірогідність знайти правильний набір опцій. Таким чином, використовуючи досвід минулого запуску, компілятор зможе виконати тільки ті перетворення які він вважатиме за потрібні.

Загалом, оптимальність такого підходу досить висока. Порівнюючи результат його використання зі звичайними рівнями компілятора GCC [11], можна сказати, що приріст продуктивності складає 15 – 34%, що є дуже великим значенням. Перед всім, це досягається завдяки визначенню та вчасному усуненню опцій, які дають найменший результат.

На початку оптимізації, компілятор використовує якийсь зазначений заздалегідь набір опцій і підраховує скільки кожна займає часу та дає

продуктивності. В результаті, він отримує графік, який вказує на оптимальність використання опції.

Score	%Prev	%Base	%Best	Flags diff	Reduction cost
25.319	0.00%	0.00%	-20.06%	-O2	
25.627	1.22%	1.22%	-19.09%	-DEVAS_UNROLL_FACTOR=0	-27.10%
25.689	0.24%	1.46%	-18.89%	-mvectorize-with-neon-quad	-21.10%
25.622	-0.26%	1.20%	-19.11%	-mfpu=neon	-19.27%
26.839	4.75%	6.00%	-15.26%	-ftree-vectorize	-19.03%
28.431	5.93%	12.29%	-10.24%	-fvectorize-misaligned	-13.53%
29.561	3.98%	16.76%	-6.67%	-fprefetch-loop-arrays	-9.25%
30.639	3.64%	21.01%	-3.27%	--param l1-cache-line-size=64	-6.31%
...					
31.515	0.32%	24.47%	-0.50%	-fno-if-conversion2	-0.27%
31.637	0.38%	24.95%	-0.12%	-funwind-tables	-0.45%
31.638	0.00%	24.96%	-0.11%	-fno-thread-jumps	-0.35%
*31.674	0.11%	25.10%	-0.00%	-fno-expensive-optimizations	-0.11%
31.565	-0.34%	24.67%	-0.34%	-fno-tree-ter	

Рисунок 1.4 – Графік оптимальності використання опцій в тестовому додатку

Даний спосіб хоча і дозволяє нам отримати спосіб авто-налаштування перетворень конфігурації оптимізації компілятора, все ж таки має низку недоліків.

По-перше, такий підхід не дає розробникам самостійно налаштовувати опції компілятору. Наприклад, може виникнути ситуація при якій, розробник програмного забезпечення буде впевнений, що опція яка включає оптимізація розподілу регістрів буде необхідна в даному випадку, а результати метрик будуть говорити о протилежному і в результаті, компілятор, базуючись на минулих даних, вимкне її.

Окрім цього, існує ще одна проблема автоматичного налаштування за допомогою профілів. Вона полягає у тому, що для того, щоб отримати профіль, компілятор повинен запустити програму. Це необхідно для того, щоб аналізатор міг порівняти та скомпільований код та зробити певний статистичний звіт, який потім можна буде використати на наступній ітерації компіляції. Для отримання кращого результату, додаток повинен відпрацювати деякий час та зробити велику кількість ітерацій, доки алгоритм пошуку не завершить своє виконання. Такі запуски програми дуже сильно впливають на результативний час.

1.5.3 Ефективна адаптивна компіляція

Описаний раніше ітеративний підхід при оптимізації параметрів компілятора, отримав досить широке розповсюдження та почав використовуватись в багатьох типах компілятору, незважаючи на описані проблеми. Однак про них не забули і паралельно намагались знайти шляхи їх вирішення.

Однією і мабуть найголовнішою проблемою ітеративного підходу був час, який необхідно було затратити на запуск та порівняння коду після кожної ітерації оптимізації. Це призводило до того, що процес можна було поділити на два однакових за часом процеси: аналіз статистичного звіту та порівняння. Вирішенням для цієї проблеми став віртуальний запуск скомпільованої програми. Без прямого запуску програми, аналізатор підраховує як багато інструкцій має виконуватись при певному сценарії і генерує статистичний звіт по цим даним. Такий підхід вже протягом довгого часу використовувався в аналізі покриття коду тестами.

Існує думка, що такий підхід до запуску коду робить статистичний аналіз менш точним та може не виконувати всі сценарії, оскільки кожен виток коду створює нові сценарії. Однак, останні дослідження в цій області довели, що степінь покриття коду таким підходом досить висока і складає 98.6%. Навіть маючи погрішність в 3 відсотки, переваги, які представляє цей спосіб занадто високі, оскільки час затрачений на виконання одного тесту оптимізації скорочується в 2 – 5 разів.

Останньою розборкою в цій області стала програмний додаток «ACME» для оптимізації опцій компілятору, розроблений в університеті Rice, який знаходиться у штаті Х’юстон, Об’єднані Штати Америки. Він використовує описаний ітеративний підхід в купі з віртуальним виконанням для знаходження найкращих параметрів оптимізації за мінімальний час.

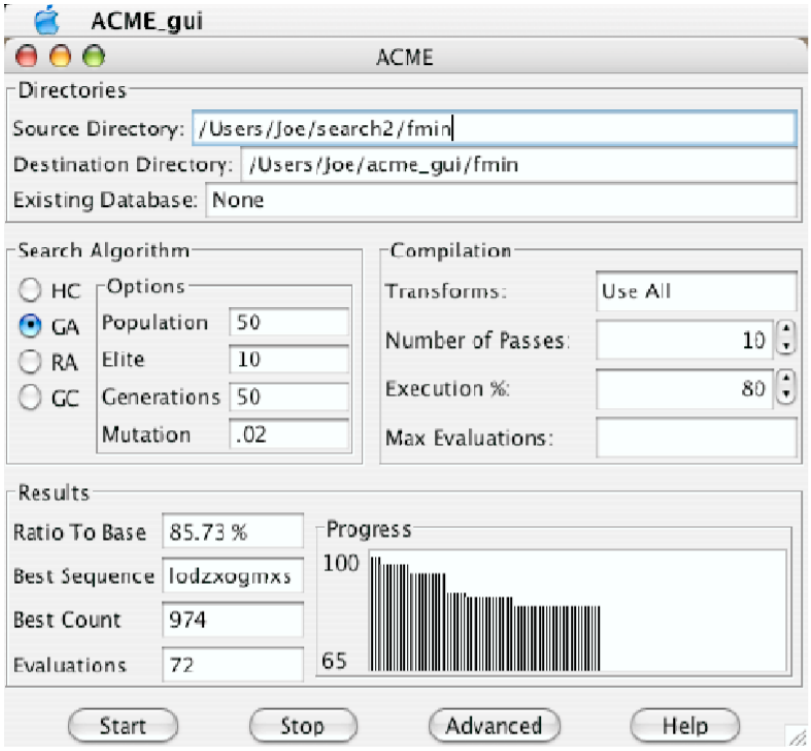


Рисунок 1.5 – Інтерфейс ACME

Для пошуку найбільш оптимальних опцій, базуючись на статистичних даних, додаток використовує модифікований спосіб віртуального виконання. Він полягає у тому, щоб не змінювати кожен шлях, який може модифікувати граф потоку керування, він будує один шлях, який може порівнювати графи. Після кожної зміни графу потоку керування компілятор буде виконувати цей шлях для виведення змін і оновлення нового графу.

Такий спосіб дозволяє проводити аналіз таких складних операцій, як очищення ітеративних блоків, які змінюють результативний граф потоку керування, оскільки для збору їх метрик необхідні реальні шляхи виконання, що дуже складно при віртуальному виконанні.

Окрім цього, зазначений додаток використовує один з чотирьох алгоритмів пошуку для знаходження найбільш оптимальних опцій з результатів прогону. На кожній ітерації, алгоритм повинен відбирати найбільш придатні опції, а ті, які погіршують продуктивність відкидати.

Наявні такі алгоритми:

- жадібний алгоритм;
- генетичний алгоритм пошуку;
- випадковий пошук простору;
- алгоритм випадкового нетерплячого спуску.

Використання такого додатку для оптимізації опцій компілятора дає зручний спосіб знаходження найбільш оптимального набору за придатний час, однак має шляхи поліпшення. Перш за все, можна покращити час знаходження оптимального набору за рахунок використання гібридних алгоритмів пошуку. Так, генетичний алгоритм пошуку [12] досить швидко знаходить рішення, яке задовольняє потребу, але для знаходження найкращого можливого набору параметрів можна використати інший алгоритм, такий як алгоритм пошуку сходженням до вершини [13]. Таким чином, можна використати спочатку один тип пошуку, для досягнення задовольняючого результату, а потім запустити інший алгоритм. Однак варто брати до уваги кількість оперативної пам’яті, яка необхідна для виконання такої зміни алгоритму. Оскільки копіювання великих об’ємів даних може сильно уповільнити виконання пошуку чи привести до ситуації, коли системні вимоги для компіляції програмного додатку будуть занадто високими.

Окрім відсутності динамічної зміни алгоритму, даний спосіб має ще один недолік, а саме обмеженість кількості циклів ітерацій при знаходженні оптимального набору. Через те, що алгоритм використовує випадкові зміни у вхідних даних для пошуку результативного набору операцій оптимізації, цей недолік має досить важливу архітектурну роль. Зазвичай, ми вказуємо наскільки гарне рішення потрібно знайти за допомогою методу, саме досягнення цього параметру і є умовою завершення пошуку. У разі неоптимальної ітерації циклу, ми можемо потрапити в ситуацію коли на пошук оптимального рішення потрібно забагато ітерацій. Саме максимальна кількість циклів може зупини алгоритм в цьому разі.

1.5.4 Порівняння методів оптимізації

Провівши аналіз кожного з доступних даний момент методів оптимізації підбору параметрів компілятора, виберемо параметри для їх порівняння, приведемо переваги та недоліки кожного з методів. Це дозволить вибрати метод підбору опцій компілятора який буде використано в прототипі додатку.

В якості параметрів оцінки будемо використовувати наступні характеристики:

а) необхідність запуску програми. Дана характеристика вказує на те, чи може бути використаний метод оптимізації без запуску програми. Це буває потрібно в разі використання оптимізації серверних додатків, оскільки запуск останнього означає повну готовність програмного додатку для поставки користувачу. Окрім цього, можливість провести оптимізації не запускаючи додаток досить сильно прискорює процес пошуку параметрів.

б) можливість паралельного обчислення. Характеристика вказує на те, чи може процес оптимізації виконуватись паралельно на декількох потоках процесору.

в) необхідність попереднього обчислення. Дана характеристика вказує на необхідність приготування пристрою для початку пошуку параметрів оптимізації. Наприклад: нейронна мережа повинна бути натренованою перед тим як її буде використано.

г) оптимізація параметрів залежних від платформи. Ця характеристика вказує на можливість оптимізувати параметри компілятора, які залежать від цільової платформи.

д) Швидкість виконання для tree[14]. Кількість часу необхідного кожному з методів оптимізації компілятора для того, щоб знайти набір оптимальних параметрів для компіляції тестового алгоритму tree. Якість результативного набору оптимізації не буде врахована, оскільки залежить від алгоритму пошуку, а не від методу оптимізації.

Використовуючи характеристики описані вище та розглянуті методи оптимізації було створено порівняння, яке відображено в таблиці 1.

Таблиця 1. – Порівняння методів оптимізації

Характеристика	Оптимізація з використанням машинного навчання	Оптимізація з використанням профілів	Ефективна адаптивна оптимізація
Необхідність запуску програми	Немає необхідності	Необхідно	Немає необхідності
Можливість паралельного обчислення	Неможливо	Неможливо	Можливо
Необхідність попереднього обчислення	Необхідно	Немає необхідності	Необхідно
Оптимізація параметрів залежних від платформи	Можливо	Можливо	Можливо
Швидкість виконання для tree	1203	6733	3301

Як можна побачити на таблиці 1, кожен з методів має свої недоліки та переваги. На перший погляд метод оптимізації з використанням нейронних мереж виглядає швидшим, однак варто мати на увазі, що перед початком оптимізації, нейронна мережа повинна бути навчена. Окрім цього, складність реалізації цього методу набагато вищий за інші.

Метод ефективної адаптивної компіляції, є більш швидшим за метод профілів, оскільки він використовує віртуальний запуск програми, хоча і

використовує схожі принципи оптимізації. Окрім цього, даний метод може бути використаний як самостійно так і в комбінації з іншим методом оптимізації, таким як нейронні мережі.

Враховуючи проведений аналіз та порівняння методів оптимізації компіляції, можна зробити висновок, що метод ефективної адаптивної компіляції є більш привабливим для оптимізації, оскільки його область використання ширша за інші описані методи та результат його оптимізації, може бути використаний в інших методах, таких як нейронні мережі.

1.6 Постановка задачі

Метою даної роботи є отримання інформації про способи налаштування параметрів оптимізації при компіляції програмного коду, та оптимізація обраного способу.

Виходячи з усього вищезазначеного, можна визначити наступні задачі:

а) розглянути алгоритми пошуку, з’ясувати їх переваги та недоліки:

1) генетичний алгоритм;

2) алгоритм пошуку сходженням до вершини;

3) жадібний алгоритм

в) оптимізувати обраний алгоритм пошуку;

г) оптимізувати ітеративний спосіб компіляції з використанням обраних алгоритмів;

д) вибір засобів розробки;

е) провести експерименти та порівняти експериментальні дані з вже існуючими аналогами.

В результаті аналізу стану ситуації в науковій сфері та інструментів для її вирішення, було прийнято рішення використовувати алгоритм адаптивної ефективної компіляції. Це пов’язано з тим, що він може бути використати і в інших методах, саме тому його оптимізація призведе до оптимізації інших підходів, таких як машинне навчання.

Оскільки зазначений метод оптимізації може використовувати декілька алгоритмів пошуку окремо, було прийнято рішення дослідити оптимальність використання трьох алгоритмів для досягнення однієї цілі і провести оптимізацію додатку, використавши обраний алгоритм.

Таким чином, в цьому розділі було проведено аналіз стану розв’язання проблеми, визначені недоліки існуючих рішень та окреслені шляхи їх усунення. Було сформовано науково-технічну задачу та обґрунтовано цілі дослідження проведеного в даній роботі.

2 ОПИС ТА АНАЛІЗ ПРОВЕДЕНИХ ЕКСПЕРИМЕНТАЛЬНИХ ДОСЛІДЖЕНЬ

2.1 Аналіз алгоритмів пошуку

Вибір необхідного алгоритму пошуку є ключовим завданням в оптимізації алгоритму адаптивної ефективної компіляції. Оскільки можна покращити час знаходження оптимального набору за рахунок використання гібридних алгоритмів пошуку.

2.1.1 Генетичний алгоритм

Першим алгоритмом, який необхідно роздивитись є генетичний алгоритм. Цей метод дуже схожий на процес природної еволюції, однак він є пристосованим для вирішення проблем і задач оптимізації. Загалом, ці типи алгоритмів засновуються на механізмах природнього відбору і використовуються в проблемах пошуку. Так само як і в реальному світі, дані алгоритми використовують еволюційний принцип виживання найбільш пристосованих особин. Такі алгоритми виконують низку певних дій по колу, доки не буде виконана умова. До дій, використаних в алгоритмі відносяться:

– ініціалізація. На цьому кроці проходить формування вихідної популяції. Відбувається вибір деякої кількості двійкових послідовностей фіксованої довжини, які мають назву – хромосоми.

оцінювання. На цьому етапі необхідно визначити наскільки обрані хромосоми є пристосованими. Для цього використовують спеціальну функцію пристосованості, яка розраховується для кожної хромосоми з зазначеної популяції. Більший отриманий результат означає вищу якість хромосоми. В залежності від типу задачі, змінюється форма функції. Прийнято вважати, що

функція пристосованості приймає ненульові значення вхідних параметрів. Окрім цього, відомо, що для розв’язання задачі, необхідно максимізувати отриману функцію. В залежності від того, чи виконує вихідна форма функції зазначені умови, використовують перетворення.

– перевірка умови зупинки. Цей етап використовується для визначення того чи був досягнений необхідний результат і чи можемо ми завершити роботу алгоритму. Умовою зупинки може бути різна характеристика результату і загалом, вона залежить від цілей алгоритму. Оптимізаційні задачі можуть використовувати як флаг зупинки те, що був отриманий результат не менше заданої точності.

Рисунок 2.1– Принцип роботи генетичного алгоритму

– селекція – це один з фінальних кроків алгоритму, протягом якого відбувається вибір хромосом, які підходять до створення наступної популяції. На цьому етапі результат функції пристосованості використовується для ранжування хромосом від наймень придатних до найбільш. Для створення наступного покоління, будуть обрані ті хромосоми, які мають найбільше значення результату.

Після того, як було вирішено, які хромосоми є найбільш придатними для створення нащадків, завершується процес створення батьківської популяції. До отриманої групи застосовують набір генетичних операторів, що дозволяє створити популяцію нащадків. Таким чином ми закінчуємо першу ітерацію і повертаємось на крок оцінювання пристосованості хромосом.

2.1.2 Алгоритм пошуку сходженням до вершини

Альтернативним способом пошуку може стати досить популярний алгоритм сходження до вершини. Він відноситься до сімейства алгоритмів локального пошуку та є технікою математичної оптимізації. Так само як і генетичний алгоритм, даний спосіб базується на ітеративному пошуку найбільш оптимального рішення. Після знаходження аби-якого рішення, алгоритм починає нову ітерації, змінюючи один з параметрів, використаних для досягнення рішення. Такий приріст буде виконаний доки новий крок алгоритму буде давати покращення в результатах.

Сходження знаходить оптимальні рішення в задачах лінійного програмування, для інших завдань алгоритм знайде тільки локальний оптимум (рішення, яке не може бути покращено переходом в сусідні вузли), яке не обов'язково буде кращим рішенням (глобальний оптимум) з усіх можливих рішень в (області допустимих рішень).

Як спроби подолати застрявання в локальному оптимумі можна спробувати почати пошук заново (тобто повторити локальний пошук) або використовувати більш складні схеми, засновані на ітераціях (як в ітеративному локальному пошуку), запам'ятовуванні в пам'яті (як в пасивному пошуку і пошуку з заборонами), або вимагають меншої запам'ятовування стохастичні модифікації алгоритму (такі як імітація відпалу).

Відносна простота алгоритму робить алгоритм популярним серед оптимізаційних алгоритмів. Він широко використовується в теорії штучного інтелекту для досягнення цільового стану з початкової точки. Метод вибору наступної точки і початкової точки може змінюватися, даючи ряд пов'язаних алгоритмів. Хоча більш просунуті алгоритми, такі як алгоритм імітації відпалу або пошук з заборонами можуть давати результати краще, в деяких ситуаціях сходження працює з тим же успіхом.

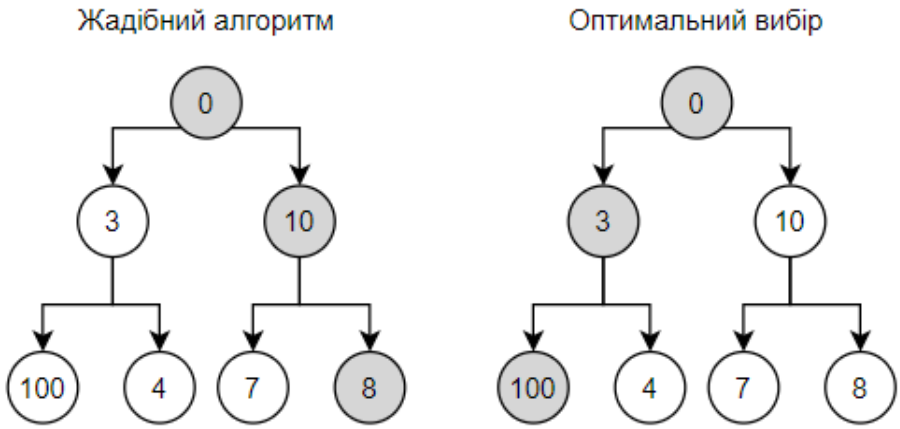
2.1.3 Жадібний алгоритм

Додатковим алгоритмом, який варто розглянути є жадібний алгоритм [15] пошуку. Загалом, цей тип знаходження рішення є альтернативою для алгоритму пошуку сходженням до вершини, оскільки використовує схожий принцип дії і є евристичним.

Цей тип алгоритму відрізняється тим, що на кожному кроці робить локально найкращий вибір в надії, що і глобальний результат буде також найкращим. Така поведінка досить сильно прискорює вибір параметрів. Однак, це призводить до того, що такий алгоритм як і алгоритм сходженням до вершини легко потрапляє в проблему хребтів і не може подолати набір локальних максимумів.

Окрім цього, даний тип алгоритму відрізняється своєю недалекоглядністю. Це пов’язано з тим, що алгоритм робить розрахунок тільки в одному напрямі та

лише на даних які є зараз. Він не може змінити нічого в минулому виборі. Саме тому, жадібний алгоритм не може бути використаним для деяких типів завдань оптимізації. Так, наприклад, проблема комівояжера буде зведена до того, що алгоритм буде вибирати найближче з невідвіданих місць. Оскільки наступні дані про сусідні міста можуть змінити оптимальність обраного кроку, алгоритм врешті решт обере не оптимальний шлях. Через описану поведінку при вирішенні проблем, описаний алгоритм не рекомендують використовувати в низці задач, віддаючи перевагу динамічним алгоритмам. Приклад неоптимального вибору жадібного алгоритму приведено на рисунку 2.2.



Рисунку 2.2 – Приклад неоптимального вибору жадібного алгоритму

Як можна побачити на рисунку 2.2. жадібний алгоритм може досить легко вирішити проблему локального максимуму, однак в глобальному плані його рішення не завжди приводить до найліпшого результату. На рисунку, жадібний алгоритм вибирав найбільш оптимальний варіант на даний момент, однак не враховував альтернативних шляхів і врешті решт обрав неоптимальний шлях.

Враховуючи вищезазначені проблеми можна прийти до висновку, що для знаходження оптимального набору опцій оптимізації на результатах роботи

генетичного алгоритму більш підходить алгоритм сходженням до вершини. Хоча цей алгоритм також має проблеми локальних максимумів та мінімумів, він є більш надійним, оскільки дозволяє прораховувати можливі шляхи та прораховувати один з варіантів до тих пір доки він є оптимальним.

2.2 Порівняння алгоритмів

Оскільки результати дослідження кожного з алгоритмів показали, що алгоритм сходження до вершини є більш перспективним та надійним за жадібний алгоритм, було вирішено не використовувати останній алгоритм. Через це, порівняння буде проводитись поміж двох алгоритмів для визначення порядку їх виконання. Загалом, як алгоритм пошуку сходженням до вершини так і генетичний алгоритм використовуються для пошуку в великих об’ємах даних, однак вони мають декілька вагомих відмін.

Алгоритм сходження до вершини має хороші показники в локальному пошуку. Починаючи з визначення початкової групи, вирішуючи кращу пошукову область до ітерації від рівня до рівня, результати кожного рівня беруться з найкращих і потім порівнюються, щоб вони могли отримати більш оптимальні.

Генетичний алгоритм є потужним і гнучким метаевристичним, а також відносно новим типом алгоритму, приймаючи ідею природного відбору та генетичних змін природним шляхом. Цей алгоритм відомий як інструмент, який може вирішувати комбінаторні методи оптимізації, такі як проблема комівояжера.

Однак незважаючи на їх можливе використання для вирішення одних проблем, дані алгоритми мають бути використані на різних масивах даних.

Як приклад, візьмемо класичну проблему комівояжера [16]. Всього будем використовувати в нашому тестуванні 8, 16 та 32 міста. Мовою програмування для реалізації алгоритмів оберемо C# і будемо використовувати бібліотеку

«GAF» [17], яка надає реалізовані генетичні алгоритми. Тестування проводиться на процесорі Intel Core i7 8750H, за тактовою частотою 2.21 GHz. Для налаштування генетичного алгоритму, будемо використовувати:

- популяцію 10;
- ймовірність перехрестя 60%;
- можливість мутації 50%;
- кількість генерацій 200.

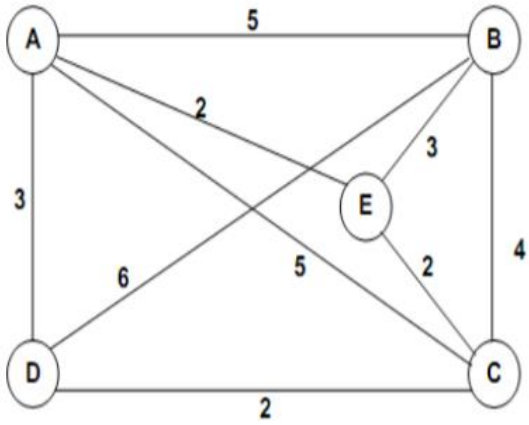


Рисунок 2.3 – Приклад вхідних даних проблеми комівояжера

Провівши по 20 тестів для кожного алгоритму, було отримано наступні результати:

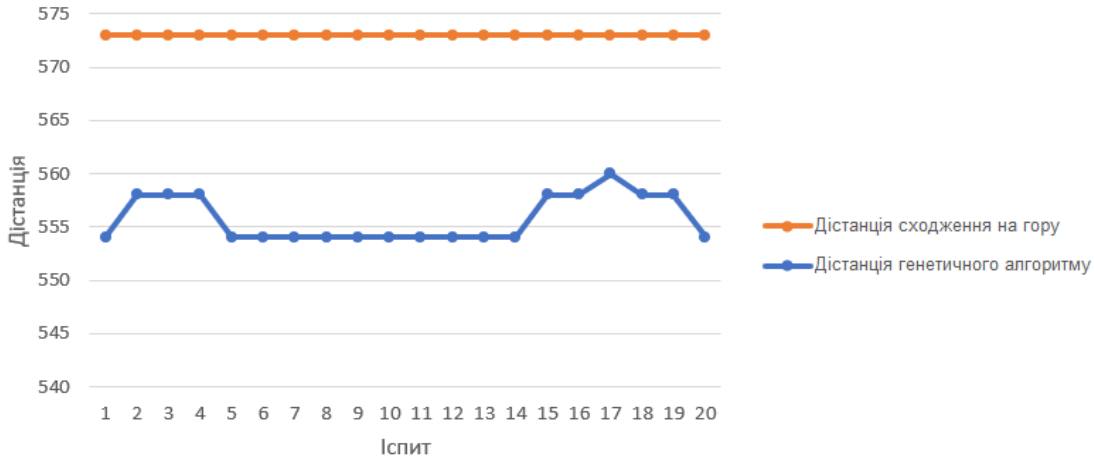


Рисунок 2.4 – Результати іспиту для 8 міст

Як можна побачити на першому тесті, при достатньо малому об’ємі даних, алгоритм сходження на гору дає більш оптимальний результат, оскільки всі прогони дають однакову вагу. Однак, варто відмітити, що результат генетичного алгоритму визначає більш дешевий шлях. Час виконання генетичного алгоритму – 820.412 секунд, для алгоритму сходження час склав 4.580 секунд.

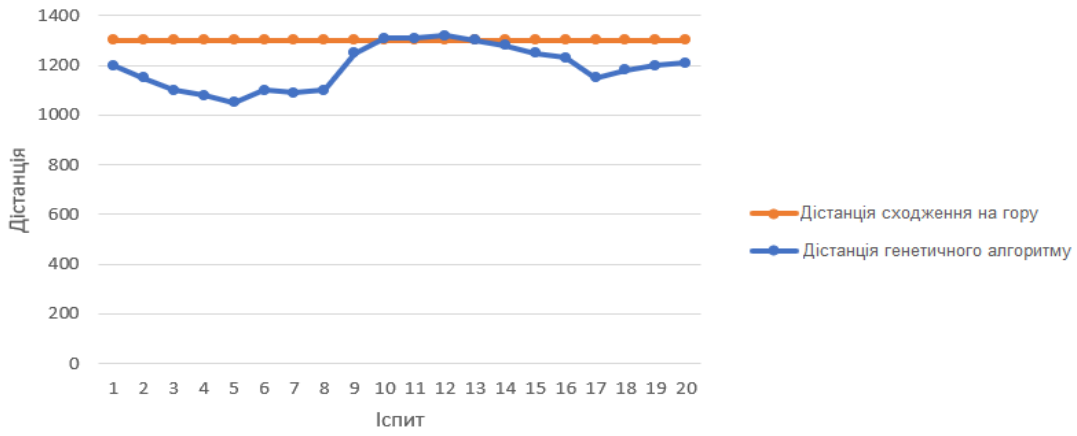


Рисунок 2.5 – Результати іспиту для 16 міст

При другому тестуванні, алгоритм сходження на гору знов дає більш оптимальний результат, оскільки всі прогони дають однакову вагу. Цього разу, генетичний алгоритм вже не знайшов більш оптимального шляху на кожному прогоні. Час виконання генетичного алгоритму – 3812.331 секунд, для алгоритму сходження час склав 73.212 секунд.

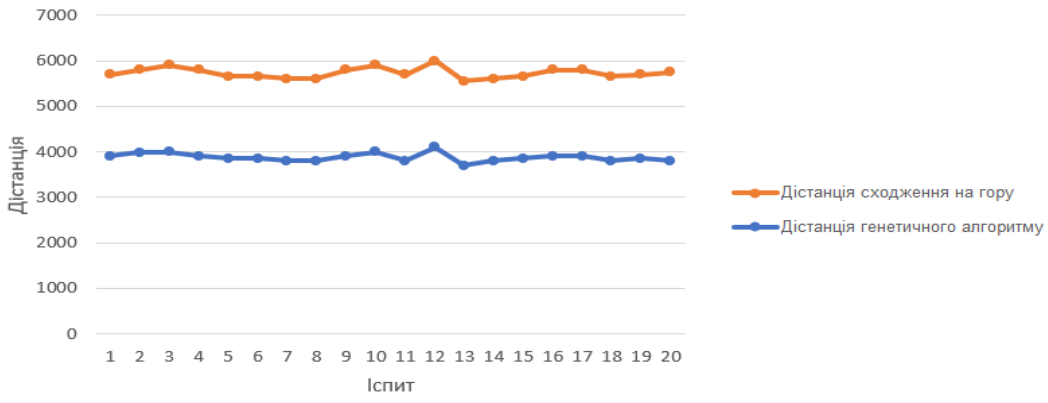


Рисунок 2.6 – Результати іспиту для 32 міст

Останній тест показав, що обидва алгоритми дали оптимальний результат. Час виконання генетичного алгоритму – 9875.311 секунд, для алгоритму сходження час склав 132.631 секунд.

Проаналізувавши результати тестувань двох алгоритмів можна зробити висновок, що генетичний алгоритм дає більш вдалі результати, але виконується повільніше за алгоритм сходження. В той час, алгоритм сходження є більш швидким за генетичний, але показав не найбільш оптимальний шлях. Це пов’язано з тим, що останній кожного разу знаходить точку локального мінімуму. Вирішенням цієї проблеми може стати обмеження кількість даних, та їх попередня фільтрація.

Зважаючи на проведений аналіз, можна зробити висновок, що використання алгоритму сходження для знаходження найбільш оптимального набору опцій налаштування компілятору є більш привабливим. Однак, для обмеження кількості вхідних даних та усунення можливих неточностей, слід використовувати генетичний алгоритм, який може знайти необхідний мінімум, який можна буде використати в якості вхідних даних до алгоритму сходження. Таким чином можна оптимізувати алгоритм пошуку, підвищити його точність та мати можливість пошуку за придатний час.

3 ОПИС ТА АНАЛІЗ ПРОГРАМНОГО ЗАСТОСУВАННЯ

3.1 Оптимізація адаптивної компіляції

Як було раніше зазначено, проблемою ефективної адаптивної компіляції є те, що використання одного методу пошуку призводить до різних проблем. Так, наприклад, використання генетичного алгоритму призводить до гарних результатів, проте час пошуку необхідного результату занадто великий і може бути оптимізований.

При використані алгоритму сходження на вершину, результати виконання оптимізації можуть бути отримані за помітно менший час, однак вони можуть бути неточними.

Враховуючи полярність зазначених вище проблем, можна зробити висновок, що два алгоритми – генетичний алгоритм та алгоритм сходження до вершини, можуть бути скомбіновані для часткового усунення проблем один-одного. На початку виконання будемо використовувати генетичний алгоритм для усунення варіантів, які точно не будуть виконуватись в результативній вибірці. Після цього, буде використаний алгоритм сходження до вершини, для знаходження локального максимуму, який представляє собою набір найліпших параметрів оптимізації компілятора. Такий підхід до поступового виконання спочатку генетичного алгоритму пошуку повинен зменшити вірогідність того, що алгоритм пошуку сходженням до вершини потрапить у локальний максимум та не може знайти найліпший результат.

Виходячи з проведеного аналізу, було прийнято рішення, використати два алгоритми поступово. [18] Такий підхід повинен оптимізувати роботу обох алгоритмів та зменшити час, затрачений на знаходження найкращої можливої комбінації опцій налаштування компілятора з використанням оптимізації.

3.2 Вибір інструментів розробки

В якості програмного продукту, який буде виконувати оптимізації, використаємо проект з відкритим кодом «ACOVEA» [19] версії 4.0, розроблений командою ентузіастів. Він дозволяє скомпілювати код з використанням ітеративної оптимізації. Окрім цього, він має можливість використання генетичного алгоритму для знаходження найбільш оптимального набору параметрів оптимізації.

Варто зазначити, що дане програмне забезпечення розповсюджується за ліцензією «GNU General Public License» [20]. GNU GPL — надає можливість розробнику копіювати, змінювати та розповсюджувати програмні додатки. Також надає зобов’язання, надавати користувачам всіх похідних програм ці самі права.

Результатом виконання пошуку набору оптимізації є набір опцій, які можливо використати при компіляції. Цей набір сортований за полем «Score» яке представляє собою вірогідність того, що використання даної опції призведе до збільшення продуктивності програми.

Окрім цього, додаток має можливість однозначно визначати чи є необхідність у деяких опціях. Так, кілька опцій в компіляторі однозначно призводять до поліпшення продуктивності. Якщо алгоритм знаходить їх, то вони помічаються як «MISSED». В разі невизначеності в необхідності використання тієї чи іншої опції, компілятор помічає їх як «WONDERED».

Після того, як бібліотека завершує виконання пошуку опцій компілятора, які можуть бути використані для оптимізації, результати роботи використовуються розробниками програмного забезпечення для того, щоб вручну налаштувати компілятор. Такий підхід гарний тим що є більш гнучким оскільки залишає за розробником остаточний вибір того, які опції оптимізації компілятора використовувати.

```
Legend:
   !!! MISSED   -- I'm of opinion these flags can _NOT_decrease_ performance of code any way
   ??? WONDERED -- I'm of opinion these flags definally _decrease_ performance of byte-code


 Score |  So?  | Switch (annotation)                  |
-------------------------------------------------------------------------------
  29.8 |  Yes  | -fcse-skip-blocks (-O2)              |
  29.6 |  Yes  | -fno-omit-frame-pointer (! -O1)      |   ??? WONDERED
  29.1 |  Yes  | -frerun-cse-after-loop (-O2)         |
  29.0 |  Yes  | -fno-thread-jumps (! -O1)            |   ??? WONDERED
  28.9 |  Yes  | -fmove-all-movables                  |
  28.0 |  Yes  | -minline-all-stringops               |
  27.6 | Maybe | -fno-math-errno (fast math)          |
  26.6 |  Yes  | -fcaller-saves (-O2)                 |
  26.0 | Maybe | -falign-jumps (-O2 GCC 3.3)          |
  25.9 |  Yes  | -fno-if-conversion2 (! -O1)          |   ??? WONDERED
  25.9 | Maybe | -fschedule-insns2 (-O2)              |
  25.8 |  Yes  | -fcse-follow-jumps (-O2)             |
  25.7 |  Yes  | -fsched-spec (-O2 GCC 3.3)           |
```

Рисунок 3.1 – Результат роботи «ACOVEA»

В якості цільового компілятора, виберемо компілятор сімейства GNU – «GCC». Це набір компіляторів для різних мов програмування, розроблений в рамках проекту GNU. GCC є вільним програмним забезпеченням та поширюється фондом вільного програмного забезпечення.

Оскільки цільовий компілятор використовує високорівневу мову програмування C++ для реалізації генетичного алгоритму, розширення за допомогою алгоритму пошуку сходження на вершину також будемо писати на C++. Орім цього, використовування означеної мови програмування, дає широкі можливості управління пам’яттю та дозволить провести найбільш можливу оптимізацію.

Проводити тестування будемо на операційні системі «Ubuntu». Зазначена система побудована на основі Debian GNU/Linux. Версія операційної системи 18.04 LTS.

Тестування проводиться на процесорі Intel Core i7 8750H, за тактовою частотою 2.21 GHz. Це мобільний шестиядерний процесор сімейства Coffee Lake, здатний обробляти до 12 потоків.

3.3 Тестування генетичного алгоритму

Для того, щоб оцінити отриманий приріст продуктивності, необхідно провести тестування генетичного алгоритму з використанням бібліотеки «ACOVEA». Тестування будемо проводити використовуючи наступні бібліотеки:

– soplex [21]. Набір тестів вибраний з офіційного стандарту тестів SPECfp2006 [22]. Цей тест використовує симплекс алгоритм для вирішення лінійної програми. Програма, яку моделює цей тест вирішує проблему планування на залізничній станції. Симплекс метод полягає у поступовому будуванні базисних рішень, на яких монотонно убуває лінійний функціонал до ситуації коли виконуються необхідні вимоги локальної оптимальності.

– povray [23]. Povray - це програма для відстеження промінів. Відстеження промінів необхідне при моделюваннях об’єктів реального оточення і часто використовується в комп’ютерних іграх. Загалом, відстеження проміння означає моделювання того, як промінь світла подорожує від джерела. Промені можуть проходити крізь прозорі предмети чи відбиватись від інших предметів. Цей тест обчислює проміні над шаховою дошкою, де вже розташовані всі фігури. Розмір вихідного зображення сцени дорівнює 1280 на 1024 пікселі.

– fft. Швидке перетворення Фур'є (FFT) [24] - алгоритм, який обчислює дискретне перетворення Фур'є (DFT) послідовності або його зворотну (IDFT). Найвідоміші алгоритми FFT залежать від факторизації N, але є FFT з O (N log N) складністю для всіх N, навіть для простих N.

– tree. Офіційний тест, який представляє собою алгоритм генерації структури даних типу дерева. Дерево використовує числовий тип даних та має максимальну глибину яка дорівнює 4. Дані, занесені у дерево були отримані генератором випадкових чисел.

Виконаємо тестування додатку 10 разів, для отримання більш точних результатів.

Рисунок 3.2 – Результати тестування soplex

Як можна побачити на приведеному рисунку, для досягнення результату алгоритм може витрачати різну кількість часу. Така поведінка є досить нормальною для алгоритму пошуку такого типу, оскільки набори початкових батьків в генетичному алгоритмі є випадковими. Крім цього на похибку у часі може впливати що й завантаження процесору. Беручи до уваги все вищесказане, можна зробити висновок що рішення про використання багаторазового тестування є вдалим рішенням.

Отримавши результати тестувань, визначимо що середній час виконання пошуку найбільш вдалих оптимізацій склав 4742.1 секунди.

Рисунок 3.3 – Результати тестування povray

В результаті тестування, час пошуку оптимізацій в наборі povray, склав 2196.6 секунди.

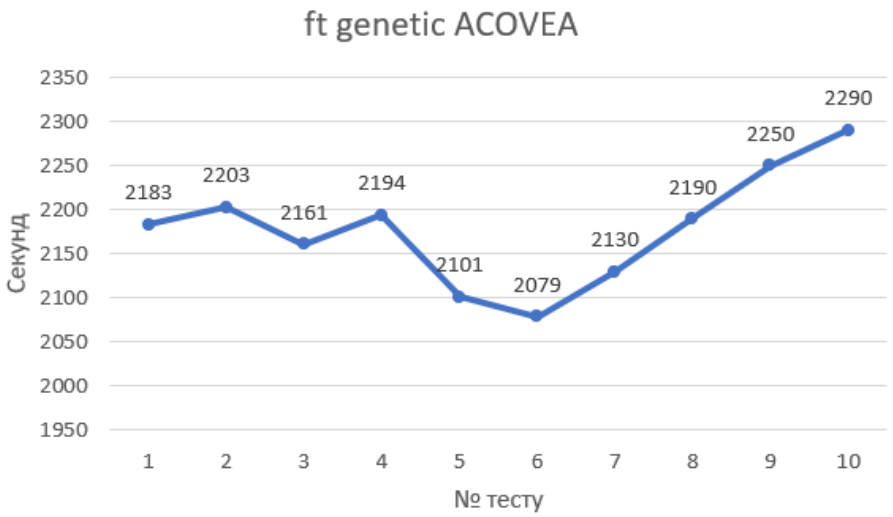


Рисунок 3.4 – Результати тестування fft

Середній час отримання оптимальних опцій налаштування ітеративного компілятора склав 2178.1 секунди.

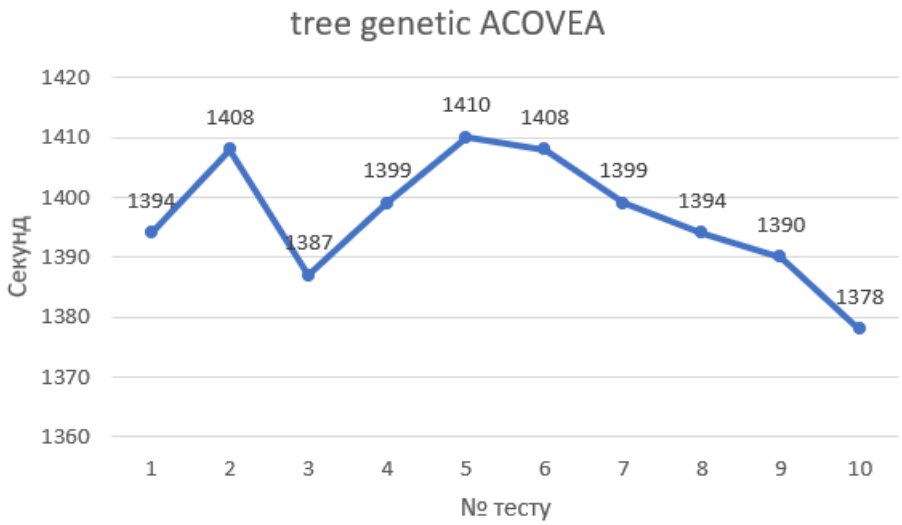


Рисунок 3.5 – Результати тестування tree

Середній час отримання оптимальних опцій налаштування склав 1396.7 секунд.

Як і в оригінальному додатку, результатом виконання модифікованої бібліотеки «ACOVEA» є набір опцій оптимізації які можуть бути використані в компіляторі та їх результат. Опції сортовані за результатом та мають флаг, який позначає чи варто їх застосовувати.

```
Legend:
   !!! MISSED   -- I'm of opinion these flags can _NOT_decrease_ performance of code any way
   ??? WONDERED -- I'm of opinion these flags definally _decrease_ performance of byte-code


 Score |  So?  | Switch (annotation)                                                    |
-------------------------------------------------------------------------------------
  34.4 | Maybe | -fgcse (-O2)                                                           |
  34.3 |  Yes  | -maccumulate-outgoing-args                                             |
  34.3 | Maybe | -fstrict-aliasing (-O2)                                                |
  33.5 | Maybe | -funsafe-math-optimizations (fast math)                                |
  31.9 | Maybe | -finline-functions (-O3)                                               |
  29.8 |  Yes  | -fcse-skip-blocks (-O2)                                                |
  29.6 |  Yes  | -fno-omit-frame-pointer (! -O1)                                        |   ??? WONDERED
  29.1 |  Yes  | -frerun-cse-after-loop (-O2)                                           |
  29.0 |  Yes  | -fno-thread-jumps (! -O1)                                              |   ??? WONDERED
  28.9 |  Yes  | -fmove-all-movables                                                    |
  28.8 |  Yes  | -frename-registers (-O3)                                               |
  28.5 |  Yes  | -falign-labels (-O2 GCC 3.3)                                           |
  28.5 |  Yes  | -fno-crossjumping (! -O1)                                              |   ??? WONDERED
  28.2 |  Yes  | -foptimize-sibling-calls (-O2)                                         |
  28.0 |  Yes  | -minline-all-stringops                                                 |
  27.6 | Maybe | -fno-math-errno (fast math)                                            |
  27.4 | Maybe | -freorder-blocks (-O2)                                                 |
  26.6 |  Yes  | -fcaller-saves (-O2)                                                   |
  26.0 | Maybe | -falign-jumps (-O2 GCC 3.3)                                            |
  25.9 |  Yes  | -fno-if-conversion2 (! -O1)                                            |   ??? WONDERED
  25.9 | Maybe | -fschedule-insns2 (-O2)                                                |
  25.8 |  Yes  | -fcse-follow-jumps (-O2)                                               |
  25.7 |  Yes  | -fsched-spec (-O2 GCC 3.3)                                             |
  25.4 | Maybe | -fno-delayed-branch (! -O1)                                            |
```

Рисунок 3.6– Результат виконання програми для povray

Загалом, результати роботи бібліотеки складаються з конкретної опції, яка налаштовує оптимізацію використану в компіляції програмного коду, її необхідності видалення та результативного рішення чи треба її видаляти чи ні.

3.4 Оптимізація адаптивної компіляції

Виходячи з аналізу предметної галузі та використаних алгоритмі, було прийнято рішення використовувати гібридний алгоритм пошуку. Після аналізу двох алгоритмів, було вирішено використовувати генетичний алгоритм на початку пошуку для звуження області. Після того, як буде досягнута необхідна

величина кількості параметрів чи буде перевищено час однієї ітерації, алгоритм буде передавати параметри, які залишились на обробку алгоритму сходження до вершини.

Такий підхід до обробки повинен покращити роботу двох алгоритмів. Генетичний алгоритм показує кращу якість результатів на великому об’ємі даних при меншій продуктивності. Як показало дослідження, алгоритм сходження до вершини дає більш неточні дані на великих об’ємах, але є швидшим за генетичний алгоритм.

Оскільки «ACOVEA» вже використовує генетичний алгоритм, нам необхідно реалізувати алгоритм сходження до вершини та змінити додаток для використання нового способу обробки даних. Реалізація алгоритму сходження:

```
void HillClimb::StartHillClimb(Options inputOptions[])
{
    BestPerfOption = CheckAllOptions(inputOptions);
    int CurrentPerf = BestPerfOption;

    while (true)
    {
        int t = 0;
        int temp = CheckAllOptions(inputOptions);
        while (t <inputOptions.size())
        {
            Node* back =inputOptions.back();
           inputOptions[Options.size() - 1] =inputOptions[t];
           inputOptions[t] = back;
            CurrentPerf = CheckAllOptions(inputOptions);

            if (CurrentPerf < BestPerfOption)
            {
                BestPerfOption = CurrentPerf;
                break;
            }
            else
            {
                back =inputOptions.back();
               inputOptions[Options.size() - 1] =inputOptions[t];
               inputOptions[t] = back;
            }
            i++;
        }

        if (CurrentPerf == temp)
        {
            break;
        }
```

```
    }
}
int HillClimb::CheckAllOptions(Options inputOptions[])
{
    int CurrentPerf = 0;
    for (unsigned int t = 0; t < inputOptions.size(); t++)
    {
        CurrentPerf += CalcOptionPerf(inputOptions[t]);
    }
    return(CurrentPerf);
}
```

Після того, як алгоритм сходження до вершини було реалізовано, його необхідно імпортувати до бібліотеки. Оскільки додаток «ACOVEA» написаний за допомогою мови програмування C++, програма має лише одну вхідну точку. Саме в цей елемент буде вбудовано виклик алгоритму сходження до вершини. Для поліпшення продуктивності, програма не буде перезавантажуватись, щоб не розгружувати оперативну пам’ять.

Окрім реалізації алгоритму пошуку сходженням до вершини потрібно ще реалізувати механізм, який буде перетворювати результативний вектор генетичного алгоритму на набір даних придатний до використання в наступному колі пошуку з використанням іншого алгоритму.

В результаті, було отримано модифіковану бібліотеку «ACOVEA», яка дозволяє отримувати формацію про найбільш-вдалі параметри оптимізації компілятора. При цьому, бібліотека використовує гібридний алгоритм пошуку, в якому першу частину фільтрації проводить генетичний алгоритм, а другу алгоритм сходження до вершини.

Код поступового виклику алгоритмів наведено нижче:

```
int main()
{
    int POPULATION_SIZE;
    int TESTS_NUMBER_PER_POPULATION = 150;

    std::vector<Options> result;
    vector<Individual> population;

    try
    {
```

```
        population = genetic_Alg();
    }
    catch(AccuracyAchieved e)
    {
        log_result_by_ex(e)
    }
    catch(...)
    {
        cout << "Error"
    }

    result = convert_population(population);

    try
    {
        result = StartHillClimb(result);
    }
    catch(...)
    {
        cout << "Error"
    }

    log_result_by_res(result)
}
```

Після завершення оптимізації бібліотеки, було проведено ряд тестувань, які використовувались для тестування генетичного алгоритму для розрахунку змін. Варто зазначити, що протягом тестування результати бібліотеки різнились лише у часі та в рамках допустимої погрішності.

Рисунок 3.7 – результат soplex тестування модифікованої версії

У результаті модифікації, алгоритм показав прискорення пошуку оптимальних опцій. Середній час складає 4406.1 секунди, що на 7.08% відсотків менше за попередній час.

Рисунок 3.8 – результат povray тестування модифікованої версії

Як можна помітити з результатів тестування, при використанні комбінованого алгоритму, пошук оптимальних параметрів було прискорено на 7.97%. Середній час складає 2021.6 секунд.

Рисунок 3.9 – результат ft тестування модифікованої версії

Цей тест також показав приріст продуктивності, скоротивши середній час пошуку до 2010.1 секунд, що становить 7.71% прискорення.

Рисунок 3.10 – результат tree тестування модифікованої версії

Останній тест також показав приріст продуктивності, скоротивши середній час пошуку до 1285.5 секунди, що становить 7.91% прискорення.

3.5 Аналіз результатів дослідження

Проаналізувавши отримані результати, можна стверджувати, що проведений аналіз існуючих алгоритмів є достовірним і використання комбінованого алгоритму прискорює пошук найкращих можливих параметрів оптимізації.

Згідно протестованим додаткам, середній приріст продуктивності складає 7-8%. Оскільки тестування було обмежене 100 ітераціями, приріст продуктивності не так сильно помітно, однак при використанні такого методу на реальних алгоритмах та додатках, економія часу буде помітною.

Так, наприклад, візьмемо компіляцію фреймворку «webkit» [25]. Це інфраструктура з відкритим вихідним кодом для відображення веб-сторінок. Зараз він використовується в декількох браузерах та інших програмах, які потребують відображенні і редагуванні вмісту веб-сторінок. Для тестування продуктивності використовувався набір тестів SunSpider для JavaScript, включений в репозиторій webkit. На 500 ітераціях потрібно близько 72 годин для того, щоб знайти оптимальні опції. При використанні комбінованого алгоритму, час який необхідно витратити на пошук буде зменшено приблизно на 5 – 6 годин.

Однак для подальшого дослідження, використаний спосіб прискорення ітеративної компіляції, може бути випробуваний разом з нейронними мережами. Вже існують праці в яких досліджують та прототипують використання байєсовських мереж разом з ітеративними компіляторами.

В досліджених працях використовування байєсовських мереж разом з ітеративними компіляторами надавало значний приріст продуктивності. Так, для двох тестових наборів CBench [26] та Polybench [27] відсоток прискорення став майже 50.

Оскільки ітеративний компілятор використовується в них для тренування моделей та підготовки набору тестових даних, оптимізація ніяк не вплине на швидкість вихідного компілятора, однак може допомогти в прискоренні навчання мережі, зменшивши таким чином результативний час виконання пошуку.

ВИСНОВКИ

В процесі виконання роботи було досліджено методи оптимізації, які використовуються при компіляції програмного коду. Окрім цього були досліджені методи вибору оптимальних параметрів компіляції для досягнення найліпшої продуктивності. Було проведено дослідження предметної галузі, проведено аналіз систем та аналіз конкурентів, які використовуються для налаштування опцій компіляції. Протягом аналізу було виявлено їх переваги, недоліки та шляхи оптимізації.

В результаті, було визначено план оптимізації рішення для тестування ітеративної компіляції «ACOVEA». Було вирішено використовувати гібридний алгоритм пошуку, який би використовував генетичний алгоритм для звуження простору шуканих параметрів, а потім передавав результати роботи на обробку алгоритму сходження до вершини, який знаходив би найоптимальніший набір опцій.

Для модифікування проекту було обрано мову програмування C++. Під час роботи повинні будуть використовуватись таки засоби розробки як Visual Studio 2019 та Visual Studio Code. Для збору метрик будуть використовуватись можливості «ACOVEA».

Окрім цього було досліджено де можуть бути використані результати отриманої роботи. Описаний спосіб підвищення продуктивності теоретично може бути використаний в нових версіях ітеративних компіляторів. Через зменшення часу необхідного для знаходження одного набору параметрів оптимізації, ми маємо можливість виконати набагато більше ітерацій для знаходження найліпшого варіанту. Іншим, більш перспективним напрямом використання результатів роботи може стати адаптація описаного алгоритму до використання разом з нейронними мережами. Такий підхід в результаті повинен прискорити процес навчання нейронної мережі та дати приріст до продуктивності.

ПЕРЕЛІК ПОСИЛАНЬ

1. Microsoft Visual Studio / Wikipedia. Дата оновлення: 15.03.2020. URL: https://ru.wikipedia.org/wiki/Microsoft_Visual_Studio (дата звернення: 14.04.2020).

2. Rider / Jetbrains. Дата оновлення: 20.03.2020. URL: https://www.jetbrains.com/ru-ru/rider/ (дата звернення: 14.04.2020).

3. Jetbrains / Jetbrains. Дата оновлення: 13.12.2019. URL: https://www.jetbrains.com/ (дата звернення: 14.04.2020).

4. Using Version Control in VS Code / Microsoft. Дата оновлення: 20.01.2019. URL: https://code.visualstudio.com/docs/editor/versioncontrol (дата звернення: 15.04.2020).

5. Реинхард В., Хак С., Сейдл Х. Compiler Design: Analysis and Transformation. Берлін: Springer, 2012. 177 с.

6. Торчон Л. Engineering a Compiler. Берлінгтон: Morgan Kaufmann, 2011. 824 с.

7. Шилдт Г. C#: Полное руководство C# 4.0: навч. посібник. 4-е вид. Санкт-Петербург: Питер, 2016. 896 с.

8. Оптимизирующий компилятор / Wikipedia. Дата оновлення: 20.08.2018. URL: https://ru.wikipedia.org/wiki/Оптимизирующий_компилятор (дата звернення: 13.04.2020).

9. ACME: Adaptive Compilation Made Efficient / Computer Science Department of Rice University. Дата оновлення: 03.08.2017. URL: http://www.cs.cmu.edu/afs/cs/academic/class/15745-f09/www/papers/p69-cooper.pdf (дата звернення: 11.04.2020).

10. COBAYN / Github. Дата оновлення: 11.03.2015. URL: https://github.com/amirjamez/COBAYN/ (дата звернення: 13.04.2020).

11. Гоф Б. An Introduction to GCC: For the GNU Compilers Gcc and G++ Брістоль: Network Theory Limited, 2005. 124 с.

12. Голдберг Д. Genetic algorithms in search, optimization, and machine learning. Бостон: Addison-Wesley Professional, 2005. 432 с.

13. Поиск восхождением к вершине / Wikipedia. Дата оновлення: 24.09.2016. URL: https://uk.wikipedia.org/wiki/Поиск_восхождением_к_вершине (дата звернення: 14.04.2020).

14. TreeBench / GtiHub. Дата оновлення: 04.01.2012. URL: https://github.com/Acovea/libacovea/blob/master/benchmarks/treebench.c (дата звернення: 16.04.2020)

15. Кормен Т., Лейзерсон Ч., Ривест Р., Штайн К. Алгоритмы: построение и анализ: навч. посіб., Москва: Диалектика-Вильямс, 2020. 1328 с.

16. Аплігейт Д., Біксбі Д., Чватал В., Кук В. The Traveling Salesman Problem: A Computational Study. Прінстон: Princeton University Press, 2007. 608 с.

17. GAF / Nuget. Дата оновлення: 12.11.2019. URL: https://www.nuget.org/packages/GAF/ (дата звернення: 16.04.2020).

18. Нємцов М, Каук В. Дослідження методів оптимізації, які використовуються у компіляторах коду. International Electronic Scientific Journal "Science Online", 2020

19. ACOVEA / Free Software. Дата оновлення: 14.11.2012. URL: https://directory.fsf.org/wiki/Acovea (дата звернення: 16.04.2020).

20.GNU General Public License / Wikipedia. Дата оновлення: 13.07.2018. URL: https://ru.wikipedia.org/wiki/GNU_General_Public_License (дата звернення: 16.04.2020).

21. 450.soplex SPEC CPU2006 Benchmark Description / SPEC CPU2006. Дата оновлення: 12.04.2014. URL: https://www.spec.org/cpu2006/Docs/450.soplex.html (дата звернення: 18.04.2020).

22. Рассел Д. SPECfp. Москва: Bookvika publishing, 2009. 238 с.

23. 453.povray SPEC CPU2006 Benchmark Description / SPEC CPU2006. Дата оновлення: 14.04.2014. URL: https://www.spec.org/cpu2006/Docs/453.povray.html (дата звернення: 19.04.2020).

24. Быстрое преобразование Фурье / Wikipedia. Дата оновлення: 19.06.2018. URL: https://ru.wikipedia.org/wiki/_Быстрое_преобразование_Фурье (дата звернення: 18.04.2020).

25. Міннік К. WebKit For Dummies Wrox, Хобокен: For Dummies 2012. – 408 с.

26. Фусин Д. Collective benchmark (cbench), a collection of open-source programs with multiple datasets assembled by the community to enable realistic benchmarking and research on program and architecture optimization. Прінстон : Princeton University Press, 2014. 638 с.

27. PolyBench/C / Штат Огайо. Дата оновлення: 19.06.2018. URL: http://web.cse.ohio-state.edu/~pouchet.2/software/polybench/ (дата звернення: 18.04.2020).