Хмельницький національний університет
Факультет інформаційних технологій
Кафедра інженерії програмного забезпечення

ДИПЛОМНА РОБОТА

Метод нефункційного тестування та автоматичного налаштування генераторів кодів
Назва теми
з урахуванням програмно-апаратної платформи

Рівень вищої освіти Другий (магістерський)

Галузь знань 12 «Інформаційні технології»

Спеціальність 121 «Інженерія програмного забезпечення»

Освітня програма Освітньо-професійна програма «Інженерія програмного забезпечення»

Шифр ДРІПЗ 170104.01.03.00 ПЗ

Виконала студентка ІІ курсу група ІПЗм-21-1 _____ Дьоміна А.І
Підпис Ініціали, прізвище

Керівник канд. техн. наук, доцент _____ Гурман І.В.
Науковий ступінь, звання Підпис Ініціали, прізвище

Нормоконтролер канд. техн. наук, доцент _____ Форкун Ю.В.
Підпис Ініціали, прізвище

До захисту допускаю:
Завідувач кафедри інженерії
програмного забезпечення _____ Л. П. Бедратюк
Підпис Ініціали, прізвище

5 грудня 2022 р.

Хмельницький 2022

ХМЕЛЬНИЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

Факультет Програмування та комп'ютерних і телекомунікаційних систем

Кафедра Інженерії програмного забезпечення

Рівень вищої освіти Другий (магістерський)

Галузь знань 12 «Інформаційні технології»

Спеціальність 121 «Інженерія програмного забезпечення»

Освітня програма Освітньо-професійна програма «Інженерія програмного забезпечення»

ЗАТВЕРДЖУЮ
Завідувач кафедри
Л. П. Бедратюк
02 09 2022 р.

ЗАВДАННЯ
НА ДИПЛОМНУ РОБОТУ

Дьоміній Анастасії Іванівні
Прізвище, ім'я, по батькові студента

1. Тема проекту (роботи) Метод нефункційного тестування та автоматичного налаштування генераторів кодів з урахуванням програмно-апаратної платформи

Керівник проекту (роботи) Гурман Іван Васильович, канд. техн. наук, доцент
Прізвище, ім'я, по батькові, науковий ступінь, вчене звання

Затверджена наказом ректора університету від 01.07.2022 р. № 102

2. Строк подання студентом проекту (роботи) на кафедру 01.12.2022 р.
3. Вихідні дані до проекту (роботи) Матеріали переддипломної практики
4. Зміст пояснювальної записки (перелік питань, які потрібно розробити) Дослідження предметної області та постановка задачі. Існуючі рішення. Автоматичне нефункціональне тестування сімейств генераторів коду. CAT: програма для автоматичного налаштування компіляторів Тестування автоматичних генераторів коду.
5. Перелік графічного матеріалу (із зазначенням обов'язкових креслень) Презентаційні матеріали (слайди)

6. Консультанти розділів дипломного проекту

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Антиплагіат	Гурман І.В. доцент		
Нормоконтроль	Федуш Ю.В. доцент		

7. Дата видачі завдання « 01 » вересня 2022р.

КАЛЕНДАРНИЙ ПЛАН

Назва етапів (розділів) дипломного проекту (роботи)	Строк виконання етапів проекту (роботи)	Примітка
1 Ознайомлення з предметною областю; формулювання мети та задач дослідження; визначення об'єкта та предмета дослідження; визначення структури дипломної роботи	01.09 – 07.09.2022	
2 Робота над розділом 1 та 2 дипломної роботи – Дослідження предметної області та постановка задачі. Існуючі рішення	08.09 – 25.09.2022	
3 Робота над розділом 3 дипломної роботи – Автоматичне нефункціональне тестування сімейств генераторів коду	26.09 – 10.10.2022	
4 Робота над науковими статтями	11.10 – 20.10.2022	
5 Робота над розділом 4 дипломної роботи – CAT: програма для автоматичного налаштування компіляторів	11.10 – 26.10.2022	
6 Робота над розділом 5 дипломної роботи – Тестування автоматичних генераторів коду.	27.10 – 15.11.2022	
7 Узгодження постановки задачі, отриманих результатів та висновків; написання вступу, загальних висновків, оформлення джерел посилання та додатків, оформлення пояснювальної записки та графічних матеріалів згідно вимог стандартів	06.11 – 30.11.2022	
8 Попередній захист ДР	Листопад (згідно графіка)	
9 Перевірка ДР на плагіат, нормконтроль, отримання відгуків та рецензій. Брошурування (зшиття) пояснювальної записки)	22.11.2022 – 30.11.2022	
10 Підготовка до захисту та захист ДР	01.12.2022-06.12.2022	

Студент _____ А.І. Дьоміна
Підпис Ініціали, прізвище

Керівник проекту (роботи) _____ І,В, Гурман
Підпис Ініціали, прізвище

РЕФЕРАТ

Тема дипломної роботи: Метод нефункційного тестування та автоматичного налаштування генераторів кодів з урахуванням програмно-апаратної платформи

Автор роботи: Дьоміна Анастасія Іванівна.

Керівник проекту: Гурман Іван Васильович.

Пояснювальна записка: 162 с., 43 рис., 15 табл., 2 дод., 125 джерел.

АЛГОРИТМ ПОШУКУ НОВИЗНИ, АВТОМАТИЧНА ГЕНЕРАЦІЯ ТЕСТОВИХ ДАНИХ, МЕТАЕВРИСТИЧНІ МЕТОДИ ПОШУКУ, ГЕНЕТИЧНІ АЛГОРИТМИ, ЗАДАЧІ ГЕНЕРАЦІЇ ДАНИХ, СТРУКТУРНЕ ТЕСТУВАННЯ НА ОСНОВІ ПОШУКУ, ЗНАЧЕННЯ ДАНИХ.

Метою роботи є удосконалення методу оцінки якості згенерованого коду з точки зору продуктивності та використання ресурсів.

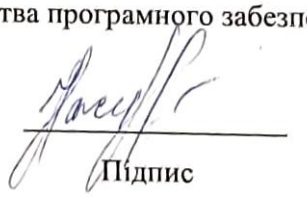
У дипломній роботі проаналізовано існуючі підходи та методи для тестування ти автоналаштування генераторів кодів, з урахуванням програмного та апаратного забезпечення. Досліджено придатність методу пошуку новизни, для покращення нефункційного тестування генераторів коду шляхом пошуку невикористаних тесткейсів. Розроблено автотюнер для компіляторів CAT, який знаходить оптимальні налаштування для конкретного програмно-апаратного забезпечення . Проведена оцінка продуктивності методу пошуку новизни для оптимізації компілятора. Виконано оцінювання розробленого методу з точки зору ефективності, порівнюючи його з існуючими рішеннями на базі GCC.

Для написання програмного коду CAT було використано мову Java.

Практична значимість отриманих результатів тому, що з точки зору користувача, генератори коду є компонентами чорної скриньки, які можна використовувати для полегшення процесу виробництва програмного забезпечення.

01.12.2022

Дата



Підпис

ABSTRACT

Master's thesis: «A method of non-functional testing and automatic adjustment of code generators taking into account the hardware and software platform»

Author: Anastasiia Ivanivna Dominf.

Head of work: Ivan Vasylovych Gurman.

Master's thesis consist of: 162 p., 43 pic, 15 tab., 2 add., 125 srs.

NOVELTY SEARCH ALGORITHM, AUTOMATIC TEST DATA GENERATION, METAHEURISTIC SEARCH METHODS, GENETIC ALGORITHMS, DATA GENERATION TASKS, SEARCH-BASED STRUCTURAL TESTING, DATA VALUES.

The aim of the work is to improve the method of assessing the quality of the generated code in terms of performance and resource usage.

The thesis analyzes existing approaches and methods for testing and auto-tuning code generators, taking into account software and hardware. The suitability of the novelty search method to improve non-functional testing of code generators by finding unused test cases is investigated. The auto-tuner for CAT compilers, which finds optimal settings for specific hardware and software, is developed. The performance of the novelty search method for compiler optimization was evaluated. The developed method was evaluated in terms of efficiency, comparing it with existing solutions based on GCC.

Java language was used to write the CAT program code.

The practical significance of the results obtained is that from the user's point of view, code generators are black box components that can be used to facilitate the software production process.

01.12.2022

Дата

Підпис

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ

NS	–	Novelty Search
ПЗ	–	Програмне забезпечення
GA	–	Genetic algoritms
ЦП	–	Центральний процессор
DSL	–	Domain-Specific Language
GP	–	Generative Programming
GPL	–	General-Purpose Language
GUT	–	Generator Under Test
LCL	–	Lower Control Limit
MR	–	Metamorphic Relation
MT	–	Metamorphic Testing
АОК	–	Аналіз основних компонентів
R	–	Range
RS	–	Random Search
S	–	Speedup
SBSE	–	Search-Based Software Engineering
SD	–	Score Distance
UCL	–	Upper Control Limit

ВСТУП

Сучасні системи програмного забезпечення сьогодні покладаються на дуже різнорідний і динамічний взаємозв'язок платформ і пристроїв, які надають широкий спектр можливостей і послуг. Ці різнорідні служби можуть працювати в різних середовищах, починаючи від хмарних серверів з практично необмеженими ресурсами до пристроїв з обмеженими ресурсами лише з кількома кілобайтами оперативної пам'яті. Ефективна розробка програмного забезпечення для багатоцільових платформ і апаратних технологій стає все більш важливою. Як наслідок, в останні роки помітна тенденція [1], що генеративна розробка програмного забезпечення стає все більше привабливою для приборкання з неоднорідністю часу виконання платформ і технологічних стеків, які існують у кількох областях, таких як мобільні пристрої або Інтернет речей [2].

Генеративне програмування [1] пропонує рівень абстракції програмного забезпечення, який розробники програмного забезпечення можуть використовувати для визначення бажаної поведінки системи (наприклад, використання доменно-спеціальних мов DSL, моделей тощо), і автоматично генерувати програмні артефакти. Як наслідок, нові досягнення в специфікаціях апаратного забезпечення та платформ проклали шлях до створення кількох генераторів, які служать основою для автоматичного створення коду для широкого діапазону програмних і апаратних платформ.

Автоматична генерація коду передбачає два види генераторів: генератори вихідного коду та компілятори. З одного боку, генератори коду необхідні для перетворення системних специфікацій високого рівня (наприклад, мови текстового або графічного моделювання) у звичайні програми з вихідним кодом (наприклад, мови загального призначення GPL, такі як Java, C++ тощо). З іншого боку, компілятори усувають розрив між вхідними програмами (тобто написаними з використанням GPL) і цільове

середовище виконання, створюючи низькорівневий машинний код (тобто двійкові файли, виконувані файли) для конкретної апаратної архітектури.

Завдяки повністю автоматичному створенню коду тепер можна легко та швидко розробляти код, покращуючи якість і послідовність програми, а також продуктивність розробки програмного забезпечення [3]. Крім того, сьогоднішні сучасні генератори (наприклад, компілятори C) стають високо конфігурованими, пропонуючи численні` параметри конфігурації (наприклад, проходи оптимізації) для користувачів, щоб налаштувати створений код щодо цільового програмного та/або апаратного забезпечення. платформа.

У цьому контексті вкрай важливо, щоб програмне забезпечення, яке автоматично генерується, пройшло відповідну техніку тестування, щоб перевірити правильну поведінку генераторів. Таким чином, користувачі можуть довіряти генератору коду та бути впевненими у його правильній роботі. Перевірка правильності згенерованого коду може бути функціональною (наприклад, перевірка того, що згенерований код демонструє таку ж функціональну поведінку, як описано у вхідній програмі), або нефункціональною (наприклад, перевірка якості згенерованого коду).

Актуальність даної теми полягає у тому, що з точки зору користувача, генератори (компілятори та генератори коду) є компонентами чорної скриньки, які можна використовувати для полегшення процесу виробництва програмного забезпечення. Якість згенерованого програмного забезпечення безпосередньо залежить від якості самого генератора. Поки якість генераторів підтримується та покращується, якість створених програмних артефактів також покращується. Будь-яка помилка з цими генераторами впливає на якість програмного забезпечення, що постачається на ринок, і призводить до втрати довіри з боку кінцевих користувачів. Зокрема, коли використовується автоматична генерація коду, існують дві основні проблеми, які загрожують якості згенерованого програмного забезпечення: з одного боку, генератори з високою можливістю налаштування контролюють якість згенерованого коду за допомогою численних параметрів оптимізації, які користувачі можуть

увімкнути/вимкнути. Цей величезний простір для конфігурації створює важливу проблему для користувачів, при виборі найкращих варіантів оптимізації, які відповідають деяким нефункціональним вимогам. З іншого боку, складність генераторів коду, а також відсутність рішень для оцінки нефункціональних властивостей згенерованого коду є перешкодою для користувачів, яким потрібні додаткові докази, щоб продовжувати використовувати їх під час розробки програмного забезпечення.

Мета проекту – удосконалити метод оцінки якості згенерованого коду з точки зору продуктивності та використання ресурсів.

Для досягнення мети проекту необхідно виконати такі задачі:

– провести дослідження предметної області, визначити особливості та специфіку процесу налаштування генераторів коду;

– виконати детальний аналіз існуючих рішень;

– провести аналіз інструментів та технологій, що використовуються для автоматичного налаштування генераторів коду;

– проаналізувати та порівняти методи тестування генераторів коду

– на основі проведених досліджень та аналізів провести специфікацію вимог до методу нефункційного тестування та автоматичного налаштування генераторів кодів з урахуванням програмно-апаратної платформи

– підвести підсумки про актуальність даної розробки;

1 ДОСЛІДЖЕННЯ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ПОСТАНОВКА ЗАДАЧІ

1.1 Різноманітність в програмній інженерії

Історія розробки програмного забезпечення демонструє постійне зростання складності в кількох аспектах процесу розробки ПЗ. Ця складність тісно пов’язана з фактичним технологічним прогресом у індустрії розробки ПЗ, оскільки на ринку з’являється все більше різнорідних пристроїв [4]. Як правило, неоднорідність може виникати з точки зору різної складності системи, різних мов програмування та платформ, типів систем, процесів розробки та розподілу між сайтами розробки [5]. Неоднорідність системи часто зумовлена різноманітністю програмного та апаратного забезпечення. Різноманітність постає як критична проблема, яка охоплює всі види діяльності в розробці програмного забезпечення, від проектування до експлуатації [6]. Вона з’являється в різних сферах, таких як мобільні пристрої, веб-розробка [7], безпека [8] тощо.

Однак різноманіття програмного та апаратного забезпечення призводить до збільшення ризику системних збоїв через постійну зміну конфігурацій і специфікацій системи. По суті, ефективна розробка програмного забезпечення для багатьох цільових платформ і апаратних технологій стає все більш важливою. Крім того, зростаюча актуальність програмного забезпечення та вищі вимоги до якості та продуктивності ускладнюють розробку.

У цьому розділі досліджується два різні виміри різноманітності:

неоднорідність апаратного забезпечення;

різноманітність програмного забезпечення.

1.1.1 Неоднорідність обладнання

Сучасні програмні системи сьогодні покладаються на дуже різнорідний та динамічний взаємозв’язок пристроїв, які надають широкий спектр можливостей і

послуг для кінцевих користувачів. Ці різнорідні служби працюють у різних середовищах – від хмарних серверів до пристроїв з обмеженими ресурсами. Неоднорідність апаратного забезпечення походить від безперервних інновацій апаратних технологій для підтримки нових конфігурацій системи та архітектурного дизайну (наприклад, додавання нових функцій, зміна архітектури процесора, доступність нового апаратного забезпечення тощо). Наприклад, до лютого 2016 року збільшення потужності мікропроцесорів відбувалося за знаменитим законом Мура для процесорів Intel. Дійсно, кількість компонентів (транзисторів), які можна встановити на чіп, подвоюється кожні два роки, підвищуючи продуктивність і енергоефективність. Наприклад, процесор Intel Core 2 Duo був представлений у 2006 році з 291 мільйоном транзисторів і тактовою частотою 2,93 ГГц. Двома роками пізніше Intel представила процесори Core 2 Quad, які мали тактову частоту 2,66 ГГц і подвійну кількість транзисторів, представлених у 2006 році з 582 мільйонами транзисторів.

Останніми роками сучасні процесори стають все більш і більш неоднорідними, використовують більш ніж один тип процесора або ядер, які називаються «співпроцесорами». ЦП може навіть використовувати різні архітектури набору інструкцій (ISA), де головний процесор має одну а решта має іншу, як правило, дуже різну архітектуру. Операціями, які виконує співпроцесор, можуть бути арифметика, графіка, обробка сигналів, обробка рядків, шифрування або взаємодія вводу/виводу з периферійними пристроями. Як приклад, ARM архітектура процесора big.Little випущена у 2011 році (див. рис 1.1) – це технологія оптимізації живлення, у якій високопродуктивні ядра ЦП ARM поєднуються з найефективнішими ядрами ЦП ARM для забезпечення максимальної продуктивності та підвищення швидкодії паралельної обробки за значно нижчої середньої потужності. Це може заощадити 75% енергії ЦП і збільшити продуктивність на 40% у високопотокових робочих навантаженнях. Ціль цієї архітектури полягає в тому, щоб створити багатоядерний процесор, який може краще адаптуватися до динамічних обчислювальних потреб. Потоки з високим пріоритетом або обчислювальною інтенсивністю в цьому випадку можуть бути

призначені «Big» ядрам, тоді як потоки з меншим пріоритетом або меншою обчислювальною інтенсивністю, наприклад, фонові завдання, можуть виконуватися «Little» ядрами. Ця модель була реалізована в Samsung Exynos 5 Octa в 2013 році.

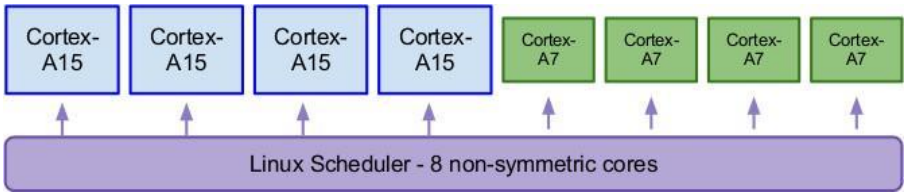


Рисунок 1.1 – ARM Big.Little гетерогенна багатопроцесорна обробка

Враховуючи складність нових архітектур процесорів (x86, x64, багатоядерних тощо) і виробників процесорів, таких як ARM, AMD і Intel, деякі з питань, на які розробники повинні відповісти, стикаючись із різнорідністю апаратного забезпечення: Чи легко забезпечити задовільний рівень продуктивності на сучасних процесорах? Як можна створити машинний код, який міг би ефективно використовувати постійні зміни апаратного забезпечення?

1.1.2 Різноманітність програмного забезпечення

У сучасних системах програмного забезпечення різні варіанти ПЗ зазвичай розробляються одночасно для вирішення широкого діапазону прикладних контекстів і вимог замовника [9]. Таким чином, ПЗ різних підходів і мов, залежно від області застосування.

У роботах [10] і [9] представлено вичерпний огляд багатьох аспектів різноманітності програмного забезпечення при розробці ПЗ. Згідно з цими дослідженнями, різноманітність ПЗ може виникати в різних типах і вимірах, таких як різноманітність операційних систем, мов програмування, структур даних, компонентів, середовищ виконання тощо. Як і всі сучасні програмні системи,

програмне забезпечення потрібно адаптувати для вирішення вимоги, що змінюються з плином часу, підтримують еволюцію системи, технології та потреби ринку, як-от розгляд нових програмних платформ, нових мов, нових варіантів клієнта, тощо.

Щоб зрозуміти навички та здібності, необхідні для розробки програмного забезпечення на основі різних класів пристроїв і доменів програм, використаємо популярний репозиторій з відкритим кодом GitHub, щоб оцінити різноманітність існуючих мов програмування. Були використані наступні набори ключових слів:

Хмара: сервер із практично необмеженими ресурсами;

Мікроконтролер: вузол з обмеженими ресурсами (кілька КБ оперативної пам’яті, кілька МГц);

Мобільний: проміжний вузол, як правило, смартфон;

Інтернет речей: пристрої з підтримкою Інтернету;

Розподілені системи;

Вбудовані системи, як велика та важлива частина впровадження послуг, працюватимуть якнайближче до фізичного світу, вбудовані в датчики та пристрої.

Рисунок 1.2 представляє результати цих запитів.

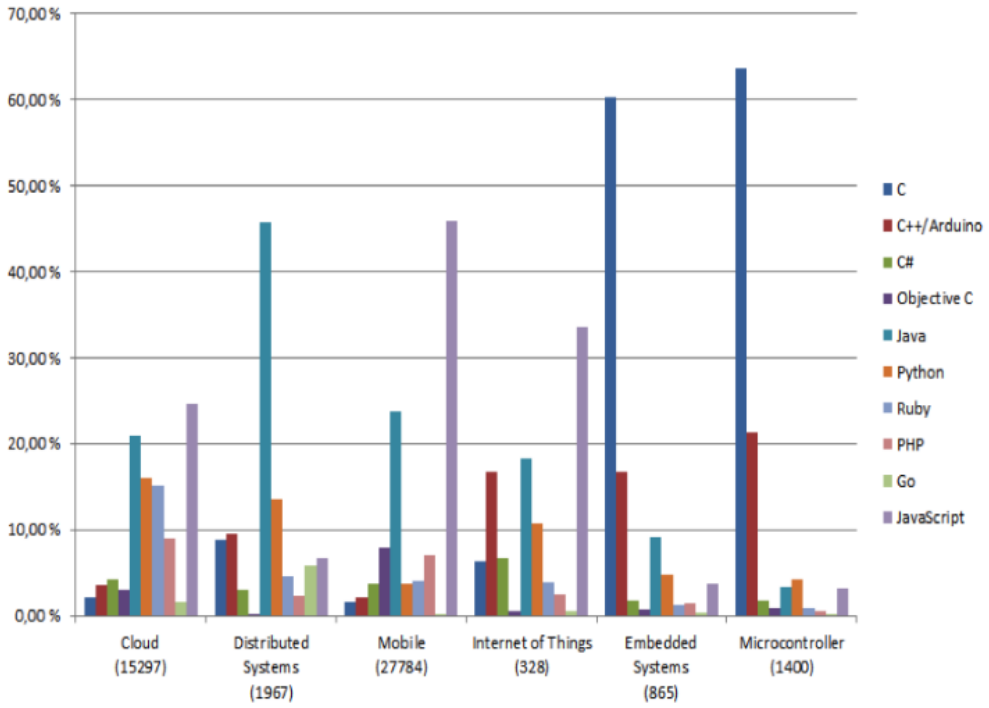


Рисунок 1.2 – Популярність 10 мов програмування в різних областях застосування

Запитані ключові слова представлені на осі X разом із кількістю збігів для цього ключового слова. Для кожного ключового слова вісь у представляє популярність (у відсотках від загальної кількості збігів) кожної з 10 найпопулярніших мов програмування.

Це просте дослідження показує, що жодна мова програмування не є популярною в різних областях. Загальна тенденція вказує на те, що Java і JavaScript (і певною мірою Python і Ruby) популярні в хмарі та мобільних пристроях, тоді як C (і певною мірою C++) є очевидним вибором для розробників, орієнтованих на вбудовані системи та системи на основі мікроконтролерів. Інші мови не набирають більше 10% для жодного з ключових слів. Для всіх ключових слів, крім Хмара, сукупна популярність Java, JavaScript і C/C++ (тобто сума відсотків) перевищує 70%. Для Хмари спостерігається широке використання Python, Ruby також дуже популярний, тому сукупна популярність Java, JavaScript і C/C++ становить лише 50%. Варто також зауважити, що найпопулярніша мова для певного ключового слова має дуже низькі оцінки (менше ніж 5%) принаймні для іншого ключового слова. Хоча може здатися, що поєднання C/C++, JavaScript і Java має охопити всі сфери, на практиці це не виключає потреби в інших мовах програмування. Наприклад, Fibaro Home Center 2 (шлюз для домашньої автоматизації на основі протоколу Z-Wave) використовує Lua як мову сценаріїв для визначення правил автоматизації. Іншим прикладом є BlueGiga BlueTooth Smart Module, для якого можна створювати сценарії за допомогою BGScript, власної мови сценаріїв. Це показує, що кожна частина інфраструктури може вимагати використання нішевої мови, проміжного програмного забезпечення або бібліотеки для повного використання свого потенціалу.

Підсумовуючи, варіація мов програмування для різних типів пристроїв і областей застосування викликає високу різноманітність програмного забезпечення. Відповідно, запропоновано наступне визначення різноманіття програмного забезпечення в контексті цієї дисертації: різноманітність програмного забезпечення – це генерація або реалізація тієї самої специфікації програми різними способами/способами, щоб задовольнити один або більше вимірів різноманітності,

таких як різноманітність мов програмування, середовищ виконання, функціональних можливостей тощо.

1.1.3 Зіставлення різноманіття програмного забезпечення з різнорідним обладнанням

Спільноти апаратного та програмного забезпечення стикаються зі значними змінами та серйозними проблемами. рисунок 1.3 показує огляд проблем, з якими стикаються обидві громади. Насправді апаратне та програмне забезпечення тягнуть нас у протилежних напрямках.

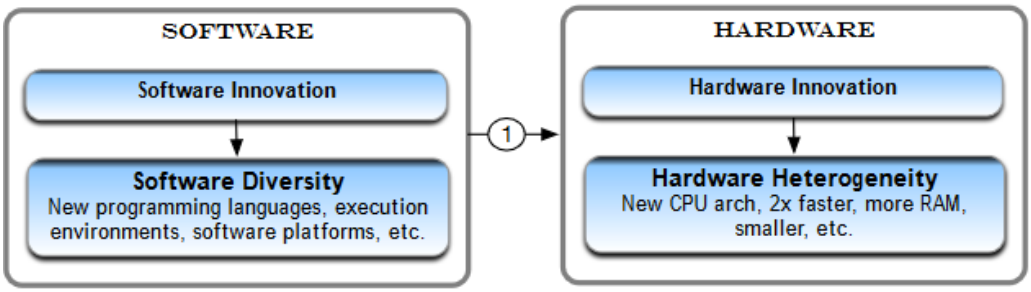


Рисунок 1.3 – Відповідність програмного забезпечення апаратному забезпеченню

З одного боку, програмне забезпечення стикається з проблемами подібного масштабу, із серйозними змінами в тому, як програмне забезпечення розгортається, продається та взаємодіє з апаратним забезпеченням. Різноманітність програмного забезпечення, як обговорювалося в розділі 1.1.2, керується інноваційним програмним забезпеченням, що спонукає розробку програмного забезпечення до складних систем із високою конфігурацією. Ця складність пов’язана з величезним розмаїттям програмних технологій, конфігурацій клієнтів, середовищ виконання, мов програмування тощо. Ця неймовірна кількість конфігурацій, з якими стикається програмне забезпечення, робить тестування та валідацію дуже складними та трудомісткими. Як наслідок, програмне забезпечення стає все вищим рівнем абстракції, керуючи складністю та склеюючи багато частин, щоб дати

програмістам правильну абстракцію щодо того, як речі насправді працюють і як насправді представлені дані.

З іншого боку, апаратне забезпечення наражає на деталі низькорівневої реалізації та неоднорідність через безперервні апаратні інновації. Інновації в апаратному забезпеченні пропонують енергоефективність, покращення продуктивності, але створюють багато складнощів для інженерів програмного забезпечення та розробників. Наприклад, у [11] автори стверджують, що системне програмне забезпечення не готове до такої неоднорідності та не може повністю отримати вигоду від нових апаратних досягнень, таких як багатоядерні та багатоядерні процесори. Хоча багатоядерні процесори використовуються в повсякденному житті, все ще не відомо, як найкраще організувати та використовувати їх. Тим часом апаратна спеціалізація для кожної окремої програми не є найкращим способом створення чіпів.

Узгодження програмного забезпечення з апаратним забезпеченням забезпечується ефективним перекладом програм високого рівня в машинний код, який краще використовує зміни в апаратному забезпеченні (співвідношення 1 на рисунку). 1.3). Це саме те, для чого призначений компілятор.

Пройшли ті часи, коли асемблерний код писався вручну та з нуля. Тепер компілятори повинні впоратися з цією неоднорідністю та ефективно згенерувати та оптимізувати код для конкретного мікропроцесора.

Як показано на рисунку 1.4, компілятор зазвичай ділиться на дві частини: інтерфейс і сервер. Інтерфейс компілятора перевіряє синтаксис і семантику та аналізує вихідний код для створення внутрішнього представлення програми, яке називається проміжним представленням або IR. Наприклад, колекція компіляторів GNU (GCC) і LLVM підтримують багато інтерфейсів з такими мовами програмування, як C, C++, Objective-C, Objective-C++, Fortran, Java, Ada та Go тощо. Внутрішня частина компілятора генерує залежний від цілі код складання та виконує оптимізацію для архітектури цільового обладнання. Як правило, результатом обробки є машинний код, спеціалізований для певного процесора та операційної системи (наприклад, процесори ARM, Sparc тощо). Як наслідок,

людям, які пишуть компілятори, доводиться постійно вдосконалювати спосіб створення цих виконуваних файлів, випускаючи нові версії компілятора для підтримки нових змін у апаратному забезпеченні (тобто вводячи нові оптимізаційні реклами, набори інструкцій тощо).

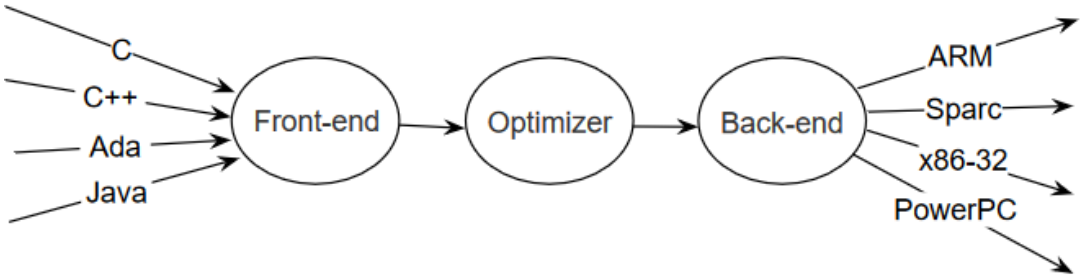


Рисунок 1.4 – Архітектура компілятора

Візьмемо приклад GCC. GCC може автоматично генерувати код приблизно для більш ніж 40 різних архітектур процесорів. Отже, GCC стає висококонфігурованим, дозволяючи користувачеві компілятора ввімкнути кілька рекламних груп для налаштування згенерованого коду. Наприклад, одним з важливих компіляторів параметр *-march*. Він повідомляє компілятору який код, який він повинен створити для архітектури процесора системи. Він повідомляє GCC, що він повинен створити код для певного типу ЦП. Використання налаштування *-march=native* увімкне всі параметри оптимізації, які можна застосувати до центрального процесора рідної системи, з усіма його можливостями, функціями, наборами інструкцій тощо. Існує багато інших параметрів конфігурації для цільового процесора, наприклад *-with-arch=i7, -with-cpu=corei7* тощо. Як правило, щоразу, коли випускається нове сімейство процесорів, розробники компілятора випускають нову версію компілятора з більш складними параметри конфігурації для цільової платформи. Наприклад, старі компілятори створюють лише 32-розрядні програми. Ці програми все ще працюють на нових 64-розрядних комп’ютерах, але вони можуть не використовувати всі можливості процесора (наприклад, вони не використовуватимуть нові інструкції, які пропонує архітектура ЦП x64). Наприклад, поточна мова асемблера x86-64 все

ще може виконувати арифметичні операції з 32-розрядними регістрами за допомогою таких інструкцій, як addl, subl, andl, orl тощо, де l означає «long», що становить 4 байти/32 біти. 64-бітна арифметика виконується за допомогою addq, subq, andq, orq тощо, де q означає «quadword», що становить 8 байтів/64 біти.

Іншим прикладом є те, що компілятори повинні підтримувати паралелізм. Сучасні комп'ютери сьогодні можуть робити багато речей одночасно, а сучасні процесори стають високопаралельними процесорами з різними рівнями паралелізму (наприклад, ARM Big.Little на рисунку 1.1). Паралелізм скрізь, від паралельних блоків виконання в ядрі ЦП до набора інструкцій SIMD (одна інструкція, кілька даних) і паралельного виконання кількох потоків. Однією з оптимізацій, які часто застосовують сучасні компілятори в паралельних обчисленнях, є векторизація. Він являє собою процес перетворення алгоритму зі скалярної реалізації, яка одночасно обробляє одну пару операндів, до векторної реалізації, яка обробляє одну операцію над декількома парами операндів одночасно. Програмісти можуть використовувати векторизацію за допомогою компіляторів для прискорення певних частин коду. Однією з гарячих тем дослідження в інформатиці є пошук методів автоматичної векторизації [12]: пошук методів, які б дозволили компілятору перетворювати скалярні алгоритми у векторизовані алгоритми без втручання людини.

Коротше кажучи, щоб впоратися з різнорідними апаратними платформами, розробники програмного забезпечення використовують ці висококонфігуровані компілятори (для скомпільованих мов, таких як C або C++), щоб ефективно компілювати свої програми з вихідним кодом високого рівня та виконувати їх на вершині діапазону плат. платформ і процесорів.

Іноді розробники програмного забезпечення намагаються уникнути апаратної неоднорідності. Таким чином, вони використовують, наприклад, керовані мови, такі як Java, Scala, C# тощо, щоб сприяти переносимості програмного забезпечення. Замість компіляції до рідного набору машинних інструкцій ці мови компілюються в проміжну мову або IL, яка схожа на бінарну мову асемблера. Ці інструкції виконуються JVM або віртуальною машиною CLR

.NET, яка ефективно перетворює їх у рідні двійкові інструкції, специфічні для архітектури центрального процесора та/або ОС машини. За допомогою керованого коду керування пам’яттю, наприклад перевірка безпеки типу та знищення непотрібних об’єктів (наприклад, збирач сміття), обробляється внутрішньо в цій пісочниці виконання ізольованого програмного середовища. Таким чином, розробники зосереджуються на бізнес-логіці додатків, щоб забезпечити більш безпечне та стабільне програмне забезпечення, не надто піклуючись про неоднорідність апаратного забезпечення. Однак використання керованих мов має недоліки. Він включає повільнішу швидкість запуску (керований код має бути JIT-компільований віртуальною машиною). Він також може бути повільнішим, ніж рідний код, і, як правило, більш жадібним з точки зору системних ресурсів. Наприклад на рисунку 1.2 видно, що мова C є найбільш широко використовуваною мовою програмування в контексті вбудованих систем де система справді обмежена в ресурсах. На відміну від керованих мов, C максимально використовує апаратне забезпечення завдяки багатопроцесорним і багатопоточним API, наданим POSIX. Він також контролює керування пам’яттю та використовує менше пам’яті (що дає більше свободи в управлінні пам’яттю порівняно з використанням збирача сміття).

Альтернативним підходом до встановлення відповідності програмного забезпечення та апаратного забезпечення є створення нових мов і компіляторів для конкретної області з нуля. Наприклад, в роботі [13] представено мову програмування на основі контейнерів для гетерогенних багатоядерних систем. Цей DSL дозволяє програмістам писати уніфіковані програми, здатні ефективно працювати на гетерогенних процесорах. Щоб зіставити цей DSL із апаратними процесорами, надається набір компіляторів і середовищ виконання для ЦП x86 і графічних процесорів CUDA. Подібним чином в роботах [14, 15] запропоновано використовувати DSL для відображення коду програми високого рівня на різнорідні пристрої. Результати показують, що представлений DSL може досягти високої продуктивності на гетерогенному паралельному обладнанні без модифікації вихідного коду. Проведені порівняння продуктивності цієї мови з кодом MATLAB показали, що вона перевершує її майже в усіх випадках.

Коротше кажучи, неоднорідність апаратного забезпечення створює багато проблем для спільноти програмного забезпечення, якій потрібно створювати або мати справу з висококонфігурованими генераторами (тобто компіляторами), щоб справді скористатися перевагами нового чіпа з більш просунутою оптимізацією для нових налаштувань апаратного забезпечення.

1.2 Від класичної розробки ПЗ до генеративного програмування

У порівнянні з класичним підходом, коли розробка програмного забезпечення виконувалася вручну, сучасна розробка програмного забезпечення вимагає більш автоматичних і гнучких підходів до безперервних інновацій у промисловому виробництві, як описано в попередніх розділах. Отже, застосовуються більш загальні інструменти, методи та техніки, щоб зробити процес розробки програмного забезпечення максимально простим для тестування та обслуговування, а також задовольнити різні вимоги задовільним та ефективним способом. Як наслідок, методи генеративного програмування (GP) все частіше застосовуються для автоматичного створення та повторного використання програмних артефактів.

Генеративне програмування – це парадигма інженерії програмного забезпечення, заснована на моделюванні програмних сімейств таким чином, що, враховуючи специфікацію конкретних вимог, високоспеціалізований і оптимізований проміжний або кінцевий продукт може бути автоматично виготовлений на вимогу з елементарних повторно використовуваних компонентів реалізації за допомогою знань про конфігурацію [16].

Генеративна розробка програмного забезпечення полягає у використанні методів програмування вищого рівня, таких як метапрограмування, моделювання, DSL тощо, щоб забезпечити нове інтегроване рішення розробки програмного забезпечення, яке дозволяє використовувати різні виміри різноманітності програмного забезпечення.

У принципі, концепцію генеративного програмування можна розглядати як відображення між простором задач і простором рішень [17] (див. рис. 1.5).

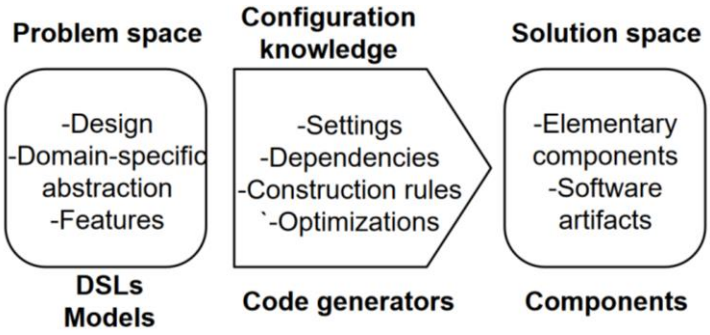


Рисунок 1.5 – Концепція генеративного програмування

Проблемний простір – це набір предметно-спеціальних абстракцій, які можуть використовуватися інженерами додатків для вираження своїх потреб і визначення бажаної поведінки системи. Цей простір зазвичай визначається як DSL або моделі високого рівня.

Простір рішення складається з набору компонентів реалізації, які можуть бути складені для створення системних реалізацій (наприклад, створення специфічних для платформи компонентів програмного забезпечення, написаних з використанням GPL, таких як Java, C++ тощо).

Знання про конфігурацію становлять відображення між обома просторами. Воно приймає специфікацію як вхідні дані та повертає відповідну реалізацію як вихідні дані. Воно визначає правила побудови (тобто правила трансляції, які застосовуються для трансляції вхідної моделі/програми в конкретні компоненти реалізації) та оптимізації (тобто оптимізацію можна застосувати під час генерації коду для покращення деяких нефункціональних такі властивості, як швидкість виконання). Воно також визначає залежності та параметри між специфічними для домену концепціями та функціями.

Ця схема об’єднує кілька концепцій розроблення, керованого моделлю, таких як предметно-спеціальні мови, моделювання функцій і генератори коду.

Деякі загальні переваги такого процесу розробки програмного забезпечення:

Менший обсяг реконструкції/обслуговування системи, спричинений вимогами специфікації;

Полегшене повторне використання компонентів/частин системи;

Збільшення модульності системи та її декомпозиції.

Вирішення проблеми неоднорідності цільових програмних платформ, автоматично генеруючи код.

Прикладом застосування генеративного програмування є використання ліній програмних продуктів (SPL) [9]. Різноманітність програмного забезпечення на основі SPL часто поєднується з методами генеративного програмування [16], які дозволяють автоматично створювати вихідний код із моделей варіативності. Цей метод передбачає використання автоматичних генераторів коду для створення коду, який задовольняє вимоги користувача (моделі SPL). Ця техніка дозволяє керувати набором пов’язаних функцій для створення різноманітних продуктів у певній області. Таким чином, це рішення здатне контролювати різноманітність програмного забезпечення, обробляючи різноманітність вимог, таких як вимоги користувачів або обмеження чи зміни середовища.

1.2.1 Огляд генеративного процесу розробки програмного забезпечення

Процес генеративної розробки програмного забезпечення включає багато різних технологій. У цьому розділі більш детально описано різні види діяльності та зацікавлених сторін, які беруть участь у автоматичному перетворенні специфікацій високорівневої системи у виконувані програми, від часу розробки до виконання.

Чотири завдання, необхідні для забезпечення автоматичної генерації коду:

Розробка програмного забезпечення: як частина процесу генеративного програмування, перший крок полягає в представленні поведінки системи. На стороні введення можна використовувати код як вхід або абстрактну форму, яка представляє дизайн. Це залежить від типу генератора коду та програми джерела вхідних даних, яка йому потрібна. Ці програми можуть варіюватися від формальної

специфікації поведінки системи до абстрактних моделей, які представляють бізнес-логіку. Наприклад, розробники можуть визначити під час проектування поведінку програмного забезпечення, використовуючи, наприклад, Domain-Specі c Models (DSM). DSM – це система абстракцій, яка описує вибрані аспекти сфери знань і концепцій реального світу, що мають відношення до домену, який необхідно спроектувати. Ці моделі специфікуються за допомогою абстрактних мов високого рівня (тобто DSL).

Генерація коду: Генерація коду – це техніка створення коду за допомогою програм. Загальною особливістю генератора є створення коду, який розробник програмного забезпечення інакше написав би вручну. Генератори коду, як правило, розглядаються як чорний ящик, який вимагає як вхідні дані програму, а як вихідні дані генерує вихідний код для конкретної цільової програмної платформи/мови. Генерація коду може створювати код для однієї чи кількох цільових мов один або кілька разів. Існують різні різновиди аспектів генерації коду, і це сильно залежить від категорії введення, як описано в попередньому кроці. Наприклад, розробники генераторів коду використовують методи моделювання, щоб автоматично генерувати код. Замість того, щоб зосередити свої зусилля на побудові коду, вони будують моделі і, зокрема, створюють перетворення моделей, які перетворюють ці моделі в нові моделі або код. Таким чином, процес генерації коду починається з використання попередньо визначеної специфікації, щоб перевести модель у реалізацію цільовою мовою.

Розробка програмного забезпечення: розробку програмного забезпечення можна розділити на дві основні частини. З одного боку, розробники програмного забезпечення можуть виконувати два попередні кроки, щоб автоматично генерувати код для конкретної цільової програмної платформи. У цьому випадку вони можуть редагувати специфікацію системи, описану на першому кроці (на високому рівні), і кожного разу повторно генерувати код, викликаючи певний генератор. У деяких випадках згенерований код можуть навіть редагувати кінцеві розробники програмного забезпечення. Це завдання залежить від складності згенерованого коду. Іноді потрібна допомога експертів із домену, які мають

достатньо досвіду та знань, щоб легко оновлювати та підтримувати автоматично згенерований код. З іншого боку, вони можуть вручну реалізувати вихідний код з нуля, не вдаючись до будь-яких абстракцій або аспектів генерації коду. У цьому випадку вони можуть інтегрувати написаний вручну код із автоматично згенерованим, щоб надати остаточний програмний продукт.

Компіляція: після створення або впровадження коду використовується класичний компілятор (за потреби) для перекладу згенерованого коду у виконуваний. Цей переклад залежить від цільової апаратної платформи, і вибір відповідного компілятора для використання залежить від розробника програмного забезпечення. Компілятори потрібні для націлювання на неоднорідні та різноманітні типи апаратних архітектур і пристроїв. Як приклад, крос-компілятори можна використовувати для створення виконуваного коду для платформи, відмінної від тієї, на якій працює компілятор. Якщо згенерований код потрібно запускати на різних машинах/пристроях, розробнику програмного забезпечення потрібно використовувати різні компілятори для кожної цільової програмної платформи та розгортати згенеровані виконувані файли на різних машинах, що є виснажливим і трудомістким завданням.

На рисунку 1.6 представлено генеративний процес розробки.

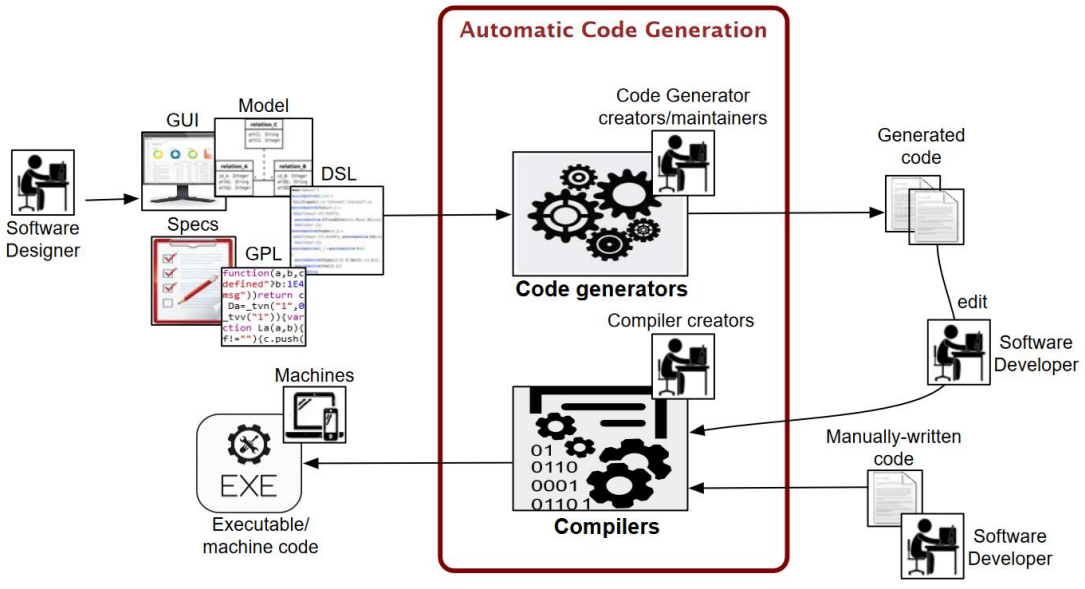


Рисунок 1.6 – Огляд генеративного процесу розробки програмного забезпечення

1.2.2 Автоматичне генерування коду в GP: процес з високою можливістю налаштування

Серед основних переваг, які пропонує GP, є автоматична генерація коду, виділена червоною рамкою на рисунку 1.6 . Автоматична генерація коду виникає у двох основних аспектах:

Використання генераторів коду для роботи з різноманіттям програмного забезпечення та автоматичного створення коду для широкого діапазону програмних платформ, яке описано в розділі 1.1.2 .

Використання компіляторів для роботи з неоднорідністю апаратного забезпечення та автоматичного створення коду для широкого діапазону апаратних платформ, яке описано в розділі 1.1.1 .

І компілятори, і генератори коду відповідають за автоматичне створення коду в GP. Щоб задовольнити різноманітні вимоги до програмного та апаратного забезпечення, сучасні генератори надають багато варіантів конфігурації, щоб легко налаштувати згенерований код:

Компілятори, з одного боку, стають висококонфігурованими та дуже зручними для користувача, дозволяючи користувачеві легко вводити оптимізацію та налаштовувати машинний код відповідно до параметрів цільового обладнання. Як приклад табл 1.1 зображено кількість оптимізацій, доступних у трьох популярних компіляторах. Користувач може налаштувати компілятор, вибравши одну з 2^n можливих послідовностей оптимізації, де n – кількість оптимізацій, доступних у компіляторі. Видно, що простір конфігурації дуже великий.

Таблиця 1.1 – Кількість оптимізацій у LLVM, GCC та ICC

Компілятор	Кількість оптимізацій	Кількість комбінацій
LLVM	100	$_2{}^{100}$
GCC	250	$_2{}^{250}$
ICC	75	$_2{}^{75}$

З іншого боку, генератори коду пропонують можливість налаштувати згенерований код для цільової програмної платформи. Генератори коду надають загальні параметри конфігурації, необхідні для створення артефактів програмного забезпечення (наприклад, вибір цільової мови програмування, залежностей, параметрів платформи, бібліотек тощо). Як приклад, JHipster є конкретним прикладом застосування генеративного програмування в промисловості. JHipster – це генератор додатків на основі генератора YO, який надає інструменти для швидкого створення сучасних веб-додатків за допомогою стека Java на стороні сервера (за допомогою Spring Boot) і адаптивного веб-інтерфейса на стороні клієнта (з AngularJS і Bootstrap). Створена веб-програма може дуже відрізнятися від одного користувача до іншого. Це справді залежить від опцій/вибору, вибраних користувачем для створення налаштованої програми. Вибрані значення параметрів налаштовуватимуть спосіб створення коду генераторами коду JHipster. Наприклад, рис 1.7 показує модель функцій деяких прикладів конфігурації, які користувач може вибрати. Під час створення додатків користувач може вибрати тип бази даних, яку він буде генерувати, версію Java, мережевий протокол тощо. Використовуючи цю функціональну модель, можна вибрати понад 10 тисяч різних типів архітектури проекту, що означає, що можна створити 10 тисяч варіантів програми в залежності від різних критеріїв. Яку б конфігурацію не вибрав користувач, поведінка програми не зміниться, а створена програма матиме схожу архітектуру та базовий код.

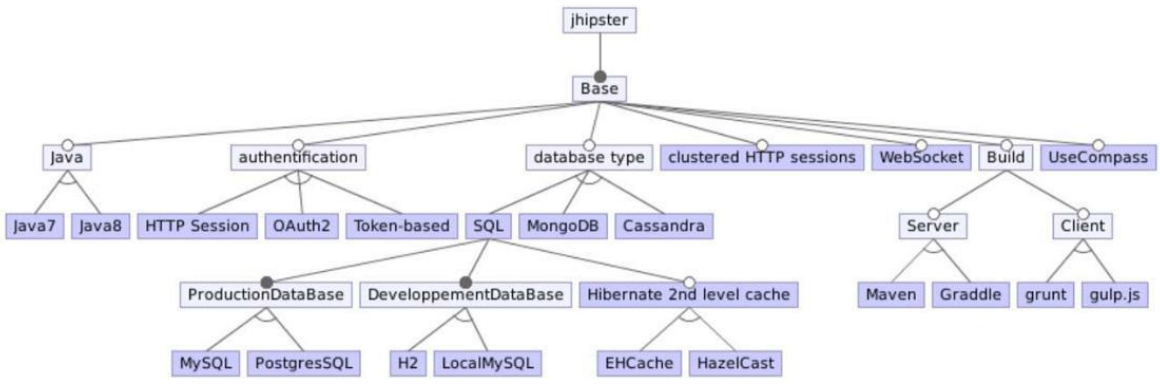


Рисунок 1.7 – Приклад моделі Jhipster

Що ж, видно, що обидва генератори мають високу конфігурацію. Однак на практиці генератори коду менше використовуються в промисловості порівняно з компіляторами. По-перше, тому що для створення та оптимізації машинного коду потрібні компілятори. Тоді користувачі популярних компіляторів, таких як GCC або LLVM, мають достатньо досвіду та впевненості щодо правильного перекладу коду. З іншого боку, генератори коду використовуються рідше, тому що користувачі не мають достатнього досвіду роботи з ними, і їм потрібно отримати впевненість у їх правильному функціонуванні шляхом їх ретельного тестування.

Таким чином, автоматична генерація коду в GP стикається з двома основними проблемами. З одного боку, генератори, які можна конфігурувати, повинні бути ефективно налаштовані, щоб виробляти високоякісні програмні продукти. З іншого боку, вони мають бути ретельно перевірені, щоб надати користувачам докази ефективності згенерованого коду.

1.2.3 Зацікавлені сторони та їхні ролі для тестування та налаштування генераторів

Розробка програмного забезпечення включає кілька зацікавлених сторін, які відіграють різні ролі для перевірки та тестування ланцюга розробки програмного забезпечення, описаного раніше. рисунок 1.8 зображено діаграму варіантів використання, яка описує ці різні проблеми, учасників і ролі для тестування та налаштування генераторів. Можна виділити двох зацікавлених сторін: користувачів-генераторів і творців/підтримувачів. Як показано в нижній частині малюнка 1.8, творці/підтримувачі несуть відповідальність за правильність функціонування генераторів. Вони використовують свій досвід і знання, пов’язані з технологіями програмного та апаратного забезпечення, що забезпечує ефективне генерування коду. Вони роблять свій внесок у спільноту розробників програмного забезпечення, створюючи та надаючи нові оновлення генераторів (наприклад,

запроваджуючи нові оптимізації, створюючи нові генератори для певної платформи або вдосконалюючи існуючі).

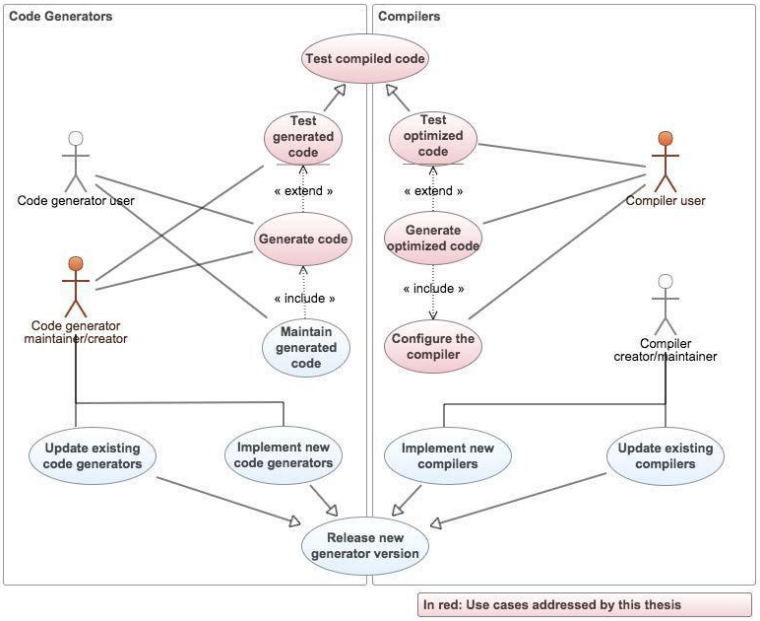


Рисунок 1.8 – Діаграма варіантів використання різних акторів/ролей, залучених до тестування та налаштування генераторів

З іншого боку, користувачі генератора представляють групу розробників програмного забезпечення, які не мають знань/досвіду про те, як генерується код. Таким чином, вони не в змозі редагувати або підтримувати внутрішню поведінку генераторів (наприклад, у випадку комерційних генераторів і генераторів у формі оболонки). У цьому випадку генератори використовуються як компоненти чорного ящика інженерами під час розробки програмного забезпечення, щоб полегшити створення коду. Розробники можуть налаштовувати генератори для ефективного створення коду для цільової апаратної платформи (наприклад, шляхом надання набору параметрів оптимізації) або підтримувати/редагувати згенерований код у разі автоматичної генерації вихідного коду.

Варіанти використання, виділені на рисунку 1.8 червоним кольором складають основні завдання, які вирішуються в цій дипломній роботі. Головне завдання – оцінити нефункціональні властивості автоматично створеного коду. Залучаються дві зацікавлені сторони в генераторах, творці/підтримувачі і

користувачі (виділені червоним). З одного боку, ця робота допоможе розробникам/супроводжувачам коду протестувати згенерований код і виявити проблеми з генератором коду, оцінивши використання ресурсів і характеристики продуктивності. Це завдання також може залучати користувачів генераторів коду, але це в основному завдання експертів з генераторів. З іншого боку – користувачам компіляторів автоматично налаштувати компілятори за допомогою оптимізації, наданої експертами з компіляторів. Він полягає в оцінці впливу цих конфігурацій на властивості продуктивності та використання ресурсів, щоб знайти найкращий набір оптимізацій для конкретної програми та цільової архітектури обладнання.

1.3 Тестування генераторів коду

1.3.1 Процес тестування

Основною метою генераторів є створення програмних систем із специфікацій вищого рівня. Як було сказано раніше, робота по створенню коду ділиться на два рівні. Починається транс формування системи у вихідний код за допомогою генераторів коду. Після цього вихідний код перетворюється на виконувані файли за допомогою компіляторів. Таким чином, розробники програмного забезпечення використовують для створення коду, редагування його (за потреби), компіляції та тестування. Якщо зміни застосовуються до компіляторів або генераторів, цикл повторюється. рисунок 1.9 представляє огляд цього циклу тестування. У правій частині малюнка показано класичну роботу з розробки та налагодження коду, яка полягає в редагуванні, компіляції та тестуванні. Користувач пише або редагує існуючий код, компілює його за допомогою спеціальних компіляторів і тестує. Генерація коду додає кілька нових елементів робочого циклу в лівій частині малюнка, де творці генератора редагують шаблони та правила перетворення (або сам генератор), а потім запускають генератор коду для створення нових вихідних файлів. Вихідні файли потім компілюються, і програма тестується.

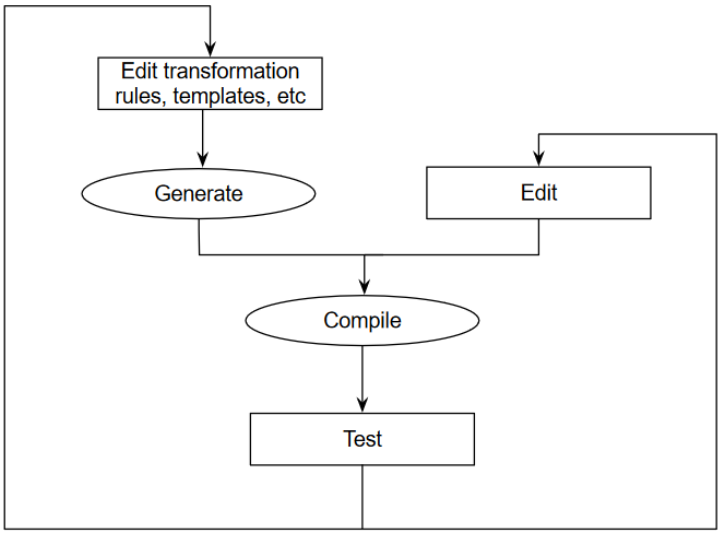


Рисунок 1.9 – Робота з генерування коду

1.3.2 Типи генераторів коду

Є багато способів класифікації генераторів. Можна відрізнити їх за складністю, використанням або за введенням/виведенням. Відповідно до [18], є дві основні категорії автоматичної генерації коду: пасивна або активна. Генератори пасивного коду створюють код лише один раз, потім користувач повинен оновлювати та підтримувати код. Найбільш поширеним використанням пасивних генераторів коду є майстри (wizards).

Активні генератори коду, запускають код кілька разів протягом життєвого циклу. З активними генераторами коду є код, який можуть редагувати користувачі, і код, який слід змінювати лише за допомогою генератора коду. Активні генератори коду широко згадуються в літературі [19, 20]. У цій дипломній роботі увага зосереджена на тестуванні цього класу генераторів коду. У літературі [18, 21, 22, 23], дослідники визначають шість категорій активних генераторів коду.

Переробник коду: переробник коду читає код як вхідні дані, а потім створює новий код як вихідні дані. Цей новий код може бути частковим або повним залежно від конструкції генератора. Генератор коду є найпоширенішою формою генераторів коду, і він широко використовується. Цей тип генераторів часто

використовується для автоматичного створення документів. Компілятор джерела до джерела, транскомпілятор або транспілер також можуть бути визначені як розбійники коду. Транскомпілятор бере код, написаний на якійсь мові програмування, і перекладає його в код, написаний на іншій мові. Внесок роботи, пов'язаний з тестуванням генераторів коду, буде зосереджено на цьому типі генераторів, щоб підтвердити обраний підхід до автоматичного виявлення невідповідностей.Приклади: C2J, JavaDoc, Jazillian, Closure Compiler, Coccinelle, Co eeScript, Dart, Haxe, TypeScript і Emscripten

Розширювач вбудованого коду: ця модель читає код як вхідні дані та створює новий файл, який використовує вхідний код як основу, але з розширеними розділами коду на основі дизайну оригінального. Це починається з розробки нової мови. Зазвичай ця нова мова є існуючою мовою з деякими розширеннями синтаксису. Розширювач вбудованого коду потім використовується для перетворення цієї мови на робочий код мовою високого рівня. Приклади: вбудовані мови SQL, такі як SQLJ (для Java) і Pro*C (для C). SQL можна вбудувати в код C або Java. Генератор будує робочий код C шляхом розширення SQL у код C, який, наприклад, реалізує запити.

Генератор змішаного коду: ця модель має таку саму швидкість обробки, що й розширювач вбудованого коду, за винятком того, що вхідний файл є справжнім вихідним файлом, який можна скомпілювати та запустити. Згенерований вихідний файл зберігає оригінальну розмітку, яка вказуватиме місце розміщення згенерованого коду. Це дозволяє генерувати код для кількох невеликих фрагментів коду в одному файлі або розподілені по кількох файлах. Зазвичай правила перетворення визначаються за допомогою регулярних виразів. Приклади: Codify – комерційний генератор змішаного коду, який може генерувати кілька фрагментів коду в одному файлі за допомогою спеціальних команд. Іншим прикладом є заміна коментарів у вхідному файлі на відповідний код.

Генератор часткового класу: Генератор часткового класу приймає абстрактне визначення як вхідні дані замість коду (наприклад, діаграму класу UML), а потім створює вихідний код. Потім користувач може розширити його, створивши похідні

класи та додавши методи для завершення дизайну. Перетворення моделей на код здійснюється через послідовність перетворень. Наприклад, платформно-незалежна модель (PIM) трансформується в платформо-спеціальну модель (PSM). Потім генерація коду виконується з PSM за допомогою певного типу перетворень коду на основі шаблонів. Приклади: ArgoUML і Codegen перекладають діаграми класів UML на мови загального призначення, такі як C#, Java і C++. Вони не створюють повних реалізацій, але вони намагаються перетворити вхідні діаграми класів UML у каркасний код, який користувач може легко редагувати. EMF GenCode також є частковим генератором.

Генератор рівня: у цій моделі генератор будує повний набір вихідного коду з абстрактного визначення. Він має ту саму концепцію, що й генератор неповних класів. Велика різниця між генерацією рівня та частковим класом полягає в тому, що в моделі рівня генератор створює весь код для рівня. Цей код призначений для використання без розширення. Однак модель часткового генератора класів дозволяє інженеру створювати решту похідних класів, які доповнять функціональність рівня. Приклади: рівень доступу до бази даних, рівень веб-клієнта, експорт, імпорт даних або рівні перетворення.

Повнодоменна мова: доменні мови – це в основному нові мови, які мають типи, синтаксис і операції, і вони використовуються для певного типу проблеми. Мови домену є крайнім кінцем автоматичної генерації коду, оскільки розробникам доводиться писати компілятор для кожної проблемної області та мови. Приклади: Matlab – це математична мова для предметної області, яка використовується для представлення математичних операцій, DSL, таких як ThingML і його генератори коду.

1.3.3 Чому тестування генераторів коду складне?

Тестування генераторів коду викликає велику кількість різних проблем. Нижче описано деякі з них.

Проблема оракула: Щоб автоматизувати процес тестування, потрібні тестові оракули, щоб оцінити, чи пройдено тест чи ні. Тестовий оракул перевіряє, чи відповідає очікуванням результат виконання тесту. У випадку функціонального тестування генераторів коду тестовий оракул може бути легко визначений. Наприклад, це може бути визначено як результат порівняння між змодельованою або виконаною моделлю та її відповідною реалізацією. Однак у випадку нефункціонального тестування генераторів коду визначити тестовий оракул складно. Насправді згенерований код має відповідати певним вимогам продуктивності (наприклад, швидкість виконання, використання ресурсів тощо).

Неможливість модульного тестування: Повністю протестувати весь генератор коду за допомогою традиційних підходів до тестування програмного забезпечення неможливо через складність інструменту. Коли справа доходить до модульного тестування генераторів коду, кожна функція перекладу повинна бути відокремлена від програмної системи та оточена тестовою системою. Це означає відокремлення кожного правила перекладу та тестування його окремо. Однак це неможливо, оскільки важко розглядати цю специфічну функціональність окремо під час тестування системи в цілому. Розглянемо, наприклад, функціональне тестування функції перекладу для оператора суми (+). Відповідно до [24], існує понад 2000 способів реалізації функції a = b + c, оскільки операція залежить від типів даних і від того, чи ввімкнено обмеження даних.

Складність генераторів коду: Генератори коду можуть бути складними для розуміння, оскільки вони зазвичай складаються з численних елементів, чиї складні взаємозалежності створюють серйозні проблеми для розробників, які виконують завдання проектування, реалізації та обслуговування. Враховуючи складність і різнорідність технологій, задіяних у генераторі коду, розробники, які намагаються перевірити та зрозуміти процес генерації коду, мають мати справу з численними різними артефактами. Як приклад, у сценарії обслуговування генератора коду розробнику може знадобитися знайти всі ланцюгові прив’язки між моделлю та моделлю до тексту, які створюють рядок коду з помилками, щоб виправити це. В таких задачах буває багато помилок, коли вони виконуються вручну [25]. Таблиця

1.2 показує, як приклад, деякі метрики генератора коду TargetLink версії 2.0. TargetLink це генератор коду, який генерує робочий код (код C) безпосередньо з графічних моделей MAT-LAB/Simulink/State. Ця таблиця показує, наскільки великий базовий код генератора коду. Маючи понад 1 800 000 рядків коду, дуже важко протестувати всю систему.

Таблиця 1.2 – Метрики генератора коду TargetLink

Метрики	TargetLink 2.0
Кількість класів	3000
Кількість файлів	6000
Кількість функцій	51 000
Рядки (всього)	1 800 000
Рядки коду	990 000
Рядки коментарів	560 000

– Невиконувана вихідна модель: Генератори коду не завжди підтримують виконувані вихідні моделі. Іноді генератори коду, такі як генератор часткових класів, генерують лише структурний код через серію перетворень із невиконуваної моделі (наприклад, UML-діаграми). Далі користувачам належить розширити згенерований код шляхом реалізації похідних класів. У випадку невиконуваних моделей стає складно автоматично перевірити правильну поведінку створеного коду, як це описано в специфікації моделі, оскільки не можна порівняти його виконання, наприклад, із симульованою моделлю.

1.4 Автоматичне налаштування компіляторів

Компілятори відіграють важливу роль у автоматичному створенні швидкого й ефективного цільового машинного коду. Це не є тривіальним завданням, оскільки

потенційно існує багато варіантів машинного коду для однієї програми. Отже, завдання компілятора полягає в тому, щоб знайти та створити найкращу версію машинного коду для будь-якої програми. З цієї причини компілятори зазвичай намагаються автоматично оптимізувати код, щоб покращити його продуктивність. Цей процес називається оптимізацією коду.

1.4.1 Оптимізація коду

Оптимізація коду в компіляторі – це процес перетворення вихідного коду програми в інший функціонально еквівалентний код з метою покращення однієї чи кількох його нефункціональних властивостей. Найпоширенішим результатом оптимізації є підвищення продуктивності. Інші менш поширені нефункціональні властивості – це розмір коду, використання пам'яті та енергоспоживання. Існує багато типів оптимізації, зокрема розгортання циклу, автоматичне розпаралелювання, перевпорядкування блоків коду та інлайнінг функцій. Апаратні характеристики, які впливають на вплив оптимізації, можуть включати: кількість регістрів ЦП (чим більше регістрів, тим легше оптимізувати продуктивність), розмір кешу, архітектуру ЦП тощо.

Оптимізацію можна умовно розділити на два типи:

- машинонезалежна оптимізація;
- машинозалежна оптимізація.

Генерація проміжного коду всередині компіляторів може викликати багато недоліків, таких як додаткові копії змінних і використання змінних замість констант. Такий тип оптимізації усуває такі недоліки та покращує код. Таким чином, компілятор приймає проміжний код і перетворює частину коду незалежно від будь-яких регістрів ЦП або місць пам'яті. Ці оптимізації загалом змінюють структуру програм. Оптимізації, які застосовуються до концепцій абстрактного програмування (структур, циклів, об’єктів, функцій), не залежать від машини, на яку націлений компілятор. Приклади: усунення надмірності, розгортання циклу,

усунення непотрібного та недоступного коду, вбудовування функцій, усунення мертвого коду тощо.

Машинозалежні оптимізації застосовуються після генерації цільового коду та коли код трансформується відповідно до архітектури цільової машини. Вони використовують переваги спеціальних функцій апаратного забезпечення для створення коду, який є коротшим або який виконується швидше на машині, наприклад вибір інструкцій, розподіл реєстрів, планування інструкцій, паралелізм тощо. Вони здебільшого включають регістри процесора та посилання на пам’ять. Машинозалежні оптимізатори докладають зусиль, щоб максимально використати переваги ієрархії пам’яті. Вони більш ефективні та мають кращий вплив на продуктивність, ніж незалежні оптимізації, оскільки вони найкраще використовують спеціальні функції цільового обладнання. Приклади: оптимізація розподілу регістрів для ефективного використання регістрів, передбачення розгалужень, оптимізація циклу тощо

1.4.2 Чому автоналаштування компіляторів є складним?

Сьогодні сучасні компілятори реалізують велику кількість оптимізацій. Кожна оптимізація намагається покращити продуктивність програми введення.

З одного боку, оптимізаційні компілятори в наш час стають досить складними. Створення оптимізацій компілятора для нового мікропроцесора є важкою та довготривалою роботою, оскільки вона вимагає всебічного розуміння базової апаратної архітектури, а також ефективного способу оцінити вплив оптимізації на нефункціональні властивості.

З іншого боку та з точки зору користувача компілятора, застосування та оцінка оптимізацій складне завдання, оскільки визначення оптимального набору оптимізації було визначено як головна дослідницька проблема в літературі [26].

Нижче описано кілька проблем, які роблять роботу з автоматичного налаштування компілятора дуже складною.

1. Сумісні цілі: оптимізація компіляторів має підтримувати різноманітні взаємозв’язані цілі, такі як час виконання/швидкість компіляції, час виконання/використання ресурсів тощо. Важко визначити набір оптимізацій, який задовольняє всі властивості.

2. Взаємодія оптимізації: взаємодія між фазами оптимізації, а також їх порядок застосування ускладнюють пошук оптимальної послідовності.

3. Величезна кількість оптимізацій: величезна кількість оптимізацій також є проблемою для користувача компілятора, щоб вибрати найкращу послідовність оптимізації, оскільки вичерпний пошук.

4. Неуніверсальна оптимізація: не існує універсальної послідовності оптимізації, яка покращить продуктивність усіх програм. Вплив оптимізації залежить від апаратного забезпечення та програми введення. Таким чином, побудувати послідовність оптимізації для різних програм і апаратних архітектур стає дуже важко.

5. Помилки компілятора: застосування оптимізацій може призвести до помилок у готовому коді та помилок компілятора [27, 28]. Тому налаштування компілятора не повинно викликати жодних змін у поведінці програми.

6. Вплив оптимізації: оптимізований код має бути швидким і ефективним. Оптимізація має покращувати деякі нефункціональні властивості, а не викликати регресію продуктивності/накладні витрати.

7. Для налаштування компіляторів потрібен досвід: якщо користувач компілятора не має знань і досвіду щодо технології компілятора та його оптимізації, йому буде досить важко вибрати набір необхідних послідовностей оптимізації для застосування.

1.5 Висновки

У цьому розділі описано основні проблеми тестування та налаштування генераторів коду.

Неоднорідні середовища виконання: різноманітність існуючих програмних середовищ і платформ, а також неоднорідність апаратного забезпечення дуже ускладнюють тестування генераторів. Розгортання та виконання автоматично згенерованих артефактів програмного забезпечення на платформі займає багато часу. Таким чином, потрібен ефективний засіб для полегшення тестування та налаштування генераторів. Як можна використовувати нові досягнення в технологіях розробки програмного забезпечення, щоб протистояти постійним інноваціям апаратного та програмного забезпечення під час тестування/налаштування генераторів?

Проблема оракула при тестуванні генераторів коду: автоматична генерація коду дає багато переваг порівняно з традиційними методами розробки програмного забезпечення (наприклад, швидкість розробки, продуктивність тощо). Однак генератори коду – це складні частини програмного забезпечення, які самі по собі можуть містити помилки. Тому вони потребують ретельного тестування. Проблема тестового оракула, розглянута в розділі 1.3.3 є однією з основних проблем, пов’язаних з автоматичним нефункціональним тестуванням генераторів коду. Ця проблема також виникає в автоматичному функціональному тестуванні, коли мова йде про невиконувані моделі. Доведення того, що згенерований код є функціонально правильним, недостатньо, щоб стверджувати про ефективність тестованого генератора коду. Як щодо нефункціональних вимог, таких як споживання ресурсів? Як можна ефективно виявити нефункціональні проблеми?

Великий простір пошуку оптимізації під час автоматичного налаштування компіляторів: компілятори можуть мати величезну кількість потенційних комбінацій оптимізації, що ускладнює та забирає багато часу для розробників програмного забезпечення, щоб знайти/конструювати послідовність оптимізацій, яка задовольняє конкретні ключові цілі користувача та критерії. Це також вимагає повного розуміння доступних оптимізацій компілятора та їх взаємодії. Крім того, важко знайти послідовність оптимізації, яка представляє обмін між двома суперечливими цілями. Отже, як можна допомогти користувачам компіляторів

автоматично налаштувати компілятори та вибрати набір для оптимізації, який задовольняє певні нефункціональні вимоги?

Моніторинг використання ресурсів автоматично згенерованого коду. Аналіз використання ресурсів оптимізованого або згенерованого коду вимагає динамічного та адаптивного рішення, яке ефективно виділяє ці властивості. Через різноманіття програмного забезпечення та неоднорідність апаратного забезпечення моніторинг використання ресурсів кожної виконавчої платформи стає складним і займає багато часу.

2 ІСНУЮЧІ РІШЕННЯ

2.1 Тестування генераторів коду

Тестування коду, написаного вручну, завжди було важливим завданням для забезпечення коректності коду. Воно має на меті довести, що код є функціонально правильним за допомогою таких методів, як модульне тестування, інтеграційне тестування, приймальне тестування тощо. Ці методи допомагають знайти помилки, які допускають інженери під час розробки коду. Коли використовується генератор коду, необхідні адекватні підходи до тестування для виявлення помилок, викликаних автоматичною генерацією коду. Перевірка правильності генератора коду підвищить довіру до інструменту, і користувачі продовжуватимуть використовувати його для генерації робочого коду.

2.1.1 Функціональне тестування генераторів коду

Більшість попередньої роботи з тестування генератора коду зосереджена на перевірці правильної функціональної поведінки згенерованого коду [29, 30, 31, 32, 33, 24, 34].

У випадку автоматичної генерації коду для виконуваних моделей були запропоновані різні підходи для автоматичної перевірки трансляції моделі в код. Метою верифікації є перевірка того, що згенерований код коректно реалізує спроектовану модель. Таким чином, модель тестується на відповідність її вимогам, а код може бути перевірений на відповідність виконуваній моделі за допомогою динамічного тестування. Для цього застосовується як модель, так і згенерований код для подальшої експлуатації.. Цей підхід представлено та обговорено в кількох дослідницьких роботах [35, 29, 32, 33, 24]. Автори цих робіт стверджують, що цей підхід застосовний не лише для генераторів коду на основі моделі, але й для всіх типів генераторів коду, якщо вхідна модель/вихідний код не є виконуваним.

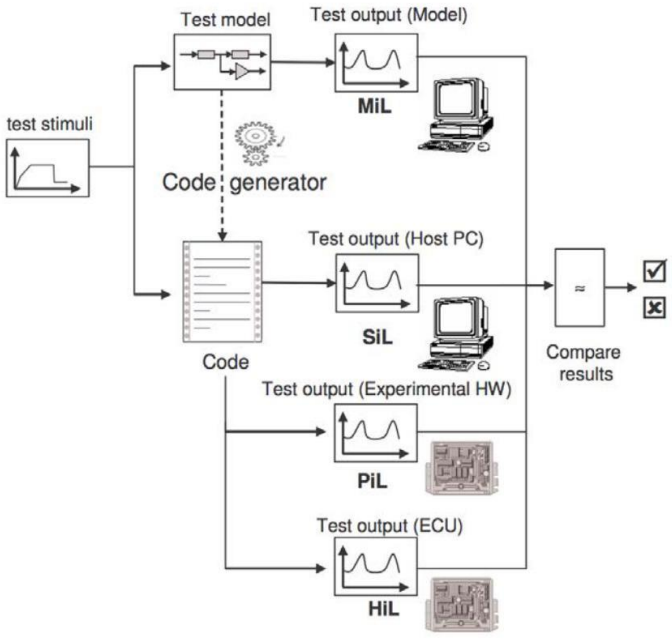


Рисунок 2.1 – Автоматичне функціональне тестування генераторів коду

Як показано на рисунку 2.1, як згенерований виконуваний файл, так і змодельована модель виконуються з однаковими вхідними даними. Визначення вхідних тестових стимулів може використовувати критерії структурного тестування на рівні моделі (покриття моделі) і на рівні коду (покриття коду) для створення високоякісних тестових векторів. Після цього два результати порівнюються за певними критеріями прийнятності. Процедура порівняння відома в спільноті тестувальників програмного забезпечення як підхід до еквівалентності, порівняння або паралельного тестування [36, 37].

Великою перевагою цього підходу є те, що тестовий оракул легко визначити. Він являє собою порівняння між двома або більше вихідними результатами. Відповідно до [38,29], існує чотири етапи порівняння, які можуть бути виконані. Вони описані наступним чином (див. рис. 2.1):

Модель у циклі (Model-in-the-Loop MiL): моделювання моделі на головній машині називається MiL. Метою тесту MiL є отримання еталонних результатів тесту (очікуваних значень). Крім того, моделювання MiL фіксує задану поведінку моделі, яка пізніше має бути реалізована мовою загального призначення. Він також перевіряє валідність моделі щодо функціональних вимог. Єдина проблема, яка

може виникнути під час виконання MiL, полягає в тому, що модель не зможе виконати.

Програмне забезпечення в циклі (Software-in-the-Loop SiL): виконання згенерованого об’єктного коду на головній машині з тими самими стимулами, що використовуються для MiL, називається SiL. Результати виконання мають бути порівнянними з результатами, отриманими під час MiL. Метою SiL є виявлення помилок перекладу, таких як арифметичні проблеми, та вимірювання покриття коду. Як тільки система виявляє дефект, середовище тестування має надати тестувальнику відповідний навігаційний інструмент для переходу до помилкової змінної даних, щоб виправити її.

Процесор у циклі (Processor-in-the-Loo PiL): PiL перевіряє об’єктний код на цільовому процесорі. Він генерує перехресно скомпільований вихідний код і виконує його на машині цільового процесора. Звичайно, можна застосувати оптимізацію компілятора для підвищення якості коду. Потім тестовий сценарій виконується на цільовому процесорі (наприклад, цільові вбудовані системи, як у [38]). Метою PiL є перевірка поведінки коду на цільовому процесорі та вимірювання ефективності

Апаратне забезпечення в циклі (Hardware-in-the-Loo HiL): нарешті, під час HiL виконується програмне забезпечення, вбудоване в цільовий чіп. З цією метою цільове обладнання підключається до системи моделювання в реальному часі, що імітує завод. Спочатку розроблена модель більше не моделює сигнали фізичного середовища, для цієї мети спеціально розроблено спеціальне обладнання. Мета HiL – перевірити правильну поведінку програмного забезпечення на реальному обладнанні.

У цьому контексті в роботі [32] застосовано підхід, описаний вище, презентовано підхід автоматизованого тестування для оцінки еквівалентності між моделями Simulink і згенерованим кодом. Цей підхід називається перевіркою генерації коду (CGV). CGV оцінює чисельну еквівалентність між використовуваною моделлю (тобто моделі Simulink) і згенерованим кодом (тобто виконувані файли, отримані зі згенерованого коду C). Фактично, за кожною

окремою трансляцією моделі в код слідує фаза перевірки, щоб оцінити, чи вхідна модель Simulink, яка використовується для генерації коду, і вихід (тобто об’єктний код, отриманий з моделі через генерацію та компіляцію коду) створюють однакові чисельні результати при стимуляції з ідентичними входами. У своєму підході до тестування на еквівалентність вони використовують для запуску моделі, яка використовується для генерації коду, за допомогою моделювання та згенерованого коду з тими самими вхідними стимулами (тестовими векторами) з подальшим чисельним порівнянням вихідних даних (векторів результатів). Потім вони перевіряють, чи збереглася семантика моделі під час генерації коду, компіляції та зв’язування, шляхом порівняння векторів результатів, які є виходами в результаті стимуляції, з ідентичними тестовими векторами моделі та згенерованого коду. Точніше, результати моделювання повинні бути схожі на результати виконання. Однак при визначенні порівняння вектора результатів вони допускають обмежені відмінності між обома результатами. Вони стверджують, що деякі фактори між симуляцією та виконанням можуть спричинити невелику різницю між обома виконаннями, такі як обмежена точність чисел набору точок, цільові оптимізовані конструкції коду тощо. Таким чином, вони визначають залежне від програми порогове значення. Отже, два вектори результатів вважаються достатньо подібними, якщо їх різниця менша за конкретне порогове значення. Вони ілюструють перевірку перекладу на основі CGV у контексті вбудованого автоматичного програмного забезпечення за допомогою Simulink і Real-Time Workshop Embedded Coder для перевірки. Вони оцінюють свій підхід, перевіряючи чисельну еквівалентність між моделями Simulink і згенерованим кодом C. Вони обчислюють абсолютну різницю між результатами моделювання та результатами виконання. Потім вони порівнюють цю різницю з визначеним порогом толерантності. Вони показують, що для деяких наборів вхідних тестів існують неузгоджені сигнали (з високим значенням варіації), які представляють невідповідність між розробленими моделями та виконаним сигналом.

У [31] інструмент автоматичного створення коду сертифіковано відповідно до певного стандарту безпеки (IEC 61508-3). Відповідність моделі стандарту

допомагає продемонструвати, що модель добре сформована відповідно до сертифікації та відповідає всім вимогам для подальшої генерації коду. Процес оцінки генератора коду (описаний вище) використовується, щоб показати, що згенерований код еквівалентний моделі.

У роботі [29] представлено підхід систематичного тестування для генераторів коду на основі моделі. Вони досліджують вплив правил оптимізації на генерацію коду на основі моделі шляхом порівняння виходу коду та виконання моделі. Якщо ці результати еквівалентні, передбачається, що генератор коду працює належним чином. Вони оцінюють ефективність цього підходу за допомогою оптимізації тестування, виконаної генератором коду TargetLink. Тестові вектори генеруються з використанням структурного покриття моделі та згенерованого коду. Вони використовували Simulink як середовище моделювання.

У [33] автори представляють підхід до тестування фреймворку генератора коду Genesys, який перевіряє переклад, виконаний генератором коду, з семантичної точки зору, а не просто перевіряє синтаксичну правильність результату генерації. По суті, Genesys реалізує послідовне тестування шляхом виконання як вихідної моделі, так і згенерованого коду на різних цільових платформах. Обидві страти створюють сліди, а сліди страти порівнюються між собою.

Іншою альтернативою перевірки генератора коду було б використання формальних доказів [39, 40]. Це включає математичний доказ того, що процес трансформації генерації коду правильний і що кожне правило генерації коду зберігає семантику моделі у. [41] розширено існуючий генератор коду таким чином, щоб він створював усі логічні анотації, необхідні для формальних доказів безпеки. Ці докази засвідчують, що програма не порушує певних умов під час її виконання. Цей підхід інтегровано в генератори коду AUTOBAYES і AUTOFILTER. Він використовувався, щоб довести, що згенерований код задовольняє як специфічні властивості мови, такі як безпека меж масиву або правильна ініціалізація змінної, так і специфічні властивості домену, такі як векторна нормалізація, симетрія матриці або правильне використання вхідних даних датчика.

Нижче наведено короткий огляд існуючих методів, які застосовуються для тестування автоматичної генерації коду,спираючись на роботу [35], деякі з них описані вище:

Таблиця 2.1 – Зведення кількох підходів, застосованих для тестування генераторів коду [35]

Рівень	Техніка тестування	Цілі
Модель	Функціональне MiL тестування/ симуляція	Перевірити, що модель відображає специфікацію функціональних вимог Перевірити працездатність моделі в середовищі розробки без ресурсних обмежень цільового середовища
	Структурне MiL тестування (модельне покриття)	Дослідити можливі шляхи в рамках моделі, визначивши тестові кейси на основі структури моделі
	Прийняття керівних принципів моделювання	Спиратися на досвід та експертні знання Виявляти помилки проектування на ранній стадії розробки
Генератор коду	Сертифікація інструменту	Незалежне схвалення, яке гарантує, що методики, застосовані для розробки та верифікації інструменту, відповідають вимогам стандарту сертифікації
	Тестування	Переконатися, що генератор коду пройшов ретельне тестування Переконатися, що специфічні функції перекладу (наприклад, оптимізації) поводяться так, як очікувалося
	Формальний доказ	Показати за допомогою математичних доведень, що кожна генерація коду (правило) зберігає семантику моделі
	Прийняття стандартів та керівних принципів	Переконатися, що генератор коду був розроблений відповідно до систематичного процесу розробки / системи управління якістю
Згенерований код	Функціональне SiL тестування	Виявлення помилок перекладу
	Функціональне PiL тестування	Перевірити коректність поведінки коду з урахуванням врахування цільової архітектури процесора
	Функціональне HiL тестування	Перевірити поведінку коду при його розгортанні на цільовому апаратному пристрої
	Структурне MiL/HiL/PiL тестування	Визначити тестові кейси на основі структури коду та дослідити можливі шляхи виконання
	Статичний аналіз	Перевірити відповідність коду керівним принципам/стандартам кодування Виявлення можливостей для оптимізації, таких як мертвий код тощо.

2.1.2 Нефункціональне тестування генераторів коду

Попередня робота з нефункціонального тестування генераторів коду зосереджена на порівнянні, як Oracle, нефункціональних властивостей рукописного коду з автоматично згенерованим кодом [42, 43]. Як приклад, у [44] реалізовано просту 2D-гру в обох Мова програмування Haxe (мова високого рівня) і цільова мова програмування, а також проведено оцінку різниці в продуктивності між двома версіями коду. Показано, що згенерований через Haxe код має кращу продуктивність, ніж написаний від руки.

У [45] автори порівнюють деякі нефункціональні властивості двох генераторів коду, генератора коду TargetLink і вбудованого кодувальника Real-Time Workshop Embedded Coder. Вони також порівнюють ці властивості з написаним вручну кодом. Код виконується на мікропроцесорі C166 як цілі, яка є вбудованою системою. Метрики, які використовуються для порівняння, це використання пам’яті ROM і RAM, швидкість виконання, читабельність і відстежуваність. Виконується багато тестів, щоб перевірити, чи контролер поводиться належним чином. Результати порівняння показують, що згенерований TargetLink код є більш ефективним, ніж написаний вручну та інший згенерований код з точки зору пам’яті та часу виконання. Вони також показують, що згенерований код можна легко відстежувати та редагувати.

Кросплатформна мобільна розробка також була частиною цілей нефункціонального тестування, оскільки багато генераторів коду все частіше використовуються в промисловості для автоматичної кросплатформної розробки. У [46, 47], автори порівняли продуктивність набору кросплатформних генераторів коду та представили найбільш ефективні інструменти.

Одним із найважливіших аспектів, який цікавить нас під час тестування генераторів коду, є тестовий оракул. Це механізм, який перевіряє, чи правильні виходи програми для виконаних тестів чи ні.

Порівняно з багатьма аспектами автоматизації тестування, проблема автоматизації тестового оракула все ще є складною та менш добре вирішеною. Для

створення тестових оракул доступно лише кілька методів. У більшості випадків розробка та реалізація тестових оракул все ще є ручною та дорогою діяльністю. Це пов’язано з тим, що тестові оракули не завжди доступні, і їх може бути важко визначити або занадто складно застосувати [48]. Це широко відоме як «проблема оракула». Як зазначено в [49], проблема оракула була «однією з найскладніших задач у тестуванні програмного забезпечення», але дослідники її часто ігнорують.

У цьому контексті автоматичне тестування генераторів коду особливо передбачає проблему оракула. Під час тестування компіляторів, наприклад, це досить важко автоматично перевірити еквівалентність між вихідним кодом і об'єктним кодом. Це завдання ще більше ускладнюється, коли до машинного коду застосовуються деякі оптимізації. Під час тестування генераторів коду перевірка еквівалентності між специфікацією системи високого рівня та згенерованим кодом також є складною.

Як описано в розділі 2.1.1, що стосується автоматичного функціонального тестування генераторів коду, тестовий оракул часто визначається як послідовне порівняння між результатом специфікації системи та згенерованим кодом. Коли справа доходить до перевірки нефункціональних властивостей, таких як використання ресурсів або швидкість виконання, ця проблема стає більш критичною. Не існує чіткого визначення того, як слід визначати оракул, за винятком кількох дослідницьких зусиль, обговорених у розділі 2.1.2 .

Існує кілька підходів [50, 48] для полегшення проблеми оракула. У дослідженні [50] класифіковано тестові оракули на три категорії: оракули, неявні оракули та похідні оракули. Нижче наведено опис цих трьох категорій.

Визначені оракули: можуть бути створені з кількох типів специфікацій, таких як алгебраїчні специфікації або формальні моделі поведінки системи. Наприклад, автори [51, 52] обговорюють підхід для отримання тестових випадків і оракул із специфікації. Ідея полягає в тому, що формальну специфікацію програмного забезпечення можна використовувати як керівництво для розробки функціональних тестів. Тоді тестові оракули можна пов’язати з окремими тестовими шаблонами (специфікаціями тестових випадків). Таким чином, вони

створюють абстрактні моделі очікуваних результатів, які називаються шаблонами оракулів. Цей підхід проілюстровано шаблонами тестового оракула для специфікації. Визначені оракули ефективні у виявленні помилок припущенихгенераторами. Однак завдання визначення та підтримки специфікацій є дуже дорогим і трудомістким. Таким чином, застосовність зазначених оракулів обмежена, і вони також менш прийняті в промисловості.

Неявні оракули: відноситься до виявлення очевидних помилок, таких як збій програми, ненормальне завершення або помилка виконання. Таким чином, визначення Oracle не потребує жодних знань домену чи формальної специфікації для реалізації, і, як наслідок, воно не потребує жодних передумов щодо поведінки чи семантики програми, що тестується. Неявні оракули [50, 48] легко отримати практично безкоштовно. У той же час неявні оракули здебільшого неповні, оскільки не здатні ідентифікувати внутрішні проблеми та складні збої, але вони допомагають виявляти за допомогою чорної скриньки загальні помилки, такі як збої системи або необроблені винятки. Як приклад, нечітке тестування [53] є одним із методів, де неявні оракули використовуються для виявлення аномалій, таких як збої. Ідея фаззингу полягає в генеруванні випадкових вхідних даних і передачі їх тестовій системі для виявлення аномалій. Виявлення помилок базується на ефективності згенерованих вхідних даних/даних. Якщо виявлено аномалію, тестер повідомляє про це, визначаючи вхід, який її запускає. Фаззинг зазвичай використовується для виявлення вразливостей безпеки, таких як перевантаження буфера, витік пам’яті, винятки тощо [54]. У [55] представлено підхід до перевірки надійності системи, що тестується, використовуючи неявні оракули. Цей підхід ґрунтується на створенні та виконанні недійсних вхідних тестів надійності. Зокрема, ці тести призначені для виявлення збоїв і зависань, спричинених недійсними введеннями для викликів функцій. Результати показують, що від 42% до 63% компонентів вимірюваних систем POSIX мали проблеми з надійністю. Робота [56] зосереджена на розробці шаблонів для виявлення аномалій. В ній розглянуто підмножину можливих аномалій, які можна знайти у веб-додатках, наприклад проблеми з навігацією, неузгодженість гіперпосилань тощо. Їхні

емпіричні результати показують, що 60% веб-додатків, розглянутих у їх дослідженні, демонструють аномалії та збої у виконанні.

Похідні оракули: є похідними від різних артефактів (наприклад, документації, виконання системи) або властивостей, відмінних від специфікацій. Наприклад, у регресійному тестуванні оракули можуть бути отримані з виконання попередніх версій тестованого програмного забезпечення. У цьому випадку похідні оракули перевірятимуть, чи нова версія програмного забезпечення поводиться як оригінальна [57]. Наприклад, EvoSuite і Randoop отримують тестові оракули з попередніх версій тестованої системи. Іншим типом похідних оракул є псевдооракули (також відомі як диференційне тестування, подвійне кодування та програмування N-версій [58]). Концепцію псевдооракула ввели Девіс і Вейюкер [59]. Псевдооракул – це програма, яка здатна надавати очікувані результати та перевіряти правильність системи, порівнюючи результати кількох незалежних реалізацій. Він перевіряє узгодженість результатів різних версій програмного забезпечення систем, коли виконується одна і та ж функція. Неузгодженість можна виявити, коли одна або кілька версій системи викликають збої. Наприклад, під час тестування компілятора різні версії однієї програми генеруються шляхом застосування деяких оптимізацій. Функціональність тестованої програми залишається однаковою для всіх версій. У цьому випадку оракул визначається як порівняння функціональних результатів різних версій [28].

В таблиці 2.2 подано короткий виклад кількох методів визначення оракула, описаних вище:

Таблиця 2.2 – Резюме підходів тестового оракула

Оракул	Метод	Цілі
Визначені оракули	Заяви та контракти Мови, засновані на специфікаціях Алгебраїчні мови специфікацій	Використання понять специфікацій як джерела оракулової інформації.
Неявні оракули	Нечітке тестування Навантажувальне тестування Перевірка на стійкість	Виявлення очевидних помилок, таких як збої, витоки пам'яті, необроблені винятки, ненормальне завершення програми тощо

Кінець таблиці 2.2

Оракул	Метод	Цілі
Похідні оракули	Метаморфічне тестування N-версійне програмування Регресійне тестування Back-to-back тестування Виявлення інваріантів	Оракули виводяться з різних артефактів (наприклад, документація, виконання системи) або властивостей (наприклад, метаморфічні зв'язки) системи, що тестується. Перевірте узгодженість результатів різних версій систем, коли виконується однакова функціональність.

2.2 Техніка автоналаштування компіляторів

2.2.1 Ітеративна компіляція

Ітеративна компіляція, також відома як вибір фази оптимізації, адаптивна компіляція або оптимізація, спрямована на зворотний зв’язок [60], полягає у застосуванні техніки розробки програмного забезпечення для створення кращих і оптимізованіших програм. Ключова мета ітеративної компіляції полягає в тому, щоб знайти найкращу послідовність оптимізації, яка веде до найшвидшого та найякіснішого машинного коду.

Основна ідея ітеративної компіляції полягає в дослідженні простору оптимізації компілятора шляхом вимірювання впливу оптимізації на продуктивність програмного забезпечення. Кілька дослідницьких зусиль досліджували цю проблему оптимізації, наприклад методи програмної інженерії на основі пошуку (SBSE), щоб спрямувати пошук до відповідних оптимізацій, що знижують продуктивність, енергоспоживання, розмір коду, час компіляції тощо. Експериментальні результати зазвичай були порівняно зі стандартними рівнями оптимізації компілятора.

Було доведено, що оптимізація сильно залежить від цільової платформи та програми введення, що робить завдання пошуку найкращої послідовності оптимізації дуже складним [60].

2.2.2 Реалізація процесу ітераційної компіляції

Реалізація ітераційної системи компіляції полягає головним чином у застосуванні послідовності кроків для підвищення якості згенерованого коду. рисунок 2.2 показує загальний огляд основних кроків, необхідних для забезпечення реалізації процесу ітераційної компіляції.

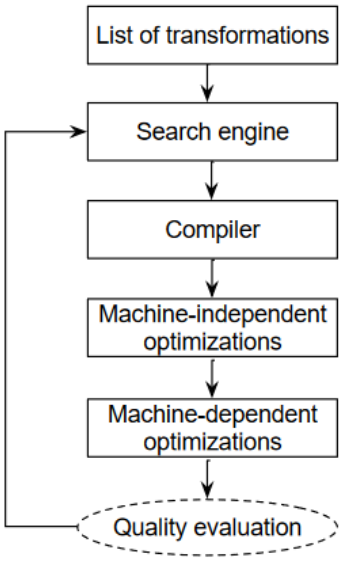


Рисунок 2.2 – Огляд процесу ітераційної компіляції

1. Список перетворень: Ітераційний процес починається з визначення простору оптимізації. Він представляє список оптимізацій, які компілятор повинен розвинути під час пошуку найкращих послідовностей оптимізації.

2. Пошукова система: Він застосовує алгоритм або метод пошуку для ефективного дослідження великого простору пошуку для оптимізації. Фактично, він приймає як вхідні дані попередньо визначений список перетворень і вирішує, які оптимізації будуть збережені в кінці пошуку.

3. Компілятор: Як тільки послідовність оптимізації визначена, цільовий компілятор (наприклад, GCC, LLVM) викликається для компіляції вхідної програми, а також виконує початкову машинно-незалежну оптимізацію.

4. Машинонезалежні оптимізації: Результатом цього є початкова машинно-незалежна оптимізована програма. Ці оптимізації виконуються під час генерації

коду та впливають на всі цільові системи. Він включає в себе оптимізації, які застосовуються під час відображення дерева аналізу на проміжний код, і оптимізацію, застосовану до самого проміжного коду.

5. Машинно-залежна оптимізація: Для подальшої оптимізації компілятор застосовує машинно-залежну оптимізацію з наданої послідовності оптимізації. Це включає оптимізацію, застосовану під час відображення проміжного коду в асемблер, і оптимізацію, застосовану до згенерованого об’єктного коду.

6. Оцінка якості: Він полягає в оцінці якості оптимізованого коду. Можна оцінити багато нефункціональних властивостей, як-от розмір коду, час виконання, використання ресурсів, енергоспоживання тощо.

Ця модель представляє класичний і типовий ітеративний процес компіляції. Звичайно, існує багато способів і адаптацій для здійснення цього процесу. Реалізація ітераційного процесу залежить від використовуваного алгоритму, вирішеної проблеми, використовуваних технологій тощо.

2.2.3 Ітеративна техніка компіляції

2.2.3.1 Автоналаштування: оптимізація монооб’єкта

Техніка автонастроювання компілятора використовувалася в багатьох сценаріях оптимізації. Спільним для всієї цієї попередньої роботи над ітеративною компіляцією є те, що вона зосереджена на одній цільовій функції, яку потрібно оптимізувати. Наприклад, дослідники зазвичай зосереджуються на прискоренні продуктивності скомпільованого коду, що є основною метою оптимізації для більшості ітеративних підходів до компіляції [61, 62, 63, 64, 65, 74].

Таким чином, цю проблему часто адаптували як одноцільову задачу пошуку, де прискорення є основною проблемою для більшості попередніх робіт. Генетичні алгоритми (GA) [75, 76] представляють привабливе вирішення цієї проблеми. Підходи на основі GA обчислюють початкову популяцію за допомогою набору оптимізацій, які зазвичай визначаються на стандартних рівнях компілятора -Ox.

Потім на кожній ітерації індивідууми (тобто набори опцій), які складають генерацію, оцінюються шляхом вимірювання часу виконання кожного рішення. Результати сортуються та проходять стадію розмноження та мутації для формування наступного покоління. Цей процес триває, доки не буде досягнуто умови завершення. Зрештою, алгоритм повертає найкращий набір оптимізації, який призвів до найвищої продуктивності.

Наприклад, ACOVEA (Analysis of Compiler Options via Evolutionary Algorithm) – це інструмент із відкритим кодом, який застосовує GA для пошуку найкращих варіантів для компіляції програм за допомогою компілятора GCC. У цьому контексті найкращі рішення визначають ті параметри, які створюють найшвидшу виконувану програму з певного вихідного коду. Цей інструмент навіть було включено до репозиторію Gentoo Linux, щоб допомогти користувачам знайти найкращий набір оптимізацій.

Структура ESTO [76] вивчає застосування GA до проблеми вибору оптимального набору параметрів для конкретного застосування та робочого навантаження. ESTO розглядає компілятор як чорну скриньку, визначену зовнішніми параметрами оптимізації. ESTO підтримує варіант GA під назвою генетичний алгоритм з обмеженим бюджетом, який експоненціально зменшує розмір популяції, а потім зменшує час, необхідний для оцінки різних оцінок. Автори провели експерименти з пакетом тестів SPEC2000 і перевірили 60 варіантів оптимізації в трьох компіляторах: GCC, XLC і FDPR-Pro. Результати ESTO порівнюються з GCC -O1 і -O3, з XLC -O3 і з FDPR-Pro -O3. Результати показують, що ESTO здатний створювати рівні оптимізації, які забезпечують кращу продуктивність, ніж стандартні варіанти.

2.2.3.2 Ескранування локального оптимуму

Поширеною проблемою ітераційної компіляції є локальний оптимум. Насправді простір пошуку оптимізацій для конкретної програми може бути дуже

величезним і, як правило, містить багато локальних мінімумів, у яких пошуковий алгоритм може бути захоплений [77]. Тому дослідники в цій галузі намагаються створити надійні методи та алгоритми, щоб уникнути такої проблеми. У [77] автори спробували проаналізувати цей пошуковий простір і вияснили, що простір оптимізації є дуже нелінійним, містить багато локальних мінімумів і деякі розриви. Цей розділ присвячено параметризованим перетворенням. Невелика область простору трансформації, яка розглядається в цій статті, складається з трьох параметризованих оптимізацій: розгортання циклу (з коефіцієнтами розгортання від 1 до 20), мозаїка циклу (з розміром мозаїки від 1 до 100) і заповнення (від 1 до 100).). Вони зосереджені на скороченні компіляції та часу виконання оптимізованих програм і використовують симулятор для націлювання на вбудовані процесори. Вони використовують структуру компілятора, розроблену для оптимізації мультимедійних програм для вбудованих систем. Вони аналізують цю оптимізацію на чотирьох архітектурах ЦП (UltraSparc, R10000, Pentium Pro і TriMedia-1000), і множення матриці вибирається як програма для оптимізації. Запропонований алгоритм пошуку відвідує кілька точок з інтервалами, застосовуючи відповідне перетворення, виконуючи перетворену програму та оцінюючи її цінність шляхом вимірювання часу виконання. Ті точки, що знаходяться між поточним глобальним мінімумом і середнім, додаються до впорядкованої черги. Ітеративно такі точки видаляються з черги, а точки в сусідньому регіоні досліджуються, знову ж таки через проміжки часу. Цей процес продовжується, доки не буде оцінено певну кількість балів і не буде повідомлено найшвидше трансформовану програму. Вони показують, що у випадку великих просторів трансформації вони можуть досягти 0,3% найкращого можливого часу, відвідавши менше 0,25% простору. Вони знаходять мінімум після відвідування менше ніж 1% простору.

У роботі [78] автори описують свій досвід дослідження простору пошуку послідовностей компіляції. Вони повідомляють про результати вичерпного перерахування кількох просторів пошуку послідовностей довжиною 10, вибраних із 5 перетворень. Вони показують, що простір пошуку має багато локальних

мінімумів, і що підйом на пагорб із випадковим повторним запуском є ефективною стратегією для подолання цієї проблеми.

Іншим способом ефективного дослідження великого простору пошуку в оптимізації компілятора є техніка Design Space Exploration (DSE) [69, 70]. DSE базується на підході кластеризації для групування функцій зі схожістю та дослідження скороченого простору пошуку в результаті поєднання оптимізацій, запропонованих раніше для функцій у кожній групі. Для виявлення подібності між функціями використовується метод інтелектуального аналізу даних, який застосовується до представлення символічного коду. Вони порівнюють свій підхід із GA, і їхні експериментальні результати показують, що підхід на основі DSE досягає значного скорочення загального часу дослідження простору пошуку (у 20 разів порівняно з підходом GA) і прискорення продуктивності (41% порівняно з базовим рівнем).

2.2.3.3 Проблема впорядкування фаз

Упорядкування фаз також є важливою проблемою в ітераційній компіляції, яка досліджує вплив різних порядків фаз оптимізації на продуктивність програми. Фактично, при використанні деяких компіляторів, таких як LLVM, важливо визначити правильний порядок застосування оптимізацій. Таким чином, дослідники в цій галузі намагаються застосувати методи пошуку, щоб знайти правильну послідовність оптимізації. Однак перевпорядкування фаз оптимізації надзвичайно важко підтримувати в більшості виробничих систем, включаючи GCC, через використання в них кількох проміжних форматів і складних властивих залежностей між оптимізаціями. Тому, як правило, компілятори внутрішньо керують порядком застосування оптимізацій і не дають користувачеві руки вибрати цей порядок, уникаючи конфліктів і проблем компіляції.

Коли порядок керується користувачами, вичерпна оцінка всіх порядків фаз оптимізації є неможливою через величезну кількість фаз оптимізації. Ця проблема

стає складнішою через те, що ці фази комплексно взаємодіють з іншими оптимізаціями. Наприклад, навіть якщо збережено той самий набір оптимізацій для вхідної програми, зміна порядку застосування цих фаз оптимізації може призвести до створення різного коду з значними варіаціями продуктивності серед них.

У цій галузі автор [79] розробив структуру, засновану на аксіоматичних специфікаціях оптимізації, включаючи умови до та після застосування оптимізації. Для вибраного набору оптимізацій структура використовується для визначення тих взаємодій між оптимізаціями, які можуть створити умови, і тих, які можуть зруйнувати умови для застосування інших оптимізацій. Потім із цих взаємодій виводиться порядок застосування, щоб отримати потенційні переваги оптимізації, які можна застосувати до програми. Цей фреймворк використовувався для переліку потенційних увімкнення та вимкнення взаємодії між оптимізаціями, які потім використовувалися для визначення порядку застосування для оптимізацій

Автори [80, 81] запропонували вичерпну стратегію пошуку для пошуку оптимальних послідовностей компіляції для кожної функції програми. Вони вичерпно перерахували всі окремі екземпляри функцій для набору програм, які будуть створені з різних упорядкованих фаз 15 оптимізацій. Цей вичерпний перелік став можливим завдяки визначенню того, які фази були активними та чи був згенерований код унікальним, що зробило фактичний простір порядку фаз оптимізації набагато меншим, ніж спробований простір. Цей вичерпний перелік дозволив їм побудувати ймовірності ввімкнення/вимкнення взаємодії між різними проходами оптимізації загалом, а не конкретно для будь-якої програми. Вони використовують цю ідею, щоб запобігти комбінаторному вибуху загальної кількості послідовностей, які потрібно перевірити. Вони змогли знайти всі можливі екземпляри функцій, які можуть бути створені за допомогою різних упорядкованих фаз для 109 із 111 функцій, які вони оцінили.

Автор [74] адаптує GA для вирішення проблеми впорядкування фаз оптимізації. Вони націлені на вбудовані системи та зосереджені на зменшенні розміру коду. Вони обирають 10 програмних перетворень для розвитку в компіляторі Fortran. Рішення, згенеровані їхнім алгоритмом, порівнюються з

рішеннями, знайденими за допомогою фіксованої послідовності оптимізації. Їхня методика була успішною для зменшення розміру коду на 40% порівняно зі стандартною послідовністю.

В іншій роботі [78] ті ж автори досліджували порядок фаз на програмному рівні за допомогою алгоритмів рандомізованого пошуку на основі GA, альпіністів і рандомізованої вибірки. Вони націлені на імітований абстрактний процесор на основі RISC із дослідницьким компілятором. Вони повідомляють про властивості кількох згенерованих підпросторів фазового впорядкування та наслідки цих властивостей для алгоритмів пошуку.

2.2.3.4 Оцінка ітераційної оптимізації для кількох наборів даних

Більшість ітеративних досліджень оптимізації знаходять найкращу оптимізацію компілятора шляхом повторних запусків того самого набору даних. Проблема полягає в тому, що якщо обрано найкращу послідовність оптимізації для вхідного набору даних за допомогою ітераційного процесу, не відомо, чи буде вона найкращою для тієї ж програми, але з іншими наборами даних. Таким чином, дослідники в цій галузі намагаються дослідити цю проблему, оцінюючи ефективність ітераційної оптимізації на великій кількості наборів даних. Зокрема, оскільки не існує існуючого набору тестів із великою кількістю наборів даних, у. [65] автори спробували зібрати 1000 наборів даних під назвою KDataSets для 32 програм, в основному отриманих із тесту MiBench. Потім вони здійснюють ітеративну оптимізацію цих зібраних наборів даних, щоб знайти найкращу комбінацію оптимізації для всіх наборів даних. Вони використовують випадковий пошук для генерації випадкових послідовностей оптимізації для компілятора ICC (53 групи) і компілятора GCC (132 оптимізації). Вони демонструють, що для всіх 32 програм (від MiBench) вони змогли знайти принаймні одну комбінацію оптимізацій компілятора, яка забезпечує 86% або більше найкращого можливого прискорення для всіх наборів даних за допомогою ICC (83% для GCC GNU). Ця

оптимальна комбінація є специфічною для програми та забезпечує прискорення до 1,71 на ICC і 2,23 на GCC порівняно з найвищим рівнем оптимізації (-Ofast і -O3 відповідно). Це означає, що програму можна оптимізувати на основі колекції наборів даних і вона може підтримувати майже оптимальну продуктивність для більшості інших наборів даних. Однак вони перевірили свій підхід лише на одному тесті та одній цільовій архітектурі.

2.2.3.5 Сукупні цілі: багатоцільова оптимізація

Кілька дослідницьких зусиль намагаються знайти обмін між двома (чи більше) нефункціональними властивості [63, 66, 61, 67, 68, 69, 71, 72].

У COLE [66] автори вважають, що проблему оптимізації компілятора можна розглядати як багатоцільову проблему, де дві нефункціональні властивості можна покращити одночасно. Таким чином, вони досліджували стандартні рівні оптимізації компілятора шляхом пошуку оптимальних рівнів Парето, які максимізують продуктивність і час компіляції. Вони показують, що за допомогою багатоцільового генетичного алгоритму (у своєму експерименті вони використовували SPEA2) можна знайти набір послідовностей оптимізації компілятора, які є більш ефективними за Парето з точки зору продуктивності та часу компіляції, ніж стандартні рівні оптимізації. (-O1, -O2, -O3 і -Os). Мотивація такого підходу полягає в тому, що ці стандартні рівні були встановлені вручну творцями компілятора на основі фіксованих тестів і наборів даних. На думку авторів, ці універсальні рівні можуть не завжди бути ефективними для невидимих програм, і існують вищі рівні, які забезпечують кращу торгівлю з точки зору якості коду. Вони використовували тест SPEC2000 CPU, який є популярним набором тестів для оцінки продуктивності компілятора. Вони розробили 60 оптимізаційних оголошень, які визначені на стандартних рівнях -O1, -O2, -O3 і -Os. Вони запускають ітераційну компіляцію на одній машині, що постачається з процесором Intel Pentium 4, і порівнюють запропонований алгоритм (SPEA2) із випадковим

пошуком, а також зі стандартними рівнями оптимізації. Експериментальні результати з використанням GCC (версія 4.1.2) показують, що автоматична конструкція рівнів оптимізації можлива на практиці, і, крім того, вона дає кращі рівні оптимізації, ніж рівні оптимізації, отримані вручну GCC. Однак вони не дають гарантії, що нові рівні оптимізації будуть оптимальними для інших програм.

У дослідженні [72] запропоновано адаптивну структуру компілятора з урахуванням часу виконання в найгіршому випадку для автоматичного пошуку послідовностей оптимізації компілятора. У порівнянні з раніше описаними підходами, автори в цій роботі зосереджуються на створенні ефективного коду для вбудованих систем. Таким чином, вони зосереджені на ключових властивостях для систем реального часу, таких як середній час виконання (ACET), найгірший час виконання (WCET), енергоспоживання та розмір коду. Вони досліджують ефективність оптимізації компілятора з пов'язаними цілями. Таким чином, вони намагаються знайти відповідні обміни між цими цілями, щоб визначити оптимальні рішення за Парето за допомогою багатоцільових алгоритмів. Цільові функції намагаються мінімізувати властивості розміру WCET-ACET і WCET-Code. Вони застосовують три еволюційні багатоцільові алгоритми, а саме IBEA, NSGA-II і SPEA2, і порівнюють свої результати зі стандартними рівнями (-O1, -O2 і -O3). Вони розробляють 30 оптимізацій у компіляторі WCC і проводять експерименти на одній машині, що постачається з чотирьохядерним процесором Intel. Вони підбирають 35 програм з різними бенчмарками, як DSPstone, MediaBench, MiBench тощо. Вони виявили, що NSGA-II є найбільш перспективним алгоритмом для даної проблеми. Виявлені послідовності оптимізації значно перевищують стандартні рівні оптимізації. Фактично, найвищий стандартний рівень оптимізації -O3 може бути перевершений для WCET і ACET в середньому на 31,33% і 27,43% відповідно. Той самий підхід працює також для оптимізації розміру WCET-коду зі зменшенням WCET на 30,6% порівняно з -O3. Однак розмір коду збільшується на 133,4%. Вони стверджують, що WCET і розмір коду є типовими взаємопов'язаними цілями. Якщо бажано значне покращення однієї цільової функції, необхідно прийняти значне погіршення іншої цілі.

У [73] представлено структуру TACT. Порівняно з попередніми підходами, TACT призначений головним чином для автоматичного налаштування вбудованих систем під керуванням Linux. Таким чином, цільовою архітектурою ЦП для цього інструменту є архітектура ARM (ARM Cortex-A9), а в компіляторі GCC для ARM використовується 200 варіантів. TACT підтримує кілька цілей оптимізації, тому він може налаштовуватися або для одного параметра оптимізації, або для двох цільових функцій одночасно, наприклад, для продуктивності та розміру коду (або часу компіляції). Таким чином, він застосовує алгоритм SPEA2 і GA для одноцільової оптимізації. Результати показують, як SPEA2 перевершує стандартні рівні GCC (-O2, -O3 і -Os) у кількох популярних програмах з відкритим кодом, таких як C-Ray, Crafty Chess, SciMark, x264 і zlib.

2.2.3.6 Прогнозування оптимізацій: оптимізація машинного навчання

Кілька дослідників також запропонували машинне навчання для налаштування компіляторів. Порівняно з еволюційними алгоритмами, використання машинного навчання для оптимізації компілятора має потенціал для повторного використання знань у різних циклах ітераційної компіляції, отримуючи переваги ітераційної компіляції для вивчення найкращих оптимізацій у кількох програмах і архітектурах.

Як правило, програми машинного навчання створюють на одній фазі модель прогнозування, яка використовуватиметься для визначення набору оптимізацій компілятора, який слід застосовувати до невидимих програм на онлайн-фазі. Основна перевага цієї методики полягає в зменшенні кількості необхідних оцінок.

У проекті Milepost [82], наприклад, автори починають зі спостереження, що подібні програми можуть демонструвати подібну поведінку та вимагати подібної оптимізації, тому можна співвіднести функції програми та оптимізацію разом, щоб передбачити хороші трансформації для невидимих програм, на основі попереднього досвіду оптимізації. Таким чином, вони забезпечують модульну,

розширювану, самонастроювальну інфраструктуру оптимізації, яка може автоматизувати оптимізацію програми для настроюваних гетерогенних процесорів на основі кореляції між функціями програми, поведінкою під час виконання та оптимізацією. Запропонована інфраструктура базується на компіляторі машинного навчання, який представляє інтерактивний інтерфейс компіляції (ICI) і плагіни для вилучення функцій програми (таких як кількість інструкцій у методі, кількість розгалужень тощо) і вибору проходів оптимізації.

Інфраструктура Milepost складається з двох окремих етапів: навчання та розгортання. Під час фази навчання збирається інформація про структуру програм (вхідні навчальні програми), що показує, як вони поводяться за різних параметрів оптимізації. Така інформація дозволяє інструментам машинного навчання співвідносити аспекти структури програми або функції з оптимізацією, створюючи стратегію, яка передбачає хороші комбінації оптимізацій. Після виконання ітераційного процесу, який оцінює різні комбінації оптимізацій на додаток до навчальних програм/функцій, створюються прогностичні моделі, щоб співвіднести заданий набір функцій програми з професійними перетвореннями програми. Потім, на етапі розгортання, структура аналізує нові невидимі програми, визначаючи функції програми та передає їх новоствореним моделям для прогнозування найбільш професійної оптимізації таблиці для покращення часу виконання або інших показників залежно від вимог оптимізації користувача. В якості інфраструктури компілятора було обрано GCC. Вони розробили 100 оптимізаційних оголошень на рівнях -O1, -O2 і -O3 та порівняли свої результати з рівнем -O3 і випадковим пошуком. Експериментальні результати показують, що можливо покращити продуктивність набору тестів MiBench автоматично за допомогою ітеративної компіляції та машинного навчання на кількох платформах, включаючи x86: Intel і AMD, а також сімейство конфігурованих ядер ARC. Використовуючи фреймворк на основі машинного навчання, вони також змогли вивчити модель, яка автоматично покращує час виконання деяких окремих програм MiBench більш ніж у 2 рази, одночасно покращуючи загальний пакет MiBench на 11% на архітектурі ARC, що перебудовується, без використання жертвуючи

розміром коду або часом компіляції. Крім того, їхній підхід підтримує загальну багатоцільову оптимізацію, коли користувач може мінімізувати не лише час виконання, але й розмір коду та час компіляції.

2.2.3.7 Підсумки: техніка компілятора з автоматичним налаштуванням

В таблиці 2.3 подано короткий виклад кількох ітеративних підходів компіляції, більшість із яких описано вище. Ці підходи класифікованозалежно від проблеми, яку вони вирішують. Іноді дослідницькі роботи стосуються більш ніж одного питання в ітераційній компіляції. Це не вичерпне дослідження всіх ітеративних підходів, але воно дає огляд кількох дослідницьких зусиль, задіяних у різних областях протягом останніх 20 років.

Таблиця 2.3 – Підсумки підходів ітераційної компіляції

	Локальний оптимум	Упорядкування етапів	Множинні набори даних	Конфлікуючі цілі	Методи навчання	Кілька процесорів	Мультикомпілятори
[79]	-	+	-	-	-	-	-
[77]	+	-	-	-	-	+	-
[81]	-	+	-	-	-	-	-
[78]	+	+	-	-	-	+	-
[76]	-	-	-	-	-	+	+
[66]	-	-	-	+	-	-	-
[83]	-	-	-	+	-	-	-
[65]	-	-	+	-	-	+	+
[82]	-	-	-	+	+	+	-
[84]	-	-	-	+	+	-	-
[73]	-	-	-	+	-	-	-

2.3 Легка віртуалізація системи для автоматичного тестування програмного забезпечення

Використання віртуалізації, як-от віртуальних машин (VM), дуже корисно для вирішення проблеми неоднорідності апаратних і програмних платформ. У

промисловості ряд сценаріїв комерційного використання виграють у методів віртуалізації для надання послуг кінцевим користувачам. Наприклад, Amazon EC2 робить віртуальні машини доступними для клієнтів, які можуть використовувати їх для запуску власних комп'ютерних програм або служб у хмарі. Таким чином, користувач може створювати, запускати та завершувати роботу нових віртуальних машин за потреби. Однак відомо, що віртуальні машини дуже дорогі з точки зору системних ресурсів і продуктивності [85]. Насправді кожен новий екземпляр віртуальної машини є віртуальною копією всього апаратного забезпечення хост-машини, що збільшує використання ресурсів і накладні витрати [86]. Віртуалізація на основі контейнерів представляє цікаву технологію, альтернативну віртуальним машинам. Контейнерна технологія – це віртуалізація на рівні операційної системи, яка практично не потребує накладних витрат. Програми у віртуальних екземплярах використовують інтерфейс системного виклику операційної системи, і їх не потрібно емулювати або запускати на проміжній віртуальній машині, як це відбувається у VMware, QEMU або Xen. Наприклад, Docker це популярний механізм, який пропонує можливість розгортати програми та їхні залежності в легких контейнерах, які дуже дешеві у створенні. Процеси, що виконуються в контейнері Docker, ізольовані від інших процесів, що виконуються в головній ОС або в інших контейнерах Docker. Рішення Docker спрямоване на вирішення проблем використання ресурсів і накладних витрат на продуктивність, спричинених віртуалізацією повного стека.

Кілька авторів [85, 87, 86] порівнювали продуктивність традиційного рішення віртуальної машини з технологією контейнерної операційної системи. Вони показали, що контейнери забезпечують кращу продуктивність, ніж віртуальні машини, оскільки вони викликають менше накладних витрат.

2.3.1 Застосування в тестуванні програмного забезпечення

У розробці програмного забезпечення технологія контейнерів все частіше використовується для створення портативного узгодженого операційного середовища для розробки, розгортання та тестування в хмарі [88]. Далі описано деякі сучасні підходи, які вибирають технологію контейнерів як механізм для вирішення деяких проблем дослідницького тестування.

Автори [89] використали Docker як технологічну основу у своїй структурі аналізу сховищ Covrig для проведення широкомасштабної та безпечної перевірки історії версій із шести вибраних сховищ коду Git. Для аналізу вони запускають кожну версію системи ізольовано та збирають статичні та динамічні показники програмного забезпечення, використовуючи полегшене контейнерне середовище, яке можна розгорнути на кластері локальних або хмарних машин. Кожен контейнер використовується для конфігурації, компіляції та тестування однієї версії програми, а також для збору цікавих показників, таких як розмір коду та покриття. Мотивація використання такої інфраструктури полягає в тому, щоб забезпечити чисте та настроюване середовище виконання для проведення експериментів.

За словами авторів, використання Docker як рішення для автоматичного розгортання та виконання різних версій програми полегшило процес тестування.

Інший підхід, заснований на Docker, представлений у проекті BenchFlow2, який зосереджений на порівняльному аналізі механізмів BPMN 2.0 [90]. Цей проект присвячений тестуванню продуктивності робочих двигунів. У цій роботі автори представити структуру для автоматичного та надійного розрахунку показників ефективності для систем управління роботою (WfMS) BPMN 2.0. На думку авторів, бенчмаркінг WfMS викликає багато проблем:

складність розгортання системи через розподілену природу виконання цих моделей;

велика кількість параметрів конфігурації, необхідних для інтеграції розгортання тестованої системи, тобто WfMS;

складність поведінки виконання, яка може бути виражена сучасними мовами моделювання та виконання, такими як BPMN2.

Тому, щоб вирішити ці проблеми, BenchFlow використовує Docker як технологію контейнеризації, щоб забезпечити автоматичне розгортання та конфігурацію WfMS. Таким чином, WfMS автоматично розгортаються та скасовуються за допомогою Docker. Кожен компонент, який бере участь у тестуванні, упаковано як образи Docker, які розгортаються та виконуються на різних серверах, підключених до виділеної локальної мережі. Для кожного екземпляра Docker під час експерименту Wf виконує новий екземпляр набору бізнес-моделей. Завдяки Docker BenchFlow автоматично збирає всі дані, необхідні для обчислення показників продуктивності та перевірки правильності виконання тестів (метрики, пов’язані з використанням оперативної пам’яті/ЦП і часом виконання). Їх експериментальні результати показують, що проста бізнес-модель, яка працює на двох популярних WfMS з відкритим кодом, виявляє важливі проблеми масштабованості продуктивності.

У [91] запропоновано Pons, веб-інструмент для розповсюдження попередніх версій мобільних додатків з метою ручного тестування. Pons полегшує створення, запуск і ручне тестування програм Android безпосередньо в браузері. Заснований на технології Docker, цей інструмент дозволяє розробникам і кінцевим користувачам залучатися до тестування програм в одному місці, полегшуючи тягар тестувальників щодо встановлення та підтримки тестових середовищ і надаючи платформу для розробників, щоб швидко повторювати програмне забезпечення та інтегрувати зміни з часом. . Таким чином, це прискорює процес тестування та знижує його вартість. Pons використовує Docker, попередньо визначаючи образи Docker, які містять необхідні сервіси та інструменти для створення додатків Android, починаючи від операційної системи до набору програмного забезпечення. Потім за допомогою одного з цих зображень створюється контейнер для зберігання вихідного коду мобільного додатку в певний момент історії в середовищі пісочниці. Після цього Pons створює емулятор Android у контейнері Docker для запуску тестів. Результати транслюються під час виконання у веб-браузері.

2.3.2 Застосування в моніторингу часу виконання

Моніторинг виконання є важливим у сфері хмарних обчислень [92]. Як і попередні віртуальні машини, контейнерам потрібен механізм моніторингу. Він повинен надавати як історичну, так і своєчасну інформацію про використання ресурсів контейнерів.

У промисловості пропонується багато комерційних рішень для ефективного моніторингу додатків, що працюють усередині контейнерів. Наприклад, Datadog і cAdvisor агенти використовують власні облікові показники Docker, щоб зібрати показники процесора, пам'яті, мережі та введення/виведення запущених контейнерів. CAdvisor дозволяє відстежувати контейнери, що працюють на одній машині. Як альтернатива Scout використовується для агрегування метрик з різних хостів і контейнерів у розподіленій архітектурі. Як систему оркестровки для контейнерів Docker обрано відкритий проект Kubernetes. Це дозволяє швидко й ефективно реагувати на попит клієнтів, розгортаючи програми за допомогою кількох хостів і контейнерів у хмарі. Ця структура кластеризації постачається з інструментом моніторингу під назвою Heapster який надає базову платформу моніторингу на Kubernetes. Heapster збирає та інтерпретує різні сигнали, такі як використання ресурсів, події життєвого циклу тощо, і експортує показники кластера через кінцеві точки REST. Він підтримує вбудований сервер зберігання, наприклад In uxDB із Grafana та Google Cloud Monitoring. Більшість із цих інструментів надають веб-панелі для візуалізації споживання ресурсів під час виконання, а також механізм попередження, який може бути запущений, якщо показники перевищують або опускаються нижче встановленого порогу. Існують інші приклади інструментів моніторингу Docker, наприклад: Sensu Monitoring Framework, Prometheus, Sysdig Cloud тощо.

Моніторинг контейнерів під час роботи також застосовувався для вирішення проблем дослідження. Як приклад, у [93] автори досліджували проблему автоматичного шардингу в базах даних NoSQL, використовуючи інфраструктуру на основі контейнерів для моніторингу часу виконання. Техніка автоматичного

шардингу використовується для розділення даних у базі даних і розподілу їх між декількома машинами з метою горизонтального масштабування. Насправді вибрати правильний ключ складно. Це може призвести або до покращення продуктивності та можливостей бази даних, або до проблем із продуктивністю (тобто через вибір неправильного ключа), що може призвести до зупинки системи. Таким чином, автори проаналізували та оцінили такі запропоновані властивості, вивчивши, як зміна вибору ключа фрагмента може вплинути на продуктивність БД. Вони змоделювали середовище за допомогою контейнерів Docker і виміряли продуктивність читання/запису різноманітних ключів. Усередині кожного контейнера вони виконували запити на запис/читання в MongoDB і використали статистику Docker для автоматичного отримання інформації про використання пам’яті та ЦП.

У роботі [94] представлено інструмент для тестування, оптимізації та автоматизації рішень щодо розподілу хмарних ресурсів для досягнення цілей якості обслеговавання для веб-додатків. Їхня інфраструктура покладається на Docker для збору інформації про використання ресурсів розгорнутих веб-серверів.

Моніторинг контейнерів застосовувався в інших дослідженнях, пов’язаних, зокрема, з хмарними обчисленнями та віртуалізацією [95, 96].

2.4 Висновки

Аналіз сучасного стану виявив кілька проблем. Нижче описано деякі проблеми, які виявлено в обох областях ітераційної компіляції та тестування генератора коду:

Обмеження існуючої роботи під час тестування генераторів коду (проблема оракула): Більшість робіт, пов’язаних з автоматичним тестуванням генераторів коду, визначають функціональний оракул еквівалентності для порівняння результатів MiL, SiL та PiL. У випадку невиконуваних моделей це порівняння стає неможливим. Зокрема, проблема е для тестування нефункціональних властивостей

згенерованого коду була уникнута та не розглянута існуючими дослідницькими зусиллями. Єдине порівняння, яке було зроблено, полягає в порівнянні рукописного коду з автоматично згенерованим кодом. Основна мета цього порівняння – показати, що згенерований код має кращі або еквівалентні характеристики продуктивності порівняно з людським кодом.

Обмеження існуючих методів під час дослідження великого простору пошуку оптимізації (мультимодальна проблема): Як було показано раніше, під час ітеративного процесу компіляції застосовуються різні методи для дослідження великого простору пошуку оптимізації. Показано, що простір оптимізаційного пошуку є мультимодальною задачею, що містить багато локальних оптимумів. Для ефективного пошуку деякі автори використовують лише кілька оптимізацій (від 2 до 10) або обрізають деякі шляхи, щоб зменшити цей великий простір пошуку. Генетичні алгоритми в основному застосовуються в більшості попередніх робіт. Однак ця методика може потрапити в задачу локального оптимуму. Простір пошуку оптимізації мультимодальний і дуже великий. Для вирішення цієї проблеми потрібне рішення, альтернативне класичним підходам.

Відсутність рішень, які мають справу з суперечливими цілями під час автоматичного налаштування компіляторів (проблема багатоцільової оптимізації): при спробі оптимізувати продуктивність програмного забезпечення багато нефункціональних властивостей і дизайнерських обмежень мають бути задіяні та задоволені одночасно, щоб краще оптимізувати код. Однак збільшення часу виконання програми може призвести до значного використання ресурсів, що може знизити продуктивність системи, особливо для пристроїв з обмеженими ресурсами. Таким чином, важливо побудувати рівні оптимізації, які представляють численні обміни між використанням ресурсів і продуктивністю, дозволяючи користувачеві вибирати серед різних оптимальних рішень, які найкраще відповідають вимогам системи. Насправді, є лише кілька робіт, які стосуються оптимізації компілятора як багатоцільової проблеми оптимізації. Щоб мати справу з суперечливими цілями, важливо знайти консенсус між декількома нефункціональними властивостями під час оптимізації коду для обробки використання ресурсів, і вимог до продуктивності

Обмеження існуючих підходів до обробки програмної платформи та вимог до апаратного забезпечення (проблема різноманітності програмного забезпечення та гетерогенності апаратного забезпечення): Тестування генераторів вимагає виконання згенерованого коду на різних апаратних та програмних платформах. Більшість існуючих методів дослідження застосовують наївний підхід, запускаючи згенерований код на різних машинах або використовуючи симулятори для деяких конфігурацій. Конфігурація цільового середовища виконання для запуску згенерованого коду та його тестування займає багато часу. Зокрема, для користувачів генераторів стає дуже складно розгортати та тестувати згенеровані програми перед зростаючою різноманітністю налаштувань апаратної та програмної платформи. Для виконання вимог програмного та апаратного забезпечення потрібне середовище виконання з можливістю налаштування.

Відсутність рішень, які б оцінювали властивості використання ресурсів під час оцінювання згенерованого коду: Майже немає жодних дослідницьких зусиль, щоб мати справу з нефункціональними властивостями автоматично згенерованого коду, такими як використання ресурсів. Більшість пов’язаної роботи (під час тестування генератора коду) зосереджується на функціональній коректності згенерованого коду, не надто наголошуючи на якості згенерованого коду. Оцінка таких властивостей, як використання ресурсів, дуже важлива для забезпечення ефективної генерації робочого коду. При ітераційній компіляції більшість існуючих робіт має тенденцію до зменшення часу виконання, розміру коду, часу компіляції тощо. Практично немає роботи, яка б оцінювала використання пам’яті та ЦП оптимізованого коду. Для оцінки властивостей використання ресурсів автоматично створеного коду потрібне ефективне рішення для моніторингу

3 АВТОМАТИЧНЕ НЕФУНКЦІОНАЛЬНЕ ТЕСТУВАННЯ СІМЕЙСТВ ГЕНЕРАТОРІВ КОДУ

Генеративна розробка програмного забезпечення проклала шлях до створення численних генераторів коду, які автоматично перетворюють специфікації системи високого рівня в багатоцільовий виконуваний код. Щоб зберегти надійність і якість програмного забезпечення, згенерований код потрібно тестувати з такими ж зусиллями, як і код, написаний вручну. Будь-які проблеми з генераторами коду слід виявляти та виправляти якомога раніше, щоб забезпечити правильну роботу доставленого програмного забезпечення. В попередньому розділі представлено кілька підходів до автоматичного функціонального тестування генераторів коду. Однак автоматичне тестування нефункціональних властивостей згенерованого коду є складним завданням, яке ще не розглядалося.

У цьому розділі описано підхід до тестування, який автоматично виявляє аномалії в генераторах коду з точки зору нефункціональних властивостей (тобто використання ресурсів і продуктивності). Насправді адаптовано ідею метаморфічного тестування до проблеми тестування генераторів коду. Отже, запропонований підхід спирається на визначення тестових оракулів високого рівня (тобто метаморфічних зв’язків), щоб перевірити потенційно неефективний генератор коду серед сімейства генераторів коду. Крім того, застосовано різні статистичні методи, щоб автоматизувати виявлення невідповідностей. Запропонований підхід оцінюється, аналізуючи продуктивність Haxe, популярної мови програмування високого рівня, яка включає набір кросплатформових генераторів коду. Експериментальні результати показують, що запропонований підхід здатний виявити деякі невідповідності продуктивності, які виявляють реальні проблеми в генераторах коду Haxe.

3.1 Контекст і мотивація

3.1.1 Сімейства генераторів коду

Сьогодні різні настроювані генератори коду можна використовувати для легкого й ефективного створення коду для різних програмних платформ, мов програмування, операційних систем тощо. Ця робота базується на гіпотезі, що генератор коду часто є членом сімейства генератори коду [97].

Сімейство генераторів коду – це набір генераторів коду, які приймають як вхідні дані ту саму мову/модель і генерують код для різних цільових платформ.

Наприклад, ця концепція широко використовується в промисловості під час застосування парадигми «write onely, run everywhere». Користувачі можуть отримати вигоду від сімейства генераторів коду (наприклад, крос-платформних генераторів коду [98]), щоб генерувати з написаного вручну (високорівневого) коду, різні реалізації однієї програми на різних мовах Ця техніка дуже корисна для вирішення різноманітних програмних платформ і мов програмування.

Як мотиваційні приклади для цієї роботи можна навести три підходи, які інтенсивно розробляють і використовують сімейства генераторів коду:

Haxe. Haxe [99] – це набір інструментів із відкритим вихідним кодом для кросплатформної розробки, який компілюється для низки різних платформ програмування, включаючи JavaScript, Flash,PHP, C++, C# і Java. Haxe включає багато функцій: мову Haxe, мультиплатформенні компілятори та різні нативні бібліотеки. Мова Haxe – це мова програмування високого рівня, яка строго типізована. Ця мова підтримує як функціональну, так і об’єктно-орієнтовану парадигми програмування. Він має загальну ієрархію типів, що робить певний API доступним на кожній цільовій платформі. Крім того, Haxe поставляється з набором генераторів коду, які перекладають написаний вручну код (мовою Haxe) на різні цільові мови та платформи. Цей проект популярний (більше 1440 зірок на GitHub).

ThingML. ThingML це мова моделювання для вбудованих і розподілених систем [100]. Ідея ThingML полягає в тому, щоб розробити практичний ланцюжок інструментів програмної інженерії, керований моделлю, який націлений на

вбудовані системи з обмеженими ресурсами, такі як малопотужні датчики та пристрої на основі мікроконтролерів. ThingML розроблено як предметно-спеціальну мову моделювання, яка включає концепції для опису компонентів програмного забезпечення та протоколів зв'язку. Використовуваний формалізм є комбінацією моделей архітектури, кінцевих автоматів і мови імперативних дій. Набір інструментів ThingML надає сімейство генераторів коду для перекладу ThingML на C, Java і JavaScript. Він містить набір варіантів для генераторів коду C і JavaScript для націлювання на різні вбудовані системи та їхні обмеження. Цей проект все ще конфіденційний, але він є хорошим кандидатом для представлення практик спільноти моделювання.

TypeScript. TypeScript – це типізований наднабір JavaScript, який компілюється у звичайний JavaScript [101]. Фактично, він не компілюється лише в одну версію JavaScript. Він може перетворювати TypeScript на EcmaScript 3, 5 або 6. Він може генерувати JavaScript, який використовує різні системні модулі («none», «commonjs», «amd», «system», «umd», «es6» або «es2015»).

Функціональне тестування сімейства генераторів коду просте. Оскільки створені програми генеруються з тієї самої програми високого рівня, оракул можна визначити як порівняння між їхніми функціональними результатами, які мають бути однаковими. Фактично, ґрунтуючись на трьох прикладах проектів, представлених вище, зауважується, що всі репозиторії коду GitHub відповідних проектів використовують модульні тести для перевірки правильності роботи генераторів коду.

Що стосується нефункціональних тестів, помітно, що ThingML і TypeScript не надають жодних спеціальних тестів для перевірки узгодженості генераторів коду щодо пам'яті або використання ЦП. Нахе надає два тести для порівняння отриманого згенерованого коду. Один служить для порівняння прикладу, у якому виділення об'єктів навмисно (надмірно) використовується для вимірювання того, як доступ до пам'яті/GC поєднується з числовою обробкою в різних цільових мовах. Другий тест оцінює швидкість мережі на різних цільових платформах.

3.1.2 Проблеми під час тестування сімейства генераторів коду

Основні труднощі під час тестування властивостей використання ресурсів генераторами коду полягають у тому, що не можна просто спостерігати за виконанням створеного коду, потрібно спостерігати та порівнювати виконання згенерованих програм з еквівалентними (або еталонними) реалізаціями (тобто іншими мовами). Навіть якщо немає явного оракула для виявлення невідповідностей для одного генератора коду, можна отримати користь від сімейства генераторів коду, щоб порівняти поведінку кількох згенерованих програм і виявити окремі профілі споживання ресурсів, які можуть виявити невідповідність генератора коду [102].

Як наслідок, визначено неузгодженість генератора коду як згенерований код, який демонструє неочікувану поведінку з точки зору продуктивності або використання ресурсів порівняно з усіма еквівалентними реалізаціями в тому самому сімействі генераторів коду.

3.2 Традиційний процес нефункціонального тестування сімейства генераторів коду

Надійним і прийнятним способом підвищення довіри до генераторів коду є перевірка та перевірка функціональної поведінки згенерованого коду, що є звичайною практикою при тестуванні генераторів [33, 29, 35]. Однак довести, що згенерований код є функціонально правильним, недостатньо, щоб стверджувати про ефективність тестованого генератора коду.

Фактично, генератори коду повинні дотримуватися різних вимог, щоб зберегти надійність і якість програмного забезпечення [103]. У цьому випадку для забезпечення якості згенерованого коду необхідно перевірити кілька нефункціональних властивостей, таких як розмір коду, споживання ресурсів або енергії, час виконання тощо [61]. рисунок 3.1 узагальнює класичні кроки, необхідні

для забезпечення генерації коду та нефункціонального тестування сімейства генераторів коду. Виділено чотири основні кроки: розробка програмного забезпечення з використанням специфікацій системи високого рівня, генерація коду, виконання коду та нефункціональне тестування згенерованого коду.

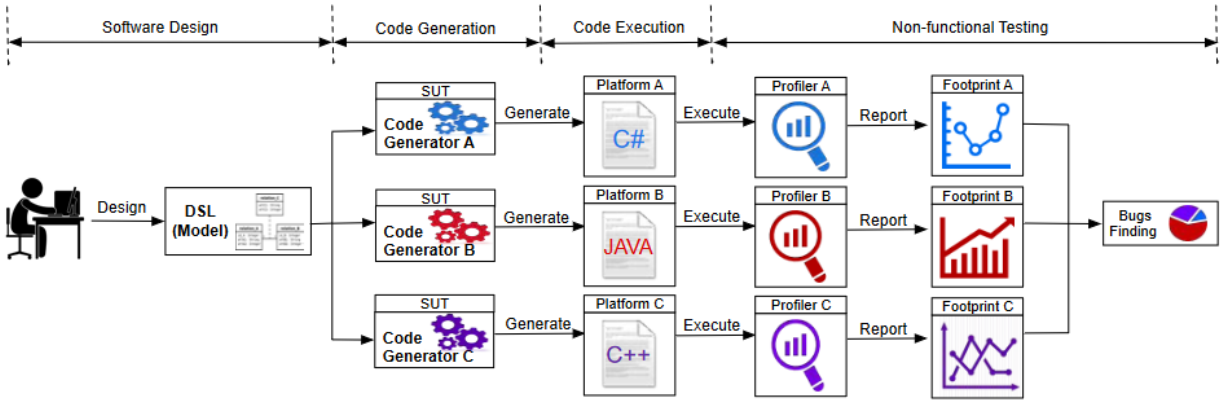


Рисунок 3.1 – Загальний огляд різних процесів, залучених для забезпечення генерації коду та нефункціонального тестування створеного коду від часу проектування до виконання: класичний спосіб

На першому кроці розробники програмного забезпечення повинні визначити, під час проектування, поведінку програмного забезпечення, використовуючи абстрактну мову високого рівня (DSL, моделі, програми тощо). Після цього розробники можуть використовувати генератори коду для певної платформи, щоб полегшити розробку програмного забезпечення та автоматично генерувати код для різних мов і платформ. На рисунку 3.1 зображено три генератори коду з одного сімейства, здатні генерувати код для трьох мов програмування (Java, C# і C++). Першим кроком є генерація коду з попередньо розробленої моделі. Після цього згенеровані програмні артефакти (наприклад, Java, C#, C++ тощо) компілюються, розгортаються та виконуються на різних цільових платформах (наприклад, Android, ARM/Linux, JVM, x86/Linux тощо). Нарешті, щоб виконати нефункціональне тестування згенерованого коду, розробники повинні збирати, візуалізувати та порівнювати інформацію про продуктивність і ефективність

виконання коду на різних платформах. Тому вони, як правило, використовують кілька специфічних для платформи профайлерів, трекерів, інструментів і інструментів моніторингу, щоб знайти деякі невідповідності або помилки під час виконання коду [104, 105]. Пошук невідповідностей у генераторах коду передбачає аналіз і перевірку коду, а також для кожної платформи виконання. Наприклад, в один спосіб, щоб упоратися з цим, потрібно проаналізувати відбиток пам’яті під час виконання програмного забезпечення та виявити витоки пам’яті [106]. Потім розробники можуть перевірити згенерований код і знайти деякі фрагменти кодової бази, які спричинили цю проблему. Потім вони повідомляють цю інформацію для х, рефакторингу та оптимізації процесу генерації коду. Порівняно з цим класичним (і ручним) підходом до тестування, запропонована робота спрямована на автоматизацію останніх трьох кроків: створення та виконання коду на різних програмних платформах, а також виявлення нефункціональних проблем.

3.3 Огляд підходу

3.3.1 Інфраструктура для нефункціонального тестування з використанням системних контейнерів

У цьому розділі основний акцент зосереджено на оцінці нефункціональних властивостей, пов’язаних із використанням ресурсів і продуктивністю згенерованого коду. Для цього потрібно враховувати багато конфігурацій системи (наприклад, середовища виконання, бібліотеки, компілятори тощо), щоб ефективно генерувати та тестувати код.

Однак налаштування різних програм (тобто згенерованого коду) з різними конфігураціями на одній машині є складним. Одна система має обмежені ресурси, і це може призвести до регресії продуктивності. Крім того, кожне середовище виконання постачається з набором відповідних інструментів, таких як компілятори, генератори коду, налагоджувачі, профайлери тощо. Тому потрібно розгорнути тестовий пакет, тобто створені двійкові файли, на еластичній інфраструктурі, яка

надає засоби для розробники генератора коду, щоб забезпечити розгортання та моніторинг згенерованого коду в різних налаштуваннях середовища. Отже, інфраструктура тестування забезпечує підтримку для автоматичного:

розгортання згенерованого код, його залежності та середовища виконання програми;

виконання створених двійкових файли в ізольованому середовищі;

контролю виконання;

збору показників використання ресурсів (ЦП, пам’ять тощо).

Щоб забезпечити виконання цих чотирьох основних кроків, використано системні контейнери [87] як динамічне та настроюване середовище виконання для запуску та оцінки згенерованих програм з точки зору використання ресурсів.

На рисунку 3.2 показано інфраструктуру на основі контейнерів, яка використовується для тестування генераторів коду.

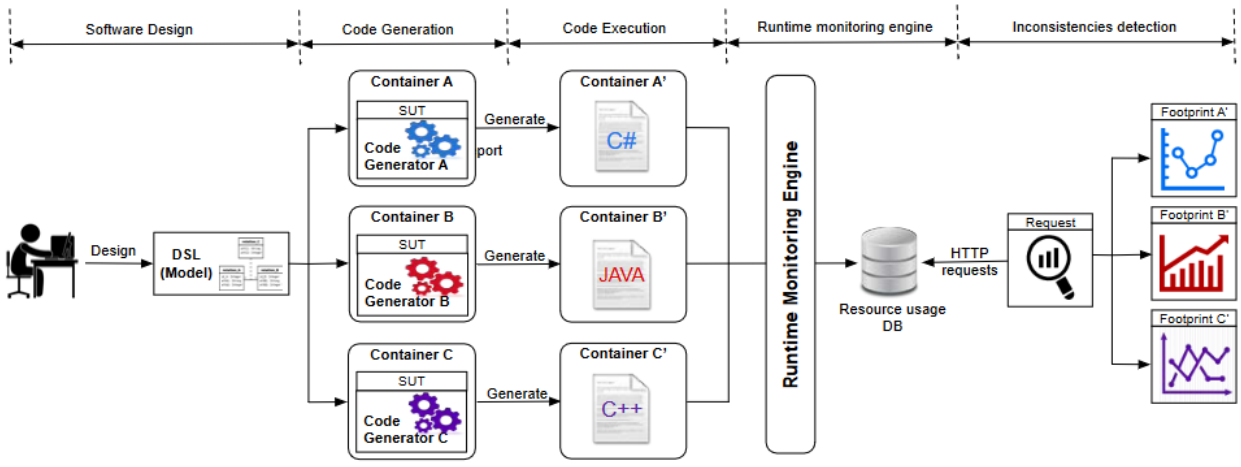


Рисунок 3.2 – Технічний огляд різних процесів, задіяних для забезпечення генерації коду та нефункціонального тестування створеного коду від часу проектування до виконання.

У порівнянні з класичним методом, представленим на рис 3.1 додано наступні функції:

На рівні генерації коду: генератори коду налаштовуються всередині різних контейнерів, щоб генерувати код для цільової платформи.

На рівні виконання коду: бібліотеки, компілятори та різні залежності конфігуруються в різних контейнерах для виконання згенерованого коду. Для кожної цільової платформи створюється новий екземпляр.

На нефункціональному рівні тестування: додано механізм моніторингу часу виконання (на основі контейнерів), щоб отримати та зберегти показники використання ресурсів усіх запущених контейнерів. Нарешті, виявлення невідповідностей передбачає аналіз даних про використання ресурсів, отриманих із бази даних, щоб знайти проблеми зі згенерованим кодом. Цей крок детально описано в наступному розділі.

3.3.2 Підхід метаморфічного тестування для автоматичного виявлення невідповідностей генератора коду

В розділі 2.1.2.2 описувалось кілька підходів, запропонованих спільнотою тестувальників програмного забезпечення, щоб полегшити проблему Oracle. Серед привабливих підходів, які можна застосувати до генераторів тестового коду, виділено підхід метаморфічного тестування (похідні оракули). Далі описано основну концепцію метаморфічного тестування та запропоновано адаптацію цього методу для нефункціонального тестування сімейств генераторів коду.

3.3.2.1 Основна концепція метаморфічних досліджень

У цьому розділі описані з основні концепції метаморфічного тестування (MT), запропоновані у [109]. Ідея MT полягає в тому, щоб виводити тестові оракули

зі зв’язку між виходами тестових випадків замість того, щоб міркувати про зв’язок між входами та виходами тесту.

МТ рекомендує, щоб, враховуючи один або кілька тестових випадків (так звані «джерельні тестові випадки», «оригінальні тестові випадки», або «успішні тестові випадки») та їхні очікувані результати (отримані через багаторазове виконання цільової програми, що тестується), один або Для перевірки необхідних властивостей (так звані Метаморфічні зв’язки MR) системи або функції, яка має бути реалізована, можна створити більше наступних тестів. У цьому випадку для створення наступних тестів і перевірки результатів тесту потрібно повага MR.

Класичним прикладом МТ є програма, яка обчислює функцію sin. Корисним метаморфічним співвідношенням для функцій *sin* є *sin(x) = sin(– x).* Таким чином, навіть незважаючи на те, що очікуване значення для вихідного тестового випадку *sin(50),* наприклад, невідоме, подальший тестовий приклад може бути створений для перевірки MR, визначеного раніше. У цьому випадку наступним тестовим прикладом є *sin(-50),* який має вивести вихідне значення, що дорівнює вихідному тестовому випадку *sin(50).* Якщо ця властивість порушується, то негайно виявляється збій. МТ генерує наступні тестові випадки, якщо дотримуються метаморфічні зв’язки. Це приклад метаморфічного відношення: вхідне перетворення, яке можна використовувати для створення нових тестових випадків із наявних тестових даних, і вихідне відношення MR, яке порівнює виходи, створені парою тестових випадків. MR може бути будь-якими властивостями, що включають вхідні та вихідні дані двох або більше виконання цільової програми, такі як рівності, нерівності, обмеження конвергенції та багато інших.

Оскільки МТ перевіряє зв’язок між кількома виконаннями, а не правильність окремих виходів, його можна використовувати для повної автоматизації процесу тестування без ручного втручання. Однак побудова метаморфічних зв’язків зазвичай є ручним завданням, яке вимагає досконалого знання програми, що тестується. Це також залежить від контексту програми та домену. Ефективність метаморфічного тестування значною мірою залежить від ідентифікованих

метаморфічних зв’язків, і, отже, розробка ефективних метаморфічних зв’язків є критичним кроком при застосуванні метаморфічного тестування.

3.3.2.2 Адаптація підходу МТ для виявлення невідповідностей генератора коду

Загалом, МТ можна застосувати до будь-якої задачі, в якій можна сформулювати необхідну властивість, що включає багаторазове виконання цільової функції. Деякі приклади успішних застосувань представлені в [108]. Зауважується, що МТ нещодавно застосовано для тестування компіляторів [110, 111, 27].

Щоб застосувати МТ, потрібно виконати чотири основні кроки:

знайдіть властивості тестованої системи: систему слід досліджувати вручну, щоб знайти передбачувані MR, що визначають зв’язок між входами та виходами. Це базується на вихідних тестах.

Згенеруйте/виберіть тестові вхідні дані, які задовольняють MR: це означає, що необхідно створити або вибрати нові наступні тестові випадки, щоб перевірити їхні результати за допомогою MR.

Виконайте систему з вхідними даними та отримайте виходи: оригінальні та наступні тестові випадки виконуються, щоб зібрати їхні виходи.

Перевірте, чи задовольняють ці результати MR.

3.3.2.3 Метаморфічний зв'язок

Крок 1 полягає в ідентифікації необхідних властивостей тестованої програми та представленні їх у вигляді метаморфічних відносин. Як уже було зазначено, метаморфічне відношення є відношенням, отриманим від виконання різних систем. Використано визначення MP, представлене в [111, 112]:

Сімейство генераторів коду можна розглядати як функцію: $C:I \rightarrow P$, де I – область дійсних вихідних програм високого рівня, а P – область цільових програм, які генеруються різними генераторами коду одного сімейства. Властивість сімейства генераторів коду передбачає, що згенеровані програми P мають таку саму поведінку, як це зазначено в I.

Наявність кількох генераторів із порівнянною функціональністю дозволяє адаптувати МТ для виявлення нефункціональних невідповідностей. Насправді, якщо можна знайти правильне співвідношення R (рівняння 3.1) нефункціональної поведінки, можна отримати метаморфічний зв'язок і провести МТ для тестування сімейств генераторів коду. Нехай $f(P(t_i))$ буде функцією, яка обчислює нефункціональний вихід (наприклад, час виконання або використання пам'яті) набір вхідних тестів (t_i), що виконується на створеній програмі (P). Оскільки наявні різні версії програм, згенеровані в одному сімействі, позначено $(P_1(t_i), P_2(t_i),..., P_n(t_i))$ набір згенерованих програм. Відповідними виходами будуть $(f(P_1), f(P_2),..., f(P_n))$. Отже, запропонований MR виглядає так:

$$R(P_1(t_i), P_2(t_i), \ldots, P_n(t_i))) \Rightarrow R(f(P_1(t_i)), f(P_2(t_i)), \ldots, f(P_n(t_i))) \quad (3.1)$$

З одного боку, використовується наступне рівняння $P_1(t_i) \equiv P_2(t_i)$ для позначення функціонального відношення еквівалентності між двома згенерованими програмами P_1 і P_2 з одного сімейства. Це означає, що згенеровані програми P_1 і P_2 мають однаковий поведінковий дизайн і для будь-якого набору тестів t_i вони мають однаковий функціональний результат. Якщо це співвідношення не виконується, то існує принаймні один несправний генератор коду, який створив неправильний код.

З іншого боку, оскільки порівнюються еквівалентні реалізації однієї програми, написаної різними мовами, можна припустити, що використання пам'яті та час виконання мають бути більш-менш однаковими з невеликою варіацією для кожного набору тестів у різних версіях. . Очікувано, що існує різниця між різними виконаннями, оскільки порівнюється час виконання та використання пам'яті

наборів тестів, які написані різними мовами та виконуються з використанням різних технологій (наприклад, інтерпретатори для PHP, JVM для Java тощо). Це спостереження також ґрунтується на початкових експериментах, у яких оцінюється використання ресурсів/час виконання кількох наборів тестів у наборі еквівалентних версій, згенерованих за допомогою сімейства генераторів коду. Як наслідок, використовується позначення $\Delta\{f(P_1(t_i)), f(P_2(t_i))\}$ для позначення варіації використання пам'яті або часу виконання тестового набору t_i для двох версій згенерованого коду P_1 і P_2, написаних різними мовами. Припускається, що ця варіація не повинна перевищувати певне порогове значення T, інакше створюється неузгодженість генератора коду. Виходячи з цієї логіки, MR можна представити у вигляді рівняння:

$$P_1(t_i) \equiv P_2(t_i) \equiv \ldots \equiv P_n(t_i)) \Rightarrow \Delta\{f(P_1(t_i)), f(P_2(t_i)), \ldots, f(P_n(t_i))\} < T \quad (n > 2) \qquad (3.2)$$

Цей MR еквівалентний тому, що: якщо набір функціонально еквівалентних програм генерується за допомогою однієї сім'ї генераторів коду $((P_1(t_i), P_2(t_i), ..., P_n(t_i))$, і з тим самим набором вхідних тестів t_i, потім порівняння їхніх нефункціональних виходів $(f(P_1(t_i)), f(P_2(t_i)), ..., f(P_n(t_i)))$ має бути однаковим, беручи до уваги інтервал допуску, визначений варіацією, який не повинен перевищувати конкретного граничного значення T.

Згенерований код, який порушує цю метаморфічну властивість, представляє невідповідність, і його відповідний генератор коду вважається дефектним.

3.3.2.4 Метаморфічні випробування

Наразі визначено MR, необхідний для виявлення невідповідностей. Нижче описано запропонований підхід до автоматичного метаморфічного тестування на основі цього співвідношення (Кроки 2, 3 і 4). рисунок 3.3 показує огляд підходу. Сімейство генераторів коду приймає ту саму вхідну програму та генерує набір

еквівалентних тестових програм *(P_1, P_2, ..., P_n)*. Це відповідає Кроку 2. У запропонованій адаптації МТ наступні тестові випадки представляють еквівалентні тестові програми, які автоматично генеруються за допомогою сімейства генераторів коду. Набори тестів також генеруються автоматично, оскільки припускається, що вони вже визначені під час розробки. Насправді той самий набір тестів (тестові приклади + значення вхідних даних) передається всім створеним програмам. Потім виконуються згенеровані програми та відповідні набори тестів (Крок 3). Після цього вимірюється використання пам'яті або час виконання цих згенерованих програм *($f(P_1(t_i))$, $f(P_2(t_i))$, ..., $f(P_n(t_i))$)*. Нарешті, результати виконання порівнюються та перевіряються за допомогою MR, визначеного раніше (Крок 4). У цьому процесі буде повідомлено про невідповідності, коли одна з наступних еквівалентних програм тестування порушує MR.

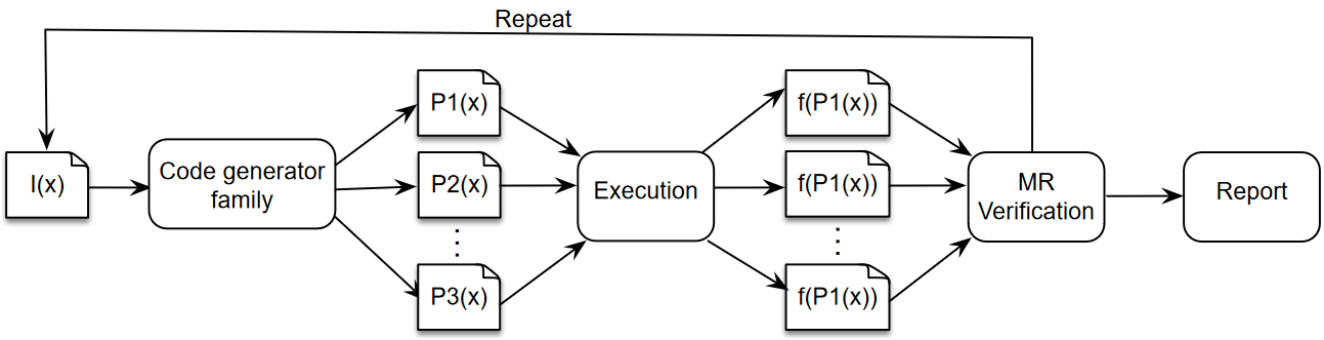


Рисунок 3.3 – Підхід до метаморфічного тестування для автоматичного виявлення невідповідностей генератора коду

3.3.2.5 Поріг варіації

Одне із питань, яке може виникнути під час застосування запропонованого підходу МТ, полягає в тому, як можна знайти правильний поріг варіації T, за якого виявляється невідповідність? Відповідь на це запитання дуже важлива для підтвердження ефективності запропонованого підходу до МТ. Проводиться статистичний аналіз нефункціональних даних, щоб знайти точне порогове значення T . Перед цим необхідно підготувати нефункціональні результати, щоб

зробити їх придатними для статистичних методів, що використовуються в методології. Отже, спочатку описується запропонований процес підготовки даних.

Підготовка даних. Як показано в таблиці 3.1 кожна програма постачається з певним набором тестів $(t_1, t_2, ..., t_m)$. Оцінка набору тестів вимагає обчислення використання пам'яті або часу виконання $f(P_1(t_i)), f(P_2(t_i)), ..., f(P_n(t_i))$, де $(1 \le i \le m)$ для всіх цільових програмних платформ. Таким чином, отримані результати являють собою матрицю, де стовпці вказують на нефункціональне значення (необроблені дані) для кожної цільової програмної платформи, а рядки вказують на відповідний набір тестів.

Таблиця 3.1 – Результати виконання наборів тестів

	Цільова платформа 1	Цільова платформа 2	...	Цільова платформа *n*
t_1	$f(P_1(t_1))$	$f(P_2(t_1))$	...	$f(P_n(t_1))$
t_2	$f(P_1(t_2))$	$f(P_2(t_2))$	...	$f(P_n(t_2))$
...	...	...	...	...
t_m	$f(P_1(t_m))$	$f(P_2(t_m))$	...	$f(P_n(t_m))$

Нефункціональні дані мають бути перетворені у формат, який можна зрозуміти статистичними методами. Одним із способів порівняння цих нефункціональних результатів є вивчення факторних відмінностей. Іншими словами, проведена оцінка для кожної цільової платформи кількість разів (коефіцієнт), які потрібні для запуску набору тестів у порівнянні з еталонним виконанням. Еталонне виконання відповідає мінімальному отриманому нефункціональному значенню t_i виконання на n цільових платформах. Отриманий коефіцієнт є співвідношенням між фактичним нефункціональним значенням і мінімальним значенням, отриманим серед n версій. Наступне рівняння застосовується до кожної комірки, щоб перетворити отримані дані:

$$F\left(f\left(P_j(t_i)\right)\right) = \frac{f\left(P_j(t_i)\right)}{\mathrm{Min}(f\left(P_1(t_i)\right), \ \ldots, f\left(P_n(t_i)\right))} \qquad (3.3)$$

Еталонне виконання автоматично отримає значення *F=1*. Максимальне значення – це те, що призводить до максимального відхилення від еталонного виконання. Наприклад, нехай P_1 буде згенерованою програмою на Java. Якщо час виконання, необхідний для запуску t_1, поступається мінімальному значенню $f(P_1(t_1))$ порівняно з іншими версіями, тоді $f(P_1(t_1))$ отримає значення фактора *F*, що дорівнює 1, а інший версії буде поділено на $f(P_1(t_1))$, щоб отримати відповідні значення фактора порівняно з Java.

Статистичний аналіз. У запропонованому підході MT невідповідність – це зміна використання/продуктивності ресурсу, яка перевищує певне порогове значення T . Можливо використовувати два методи варіаційного аналізу [113]: аналіз головних компонентів (AOK) і діапазонні діаграми (R-chart). Таблиця 3.2 дає огляд цих двох статистичних методів. Основною метою цих методів є оцінка використання пам’яті та зміни продуктивності, а отже, визначення відповідного значення T для запропонованого MR.

Таблиця 3.2 – Підходи до варіаційного аналізу

Технологія	Метод
R-chart	Визначення T як варіацію між верхньою та нижньою контрольною межею
AOK	Відсікаюче значення оціночних відстаней PC визначає T

R-chart. У цьому підході оцінка варіації між різними версіями визначається шляхом порівняння нефункціональних вимірювань на основі методики статистичного контролю якості, яка називається R-chart або діаграма діапазону [113]. R-Chart використовується для аналізу варіацій у межах процесів. Він призначений для виявлення змін у варіаціях з часом і для оцінки узгодженості варіацій процесу. R-Chart використовує контрольні межі (LCL і UCL), щоб представити межі варіації, які слід очікувати від процесу. LCL позначає нижню контрольну межу, а UCL позначає верхню контрольну межу.

Коли процес знаходиться в контрольованих межах, будь-які зміни є нормальними. Кажуть, що процес на контролі. Проте відхилення поза граничними значеннями вважаються відхиленнями, а R-chart вважається неконтрольованою, що

означає, що зміна процесу нестабільна. Існує невідповідність, що призводить до цього високого відхилення варіації (Рисунок 3.4).

Процес представляє n нефункціональних виходів, отриманих після виконання набору тестів t_i. Як визначено MR, варіація в межах одного процесу має бути нижчою за порогове значення T. У наших налаштуваннях цей варіант має бути між LCL і UCL.

Тому для кожного набору тестів обчислюється діапазон R, що відповідає різниці між максимальним і мінімальним нефункціональним виходом на всіх цільових платформах.

$$R(t_i) = Max(f(P_1(t_i)), \ldots, f(P_n(t_i))) - Min(f(P_1(t_i)), \ldots, f(P_n(t_i))) \quad (3.4)$$

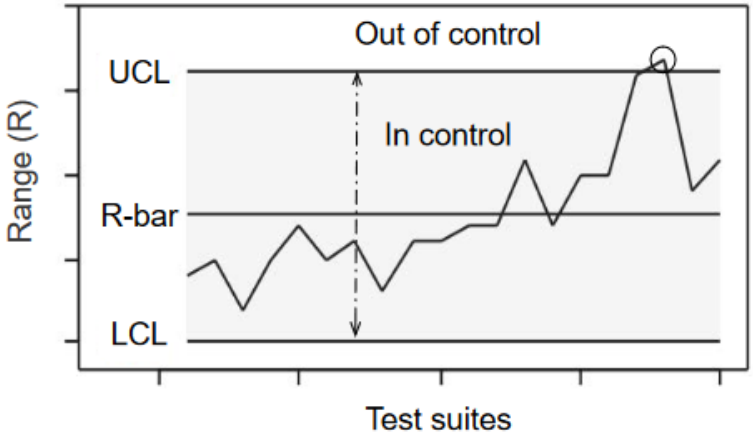


Рисунок 3.4 – Процес R-Chart

R кількісно оцінює результати варіації при виконанні одного і того ж тестового набору ti на різних версіях програми. Для того, щоб визначити, чи є варіація контрольованою чи ні, потрібно визначити значення контрольних меж. UCL і LCL відображають фактичну величину варіації, яка спостерігається. Обидві метрики є функцією R-bar($\bar{R}$) – це середнє значення R для всіх наборів тестів. UCL та LCL розраховуються наступним чином

$$\begin{aligned} UCL &= D_4\bar{R} \\ LCL &= D_3\bar{R} \end{aligned} \quad (3.5)$$

де D_4, D_3 – константи контрольної діаграми, які залежать від кількості змінних у кожному процесі.

Наприклад, для сімейства, що складається з менше ніж 7 генераторів коду, значення D_3 дорівнює 0, і, як наслідок, LCL = 0. У цьому випадку UCL представляє поріг значення, від якого виявлено високе відхилення варіації, що призводить до невідповідності. Як значення UCL (або T) чутливе до нових наборів тестів. Отже, коли виконується новий набір тестів, значення T оновлюється, а варіація оцінюється за новим пороговим значенням.

Нижче представлено альтернативний статистичний підхід до аналізу варіацій усіх наших даних.

АОК. При великій кількості версій програми матриця нефункціональних даних (табл. 3.1) може бути занадто великим для належного вивчення та інтерпретації варіації. Між різними версіями було б занадто багато парних кореляцій, щоб їх було враховано, і варіацію неможливо відобразити (графічно), коли набори тестів виконуються на більш ніж трьох цільових програмних платформах. З 12 змінними, наприклад, буде більше 200 тривимірних діаграм розсіювання, які будуть розроблені для вивчення варіації та кореляції. Щоб інтерпретувати дані в більш значущій формі, необхідно зменшити кількість змінних, що складають наші дані.

Аналіз основних компонентів (АОК) – це багатовимірний статистичний підхід, який використовує ортогональну трансформацію для перетворення набору спостережень можливо корельованих змінних у набір значень лінійно некорельованих змінних, які називаються головними компонентами (PC). Його можна застосовувати, коли дані збираються про велику кількість змінних з одного спостереження. Таким чином, застосовано підхід АОК до запропонованого прикладу, оскільки запропонований розмірний простір, як він представлений у табл 3.1, складається з набору процесів (наборів тестів), де n змінних (наприклад, цільові мови програмування) складають кожне спостереження. Варіативність у пропонованій моделі пов’язана з цими n змінними, що представляють набори тестів, що працюють на n цільових платформах.

Основна мета застосування АОК полягає в тому, щоб зменшити розмірність вихідних даних і пояснити максимальну кількість дисперсії за допомогою найменшої кількості головних компонентів. Для цього АОК займається узагальненням дисперсійно-коваріаційної матриці. Він передбачає обчислення власних векторів і власних значень дисперсійно-коваріаційної матриці. Власні вектори використовуються для проектування даних від n вимірів до нижчого вимірного представлення. Власні значення дають дисперсію даних у напрямку власного вектора. Перший власний вектор – це вектор, який визначає напрямок максимальної дисперсії в даних. Перший головний компонент обчислюється таким чином, щоб він враховував найбільшу можливу дисперсію в наборі даних. Другий головний компонент обчислюється таким же чином, за умови, що він не корельований (тобто перпендикулярний) першому головному компоненту та відповідає за наступну найбільшу дисперсію. Власний вектор, пов’язаний з найбільшим власним значенням, має той самий напрямок, що й перший головний компонент. Власний вектор, пов'язаний з другим за величиною власним значенням, визначає напрямок другого головного компонента. АОК використовує багато перетворень даних і статистичних концепцій. Таким чином, використано існуючий пакет R трансформувати та зменшувати наші дані на два PC, щоб візуалізувати варіації всіх наших точок даних у 2-вимірному просторі.

Логіка, що лежить в основі підходу АОК, полягає в тому, щоб провести загальний і повний аналіз варіацій, щоб знайти крайні точки варіації на межах багатовимірних даних. Зі статистичної точки зору ці крайні точки представляють викиди. Відповідно до запропонованого підходу МТ, ці точки відповідають невідповідностям (або відхиленням), які виявлено. Викиди мають важливий вплив на PC. Викид визначається як спостереження, яке не відповідає моделі, до якої дотримується більшість даних. Один із способів виявлення викидів – використання метрики під назвою Score Distance (SD). SD вимірює дисперсію спостережень у просторі АОК. Таким чином, він вимірює, наскільки далеко лежить спостереження від решти даних у підпросторі АОК. SD вимірює статистичну відстань від оцінки PC до центру оцінок. Для спостереження x_i оціночна відстань визначається як:

$$SD_i = \sqrt{\sum_{j=1}^{a} \frac{t_{ij}^2}{\lambda_j}} \quad (3.8)$$

де a – кількість PC, що утворюють простір АОК, t_{ij} – елементи матриці оцінок, отримані після проведення АОК, а λ_j – дисперсія j-го PC, що відповідає j-му власному значенню. Для того, щоб знайти викиди, обчислюється 97,5%-квантиль Q розподілу Хі-квадрат як граничне значення SD ($\sqrt{\chi^2_{a,0,975}}$). Це відповідає довірчому еліпсу, який охоплює 97,5% точок даних. Згідно з таблицею розподілу Хі-квадрат, це значення дорівнює $\sqrt{7,38}$ = 2,71. Будь-яка вибірка, середньоквадратичне відхилення якої є більшим за граничне значення, ідентифікується як викид (або неузгодженість). Це значення відсікання являє собою поріг варіації *T*, який визначено для запропонованого МР за допомогою методу PCA.

3.4 Оцінка

3.4.1 Експериментальна установка

3.4.1.1 Тестуються генератори коду: компілятори Haxe

Щоб перевірити застосовність запропонованого підходу, проведено експерименти з популярною мовою програмування високого рівня під назвою Haxe та її генераторами коду. Haxe – це набір інструментів з відкритим вихідним кодом для кросплатформної розробки, який компілюється для кількох різних платформ програмування, включаючи JavaScript, Flash, PHP, C++, C# і Java. Haxe включає багато функцій: мову Haxe, мультиплатформенні компілятори та різні нативні бібліотеки. Мова Haxe – це мова програмування високого рівня, яка строго типізована. Ця мова підтримує парадигми як функціонального програмування, так

і об’єктно-орієнтованого програмування. Він має загальну ієрархію типів, що робить певний API доступним на кожній цільовій платформі. Нахе поставляється з набором компіляторів, які перекладають написаний вручну код (мовою Нахе) на різні цільові мови та платформи. Код Нахе можна скомпілювати для програм, що працюють на настільних комп’ютерах, мобільних і веб-платформах. Він також постачається з набором стандартних бібліотек, які можна використовувати на всіх підтримуваних цілях, і бібліотек, що стосуються платформи, для кожної з них.

Процес трансформації та генерації коду можна описати так: Компілятори Нахе аналізують вихідний код, написаний мовою Нахе. Потім код перевіряється та аналізується в типізовану структуру, в результаті чого утворюється типізоване абстрактне синтаксичне дерево (AST). Цей AST оптимізується та перетворюється згодом для створення вихідного коду для цільової платформи/мови. Нахе пропонує можливість вибору платформи для кожної програми за допомогою параметрів командного рядка. Крім того, деякі оптимізації та інформацію про налагодження можна ввімкнути через інтерфейс командного рядка, але в наших експериментах не вмикалось жодних додаткових параметрів.

3.4.1.2 Міжплатформний тест

Одним із способів довести ефективність запропонованого підходу є створення контрольних показників. Таким чином, використано мову Нахе та її генератори коду для створення кросплатформного тесту. Пропонований еталонний тест складається з набору кросплатформних бібліотек, які можна скомпілювати для різних цілей. У цих експериментах розглядається сімейство генераторів коду, що складається з п’яти цільових компіляторів Нахе: генераторів коду Java, JS, C++, CS і PHP. Щоб вибрати міжплатформні бібліотеки, досліджується github і використовується репозиторій бібліотек Нахе. Отже, обрано сім бібліотек, які надають набір тестових наборів із високими показниками покриття коду.

Насправді кожна бібліотека Haxe постачається з API та набором тестових наборів. Ці тести, написані на Haxe, являють собою набір модульних тестів, які охоплюють різні функції API. Основним завданням цих тестів є перевірка правильності функціональної поведінки згенерованих програм. Щоб підготувати запропонований контрольний тест, видаляються всі тести, які не компілюються з нашими п’ятьма цілями (тобто помилки, збої та збої), і зберігаються лише набори тестів, які є функціонально правильними, щоб зосередитися лише на нефункціональних властивостях. Крім того, вручну додано нові тести до деяких бібліотек, щоб розширити кількість наборів тестів. Кількість наборів тестів залежить від кількості існуючих функцій у бібліотеці Haxe.

Таблиця 3.3 – Опис вибраних бібліотек тестів

Бібліотека	№ набору тестів	Опис
Color	19	Перетворення кольору з/в будь-який колірний простір
Core	51	Забезпечує розширення для багатьох типів
Hxmath	6	Математична бібліотека 2D/3D
Format	4	Бібліотека форматів, як-от формати дат, чисел
Promise	5	Бібліотека легких обіцянок і ф’ючерсів
Culture	5	Бібліотека локалізації для Haxe
Math	5	Генерація випадкових значень

3.4.1.3 Використані показники оцінювання

Ефективність згенерованого коду оцінюється за допомогою таких нефункціональних показників:

використання пам'яті: відповідає максимальному споживанню пам'яті запущеним набором тестів;

час виконання: час виконання наборів тестів вимірюється в секундах.

Також, інфраструктура тестування, що розглядається здатна оцінювати інші нефункціональні властивості згенерованого коду, такі як час генерації коду, час компіляції, розмір коду, використання ЦП. У цьому експерименті увага була зосереджена на продуктивності (тобто часу виконання) і використанні ресурсів

(тобто використанні пам’яті). Збір показників використання ресурсів забезпечує інфраструктура моніторингу.

3.4.1.4 Налаштування інфраструктури

Щоб оцінити запропонований підхід, налаштовано раніше запропоновану інфраструктуру на основі контейнерів, щоб провести експерименти на прикладі Haxe. рисунок 3.5 показує загальну картину інфраструктури тестування, яка розглядається в цих експериментах.

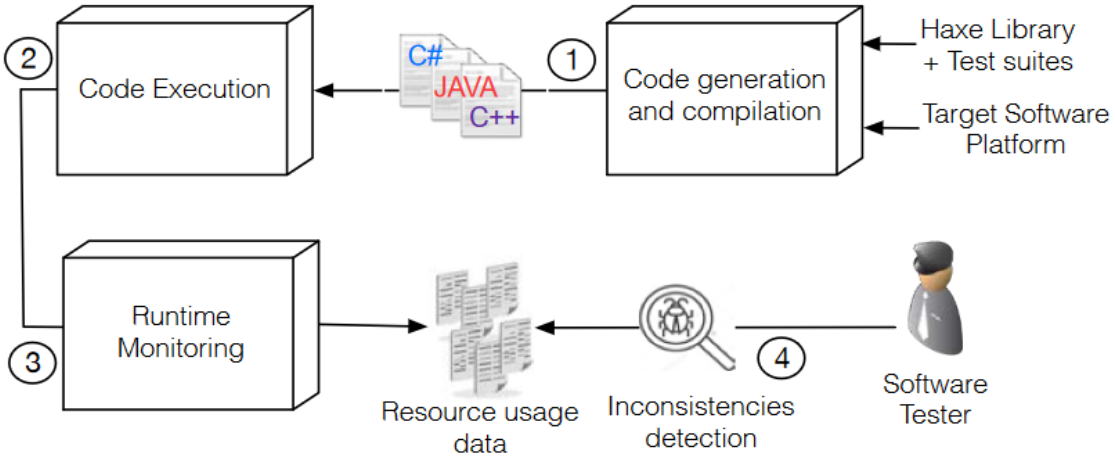


Рисунок 3.5 – Параметри інфраструктури для проведення експериментів

Спочатку створюється перший компонент, де встановлюються генератори коду Haxe і компілятори. В якості вхідних даних приймається бібліотека Haxe, яка оцінюється, і список наборів тестів (Крок 1). На виході він створює файли вихідного коду щодо цільових програмних платформ. Після цього згенеровані файли компілюються (за потреби) і автоматично виконуються в контейнері виконання (Крок 2). Цей компонент є попередньо налаштованим екземпляром контейнера, де встановлюються необхідні середовища виконання, такі як інтерпретатор PHP, вузол (для JS), mono (для C#) тощо. Тим часом, під час запуску тестових наборів усередині контейнера, збираються дані про використання

ресурсів виконання (Крок 3). Нарешті, на кроці 4 аналізуються нефункціональні дані, щоб виявити невідповідності генератора коду.

3.4.2 Експериментальна методика та результати

Результати R-chart для семи порівняльних програм щодо варіацій продуктивності та використання ресурсів представлені на малюнках 3.6 -3.19 . На рисунках 3.6-3.12, показано зміну продуктивності, що відповідає різниці діапазону R між максимальним і мінімальним часом виконання кожного набору тестів для п’яти цілей (Java, JS, C++, C# і PHP). Дані нормалізуються, усі значення діляться на мінімальний час виконання для кожного набору тестів. LCL для експериментів завжди дорівнює 0, оскільки постійне значення $D_{3,}$ як визначено в рівнянні 3.5, дорівнює нулю відповідно до таблиці констант R-chart. Фактично константа D_3 змінюється в залежності від кількості підгруп. У експериментах запропонований запис даних складається з п’яти підгруп, що відповідають п’яти цільовим мовам програмування. Центральна лінія (зеленого кольору) відповідає R-bar. Це значення змінюється від одного тесту до іншого залежно від середнього R для всіх наборів тестів у тесті.

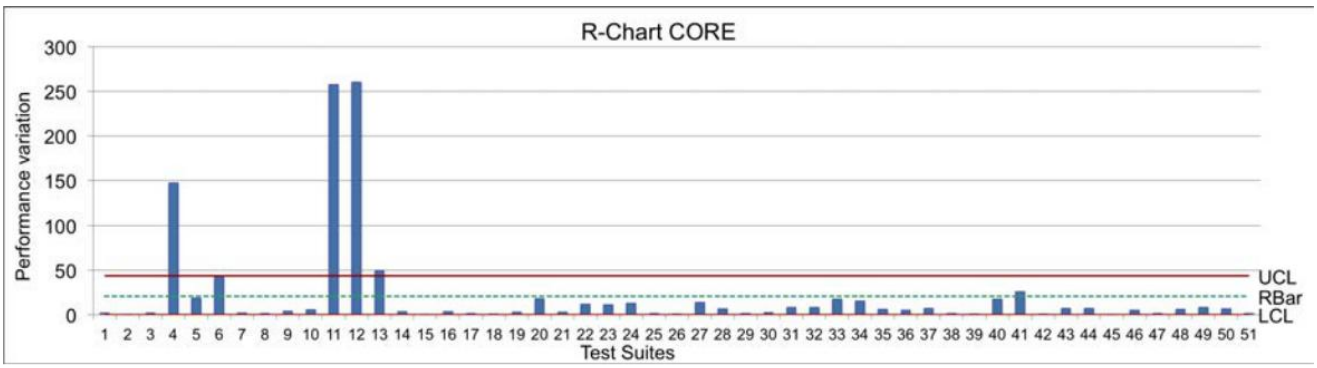


Рисунок 3.6 – швидкодія Core у Haxe

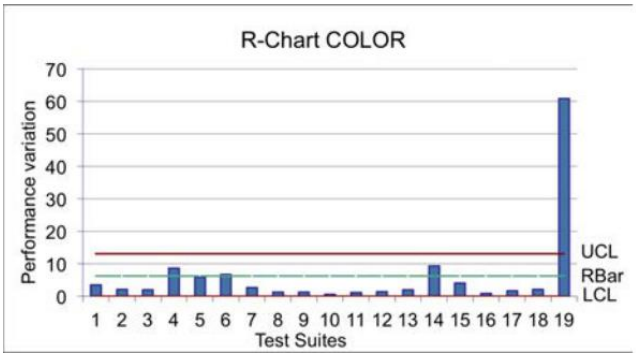


Рисунок 3.7 – швидкодія Color у Haxe

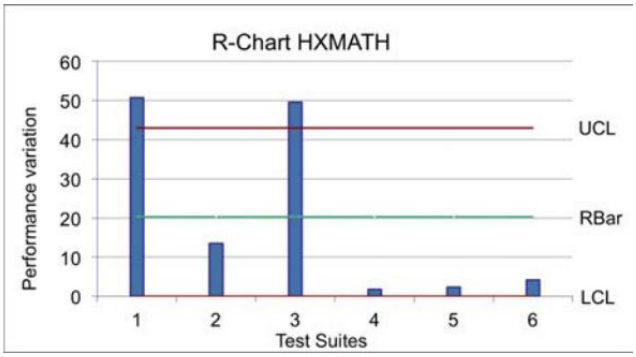


Рисунок 3.8 – швидкодія Hxmath у Haxe

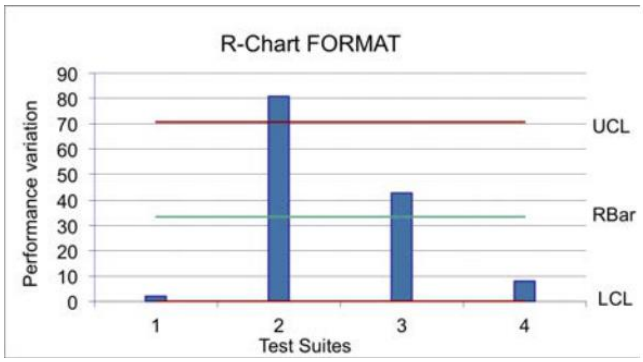


Рисунок 3.9 – швидкодія Format у Haxe

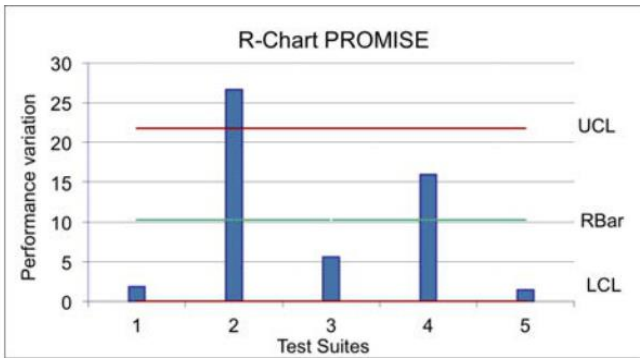


Рисунок 3.10 – швидкодія Promise у Haxe

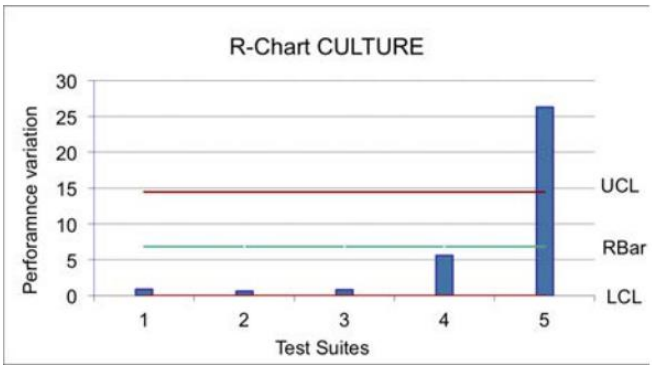


Рисунок 3.11 – швидкодія Culture у Haxe

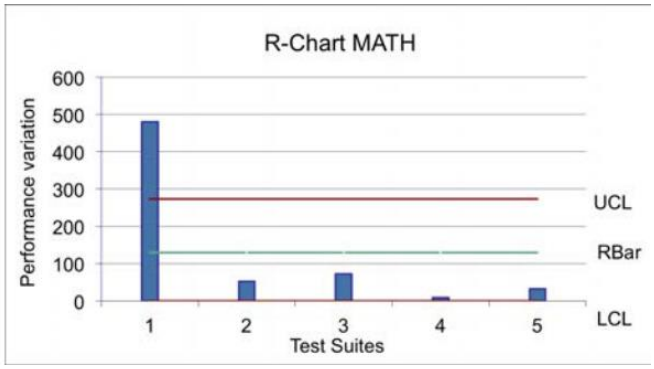


Рисунок 3.12 – швидкодія Math у Haxe

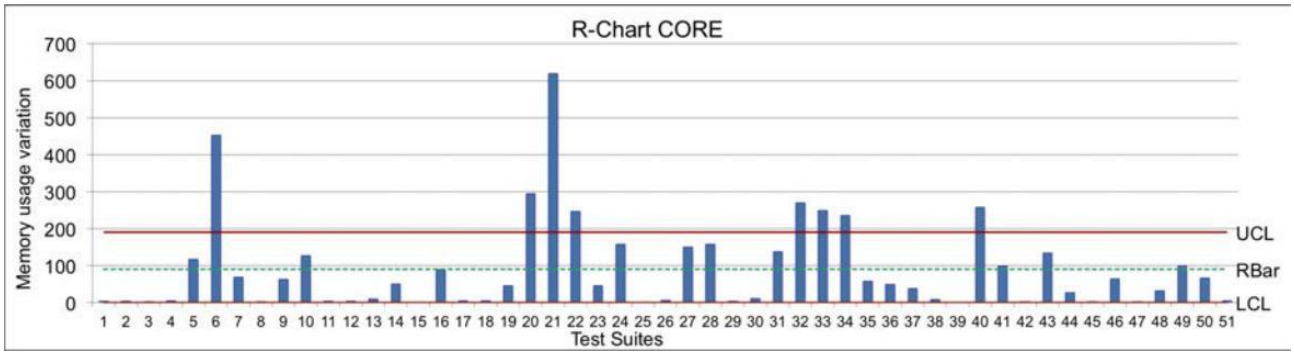


Рисунок 3.13 – використання пам’яті Core у Haxe

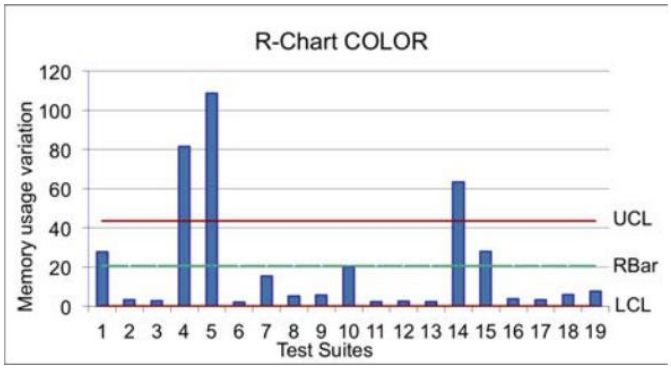


Рисунок 3.14 – використання пам’яті Color

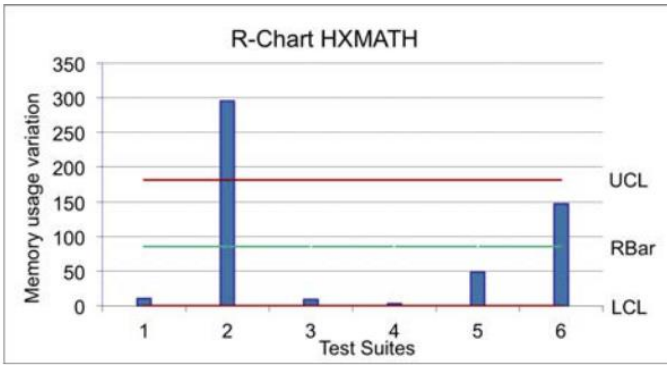


Рисунок 3.15 – використання пам’яті Hxmath

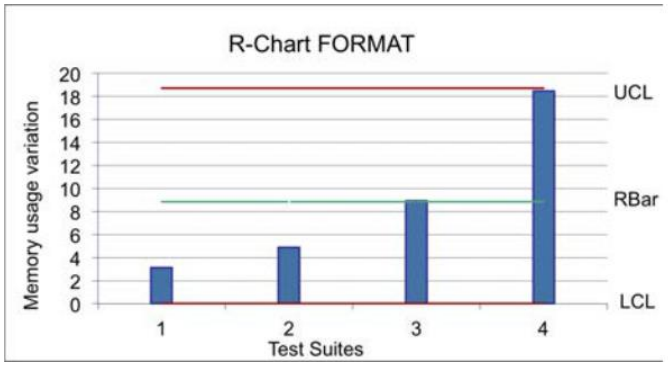


Рисунок 3.16 – використання пам’яті Format

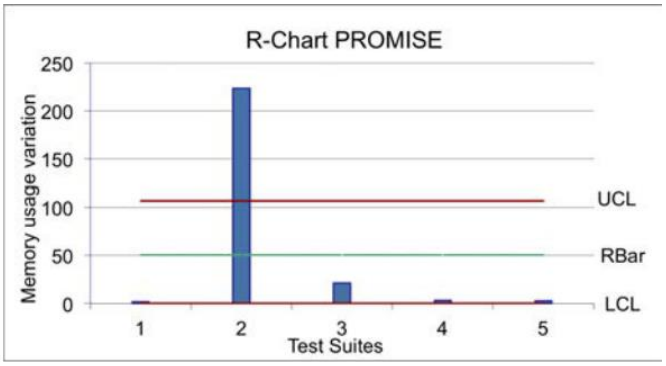


Рисунок 3.17 – використання пам’яті Promise

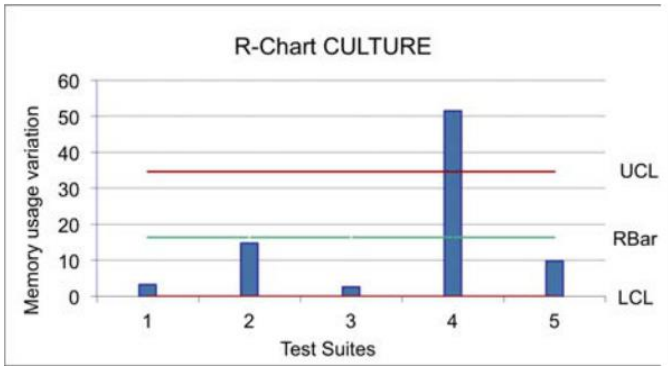


Рисунок 3.18 – використання пам’яті Culture

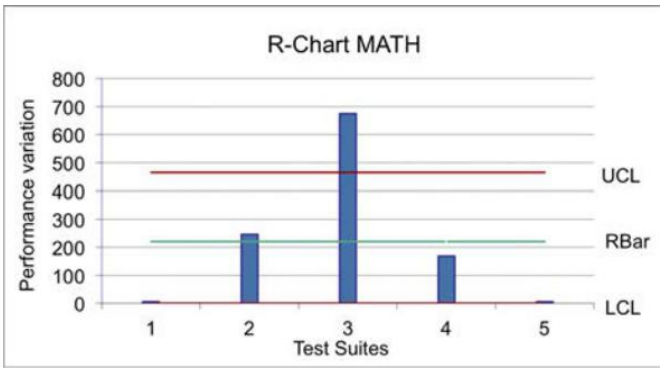


Рисунок 3.19 – використання пам’яті Math

Результати на рисунках 3.6-3.12 показують, що більшість варіацій продуктивності знаходяться в інтервалі [0 UCL], який відповідає контрольній зоні

варіації, як описано в розділі 3.3.2.5 . Однак зауважується, що для кількох наборів тестів змінюється продуктивність відносно високий (вищий за значення UCL відповідної програми тестування). Виявлено 11 серед 95 відхилень продуктивності, що лежать у неконтрольованій зоні варіації. Для інших наборів тестів варіація навіть менша, ніж загальна середня варіація R-bar. Є лише 7 наборів тестів серед решти 84, де варіація полягає в $[\bar{R} - UCL]$ відхилення, оскільки відповідно до R-chart варіації в цій зоні все ще знаходяться під контролем. Виявлені 11 відхилень у продуктивності можна пояснити тим фактом, що час виконання одного або кількох наборів тестів значно відрізняється від однієї мови до іншої. Це підтверджує ідею про те, що генератор коду створив підозрілу поведінку коду, яка призвела до високої варіації продуктивності.

Подібним чином рис 3.13-3.19 відновлює результати порівняння виконання пакетів тестів щодо використання пам'яті. Варіації в цьому експерименті важливіші за попередні результати. Це можна аргументувати тим фактом, що шаблони використання та розподілу пам’яті відрізняються від однієї мови до іншої. Тим не менш, можна визначити деякі точки, де варіація надзвичайно висока. Таким чином, виявлено 15 із 95 наборів тестів, які перевищують відповідне значення UCL. Коли варіація нижче UCL, виявлено 14 серед відносно високий. Однією з причин, що спричинила цю варіацію, може бути те, що набір тестів виконує деякі частини коду (спеціальною мовою), які дивним чином споживають багато ресурсів. Це може бути не так, коли варіація нижча, ніж $\bar{R}$ наприклад.

Отже, виявлено 11 екстремальних варіацій продуктивності та 15 екстремальних варіацій використання пам’яті серед 95 виконаних наборів тестів. Отже, припускається, що несправні генератори коду, у визначених точках, представляють загрозу для якості програмного забезпечення, оскільки згенерований код демонструє ознаки неякісного дизайну.

Результати АОК. Підхід АОК застосовано як альтернативу підходу R-chart. рисуноки 3.20-3.21 показують дисперсію точок даних у підпросторі PC. PC1 і PC2 представляють напрямки двох перших головних компонентів, що мають найбільші ортогональні варіації. Точки даних представляють варіацію продуктивності (рис

3.20) і варіація використання пам’яті (рис 3.21) із 95 наборів тестів. Точки варіації забарвлені відповідно до програми тестування, до якої вони належать (відображаються в легенді малюнка). На перший погляд можна чітко побачити, що точки варіації розташовані в одній області, за винятком деяких точок, які лежать далеко від цього кластера даних. На рисунку 3.20, рожеві точки, що відповідають математичному тесту, візуально показують найбільшу варіацію. Три набори основних тестів (червоним), які ідентифікуються як відхилення продуктивності в R-чаті, також демонструють відхилення на діаграмі розсіювання АОК. Точки 91 відносно орієнтира Math відхиляються від точки помутніння. Однак на діаграмі R-chart це не визначається як відхилення продуктивності (див. набір тестів 3 на рисунку 3.12). Насправді цей набір тестів виконується понад 80 разів. Порівняно з іншими наборами тестів, варіація продуктивності не перевищує

По суті, АОК проводить повний аналіз усіх даних, які було зібрано в усіх тестах. Таким чином, варіації відображаються щодо всіх варіантів наборів тестів у всіх тестах. Оцінка варіацій не обмежена програмою порівняння. Зображено ті самі результати на рисунку 3.22 про різницю використання пам’яті в АОК.

Щоб підтвердити це спостереження, на рисунку 3.9 наведоно результати пропонованого підходу до виявлення викидів. Визначено 4 невідповідності (або викиди) у кожній діагностичній графі. Невідповідності на рисунку 3.20 відносяться до відхилень продуктивності. Точки 31 і 32 відповідають наборам тестів 12 і 11 у тесті Core на рисунку 3.6 . Точки 91 і 95 відповідають наборам тестів 3 і 1 у тесті Math на рисунку 3.12. Для варіації використання пам’яті виявлено точки 25, 29, 82 і 92, які відповідають тестам 21 і 6 тесту Core, 2 тесту Promise і 3 тесту Math. Чітко видно, що ця методика допомагає ідентифікувати екстремальні значення, що викидаються, які здебільшого охоплюються підходом R-chart. Використовується 97,5% квантиль розподілу хі-квадрат для визначення значення куто, яке зазвичай використовується в літературі [114, 115]. Якщо зменшити це значення, можна буде виявити більше точок варіації.

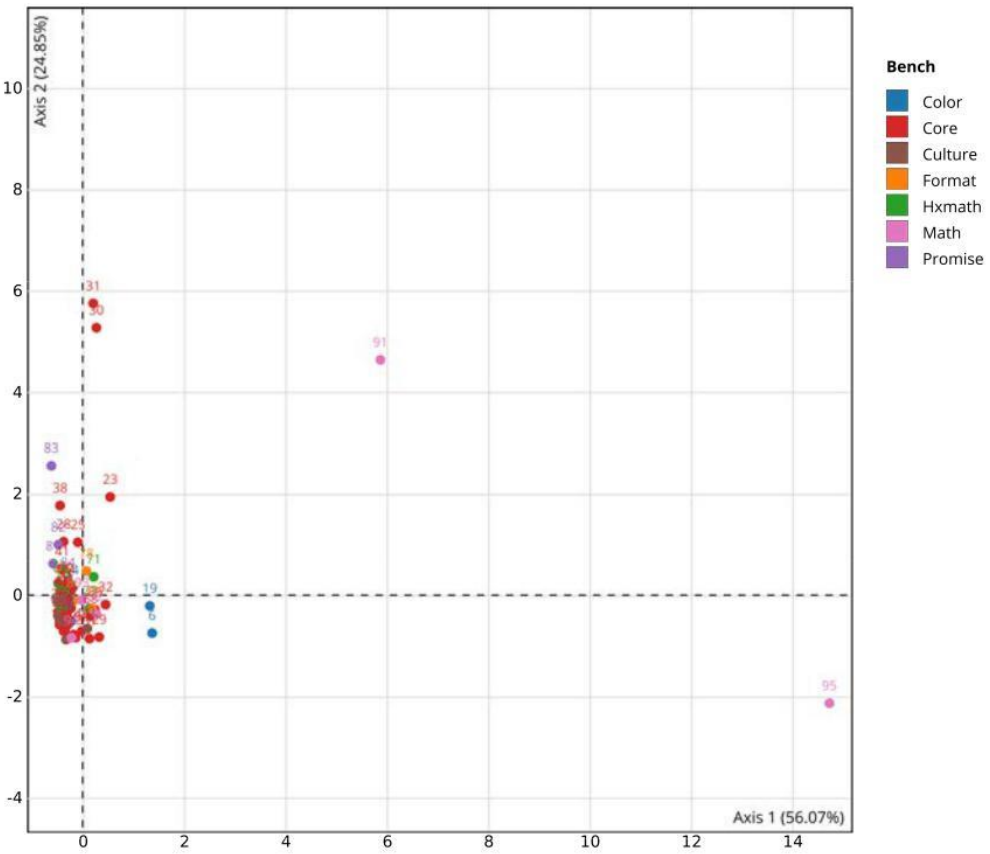


Рисунок 3.20 – Набори тестів відносно часу виконання

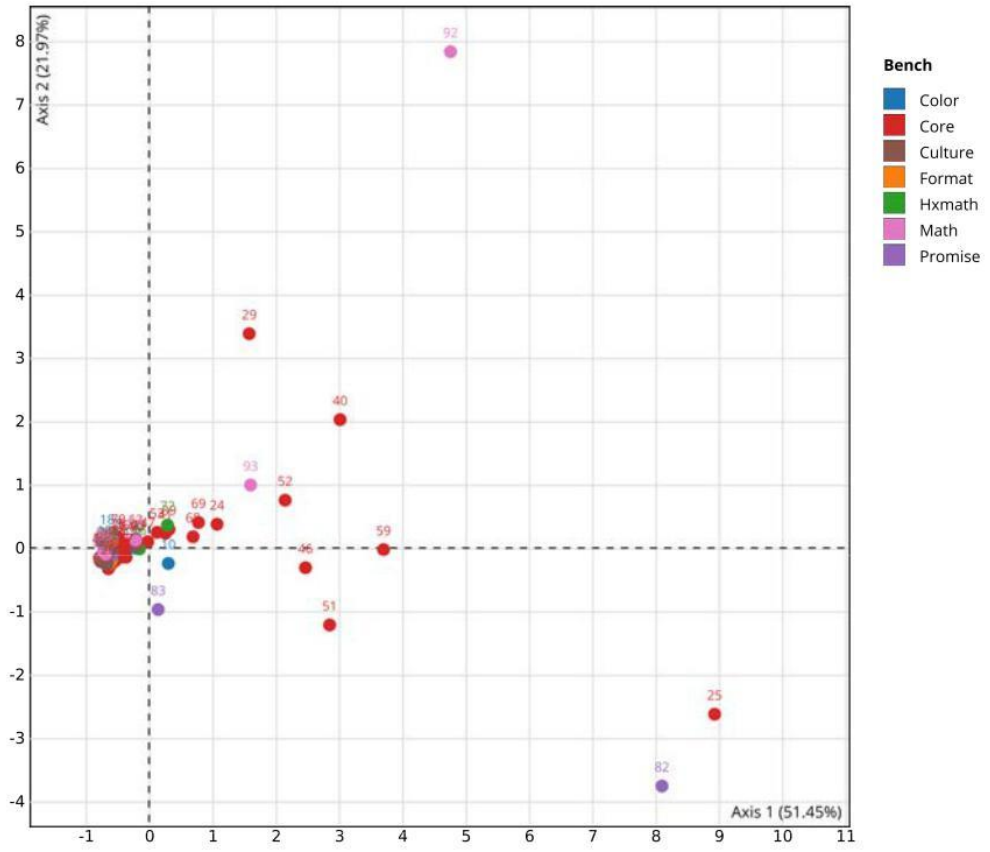


Рисунок 3.21 – Набори тестів щодо споживання пам’яті

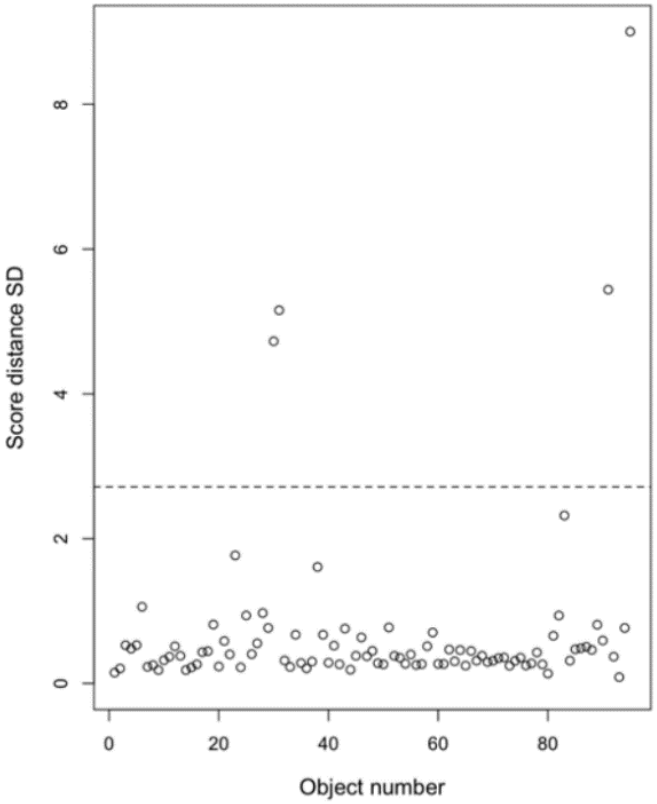


Рисунок 3.22 – Відхилення продуктивності з використанням оцінки відстані SD

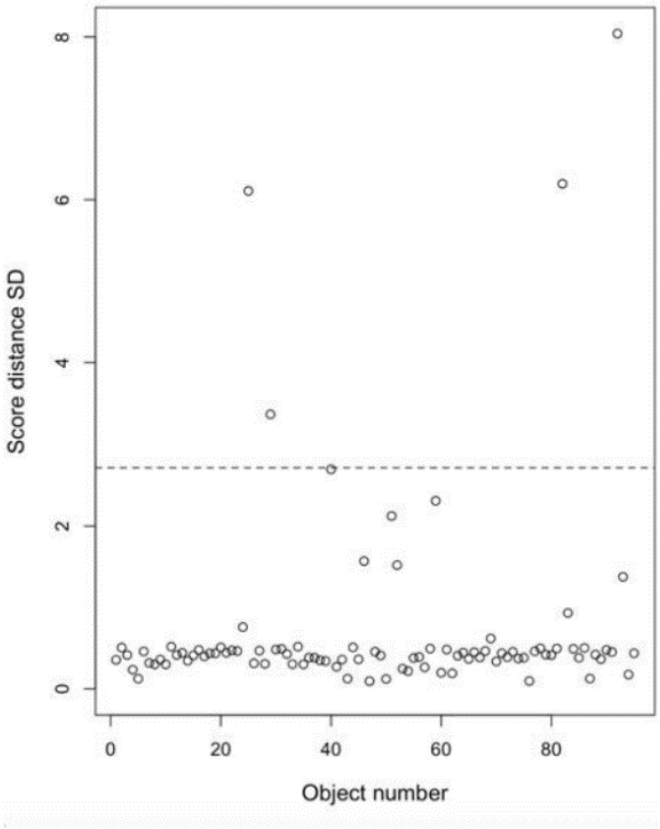


Рисунок 3.23 – Відхилення у використанні пам’яті з використанням оцінки відстані SD

Тепер, після перевірки варіації продуктивності та використання пам’яті під час виконання наборів тестів, можна проаналізувати екстремальні точки, які виявлено за допомогою R-chart, щоб більш детально спостерігати джерело такого відхилення. З цієї причини наведено в табл 3.4 і 3.5 значення необроблених даних цих наборів тестів, що призводить до екстремальних варіацій щодо часу виконання та використання пам’яті.

Таблиця 3.4 показує коефіцієнт часу виконання кожного набору тестів на певній цільовій мові. Цей коефіцієнт масштабується відносно найнижчого часу виконання серед 5 цілей.

Таблиця 3.4 – Значення необроблених даних наборів тестів, які призвели до найбільшої варіації з точки зору часу виконання

Бенчамрк	Набір тестів	Java	JS	CPP	CS	PHP	UCL(R)	Дефект CG
Color	TS19	1.90	1	2.37	3.31	61,84	13.08	PHP
Core	TS4	1	1.59	1.67	2.78	148,20	43,62	PHP
		1.14	2.71	1	3.63	258,94		PHP
		1.28	2.94	1	3.36	261,36		PHP
		1	1.05	1,86	2.39	50.30		PHP
Hxmath	TS1	2.38	1.43	1	2.82	51,72	42,97	PHP
		2.14	1.10	1	2.25	50,56		PHP
Format	TS2	1.16	1.27	1	3.35	81,85	70,66	PHP
Promise	TS2	1.52	1,85	1	1.51	27,67	21.76	PHP
Culture	TS5	1.62	1	1.27	2.02	27.29	14.47	PHP
Math	TS1	4.15	1	5.41	4.70	481,68	273,24	PHP

Також наведено UCL, визначений для порівняльного тесту. В останній колонці наведено генератор коду, який спричинив таке велике відхилення. Для цього позначено несправним CG генератор коду, який призвів до зміни продуктивності вище значення UCL.

Чітко видно, що PHP-код має особливості продуктивності з коефіцієнтом у діапазоні від x27.29 для набору тестів 5 у тесті Culture (Culture TS5) до x481.7 для Math TS1. Наприклад, якщо Math TS1 виконується за 1 хвилину в JS, той самий набір тестів у PHP займе приблизно 8 годин, що є дуже великим розривом. Найвищим фактором, виявленим для інших мов, є x5.41, що не є незначним, але це невелике відхилення порівняно з відхиленнями PHP. Хоча це правда, що порівнюється різні версії згенерованого коду, очікувалося, що під час виконання тестових випадків будуть деякі варіації з точки зору часу виконання. Однак у випадку генератора коду PHP це далеко не проста варіація, але саме невідповідність генератора коду призвела до такого зниження продуктивності.

Тим часом збирається інформацію про моменти, які призвели до найвищої варіації використання пам’яті. Таблиця 3.5 показує ці результати. Знову ж таки,

можна ідентифікувати особливу поведінку коду PHP щодо використання пам’яті з коефіцієнтом у діапазоні від x52.47 до x675. Для інших версій пакетів тестів коефіцієнт змінюється від x1 до x160,84. Спостерігається особлива поведінка коду Java для Core TS6, Core TS32 і Promise TS2, поступаючись варіації, вищій за UCL. Ці результати доводять, що генератори коду PHP і Java не завжди ефективні та становлять загрозу для згенерованого програмного забезпечення з точки зору використання пам’яті.

Таблиця 3.5 – Значення необроблених даних наборів тестів, які призвели до найбільшої варіації використання пам’яті

Бенчмарк	Набір тестів	Java	JS	CPP	CS	PHP	UCL	Дефект CG
Color	TS4	1	2.29	1.47	3.59	82,46	43,53	PHP
	TS5	1	3.08	1.83	4.53	109,69		PHP
	TS14	1	1.32	1,00	2.03	64,45		PHP
Core	TS6	250,77	71,71	1	69,90	454,15	190.03	PHP і Java
	TS20	2.31	1.34	1	3.27	296.10		PHP
	TS21	11.90	1	14.63	36.18	620,22		PHP
	TS22	1	2.70	1.74	4.69	247,32		PHP
	TS32	270,78	2.27	1	5.61	153,37		Java
	TS33	1.82	1.12	1	54.19	250,35		PHP
	TS34	1	1.17	1.48	3.90	236,97		PHP
	TS40	160,84	1.10	1	49,43	259,20		PHP
Hxmath	TS2	1	1.16	1.91	2.82	296.16	181.11	PHP
Promise	TS2	214,53	92,45	1	57,68	224,41	106,82	PHP і Java
Culture	TS4	2.75	1.01	2.52	1	52,47	34,63	PHP
Math	TS3	1.29	1	1.72	3.60	675,00	464,80	PHP

3.4.2.1 Аналіз

Ці невідповідності повинні бути виправлені творцями генераторів коду, щоб підвищити якість згенерованого коду (наприклад, код PHP). Оскільки пропонується підхід тестування «чорної скриньки», запропоноване рішення не може надати точнішу та детальнішу інформацію про частину коду, яка спричиняє ці проблеми з продуктивністю, що є одним із обмежень запропонованого підходу до тестування.

Тому, щоб зрозуміти цю несподівану поведінку PHP-коду під час застосування, наприклад, тестового набору Core TS4, перевірено PHP-код, що відповідає цьому тестовому набору. Насправді спостерігається інтенсивне використання масивів у більшості перевірених функцій. Відомо, що масиви працюють повільно в PHP, а бібліотека PHP представила набагато більш просунуті функції, такі як заповнення масиву та спеціалізовані абстрактні типи, такі як «SplFixedArray» щоб подолати це обмеження. Отже, просто змінивши ці дві частини згенерованого коду, покращено швидкість коду PHP у 5 разів.

Також зменшено використання пам’яті цим набором тестів у 2 рази. Коротше кажучи, відсутність використання конкретних типів у рідній стандартній бібліотеці PHP генератором коду PHP, таким як SplFixedArray, демонструє реальний вплив на нефункціональну поведінку згенерованого коду. Очевидно, що під час генерації коду використовуються не найкращі типи. Навпаки, ретельний вибір адекватних типів і функцій для створення коду може призвести до підвищення продуктивності.

3.4.3 Можливі проблеми

Зовнішня валідність означає можливість узагальнення висновків. У цьому дослідженні проведено експерименти на Haxe та наборі тестів, вибраних із Github і спільноти Haxe. Наприклад, не гарантується, що ці бібліотеки покривають усі функції мови Haxe. Отже, не можна гарантувати, що запропонований підхід здатний знайти всі проблеми з генераторами коду, якщо буде розроблено більш повний набір тестів. Крім того, поріг, визначений для виявлення особливої продуктивності, має величезний вплив на точність і запам’ятовуваність запропонованого підходу. Експерименти слід відтворити в інших прикладах, щоб підтвердити зроблені висновки.

Внутрішня валідність пов’язана з використанням підходу на основі контейнерів. Навіть якщо існують такі емулятори, як Qemu [14] які дозволяють відобразити поведінку різнорідного апаратного забезпечення, вибрана

інфраструктура не була оцінена для тестування згенерованого коду, націленого на гетерогенні апаратні машини. Крім того, незважаючи на те, що системні контейнери, як відомо, легкі та менш ресурсомісткі порівняно з віртуалізацією з повним стеком, підтверджено надійність запропонованого підходу, порівнявши його з невіртуалізованим підходом, щоб побачити вплив використання контейнерів на точність результатів.

3.5 Висновок

У цій роботі опиисано підхід метаморфічного тестування для автоматичного виявлення невідповідностей генератора коду. Запропонований підхід базується на інтуїції, що генератор коду часто є членом сімейства генераторів коду. Таким чином, отримується вигода від існування кількох генераторів із порівняльною функціональністю (тобто родини генераторів коду), щоб застосувати ідею метаморфічного тестування, визначаючи тестові оракули високого рівня (тобто метаморфічні відносини) як тестові оракули. Визначено метаморфічне відношення як порівняння між варіаціями продуктивності та використання ресурсів коду, створеного з однієї сім'ї генераторів коду. Будь-яка зміна, що перевищує певне порогове значення, автоматично визначається як аномалія. Застосовано два статистичні методи (тобто аналіз головних компонентів і діаграми діапазонів), щоб автоматизувати виявлення невідповідностей. Оцінюється запропонований підхід, аналізуючи продуктивність Нахе, популярної мови програмування високого рівня, яка включає набір кросплатформних генераторів коду. Оцінюються властивості, пов'язані з використанням ресурсів і продуктивністю для різних цільових програмних платформ. Запущено тестові набори в 7 бібліотеках тестів Нахе, щоб перевірити метаморфічний зв'язок (тобто варіації продуктивності та використання пам'яті) для кожної з них. Експериментальні результати показують, що запропонований підхід здатний виявити серед 95 виконаних наборів тестів 11 невідповідностей продуктивності та 15 використання пам'яті, які порушують

метаморфічний зв’язок. Показано, що запропонований підхід може автоматично виявляти реальні проблеми в сімействах генераторів коду. Зокрема, показано, що можна знайти принаймні два типи помилок: відсутність використання певної функції та абстрактний тип, який існує в стандартній бібліотеці цільової мови, що може зменшити використання пам’яті/час виконання результуюча програма. Ці проблеми мають бути досліджені та виправлені супроводжуючими/експертами генератора коду.

4 CAT: ПРОГРАМА ДЛЯ АВТОМАТИЧНОГО НАЛАШТУВАННЯ КОМПІЛЯТОРІВ

Забезпечення якості коду під час розробки програмного забезпечення є дуже важливим у програмній інженерії. Він надає розробникам програмного забезпечення засоби для підтримки, тестування та налагодження вихідного коду. Коли справа доходить до рівня компілятора, забезпечення якості згенерованого коду сильно залежить від застосованих конфігурацій до компілятора.

Однак, як обговорювалося раніше, налаштування компілятора є складним завданням через величезну кількість потенційних комбінацій оптимізації, що ускладнює та забирає багато часу для розробників програмного забезпечення, щоб знайти/конструювати послідовність оптимізацій, яка задовольняє конкретні нефункціональні вимоги користувача. Це також вимагає повного розуміння базової архітектури системи, цільової програми та доступних оптимізацій компілятора.Також зазначається що, намагаючись оптимізувати продуктивність програмного забезпечення, багато нефункціональних властивостей і дизайнерських обмежень повинні бути задіяні та задоволені одночасно, щоб краще оптимізувати код. Іноді збільшення часу виконання програми може призвести до значного використання ресурсів, що може знизити продуктивність системи. Наприклад, вбудовані системи, для яких генерується код, часто мають обмежені ресурси. Таким чином, методи оптимізації повинні застосовуватися, коли це можливо, для створення ефективного коду та покращення продуктивності (з точки зору часу виконання) щодо доступних ресурсів (використання ЦП або пам’яті) [116]. Тому важливо побудувати рівні оптимізації, які представляють численні обміни між нефункціональними властивостями, дозволяючи розробнику програмного забезпечення вибирати серед різних оптимальних рішень, які найкраще відповідають специфікаціям системи.

Як обговорювалося в розділі про стан техніки, існує багато підходів, які вирішують ці проблеми оптимізації, щоб допомогти користувачам конфігурувати

(або налаштовувати) компілятори з багатьма нефункціональними властивостями, таких як розмір коду, споживання енергії, час виконання тощо.

Цей розділ представляє альтернативний підхід до попередніх дослідницьких зусиль. У ньому представлено CAT (Compiler Auto-Tuner), підхід для автоматичного налаштування компіляторів. Запропонований підхід базується на мікросервісах для автоматизації розгортання та моніторингу різних варіантів оптимізованого коду. CAT – це інструмент на вимогу, який використовує моно- та багатоцільові алгоритми еволюційного пошуку для створення послідовностей оптимізації, які задовольняють основні цілі користувача (час виконання, розмір коду, час компіляції, використання ЦП або пам’яті тощо).

У минулому дослідники показали, що вибір послідовностей оптимізації може вплинути на продуктивність програмного забезпечення [63, 68]. Як наслідок, оптимізація продуктивності програмного забезпечення стає ключовою метою як для індустрії програмного забезпечення, так і для розробників, які часто готові платити додаткові витрати для досягнення конкретних цілей продуктивності, особливо для систем з обмеженими ресурсами.

Універсальні та попередньо визначені послідовності, наприклад, від O1 до O3 у GCC, можуть не завжди давати хороші результати продуктивності та можуть сильно залежати від тесту та вихідного коду, на якому вони були протестовані [66, 65, EAC15]. Кожна з цих оптимізацій взаємодіє з кодом і, у свою чергу, з усіма іншими оптимізаціями складним чином. Як наслідок, більшість програмістів, які не знайомі з оптимізацією компілятора, мають труднощі з вибором ефективних послідовностей оптимізації [63].

Щоб дослідити великий простір оптимізації, користувачі мають оцінити ефект оптимізації відповідно до конкретної нефункціональної цілі (див. рис. 4.1). Оптимізація використання різних властивостей, таких як час виконання, час компіляції, споживання ресурсів, розмір коду тощо. Таким чином, пошук оптимальної комбінації оптимізації для вхідного вихідного коду є складною та трудомісткою проблемою.

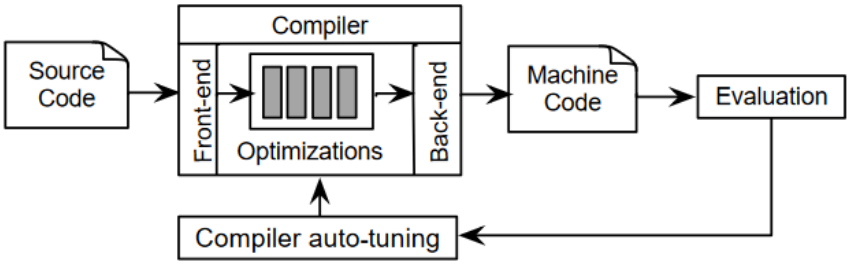


Рисунок 4.1 – Процес дослідження оптимізації компілятора

Важливо зауважити, що виконання оптимізації вихідного коду може бути настільки дорогим при використанні ресурсів, що може викликати помилки компілятора або збої. Дійсно, в середовищі з обмеженими ресурсами оптимізація компілятора може призвести до витоків пам'яті або збоїв у виконанні [28]. Таким чином, необхідно перевірити нефункціональні властивості оптимізованого коду та перевірити його поведінку щодо оптимізацій, які можуть призвести до покращення чи погіршення продуктивності.

Приклад: компілятор GCC. Колекція компіляторів GNU, GCC, є дуже популярною колекцією програмних компіляторів, доступних для різних платформ. GCC демонструє свої різні оптимізації за допомогою ряду рекламних блоків, які можна ввімкнути або вимкнути за допомогою перемикачів компілятора командного рядка.

Наприклад, версія 4.8.4 надає широкий спектр параметрів командного рядка, які можна ввімкнути або вимкнути, включаючи понад 150 параметрів для оптимізації. Різноманітність доступних варіантів оптимізації робить простір проектування для рівня оптимізації дуже величезним, збільшуючи потребу в евристиках для дослідження простору пошуку можливих послідовностей оптимізації. Як показано в табл 4.1 існує 76 оптимізаційних параметрів, які ввімкнено чотирма рівнями оптимізації за замовчуванням (O1, O2, O3, Ofast). Кожен стандартний рівень складається з низки оптимізацій. Ці рівні визначаються розробниками компіляторів на основі їхнього досвіду та попередніх експериментів. Метою визначення цих стандартних рівнів є побудова загальних і програмних незалежних послідовностей, які представляють обмін між кількома нефункціональними властивостями.

Таблиця 4.1 – Параметри оптимізації компілятора, доступні рівнями стандарту GCC

Рівень	Параметр оптимізації	Рівень	Параметр оптимізації
O1	-fauto-inc-dec -fcompare-elim -fcprop-registers -fdce -fdefer-pop -fdelayed-branch -fdse -fguess-branch-probability - f-conversion2 - f-conversion - pa-pure-const - pa-pro le - pa-reference -fmerge-constants -fsplit-wide-types -ftree-bit-ccp -ftree-builtin-call-dce -ftree-ccp -ftree-ch -ftree-copyrename -ftree-dce -ftree-dominator-opts -ftree-dse -ftree-forwprop -ftree-fre -ftree-phiprop -ftree-slsr -ftree-sra -ftree-pta -ftree-ter -funit-at-a-time	O2	-fthread-jumps -falign-functions -falign-jumps -falign-loops -falign-labels -fcaller-saves -fcrossjumping -fcse-follow-jumps -fcse-skip-blocks -fdelete-null-pointer-checks -fdevirtualize -fexpensive-optimizations -fgcse -fgcse-lm -fhoist-adjacent-loads - nline-small-functions - ndirect-inlining - pa-sra -foptimize-sibling-calls -fpartial-inlining -fpeephole2 -fregmove -freorder-blocks -freorder-functions -frerun-cse-after-loop -fsched-interblock -fsched-spec -fschedule-insns -fschedule-insns2 -fstrict-aliasing -fstrict-over ow -ftree-switch-conversion
O3	- nline-functions -funswitch-loops -fpredictive-commoning -fgcse-after-reload -ftree-vectorize -fvect-cost-model -ftree-partial-pre - pa-cp-clone		-ftree-tail-merge -ftree-pre -ftree-vrp
Ofast	- ast-math		

Наприклад, O1 вмикає оптимізацію, яка зменшує розмір коду та час виконання без виконання будь-якої оптимізації, яка зменшує час компіляції. Він вмикає 32 параметра. O2 збільшує час компіляції та скорочує час виконання згенерованого коду. Він вмикає всі параметри оптимізації, визначені O1, а також 35 інших параметрів. O3 є більш агресивним рівнем, який включає всі опції O2 плюс ще 8 оптимізацій. Нарешті, Ofast є найагресивнішим рівнем, який дозволяє оптимізувати не всі стандартні програми. Він вмикає всі оптимізації O3 плюс ще одну агресивну оптимізацію. Це призводить до величезного простору з ^{276} можливими комбінаціями оптимізації.

4.1 Еволюційне дослідження оптимізації компілятора

Для вивчення великого набору комбінацій оптимізації сучасних компіляторів можна використовувати багато методів (метаевристики, випадковий пошук тощо). У представленому підході, зокрема, вивчається використання техніки пошуку новизни (NS) для визначення набору параметрів оптимізації компілятора, які оптимізують нефункціональні властивості коду.

4.1.1 Адаптація пошуку новизни

У цій роботі надано нову альтернативу для вибору ефективних варіантів оптимізації компілятора порівняно з сучасними підходами. Оскільки простір пошуку можливих комбінацій занадто великий, використано нову техніку на основі пошуку під назвою «Пошук новизни» [117], щоб вирішити цю проблему. Ідея цього методу полягає в тому, щоб дослідити простір пошуку можливих опцій компілятора, розглядаючи різноманітність послідовностей як єдину мету. Замість вибору на основі ефективності, який максимізує одну з нефункціональних цілей, обираються послідовності оптимізації на основі оцінки новизни, показуючи, наскільки вони відрізняються від усіх інших оцінених досі комбінацій. NS –

дивергентний еволюційний алгоритм, який винагороджує послідовності оптимізації, що відрізняються від раніше відкритих. Таким чином, еволюцію можна розглядати як дивергентний процес порівняно з традиційними конвергентними підходами, такими як GA, які чинять тиск відбору на основі значень що підходять.

Крім того, пошук ефективних послідовностей оптимізації не є простим, оскільки взаємодія між оптимізаціями є надто складною і важкою для визначення. Наприклад, у попередній роботі [68] автори показали, що кілька оптимізацій можуть призвести до вищої продуктивності, ніж інші методи ітераційної оптимізації. Насправді, пошук на основі сумісності може потрапити в пастку деяких локальних оптимумів, які неможливо уникнути [77]. Це явище відоме як втрата різноманіття". Наприклад, якщо найбільш ефективна послідовність оптимізації, яка викликає менший час виконання, лежить далеко від простору пошуку, визначеного градієнтом функції сумісності, тоді деякі перспективні області пошуку можуть бути недоступні. . Проблема передчасної збіжності до локальних оптимумів була загальною проблемою в еволюційних алгоритмах. Як було описано в раніше, зараз пропонується багато методів для подолання цієї проблеми [118]. Однак усі ці спроби використовують відбір на основі цілісності для керування пошуком. Розгляд різноманітності як унікальної цільової функції, яку потрібно оптимізувати, може бути ключовим рішенням цієї проблеми.

Тому під час еволюційного процесу обираються оптимізаційні послідовності, які залишаються в розріджених регіонах простору пошуку, щоб спрямувати пошук до новизни. Тим часом було вирішено зібрати нефункціональні показники щодо споживання ресурсів (пам’яті та ЦП) оптимізованого коду. Нижче описано псевдокод представленої адаптації.

Алгоритм 1: Пошук новизни для генерації тестових даних
Require: Java interface I
Require: Source code package Pack
Require: Number of iterations N
Require: Population size PopSize
Require: Coverage threshold minCoverage

Require: Novelty threshold T
Require: Limit L
Require: Nearest neighbors k
Ensure: Set of relevant test cases bestSolutions
1: *targetClasses* ← *loadAllM ethods(I, P ack)*
2: *testCases* ← *generateTestCases(I)*
3: **repeat**
4: *testSuite* ← *generateTestData(testCases)*
5: *P* ← *setOf (testSuite)*
6: **for** *testSuite* ∈ *P* **do**
7: **for** *testCase* ∈ *testSuite* **do**
8: *coverage* ← *execute(testCase, targetClasses)*
9: **end for**
10: *noveltyM etric* ←*distFromkNearest(testSuite, archive, P, k)*
11: **if** *noveltyM etric* > *T* **then**
12: *archive* ← *archive* + *testSuite*
13: *selectedT S* ← *selectedTS* + *testSuite*
14: **end if**
15: **if** *coverage* ≥ *minCoverage* **then**
16: *bestSolutions* ← *bestSolutions* + *testSuite*
17: **end if**
18: **end for**
19: *P* ← *generateNewPopulation(selectedT S)*
20: *generation* ← *generation* + *1*
21: **until** *generation* = *N*
22: **return** *bestSolutions*

Алгоритм 1 описує загальну ідею адаптації NS для генерації тестових даних: алгоритм приймає як вхідні дані інтерфейс Java і пакет вихідного коду. Ініціюється кількість ітерацій N, розмір популяції та мінімальне покриття значення. Останній визначає поріг охоплених тверджень, який має бути досягнутий. Тестові випадки, які перевищують цей поріг, автоматично додаються до набору відповідних тестових випадків. Оскільки порівнюється підхід NS з підходами на основі придатності та випадковими, встановлено мінімальне значення покриття на основі максимального значення покриття, досягнутого цими двома підходами. Таким чином, можна записати лише тестові випадки, які можуть подолати ті, що створені підходами на основі придатності та випадковими підходами. Те ж саме стосується порогу новизни T, який визначає поріг того, наскільки новим має бути набір тестів, перш ніж його буде додано до архіву. визначається максимальний розмір для архіву L і число k, яке використовуватиметься для обчислення відстані від k-

найближчих сусідів. k – фіксований параметр, який визначається експериментальним чином.

Спочатку завантажується набір класів що тестуються, які реалізують заданий інтерфейс I. Перед початком процесу генерації тестових даних визначається набір тестових випадків, які використовуватимуться для генерації тестових даних. Рядки 3-21 кодують основний цикл NS, який шукає найкращий набір тестів. Під час кожної ітерації створюється нову популяцію, яка відповідає набору тестових наборів. Потім генеруються випадкові тестові дані і виконується набір тестів на цільових класах. Значення покриття обчислюється та призначається кожному розв'язку (виконаним тестам).

Таким же чином, для кожного набору тестів у сукупності обчислюється середня відстань від k-найближчих сусідів. Якщо новизна є достатньо високою (вищою за заданий поріг T), то вибирається набір тестових випадків та занесено до постійного архіву. Генетичні оператори пізніше застосовуються до цих вибраних тестових випадків, щоб створити нащадків і створити наступну популяцію. Відповідні тестові випадки, які досягають значення покриття, зберігаються, що перевищує мінімальний поріг покриття, визначений спочатку.

4.1.1.1 Представлення послідовності оптимізації

Для прикладу рішення-кандидат представляє всі перемикачі компілятора, які використовуються на чотирьох стандартних рівнях оптимізації (O1, O2, O3 і Ofast). Таким чином, рішення представляється як вектор, де кожен вимір є компілятором. Змінні, які представляють параметри компілятора, представлені як гени в хромосомі. Таким чином, рішення представляє значення CFLAGS, яке використовується GCC для компіляції програм. Рішення завжди має однаковий розмір, що відповідає загальній кількості залучених параметрів. Однак під час еволюційного процесу ці реклами вмикаються або вимикаються залежно від операторів мутації та кросинговеру (див. приклад на рисунку 4.2). Крім того, зберігається той самий порядок виклику кодів компілятора, оскільки це не впливає на процес оптимізації та обробляється внутрішньо GCC.

4.1.1.2 Показник новизни

Метрика новизни виражає розрідженість вхідної послідовності оптимізації. Він вимірює відстань до всіх інших послідовностей у поточній сукупності та до всіх послідовностей, які були виявлені в минулому (тобто послідовностей в архіві). Можна кількісно визначити розрідженість рішення як середню відстань до k-найближчих сусідів.

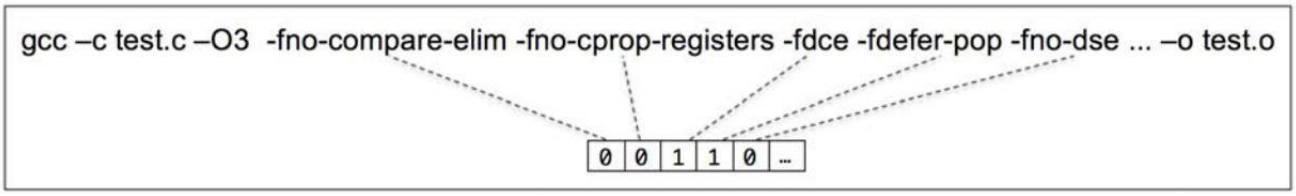


Рисунок 4.2 – Представлення рішення

Якщо середня відстань до найближчих сусідів певної точки велика, то вона належить до розрідженої області та отримає високий бал новизни. Інакше, якщо середня відстань невелика, тому вона точно належить до щільної області, тоді вона отримає низький бал новизни. Відстань між двома послідовностями обчислюється як загальна кількість симетричних відмінностей між послідовностями оптимізації. Формально ця відстань визначається наступним чином:

$$distanse(S1, S2) = |S1 \triangle S2| \qquad (4.1)$$

де S1 і S2 – дві обрані оптимізаційні послідовності (рішення). Значення відстані дорівнює 0, якщо дві оптимізаційні послідовності подібні, і більше 0, якщо існує принаймні одна оптимізаційна різниця. Максимальне значення відстані дорівнює загальній кількості вхідних параметрів.

Щоб виміряти розрідженість рішення, використано раніше визначену відстань для обчислення середньої відстані послідовності до її k-найближчих

сусідів. У цьому контексті визначається метрика новизни конкретного рішення наступним чином:

$$NM(S) = \frac{1}{k}\sum_{i=1}^{k} distanse(S, \mu_i) \quad (4.2)$$

де μ_i – і-й найближчий сусід розв'язку S в межах популяції та архіву нових значень в базі данних.

4.1.2 Пошук новизни для багатоцільової оптимізації

Багатоцільовий підхід забезпечує обмін між двома цілями, де розробники можуть вибрати бажане рішення з Парето-оптимального фронту. Ідея цього підходу полягає у використанні багатоцільових алгоритмів для пошуку обміну між нефункціональними властивостями згенерованого коду, наприклад <ExecutionTime-MemoryUsage>. Кореляції які досліджуються, більше пов'язані з обміном між споживанням ресурсів і часом виконання.

Наприклад, NS можна легко адаптувати до багатоцільових задач. У цій адаптації формулювання SBSE залишається таким самим, як описано в Алгоритмі 1 . Однак, щоб оцінити нові відкриті рішення, потрібно розглянути дві основні цілі та додати недоміновані рішення до недомінованого набору Парето. Застосовано відношення домінування Парето до рішень, які не домінуються за Парето будь-яким іншим рішенням, виявленим на даний момент, як-от у NSGA-II [83, 119]. Потім ця недомінована множина Парето повертається як результат. Як правило, наприкінці NS є більше одного оптимального рішення. Максимальний розмір остаточного набору Парето не може перевищувати розмір початкової сукупності.

4.2 Оцінювання

4.2.1 Експериментальна установка

4.2.1.1 Програми, використані в емпіричному дослідженні

Щоб дослідити вплив оптимізації компілятора, потрібен набір програм введення. З цією метою використовується генератор випадкових програм C під назвою Csmith [28]. Csmith – це інструмент, який може генерувати випадкові програми на C, які статично та динамічно відповідають стандарту C99. Його широко використовували для виконання функціонального тестування компіляторів [120, 27, 121], але не для перевірки нефункціональних вимог. Csmith може генерувати програми на C, які використовують набагато ширший діапазон можливостей C, включаючи складні елементи управління та структури даних, такі як покажчики, масиви та структури. Програми Csmith постачаються зі своїми наборами тестів, які досліджують структуру згенерованих програм (тобто високоякісне покриття коду). Автори [28] стверджують, що Csmith є ефективним інструментом виявлення помилок, оскільки він створює тести, які досліджують нетипові комбінації можливостей мови C. Вони також стверджують, що більші програми більш ефективні для функціонального тестування. Таким чином, запустивши Csmith протягом 24 годин і зібравши найбільші згенеровані програми. Було згенеровано 111 програм на C із середньою кількістю вихідних рядків 12K.

Крім того, проведено експерименти на загальновживаних тестах під назвою Collective Bench-mark (cBench) [122]. Це набір послідовних програм з відкритим вихідним кодом на C, орієнтованих на певні сфери ринку вбудованих систем. Він поставляється з кількома наборами даних, зібраними спільнотою, щоб забезпечити реалістичний порівняльний аналіз і дослідження оптимізації програм і архітектури. cBench містить понад 20 програм C. Таблиця 4.2 описує програми, які обрано з цього тесту для оцінки пропонованого підходу. Ці реальні тестові програми використовуються для вивчення впливу оптимізації компілятора на використання ресурсів в експериментах.

4.2.1.2 Налаштування параметрів

Важливим аспектом для алгоритмів метаевристичного пошуку є налаштування та вибір параметрів, які необхідні для забезпечення не тільки справедливого порівняння, але й для потенційної реплікації. CAT реалізує три монооб'єктивні алгоритми пошуку.

Таблиця 4.2 – Опис вибраних програм тестування

Програма	Вихідні рядки	Опис
automative susan s	1376	Пакет розпізнавання зображень
bzip2e	5125	Стиснення вихідного коду будь-якого файлу
bzip2d	5125	Розархівовування заархівованих файлів
office_rsynth	4111	Програма перетворення тексту в мову, створена шляхом інтеграції різних фрагментів коду
consumer_tiffmedian	15870	Застосування алгоритму медіанного відсікання до даних у файлі TIFF
consumer_tiffdither	15399	Перетворення чорно-білого зображення в кольорове

Пошук (RS), NS і GA [123]) і дві багатоцільові оптимізації (NS і NSGA-[119]). Кожна початкова сукупність/розв'язок різних алгоритмів є абсолютно випадковою. Критерій зупинки – коли досягнуто максимальної кількості оцінок щільності. Отримані значення параметрів наведено в табл 4.3 . До всіх порівнюваних алгоритмів застосовуються однакові налаштування параметрів. Під час еволюційного процесу кожне рішення оцінюється за допомогою метрики новизни. Новизна обчислюється для кожного рішення, беручи середнє з його 15 найближчих послідовностей оптимізації з точки зору подібності (враховуючи всі послідовності в поточній популяції та в архіві). Спочатку архів порожній. Відстань новизни нормується в діапазоні [0-100]. Потім, щоб створити наступні популяції, еліта з 10 найновіших організмів копіюється без змін, після чого решта нової популяції створюється шляхом турнірного відбору відповідно до новизни (розмір турніру = 2). Стандартні оператори кросинговеру та мутації генетичного програмування застосовуються до цих нових послідовностей, щоб створити нові популяції та повноцінну наступну популяцію (кросинговер = 0,5, мутація = 0,1). Тим часом

унікальні значення, які отримують бал вище 30 (поріг T), також автоматично додаються до архіву. Фактично цей поріг є динамічним. Кожні 200 оцінок перевіряється, скільки осіб було скопійовано в архів. Якщо це число менше 3, поріг збільшується шляхом множення його на 0,95, тоді як якщо додані в архів рішення вище 3, поріг зменшується шляхом множення його на 1,04. Крім того, у міру збільшення розміру архіву обчислення найближчого сусіда, яке визначає бали новизни для унікальних значень стають більш вимогливими до обчислень. Отже, щоб уникнути низької точності новизни, вирішено обмежити розмір архіву (розмір архіву = 500). Отже, він дотримується структури даних «перший увійшов, перший вийшов», що означає, що коли додається нове рішення, найстаріше рішення в архіві новинок буде відкинуто. Таким чином, забезпечується популяційнв різноманітність, видаляючи старі послідовності, які більше не можуть бути доступні для поточної популяції.

Таблиця 4.3 – Параметри алгоритму

Папаметр	Значення	Параметр	Значення
Novelty nearest-k	15	Tournament size	2
Novelty threshold	30	Mutation prob.	0.1
Max archive size	500	Crossover	0.5
Population size	50	Elitism	10
Individual length	76	Scaling archive prob.	0.05

Вибрані значення параметрів найчастіше використовуються в літературі [124, 125]. Було обрано значення, яке дало найвищу оцінку ефективності.

4.2.1.3 Використані показники оцінювання

Для монооб’єктивних алгоритмів для оцінки рішень використовуються наступні показники:

Зменшення споживання пам’яті (MR): відповідає відсотковому відношенню зменшення використання пам’яті запущеного контейнера до базового рівня.

Базовим у експериментах є рівень O0, що означає неоптимізований код. Більші значення цього показника означають кращу продуктивність. Використання пам'яті вимірюється в байтах.

Зменшення споживання ЦП (CR): відповідає відсотковому відношенню зменшення використання ЦП порівняно з базовим рівнем. Більші значення цього показника означають кращу продуктивність. Споживання процесора вимірюється в секундах, як процесорний час.

Speedup (S): відповідає відсотковому покращенню швидкості виконання оптимізованого коду порівняно з часом виконання базової версії. Час виконання програми вимірюється в секундах.

4.2.1.4 Створення інфраструктури

Налаштовуємо CAT для проведення різних експериментів. рисунок 4.3 показує загальну картину інфраструктури тестування та моніторингу розглянуті в цих експериментах.

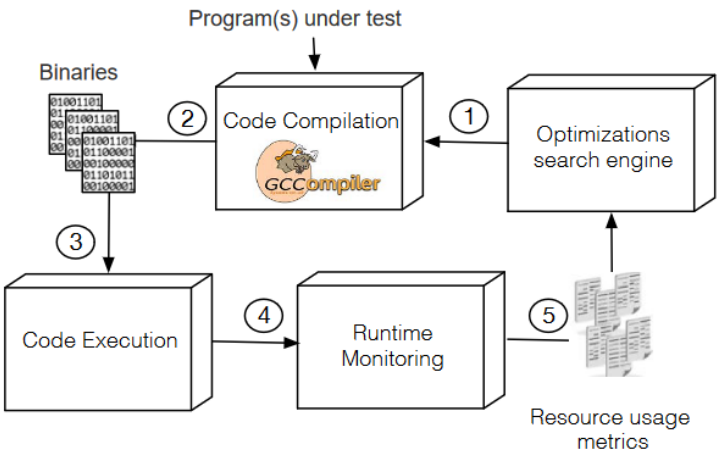


Рисунок 4.3 – CAT Експериментальна інфраструктура

По-перше, мета-евристика (моно- або багатоцільова) застосовується для створення певних послідовностей оптимізації для компілятора GCC (крок 1). Під час усіх експериментів використовується GCC 4.8.4, як це вказано в розділі

мотивації, хоча можна вибрати іншу версію компілятора за допомогою CAT, оскільки процес вилучення оптимізацій виконується автоматично. Потім компілюється програма вхідного вихідного коду, використовуючи набір згенерованих оптимізацій (крок 2). Після цього виконується оптимізований код у новому екземплярі контейнера (крок 3). Під час виконання оптимізованого коду збираються показники використання ресурсів (крок 4). Нарешті CAT оцінює набір оптимізацій, надаючи оцінку продуктивності/використання ресурсів для поточного рішення (крок 5). Вибір нефункціональних показників залежить від цілей експерименту (поліпшення пам’яті, прискорення, заміна тощо).

4.2.2 Експериментальна методологія та результати

4.2.2.1 Монооб'єктивна перевірка SBSE

Проведено одноцільовий пошук для дослідження оптимізації компілятора, щоб оцінити нефункціональні властивості оптимізованого коду. Таким чином, генеруються послідовності оптимізації за допомогою трьох методів на основі пошуку (RS, GA та NS) і порівнюємо результати їх продуктивності зі стандартними рівнями оптимізації GCC (O1, O2, O3 та Ofast).

У цьому експерименті оптимізується час виконання (S), використання пам’яті (MR) і споживання ЦП (CR). Кожна нефункціональна властивість покращується окремо та незалежно від інших показників. Інші властивості також можна оптимізувати за допомогою CAT (наприклад, розмір коду, час компіляції тощо), але в цьому експерименті увагу зосереджено лише на трьох властивостях.

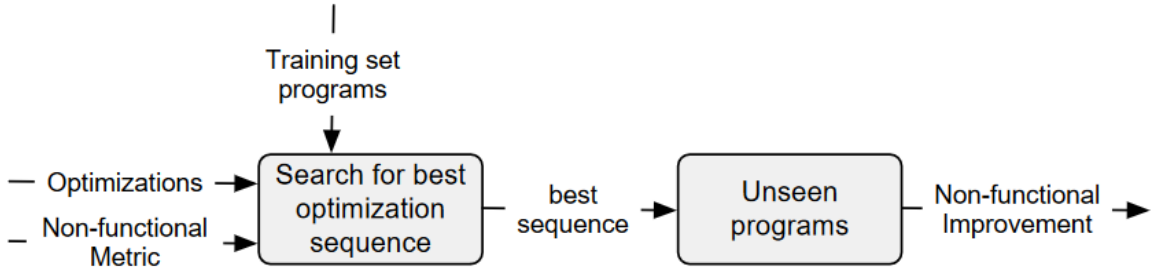


Рисунок 4.4 – Стратегія оцінювання

Як показано зліва на рисунку 4.4, враховуючи список оптимізацій і нефункціональну ціль, CAT використовується для пошуку найкращої послідовності оптимізації в наборі вхідних програм, який названо «навчальним набором». Цей «навчальний набір» складається з випадкових програм Csmith. (10 програм). Використовуються згенеровані послідовності до цих програм. Таким чином, показник якості коду в цьому параметрі дорівнює середньому покращенню продуктивності (S, MR або CR), і це для всіх тестованих програм.

Підводячи підсумок, мета експерименту:

порівняти ефективність запропонованого нами дослідження оптимізаційних послідовностей (NS) на основі різноманітності з GA та RS;

продемонструвати, що CAT може знайти оптимальне рішення щодо вхідного навчального набору.

Таблиця результатів 4.4 демонструє про результати порівняння трьох нефункціональних властивостей CR, MR і S.

Таблиця 4.4 – Результати одноцільової оптимізації

	O1	O2	O3	Ofast	RS	GA	NS
S	1.051	1.107	1.107	1.103	1.121	1.143	1.365
MR(%)	4.8	-8.4	4.2	6.1	10.70	15.2	15.6
CR(%)	-1.3	-5	3.4	-5	18.2	22.2	23.5

На перший погляд чітко видно, що всі алгоритми на основі пошуку перевершують формувати стандартні рівні GCC з мінімальним покращенням на 10% для використання пам’яті та 18% для процесорного часу (при застосуванні RS).

Запропонований підхід NS має найкращі результати покращення за трьома показниками: 1,365 прискорення, 15,6% зменшення пам’яті та 23,5% скорочення процесорного часу в усіх тестованих програмах. NS явно кращий за GA з точки зору прискорення. Однак для MR і CR NS трохи кращий, ніж GA: 0,4% покращення для MR і 1,3% для CR. RS має майже найнижчу ефективність оптимізації, але все одно краща за стандартні рівні GCC.

Застосування стандартної оптимізації впливає на час виконання з прискоренням 1,107 для O2 і O3. Ofast має такий самий вплив, як O2 і O3 на швидкість виконання. Однак вплив рівнів GCC на споживання ресурсів не завжди є ефективним. O2, наприклад, збільшує споживання ресурсів порівняно з O0 (-8,4% для MR і -5% для CR).

Це можна пояснити тим, що стандартні рівні GCC застосовують деякі агресивні оптимізації, які підвищують продуктивність згенерованого коду та погіршують системні ресурси.

4.2.2.2 Чутливість

Іншим цікавим експериментом є перевірка чутливості вхідних програм до оптимізацій компілятора та оцінка загальної застосовності найкращих оптимальних наборів оптимізації. Ці послідовності відповідають найкраще згенерованим послідовностям з використанням NS для трьох нефункціональних властивостей S, MR і CR (тобто послідовностей, отриманих у стовпці 8 табл. 4.4).

Таким чином, застосовуються найкращі виявлені оптимізації в попередньому розділі до нового невідомого Csmith (100 нових випадкових програм), а потім порівнюємо покращення продуктивності в цих програмах (див. правий бік рисунка). 4.4). Також застосовуються стандартні оптимізації, O2 і O3, до нових програм Csmith, щоб порівняти результати продуктивності. Ідея цього експерименту полягає в тому, щоб перевірити, чи чутливі нові згенеровані програми Csmith до раніше виявлених оптимізацій.

Користувачам та дослідникам компіляторів буде корисно використовувати CAT для побудови загальних послідовностей оптимізації з їхніх репрезентативних навчальних програм для кожної нефункціональної властивості: прискорення (S), пам’ять (MR) і ЦП (CR) і для кожної стратегії оптимізації: O2, O3 і NS.

На рисунку 4.5 показано розподіл пам’яті, процесора та поліпшення прискорення в 100 нових програмах Csmith. Для кожної нефункціональної

властивості застосовуються послідовності O2, O3 і найкращі NS. Результати прискорення показують, що три стратегії оптимізації призводять до майже однакового розподілу з середнім значенням 1,12 для прискорення. Це можна пояснити тим, що NS може знадобитися більше часу, щоб знайти послідовність, яка найкраще оптимізує швидкість виконання. Тим часом O2 і O3 також мають однаковий вплив на CR і MR, який є майже однаковим для обох рівнів (середнє значення CR становить 8% і близько 5% для MR).

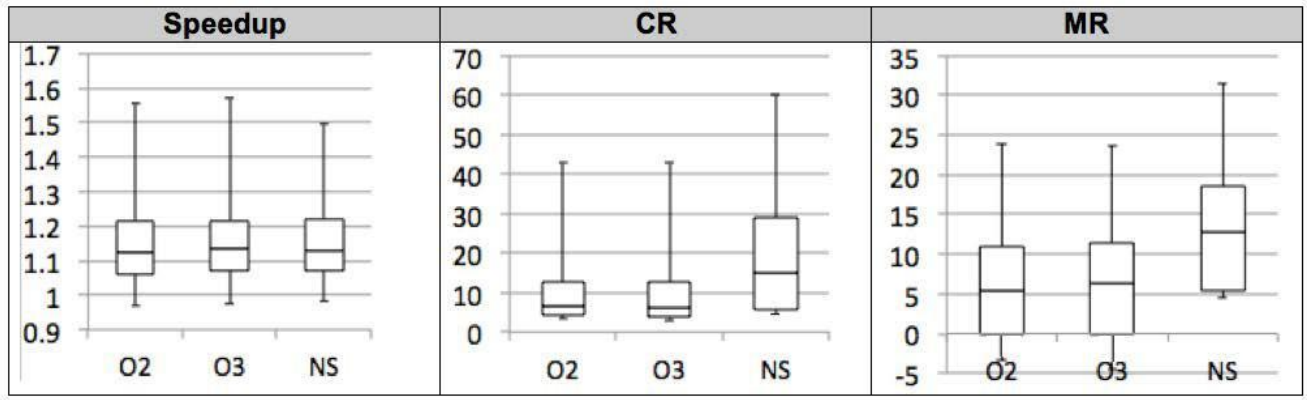


Рисунок 4.5 – Розподіл пам’яті, процесора та поліпшення прискорення в 100 нових програмах Csmith

Однак вплив застосування найкраще згенерованих послідовностей за допомогою NS явно перевершує O2 і O3 з майже 10% покращенням ЦП і 7% покращенням пам’яті. Це доводить, що послідовності NS є ефективними та можуть бути використані для оптимізації споживання ресурсів новими програмами Csmith. Цей результат також доводить, що генератор коду Csmith застосовує ті самі правила та структури для створення коду C. З цієї причини застосовані послідовності оптимізації завжди позитивно впливають на нефункціональні властивості.

4.2.2.3 Вплив оптимізації на використання ресурсів

У цьому експерименті оцінюється вплив застосування стандартних рівнів оптимізації та нових виявлених послідовностей на використання ресурсів. Також вивчається кореляція між прискоренням і споживанням ресурсів згенерованого коду. Ідея цього експерименту полягає в тому, щоб:

довести чи ні корисність залучення показників використання ресурсів як ключових цілей для покращення продуктивності;

довести чи ні необхідність багатоцільової стратегії пошуку для обробки різних нефункціональних вимог, таких як використання ресурсів і продуктивність.

Далі описано два методи проведення експериментів. Перший базується на програмах Csmith, а другий – на програмах Cbench.

Спосіб 1. У цьому експерименті CAT використовується, щоб забезпечити розуміння впливу оптимізації з точки зору споживання ресурсів під час спроби оптимізувати час виконання. Таким чином, обирається один екземпляр отриманих результатів у 1 експерименті, пов’язаних із найкращим покращенням прискорення (тобто результати, отримані в рядку 1 табл. 4.4), і вивчається вплив покращення прискорення на споживання пам’яті та ЦП. Також порівнюються дані про використання ресурсів із стандартними рівнями GCC, як вони представлені в таблиці 4.4 . Покращення завжди розраховуються щодо неоптимізованої версії (O0). Наступні вимірювання базуються на навчальному наборі з 10 програм Csmith.

Результати 1. Рис 4.6 показує вплив оптимізації прискорення на споживання ресурсів. Наприклад, O2 і O3, які призвели до найкращого покращення прискорення серед стандартних рівнів оптимізації в 1 експерименті, мають протилежний вплив на використання ресурсів. Застосування O2 викликає -8,4% MR і -5% CR. Однак застосування O3 покращує MR і CR відповідно на 3,4% і 4,2%. Під час застосування стандартних рівнів немає чіткої кореляції між прискоренням і використанням ресурсів, оскільки оптимізація компілятора зазвичай використовується для оптимізації швидкості виконання та ніколи не оцінюється для зменшення системних ресурсів.

З іншого боку, результати застосування різних одноцільових алгоритмів для оптимізації прискорення також доводять, що споживання ресурсів завжди залежить від швидкості виконання. Найвищий MR і CR досягається при використанні NS з відповідно 1,2% і 5,4%. Це покращення є значно низьким порівняно з показниками, досягнутими, коли було застосовано показники використання ресурсів як ключові цілі в gthijve trcgthbvtyns (тобто 15,6% для MR і 23,5% для CR). Крім того, відзначається, що згенеровані послідовності з використанням RS і GA мають значний вплив на системні ресурси, оскільки всі значення використання ресурсів гірші за базові.

Ці результати узгоджуються з ідеєю, що оптимізація компілятора не робить надто великого акценту на обміні між часом виконання та споживанням ресурсів.

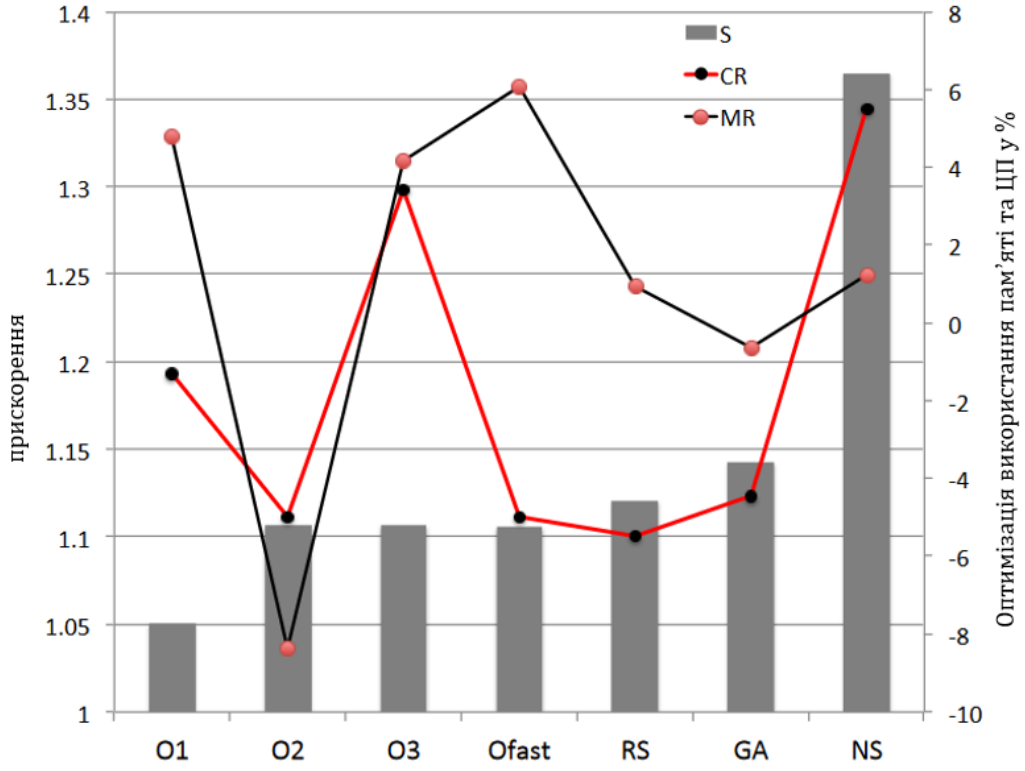


Рисунок 4.6 – Вплив покращення прискорення на споживання пам'яті та ЦП для кожної стратегії оптимізації

Спосіб 2. Тепер вивчається вплив застосування стандартних рівнів (01, O2, O3, Ofast) на використання пам'яті 5 різними програмами Cbench. Порівнюємо

результати з рішеннями, створеними за допомогою CAT, які мають найкраще скорочення споживання пам’яті (тобто створені NS). рисунок 4.7 показує це порівняння в різних програмах тестування. Він представляє відсоток збереженої пам’яті за допомогою стандартної та нової оптимізації понад рівень O0 (оптимізація не застосовується).

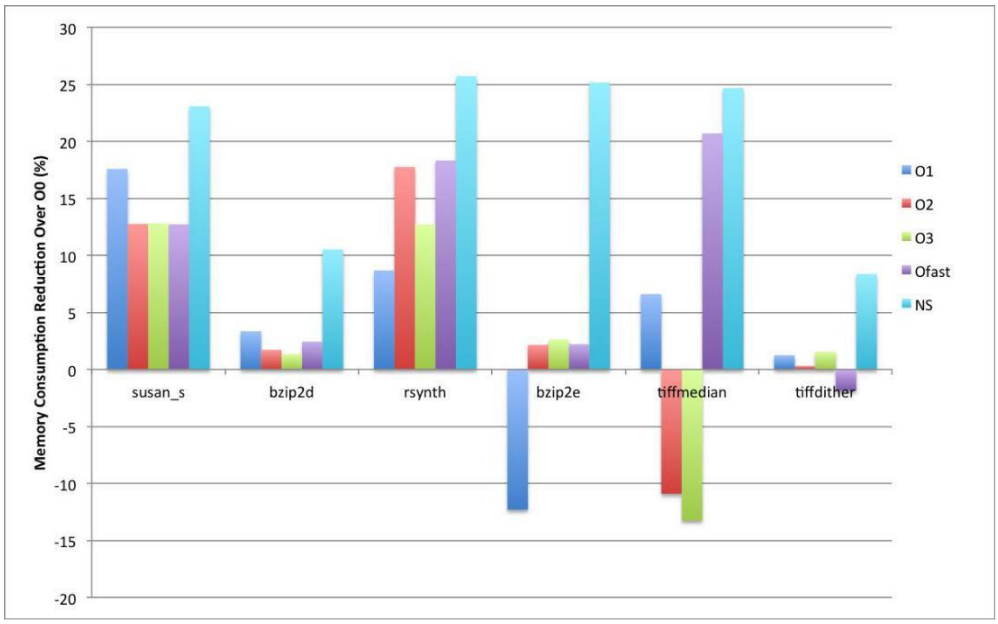


Рисунок 4.7 – Оцінка обсягу збереженої пам’яті після застосування стандартних параметрів оптимізації порівняно з найкращою згенерованою оптимізацією за допомогою NS

Результати 2. Як і очікувалося, результати показують, що NS явно перевершує стандартні оптимізації для всіх програм тестування. Використовуючи NS, можна досягти максимального зменшення споживання пам’яті майже на 26% для програми case rsynth проти максимального зменшення на 18% за допомогою параметра Ofast. Вплив застосування стандартної оптимізації на споживання пам’яті для кожної програми відрізняється від однієї програми до іншої. Використання O1 для bzip2e та O2, O3 для медіани ti збільшує споживання пам’яті майже на 13 %. Це узгоджується з ідеєю, що стандартні оптимізації не завжди дають однакові результати впливу на споживання ресурсів і можуть сильно залежати від еталонного тесту та вихідного коду, на якому вони були протестовані.

4.2.2.4 Торгівля між нефункціональними властивостями

У цьому експерименті увагу зосереджено на trade-o <ExecutionTime{ MemoryUsage>. На додаток до пропонованої адаптації NS для багатоцільової оптимізації, реалізовано широко використовуваний багатоцільовий підхід, а саме NSGA-II [119]. Позначимо пропоновану адаптацію NS через NS-II. NS-II не є багатоцільовим підходом, як NSGA-II. Він використовує той самий алгоритм NS. Однак у цьому експерименті він повертає оптимальні передні рішення Парето замість того, щоб повертати одне оптимальне рішення відносно однієї цілі. Крім того, застосовуються різні стратегії оптимізації для оцінки пропонованого підходу. По-перше, застосовано стандартні рівні GCC. По-друге, застосовуються найкраще згенеровані послідовності щодо пам’яті та оптимізації прискорення (ті самі послідовності, які використовувались в другому експертменті). Таким чином, позначаємо NS-MR послідовність, яка дає найкраще покращення пам’яті MR, а NS-S – послідовність, яка веде до найкращого прискорення. Це корисно для порівняння одноцільових рішень із новими згенерованими. У цьому експерименті оцінюється ефективність згенерованих послідовностей за допомогою лише однієї програми Csmith. Оцінюється якість отриманої оптимізації за Парето на основі необроблених даних значень пам'яті та часу виконання. Потім порівнюються результати шляхом візуального огляду передніх ярусів Парето. Мета цього експерименту – перевірити, чи існує послідовність, яка може зменшити як час виконання, так і використання пам’яті.

Результати. Рис 4.8 показує оптимальні рішення за Парето, які досягли найкращої оцінки продуктивності для trade-o <ExecutionTime{MemoryUsage>. Горизонтальна вісь вказує на використання пам’яті необробленими даними (у байтах), коли їх збирають за допомогою CAT. Подібним чином вертикальна вісь показує час виконання в секундах. Крім того, рисунок показує вплив застосування стандартних параметрів GCC і найкращих послідовностей NS на пам’ять і час виконання.

На основі цих результатів видно, що NSGA-II працює краще, ніж NS-II. Насправді NSGA-II поступається найкращому набору рішень, які представляють оптимальну торгівлю між дві цілі. Потім користувач компілятора має використовувати одне рішення з цього фронту Парето, яке задовольняє його нефункціональні вимоги (шість рішень для NSGA-II і ve для NS-II). Наприклад, він може вибрати одне рішення, яке максимізує швидкість виконання на користь зменшення пам’яті. З іншого боку, NS-II здатний генерувати лише одне недомінований рішення. Для NS-MR це, як і очікувалося, зменшує споживання пам’яті порівняно з іншими рівнями оптимізації. Такий самий вплив спостерігається на час виконання при застосуванні найкращої послідовності прискорення NS-S. Також зазначається, що на всіх стандартних рівнях GCC домінують NS-II, NSGA-II, NS-S і NS-MR.

Це узгоджується з твердженням про те, що стандартні рівні компілятора не представляють відповідного обміну між часом виконання та використанням пам’яті.

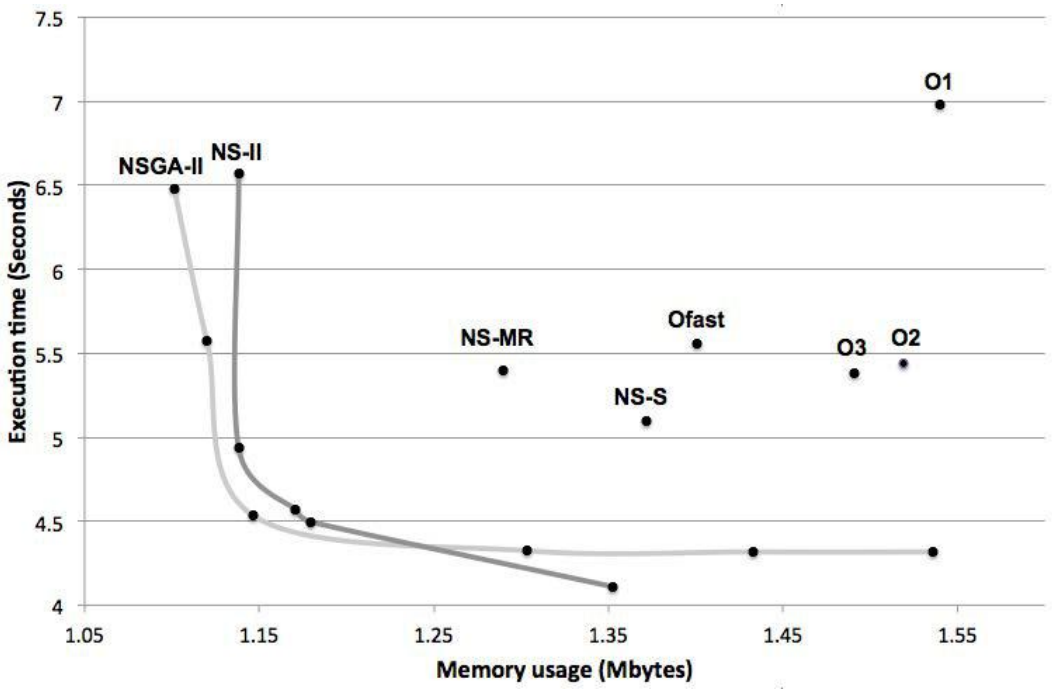


Рисунок 4.8 – Результати порівняння отриманих фронтів Парето за допомогою NSGA-II та NS-II

4.2.3 Обговорення

За допомогою цих експериментів показано, що CAT може надати користувачам компілятора засоби для тестування нефункціональних властивостей згенерованого коду. Він забезпечує підтримку для пошуку найкращих послідовностей оптимізації за допомогою однооб’єктивних і багатооб’єктивних алгоритмів пошуку. CAT. Інфраструктура продемонструвала свою здатність і масштабованість, щоб задовольнити вимоги користувачів і ключові цілі для створення ефективного коду з точки зору нефункціональних властивостей.

Під час усіх експериментів стандартні рівні оптимізації були значно перевершені різними евристиками. Крім того, також показано, що оптимізація продуктивності в деяких випадках може бути ненажерливою з точки зору використання ресурсів. Наприклад, вплив стандартних рівнів оптимізації на використання ресурсів не завжди ефективний, навіть якщо він призводить до підвищення продуктивності. Таким чином, користувачі компілятора можуть використовувати CAT, щоб оцінити вплив оптимізації на нефункціональні властивості та побудувати свої конкретні послідовності, намагаючись знайти торгові пропозиції серед цих нефункціональних властивостей ().

4.2.4 Можливі проблеми

Будь-який автоматизований підхід має обмеження. У наступних розділах описано зовнішні та внутрішні проблеми, які можуть виникнути.

Зовнішня валідність означає можливість узагальнення висновків. У цьому дослідженні проведено експерименти з випадковими програмами за допомогою Csmith і використовуємо методи ітераційної компіляції для створення найкращих послідовностей оптимізації. Використання програм Csmith як програм введення є дуже актуальним, оскільки компілятори були широко протестовані в програмах Csmith [120, 28]. Програми Csmith використовувалися лише для функціонального

тестування, але не для нефункціонального тестування. Однак не можна стверджувати, що найкраще знайдений набір оптимізацій можна узагальнити для промислових застосувань, оскільки оптимізація сильно залежить від вхідних програм і цільової архітектури.

Внутрішня валідність стосується причинно-наслідкового зв'язку між лікуванням і результатом. Метаевристичні алгоритми є стохастичними оптимізаторами, вони можуть надавати різні результати для того самого екземпляра проблеми від одного запуску до іншого. Надано статистично обґрунтований метод чи це просто випадковий результат? Через обмеження часу всі експерименти проведено лише один раз. Дотримуючись найсучасніших підходів до ітеративної компіляції, попередні дослідницькі зусилля [66, 72] не надали статистичних тестів, щоб підтвердити ефективність своїх підходів. Це тому, що експерименти займають багато часу. Однак можна впоратися з цими внутрішніми загрозами валідності, виконавши принаймні п'ять незалежних прогонів моделювання для кожного екземпляра проблеми.

4.2.5 Огляд підтримки інструментів

CAT також надає інтерфейс GUI. Мета підтримки цього інструменту – допомогти користувачам легко використовувати CAT і автоматично налаштовувати компілятори GCC. Цей інструмент використовувався для відповіді на всі попередні запитання дослідження.

Як показано на рисунку 4.9, CAT надає різні функції, щоб допомогти користувачам компілятора:

Вибір тестової програми введення: згенерувавши нову програму Csmith або вибравши наявну програму C, наприклад, програми Cbench. Генерація нової програми Csmith виконується випадковим чином.

Вибір набориів даних: якщо для вибраної програми потрібен набір даних, наприклад, для програм Cbench, CAT дозволяє користувачеві вибрати набір даних

для вибраної програми. Нагадаємо, що Cbench поставляється з набором із 20 наборів даних для кожної програми benchamrk.

Вибір архітектури цільового комп'ютера: виберіть архітектуру процесора, на якому будуть проводитися експерименти, наприклад x64, x86, ARM. У описаних експериментах використовується власний компілятор GCC хост-машини з архітектурою x64.

Визначення версії компілятора: на даний момент CAT підтримує всі версії компілятора GCC від 3.x до 5.x. Процес вилучення цільових оптимізацій для розвитку виконується автоматично

Налаштування компонентів моніторингу: це стосується контейнерів, необхідних для отримання всієї інформації про споживання ресурсів. Конфігурація таких компонентів, як версії зображень, мітки, порти, логіни, паролі, можливі за допомогою CAT.

Вибір ip-адреси хмарного хост-комп'ютера: CAT дозволяє проводити експерименти віддалено завдяки інфраструктурі мікросервісів. Таким чином користувач може вибрати IP-адресу віддаленої машини.

Визначення обмеження ресурсів для запущеного контейнера: якщо запускати оптимізацію в умовах обмежень ресурсів, можна визначити обмеження пам'яті та ЦП. За замовчуванням ці параметри вимкнено.

Вибір методу пошуку: користувач може вибрати однооб'єктивний або багатооб'єктивний пошук.

Вибір метаевристичний алгоритмів: CAT підтримує GA, RS і NS для монооб'єктивного пошуку та NS, RS і NSGA-II для багатооб'єктивної оптимізації.

Вибір кількості ітерацій: користувач може визначити певну кількість ітерацій для кожного алгоритму, яка відповідає кількості згенерованих послідовностей оптимізації.

Вибір часу пошуку: замість обмеження кількості ітерацій користувач може обмежити час налаштування (у годинах).

Вибір цілі налаштування: метою може бути зменшення часу виконання, пам’яті, ЦП, розміру коду або часу компіляції. Для пошуку з кількома цілями користувачі можуть вибрати обмін між цими цілями.

Редагування параметрів еволюційного алгоритму: Налаштування еволюційних параметрів (показано в табл. 4.3), такі як розмір популяції, параметри пошуку новизни, ймовірності мутації та кросинговеру тощо.

Наприкінці виведення на консоль (тобто результат виконання цього інструменту) відображає результати порівняння стандартних рівнів оптимізації з новими виявленими рішеннями.

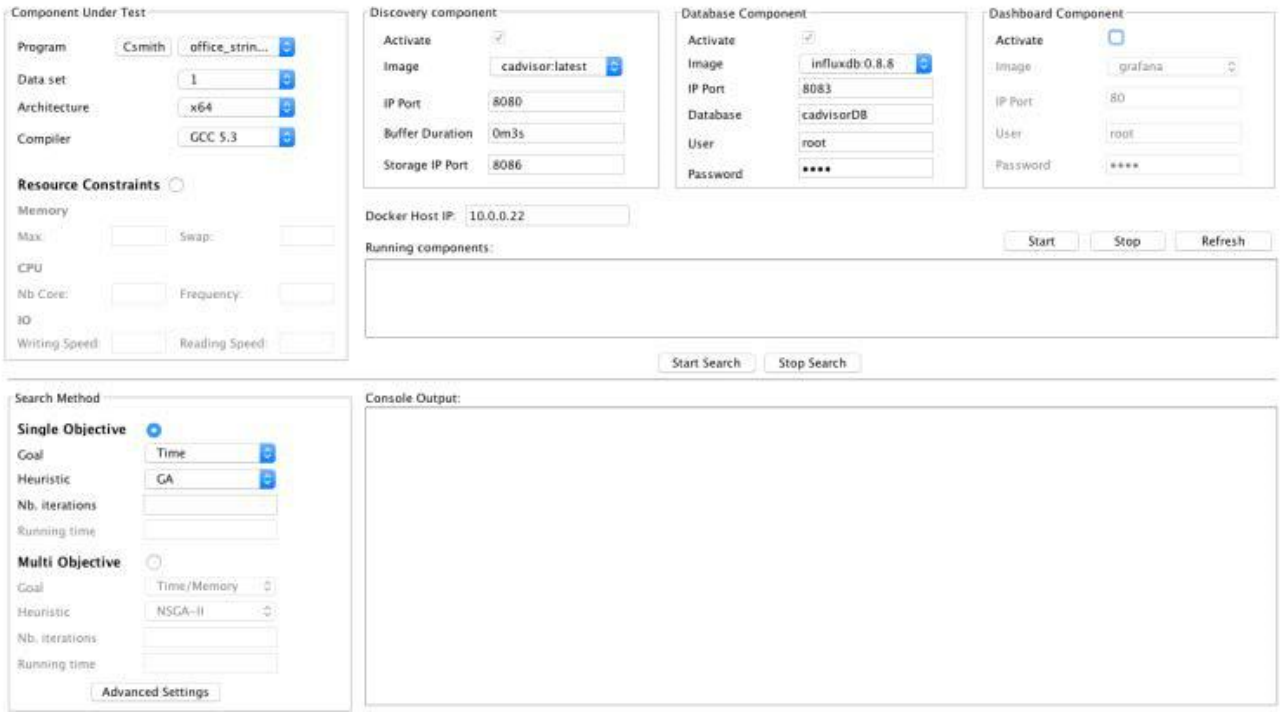


Рисунок 4.9 – Знімок інтерфейсу CAT GUI

4.3 Висновок

Сучасні компілятори мають величезну кількість оптимізацій, що ускладнює для користувачів компілятора пошук найкращих послідовностей оптимізації. Крім того, автоматичне налаштування компіляторів для задоволення вимог користувача є складним завданням, оскільки оптимізація може залежати від різних властивостей

(наприклад, архітектури платформи, програмного забезпечення, цільового компілятора, цілі оптимізації тощо). Отже, користувачі компілятора просто використовують стандартні рівні оптимізації (O1, O2, O3 і Ofast), щоб покращити якість коду, не надто піклуючись про вплив оптимізації на системні ресурси.

У цьому розділі представлено нове формулювання проблеми оптимізації компілятора на основі пошуку новизни. Ідея цього підходу полягає в тому, щоб спонукати пошук найкращих оптимізацій до новизни. Експерименти показали, що пошук новизни можна легко застосувати до задач моно- та багатоцільового пошуку. Отримані результати емпіричного дослідження пропонованого підходу в порівнянні з різними сучасними підходами надали докази на підтримку твердження про те, що Novelty Search здатний генерувати ефективну оптимізацію. По-друге, було представлено автоматизований підхід для автоматичного вилучення нефункціональних властивостей оптимізованого коду під назвою CAT. CAT застосовує різні евристики (включно з пошуком новизни) і виконує автоматичне налаштування компілятора шляхом моніторингу згенерованого коду в контрольованому середовищі ізольованого програмного середовища. Насправді CAT використовує набір мікросервісів, щоб забезпечити чітке розуміння впливу оптимізації на споживання ресурсів. Проведено оцінку ефективності пропонованого підходу, перевіривши оптимізацію, виконану компілятором GCC. Потім було вивчено вплив оптимізації на споживання пам’яті та час виконання. Результати показали, що пропонований підхід здатний автоматично отримувати інформацію про споживання пам’яті та ЦП. Також було знайдено кращі послідовності оптимізації, ніж стандартні рівні оптимізації GCC, і побудувати оптимізацію, яка представляє оптимальний обмін між прискоренням і використанням пам’яті.

5 ТЕСТУВАННЯ АВТОМАТИЧНИХ ГЕНЕРАТОРІВ КОДУ

Різноманітність програмних платформ і неоднорідність апаратного забезпечення, як обговорювалося раніше, є основною перешкодою для тестування програмного забезпечення. Насправді виконання тестів потребує багатьох конфігурацій та налаштувань середовища, щоб перевірити всю програму. Наприклад, тестування веб-програми вимагає встановлення залежностей Maven, веб-сервера, бібліотек тощо. Наприклад, коли розробники програмного забезпечення оновлюють версію веб-сервера, їм потрібно перебудувати програму та запустити ті самі інтеграційні тести, щоб перевірити, що жодних помилок не включено. Таким чином, тестування програм з використанням різних середовищ виконання та системних налаштувань стає дуже трудомістким і виснажливим.

Наприклад, щоб оцінити автоматично згенерований код (генераторами коду або компіляторами), використовуютьсяйого компіляції та запуску тестових випадків. Для цього потрібні були різні конфігурації системи, щоб забезпечити виконання цих кроків, наприклад встановлення версії генератора (версії GCC або Haxe), встановлення інтерпретаторів, компіляторів, залежностей Maven тощо.

Одним із способів тестування цих настроюваних генераторів є використання технології віртуалізації. Наприклад, альтернативний метод використовує віртуалізацію системи на основі контейнерів (наприклад, Docker, як описано в розділі 2.3), щоб автоматизувати генерацію коду, розгортання та тестування в попередньо налаштованих контейнерах програмного забезпечення. Ця технологія дає змогу імітувати параметри середовища виконання та відтворювати тести в ізольованих системних контейнерах із високою конфігурацією.

Коли справа доходить до оцінки споживання ресурсів автоматично згенерованого коду, ця технологія стає дуже цінною, оскільки вона дозволяє детально керувати ресурсами та ізолювати їх. Крім того, це полегшує вилучення ресурсів і обмеження програм, що працюють всередині контейнерів.

У цьому розділі представлено технічний опис цього легкого середовища виконання та його переваги для автоматизації нефункціонального тестування

згенерованого коду. Ця інфраструктура використовується в наших двох перших внесках як засіб для запуску тестів у настроюваному середовищі виконання та для ефективного збору показників споживання ресурсів.

5.1 Системні контейнери як полегшене середовище виконання

Системні контейнери – це метод віртуалізації на рівні операційної системи, який дозволяє запускати кілька ізольованих систем Linux на керуючому хості за допомогою одного ядра Linux. Контейнери використовують ту саму ОС і апаратне забезпечення, що й хост-машина, і дуже корисно використовувати їх для створення нових конфігурованих та ізольованих екземплярів для запуску. Віртуалізація на основі контейнерів зменшує накладні витрати, пов’язані з тим, що кожна гостьова система запускає нову встановлену операційну систему, як у випадку з віртуальними машинами. Цей підхід також може підвищити продуктивність, оскільки існує лише одна операційна система, яка піклується про апаратні виклики. Ядро Linux надає контрольні групи (Cgroups) функціональні можливості, які дозволяють обмежувати та пріоритезувати ресурси (ЦП, пам’ять, блоки вводу-виводу, мережу тощо) всередині контейнерів, щоб один контейнер не обмежував ресурси інших.

Наприклад, Docker це популярна контейнерна технологія, яка автоматизує розгортання будь-якої програми як легкого, портативного та автономного контейнера, що працює практично на головній машині [86]. Сьогодні Docker є однією з найпопулярніших інфраструктурних технологій, які застосовуються в хмарних обчисленнях [95]. Наприклад, у 2015 році Docker мав близько 3% частки ринку, а до 2017 року він працює на 15% хостів. Використовуючи Docker, можна визначити попередньо налаштовані програми та сервери для розміщення як віртуальні образи. Він також визначає спосіб розгортання служби на головній машині за допомогою файлів конфігурації, які називаються файлами Docker. Крім того, можна ввімкнути деякі параметри конфігурації для контролю та обмеження

ресурсів. Наприклад, можна надати параметри ags, щоб обмежити обсяг пам’яті чи використання ЦП кожній службі, пов’язати ядра ЦП з кожною службою тощо.

Нижче описано, основні переваги мікросервісів.

Використання контейнерів зменшує накладні витрати на продуктивність порівняно з використанням повного стекового рішення віртуалізації [85]. Дійсно, засоби вимірювання та моніторингу для профілювання пам’яті, такі як Valgrind [106], можуть спричинити надто великі витрати.

Завдяки використанню файлів Docker можна легко налаштувати середовище виконання, щоб створювати та налаштовувати програми за допомогою численних налаштувань (наприклад, версія генератора, залежності, IP-адреса хоста та ОС, параметри оптимізації тощо). Таким чином, можна використовувати той самий налаштований образ Docker для запуску різних екземплярів однієї програми.

Docker використовує групи керування Linux (Cgroups) для групування процесів, що виконуються в контейнері. Це дозволяє нам керувати ресурсами групи процесів, що дуже цінно. Цей підхід збільшує гнучкість, коли потрібно керувати ресурсами, оскільки можна керувати кожною групою окремо. Наприклад, якщо нефункціональні вимоги згенерованого коду будуть оцінюватись в середовищі з обмеженими ресурсами, можна запросити та обмежити ресурси всередині контейнера виконання відповідно до потреб.

Хоча контейнери працюють ізольовано, вони можуть обмінюватися даними з хост-машиною та іншими запущеними контейнерами. Таким чином, нефункціональні дані щодо споживання ресурсів можна збирати та керувати іншими контейнерами (тобто з метою зберігання, візуалізації)

5.2 Механізм моніторингу виконання

Щоб відстежувати програми (тобто тести), що виконуються всередині контейнерів, використовується набір компонентів Docker, щоб полегшити отримання інформації про використання ресурсів.

5.2.1 Контейнер моніторингу

По-перше, компонент моніторингу необхідний для збору характеристик використання ресурсів і продуктивності запущених контейнерів. Як обговорювалося раніше, Docker покладається на файлові системи Cgroups для показу багатьох показників щодо накопичених циклів ЦП, пам’яті, використання блокового вводу-виводу тощо. Таким чином, наш компонент моніторингу автоматизує вилучення цих показників продуктивності під час виконання, що зберігаються у файлах Cgroups. . Серед популярних способів зробити це – моніторинг кожного контейнера через Docker API або встановлення агента для детальної видимості всередині кожного контейнера. Клієнт Docker уже надає інструмент командного рядка для перевірки споживання ресурсів контейнерами. Команда docker stats, наприклад, може бути використана для отримання статистики про запущені контейнери під час виконання. Якщо потрібно зробити це вручну, можна отримати доступ до поточного споживання ресурсів кожного контейнера, доступного в файловій системі Cgroups, через статистику, знайдену в «/sys/fs/cgroup/cpu/docker/(longid)/» (для споживання ЦП) та «/sys/fs/cgroup/memory/docker/(longid)/» (для статистики, пов’язаної зі споживанням пам’яті). Наш компонент моніторингу автоматизує процес виявлення послуг і агрегації показників для кожного нового контейнера. Таким чином, замість того, щоб вручну збирати метрики, розташовані у файлових системах Cgroups, він автоматично витягує статистику використання ресурсів під час виконання щодо запущеного компонента (тобто виконаного набору тестів у контейнері). Зверніть увагу, що інформація про використання ресурсів збирається у вигляді необроблених даних. Цей процес може спричинити невеликі накладні витрати, оскільки він виконує дуже тонкий облік використання ресурсів під час роботи контейнера. На щастя, це може не вплинути на зібрані дані, оскільки запускається лише один набір тестів або програму в кожному контейнері. Щоб полегшити процес моніторингу, використовується cAdvisor, Container Advisor. cAdvisor

відстежує контейнери служби під час виконання, як описано вище. Він широко використовувався в різних проектах, таких як Heapster і Google Cloud Platform.

Однак cAdvisor відстежує та агрегує поточні дані з інтервалом лише в 60 секунд. Таким чином, записуються всі дані протягом певного часу, починаючи з моменту створення контейнера, у базі даних часових рядів. Це дозволяє кінцевим користувачам виконувати запити та визначати нефункціональні показники з історичних даних. Таким чином, щоб зробити зібрані дані справді цінними для моніторингу використання ресурсів, компонент моніторингу звязано з внутрішнім контейнером бази даних.

5.2.2 Внутрішній контейнер бази даних

Цей компонент представляє серверну частину бази даних часових рядів. Він підключений до описаного раніше компонента моніторингу для збереження нефункціональних даних для довгострокового зберігання, аналітики та візуалізації. Під час виконання програми статистичні дані про використання ресурсів постійно надсилаються цьому компоненту. Коли контейнер вимикається, можна отримати доступ до його відносних показників використання ресурсів через базу даних. Обирається база даних часових рядів, оскільки збираються дані часових рядів, які відповідають параметрам використання ресурсів виконання програм.

Використано InfluxDB, розподілену базу даних часових рядів із відкритим вихідним кодом як серверну частину для запису даних. В InfluxDB користувач може виконувати SQL-подібні запити до бази даних. Наприклад, наступний запит повідомляє про максимальне використання пам’яті контейнером «generated_code v1» з моменту його створення:

```
select max (memory usage) from stats
where container name=’generated code v1’
```

Щоб дати уявлення про дані, зібрані компонентом моніторингу та збережені в базі даних часових рядів, описано в табл. 5.1 ці зібрані показники:

Таблиця 5.1 – Показники використання ресурсів, записані в InfluxDB

Метрика	Опис
Name	Назва контейнера
T	Час, що минув з моменту створення контейнера
Network	Статистика мережевих байтів і пакетів
Disk IO	Статистика дискового введення/виведення
Memory	Використання пам'яті
CPU	використання ЦП

Окрім цього, надаєтьсямя інформацію про розмір програми (наприклад, розмір згенерованих двійкових файлів) і час компіляції, необхідний для створення коду. Наприклад, статистика використання збирається та зберігається за допомогою описаних раніше компонентів. Це актуально, щоб показати профілі використання ресурсів запущених програм понаднормово. Для цього використано інтерфейсний контейнер візуалізації для підвищення рівня використання ресурсів.

5.2.3 Внутрішній контейнер візуалізації

Після збору та збереження даних про використання ресурсів, наступним кроком є їх візуалізація. Це роль контейнера візуалізації. Це буде компонент кінцевої точки, який використовується для візуалізації записаних даних. Тому створено інформаційну панель для виконання запитів і перегляду різних характеристик споживання ресурсів запущеними компонентами через веб-інтерфейс користувача. Таким чином, можна візуально порівняти показники споживання ресурсів між контейнерами. Крім того, можна використовувати цей компонент для експорту даних, які зараз переглядаються, у статичний документ CSV. Отже, можна виконати статистичний аналіз цих даних, щоб виявити невідповідності або аномалії продуктивності. Як компонент візуалізації використовується Grafana, інструмент візуалізації часових рядів, доступний для

Docker. Grafana дозволяє нам відображати результати в реальному часі за часом у дуже гарних графіках. Так само, як і в InfluxDB, використано запити SQL для вилучення нефункціональних даних із бази даних для візуалізації та аналізу.

5.3 Приклад генератора

У цьому розділі представлено адаптацію цієї мікросервісної інфраструктури до прикладу генератора, як застосовано в розділах 3 і 4. Контейнери вже використовувались як засоби для запуску різних варіантів оптимізованого коду в розділі 4, а також для запуску стенда тестових наборів на різних програмних платформах у розділі 3 .Огляд мікросервісу та технічних рішень, застосованих для прикладу генератора, показано на малюнку 5.1. Далі детально описано налаштування інфраструктури. Експериментальний матеріал також доступний онлайн.

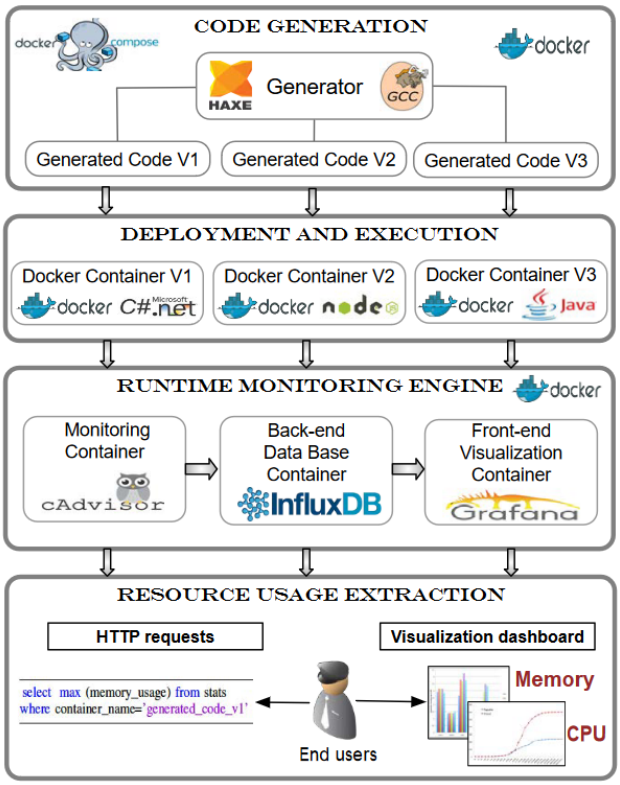


Рисунок 5.1 – Контейнерна інфраструктура для тестування автоматичних генераторів

Генерація коду Перш ніж почати моніторинг і тестування програм, потрібно розгорнути згенерований код (компіляторами або генераторами коду) у різних контейнерах Docker. Таким чином, замість конфігурації всіх тестованих генераторів (GUT) на одній головній машині розгорнуто кожен GUT в контейнері. Для цього створюється новий образ конфігурації для кожного GUT (тобто образ Docker), де встановлюються всі бібліотеки, компілятори та залежності, необхідні для забезпечення генерації та компіляції коду. Таким чином, GUT створює код у кількох екземплярах попередньо налаштованих зображень Docker. Отже, файли Docker використовуються для налаштування всіх цих параметрів. Публічний реєстр Docker (хмара-служба реєстру) використовується, щоб зберігати та керувати всіма нашими образами Docker. Після створення коду згенеровані вихідні файли зберігаються в спільному репозиторії. У середовищі Docker це сховище називається томом даних. Це спеціально призначений каталог усередині контейнерів, який ділиться даними з головною машиною та всіма іншими запущеними контейнерами.

Розгортання та виконання Далі згенерований код (у томі даних) виконується окремо всередині ізольованого контейнера Docker. Таким чином гарантується, що кожна виконана програма працює ізольовано, не піддаючись впливу головної машини чи будь-яких інших процесів. Крім того, оскільки створити контейнер дешево, можна створити занадто багато контейнерів, якщо у нас є нові програми для виконання (наприклад, новий оптимізований код, набір тестів для певної програмної платформи тощо). Оскільки кожне виконання програми вимагає створення нового контейнера, надзвичайно важливо видалити та знищити контейнери, які завершили свою роботу, щоб усунути навантаження на систему. Експеримент запускається у приватному центрі обробки даних, який забезпечує установку Docker на «голому залізі». На одній машині контейнери працюють послідовно. Після завершення виконання ресурси, зарезервовані для контейнера, автоматично звільняються для створення наступних контейнерів. Таким чином, хост-машина не надто постраждає від зміни продуктивності.

Моніторинг виконання. Під час роботи контейнерів запускаються три описані вище контейнери моніторингу, щоб контролювати запущене робоче навантаження. Для цього використовується Docker Compose щоб запустити всі контейнери одночасно. Концепція Docker Compose подібна до файлів Docker. Він використовує файл конфігурації для запуску та зв'язування багатоконтейнерних служб Docker. Файл Compose налаштовано, щоб запускати всі служби, зокрема, щоб зіставляти запущені контейнери з cAdvisor і InfluxDB, використовуючи порти Docker і мережеві посилання для потокової передачі даних про використання ресурсів комп'ютера.

Вилучення інформації про використання ресурсів. Кінцеві користувачі мають два способи отримати інформацію про використання ресурсів створеного коду. Можна напряму запитувати віддалену базу даних часових рядів через HTTP-запити, виконуючи SQL-подібні запити, як у прикладі, представленому в розділі 5.2.2. Вони можуть запитувати різні показники, такі як ЦП, використання пам'яті, швидкість запису на диск тощо. Альтернативним рішенням є візуалізація споживання ресурсів згенерованим кодом на веб-панелі, наданій Grafana. Інструмент візуалізації не використовується під час автоматичного налаштування та тестування генераторів, тому що нам просто потрібно було витягти дані про використання ЦП або пам'яті для кожного тесту чи оптимізованого коду.

Використовується те саме апаратне забезпечення для всіх експериментів у розділах 3 і 4.

Обмеження. Ця інфраструктура тестування може бути узагальнена та адаптована до інших прикладів, окрім генераторів. Використовуючи системні контейнери, будь-який програмний додаток/згенерований код можна легко виконувати та контролювати за допомогою Docker. Однак серед обмежень цієї інфраструктури мікросервісів є те, що для роботи контейнерів потрібне ядро Linux. Для запуску механізму Docker у macOS або Windows потрібна віртуальна машина для розгортання невеликої ОС на базі Linux із попередньо встановленим Docker. Збір даних про використання ресурсів цих контейнерів Windows, наприклад, може спричинити накладні витрати через використання Docker у віртуальних машинах,

що може вплинути на точність зібраних даних про використання ресурсів. Крім того, ізоляція ресурсів у Docker має деякі обмеження, особливо для метрик дискового вводу-виводу. Нещодавно було доведено [107], що рівень ізоляції не ізолює належним чином спільні ресурси під час виконання робочих навантажень із інтенсивним використанням диска всередині контейнерів, що спричиняє деякі перешкоди продуктивності.

5.4 Висновок

У цьому розділі представлено технічні деталі інфраструктури, яка використовується для збору нефункціональних показників (наприклад, споживання пам’яті та ЦП) автоматично створеного коду (або компіляторами, або генераторами коду). Це рішення забезпечує ефективну підтримку автоматичного розгортання, виконання та тестування згенерованого коду в різних налаштуваннях середовища. Та сама інфраструктура моніторингу використовується для оцінки якості згенерованого коду в двох перших статтях цієї дисертації. Експерименти, проведені в розділах 3 і 4 показали корисність цієї інфраструктури для налаштування та тестування генераторів.

ВИСНОВКИ

Розробка генеративного програмного забезпечення проклала шлях до створення кількох генераторів, які служать основою для автоматичної генерації коду для широкого діапазону програмних і апаратних платформ. Завдяки повній автоматичній генерації коду користувачі можуть швидко синтезувати програмні артефакти для різних програмних платформ. Крім того, вони можуть легко налаштувати згенерований код для цільової апаратної платформи, оскільки сучасні генератори стають висококонфігурованими, пропонуючи численні параметри конфігурації, які користувач може застосувати. Якість автоматично створеного програмного забезпечення сильно залежить від параметрів конфігурації, а також від самого генератора

Оглядаючи сучасний рівень технологій, було визначено численні підходи до тестування генераторів. Деякі з них оцінюють нефункціональні властивості автоматично створеного коду, а саме характеристики продуктивності та використання ресурсів. Основною проблемою, яку було виявлено під час тестування нефункціональних властивостей, є проблема оракула, оскільки немає чіткого визначення того, як оракул може бути визначений, коли мова йде про тестування властивостей продуктивності та використання ресурсів.

З точки зору розробки програмного забезпечення, ця робота сприяє підвищенню якості та надійності генераторів. Зокрема, в ній запропоновано алгоритм, який допомагає розробникам/супроводжувачам генераторів ефективно тестувати створені ними генератори коду та таким чином надавати кінцевим користувачам докази якості згенерованого коду, також користувачам генераторів надано засоби для ефективного автоматичного налаштування вихідних генераторів для створення високоякісного коду.

Перший внесок представленої роботи стосується проблеми нефункціонального тестування генераторів. Зокрема, вирішено проблему оракула в області тестування генераторів коду. Запропоновано підхід для автоматичного виявлення невідповідностей у генераторах коду з точки зору нефункціональних

властивостей (тобто використання ресурсів і продуктивності). Представлений підхід базується на припущенні, що генератор коду часто є членом сімейства генераторів коду. Таким чином, використовується факт існування кількох генераторів із порівнянною функціональністю (тобто родини генераторів коду), щоб застосувати ідею метаморфічного тестування [108], визначаючи тестові оракули високого рівня (тобто метаморфічні зв'язки) як тестові оракули. Визначено метаморфічне відношення як порівняння між варіаціями продуктивності та використання ресурсів коду, створеного з однієї сім'ї генераторів коду. Будь-яка зміна, що перевищує певне порогове значення, автоматично визначається як аномалія. Застосовано два статистичні методи (тобто аналіз головних компонентів і діаграми діапазонів), щоб автоматизувати виявлення невідповідностей. Запропонований підхід оцінюється, аналізуючи продуктивність Haxe, популярної мови програмування високого рівня, яка включає набір кросплатформних генераторів коду. Оцінюються властивості, пов'язані з використанням ресурсів і продуктивністю для різних цільових програмних платформ. Результати досліджень показують, що запропонований підхід може автоматично виявляти реальні проблеми в сімействах генераторів коду.

Другий внесок стосується проблеми автоналаштування генераторів. Зокрема, стосується компіляторів з автонастроюванням через велику кількість параметрів конфігурації (тобто оптимізації), які вони пропонують для контролю якості згенерованого коду. У цьому контексті використано останні досягнення в розробці програмного забезпечення на основі пошуку, щоб забезпечити ефективний підхід до налаштування компіляторів (тобто за допомогою оптимізації) відповідно до нефункціональні вимоги користувача (наприклад, продуктивність і використання ресурсів). Запропонований підхід, який називається CAT, застосовує нове формулювання, порівняно з попередньою пов'язаною роботою, проблеми оптимізації компілятора з використанням алгоритму пошуку новизни [117]. Пошук за новинками використовується для вирішення проблеми різноманітності оптимізацій, а потім забезпечує новий спосіб дослідження величезного простору пошуку для оптимізації. Насправді, оскільки простір пошуку можливих комбінацій

є мультимодальним [77] і занадто великим, цю техніку застосовано, щоб дослідити простір пошуку можливих варіантів оптимізації компілятора, розглядаючи різноманітність послідовностей як єдину мету. Проведено емпіричне дослідження, щоб оцінити ефективність запропонованого підходу шляхом перевірки оптимізації, виконаної компілятором GCC. Представлені експериментальні результати показують, що CAT може автоматично налаштовувати компілятори відповідно до вибору користувача (евристики, цілі, програми тощо) і створювати оптимізації, які забезпечують кращі результати продуктивності, ніж стандартні рівні оптимізації та класичні генетичні алгоритми. Також продемонстровано, що CAT можна використовувати для автоматичної побудови рівнів оптимізації, які представляють оптимальну взаємодію між прискоренням і використанням пам’яті за допомогою багатоцільових алгоритмів.

Оцінка використання ресурсів автоматично створеного коду складна через різноманітність програмного та апаратного забезпечення, які існують на ринку. Щоб вирішити цю проблему, в третьому внеску представлено технічні деталі інфраструктури, що використовується для збору нефункціональних показників (наприклад, споживання пам’яті та ЦП) автоматично створеного коду (або компіляторами, або генераторами коду). Насправді використано нещодавніми досягненнями у віртуалізації легких систем, зокрема віртуалізації на основі контейнерів, щоб запропонувати ефективну підтримку автоматичного розгортання, виконання та моніторингу згенерованого коду в гетерогенному середовищі. Та сама інфраструктура моніторингу використовується для оцінки експериментів, проведених у двох перших дописах.

Представлений метод може бути використаний при розробці нових генераторів коду для виявлення помилок та їх усунення.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. M. Roper, “Computer aided software testing using genetic algorithms,” 10th International Quality Week, 1997.

2. A. L. Watkins, “The automatic generation of test data using genetic algorithms,” in Proceedings of the 4th Software Quality Conference, vol. 2, 1995, pp. 300–309.

3. W. Banzhaf, F. D. Francone, and P. Nordin, “The effect of extensive use of the mutation operator on generalization in genetic programming using sparse data sets,” in Parallel Problem Solving from NaturePPSN IV. Springer, 1996, pp. 300–309.

4. Tobias Betz, Lawrence Cabac, and Matthias Guttler. Improving the development tool chain in the context of petri net-based software development. In PNSE, pages 167-178. Citeseer, 2011.

5. Ahmad Nauman Ghazi, Kai Petersen, and J¨urgen B¨orstler. Heterogeneous systems testing techniques: An exploratory survey. In International Conference on Software Quality, pages 67–85. Springer, 2015

6. Mathieu Acher, Olivier Barais, Benoit Baudry, Arnaud Blouin, Johann Bourcier, Benoit Combemale, Jean-Marc J´ez´equel, and No¨el Plouzeau. Software diversity: Challenges to handle the imposed, opportunities to harness the chosen. In GDR GPL, 2014

7. Charalampos Doukas and Fabio Antonelli. Compose: Building smart &context-aware mobile applications utilizing iot technologies. In Global Information Infrastructure Symposium, 2013, pages 1–6. IEEE, 2013.

8. Simon Allier, Olivier Barais, Benoit Baudry, Johann Bourcier, Erwan Daubert, Franck Fleurey, Martin Monperrus, Hui Song, and Maxime Tricoire. Multitier diversification in web-based software applications. IEEE Software, 32(1):83–90, 2015.

9. Ina Schaefer, Rick Rabiser, Dave Clarke, Lorenzo Bettini, David Benavides, Goetz Botterweck, Animesh Pathak, Salvador Trujillo, and Karina Villela. Software diversity: state of the art and perspectives. International Journal on Software Tools for Technology Transfer, 14(5):477–495, 2012.

10. Benoit Baudry and Martin Monperrus. The multiple facets of software diversity: Recent developments in year 2000 and beyond. ACM Computing Surveys (CSUR), 48(1):16, 2015.

11. Liqiang He. Computer architecture education in multicore era: Is the time to change. In Computer Science and Information Technology (ICCSIT), 2010 3rd IEEE International Conference on, volume 9, pages 724–728.IEEE, 2010.

12. Dorit Nuzman, Ira Rosen, and Ayal Zaks. Auto-vectorization of interleaved data for simd. ACM SIGPLAN Notices, 41(6):132–143, 2006.

13. Qiming Hou, Kun Zhou, and Baining Guo. Spap: A programming language for heterogeneous many-core systems. Technical report, Zhejiang University Graphics and Parallel Systems Lab, 2010.

14. Hassan Chafi, Zach DeVito, Adriaan Moors, Tiark Rompf, Arvind K Sujeeth, Pat Hanrahan, Martin Odersky, and Kunle Olukotun. Language virtualization for heterogeneous parallel computing. In ACM Sigplan Notices, volume 45, pages 835–847. ACM, 2010.

15. Hassan Chafi, Arvind K Sujeeth, Kevin J Brown, HyoukJoong Lee, Anand R Atreya, and Kunle Olukotun. A domain-specific approach to heterogeneous parallelism. ACM SIGPLAN Notices, 46(8):35–46, 2011

16. Krzysztof Czarnecki and Ulrich W. Eisenecker. Generative Programming: Methods, Tools, and Applications. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000.

17. Krzysztof Czarnecki. Overview of generative software development. In Unconventional Programming Paradigms, pages 326–341. Springer, 2005.

18. Jack Herrington. Code generation in action. Manning Publications Co., 2003.

19. Rui Pais, SP Barros, and Lu´ıs Gomes. A tool for tailored code generation from petri net models. In 2005 IEEE Conference on Emerging Technologies and Factory Automation, volume 1, pages 8–pp. IEEE, 2005.

20. N Amanquah and OT Eporwei. Rapid application development for mobile terminals. In 2009 2nd International Conference on Adaptive Science & Technology (ICAST), pages 410–417. IEEE, 2009.

21. Andrew Hunt and David Thomas. The pragmatic programmer: from journeyman to master. Addison-Wesley Professional, 20

22. Kresimir Fertalj and Mario Brcic. A source code generator based on uml specification. International journal of computers and communications, (1):10–19, 2008

23. Andrejs Bajovs, Oksana Nikiforova, and Janis Sejans. Code generation from uml model: State of the art and practical implications. Applied Computer Systems, 14(1):9–18, 2013.

24. Andrew Burnard and Land Rover. Verifying and validating automatically generated code. In Proc. of International Automotive Conference (IAC). Citeseer, 2004.

25. Victor Guana and Eleni Stroulia. How do developers solve software-engineering tasks on model-based code generators? an empirical study design. In First International Workshop on Human Factors in Modeling (HuFaMo 2015). CEUR-WS, pages 33–38, 2015.

26. Peter MW Knijnenburg, Toru Kisuki, and Michael FP OBoyle. Iterative compilation. In Embedded processor design challenges, pages 171–187. Springer, 2002.

27. Vu Le, Mehrdad Afshari, and Zhendong Su. Compiler validation via equivalence modulo inputs. In ACM SIGPLAN Notices, volume 49, pages 216–226. ACM, 2014

28. Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in c compilers. In ACM SIGPLAN Notices, volume 46, pages 283–294. ACM, 2011

29. Ingo Stuermer, Mirko Conrad, Heiko Doerr, and Peter Pepper. Systematic testing of model-based code generators. IEEE Transactions on SoftwareEngineering, 33(9):622, 2007.

30. Sergey V Zelenov, Denis V Silakov, Alexander K Petrenko, Mirko Conrad, and Ines Fey. Automatic test generation for model-based code generators.In ISoLA, pages 75–81, 200

31. Mirko Conrad. Testing-based translation validation of generated code in the context of iec 61508. Formal Methods in System Design, 35(3):389–401, 2009.

32. T Erkkinen Conrad, T Maier-Komor, G Sandmann, and M Pomeroy. Code generation verification–assessing numerical equivalence between simulink models and generated code. In 4th Conference Simulation and Testing in Algorithm and Software Development for Automobile Electronics, 2010.

33. Sven J̈orges and Bernhard Steffen. Back-to-back testing of model-based code generators. In International Symposium On Leveraging Applications of Formal Methods, Verification and Validation, pages 425–444. Springer, 2014.

34. Igno Sturmer and Mirko Conrad. Test suite design for code generation tools. In Automated Software Engineering, 2003. Proceedings. 18th IEEE International Conference on, pages 286–290. IEEE, 2003.

35. Ingo St ̈urmer, Daniela Weinberg, and Mirko Conrad. Overview of existing safeguarding techniques for automatically generated code. In ACM SIG-SOFT Software Engineering Notes, volume 30, pages 1–6. ACM, 2005.

36. Mladen A Vouk. Back-to-back testing. Information and software technology, 32(1):34–45, 1990.

37. William M McKeeman. Differential testing for software. Digital Technical Journal, 10(1):100–107, 1998.

38. Hesham Shokry and Mike Hinchey. Model-based verification of embedded software. IEEE Computer, 42(4):53–59, 2009.

39. Nurlida Basir, Ewen Denney, and Bernd Fischer. Constructing a safety case for automatically generated code from formal program verification information. In International Conference on Computer Safety, Reliability, and Security, pages 249–262. Springer, 2008.

40. Xavier Leroy. Formal verification of a realistic compiler. Communications of the ACM, 52(7):107–115, 2009.

41. Ewen Denney and Bernd Fischer. Certifiable program generation. In International Conference on Generative Programming and Component Engineering, pages 17–28. Springer, 2005.

42. Stepan Stepasyuk and Yavor Paunov. Evaluating the haxe programming language-performance comparison between haxe and platform-specific languages. 2015.

43. Julien Richard-Foy, Olivier Barais, and Jean-Marc J´ez´equel. Efficient highlevel abstractions for web programming. In ACM SIGPLAN Notices, volume 49, pages 53–60. ACM, 2013.

44. Domagoj Strekelj, Hrvoje Leventi´c, and Irena Gali´c. Performance overhead of haxe programming language for cross-platform game development. International Journal of Electrical and Computer Engineering Systems, 6(1):9–13, 2015.

45. Nora Ajwad. Evaluation of automatic code generation tools. MSc Theses,2007

46. Alireza Pazirandeh and Evelina Vorobyeva. Evaluation of cross-platform tools for mobile development. 2015

47. Gustavo Hartmann, Geoff Stead, and Asi DeGani. Cross-platform mobile development. Mobile Learning Environment, Cambridge, pages 1–18, 2011.

48. Earl T Barr, Mark , Phil McMinn, Muzammil Shahbaz, and Shin Yoo. The oracle problem in software testing: A survey. IEEE transactions on software engineering, 41(5):507–525, 2015.

49. LI Manolache and Derrick G. Kourie. Software testing using model programs. Software: Practice and Experience, 31(13):1211–1236, 2001

50. Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. A comprehensive survey of trends in oracles for software testing. University of Sheffield, Department of Computer Science, Tech. Rep. CS-13-01, 2013

51. Phil Stocks and David Carrington. A framework for specification-based testing. IEEE Transactions on software Engineering, 22(11):777–793, 1996.

52. Debra J Richardson, Stephanie Leif Aha, and T Owen O'malley. Specification-based test oracles for reactive systems. In Proceedings of the 14th international conference on Software engineering, pages 105–118. ACM, 1992.

53. Barton P Miller, Louis Fredriksen, and Bryan So. An empirical study of the reliability of unix utilities. Communications of the ACM, 33(12):32–44,1990.

54. Sofia Bekrar, Chaouki Bekrar, Roland Groz, and Laurent Mounier. Finding software vulnerabilities by smart fuzzing. In Software Testing, Verification and Validation (ICST), 2011 IEEE Fourth International Conference on, pages 427–430. IEEE, 2011.

55. Nathan P Kropp, Philip J Koopman, and Daniel P Siewiorek. Automated robustness testing of off-the-shelf software components. In Fault-Tolerant Computing, 1998. Digest of Papers. Twenty-Eighth Annual International Symposium on, pages 230–239. IEEE, 1998

56. Filippo Ricca and Paolo Tonella. Detecting anomaly and failure in web applications. IEEE MultiMedia, 13(2):44–51, 2006.

57. Leonardo Mariani, Sofia Papagiannakis, and Mauro Pezze. Compatibility and regression testing of cots-component-based software. In Software Engineering, 2007. ICSE 2007. 29th International Conference on, pages 85–95. IEEE, 2007.

58. Matthew Patrick, Andrew P Craig, Nik J Cunniffe, Matthew Parry, and Christopher A Gilligan. Testing stochastic software using pseudo-oracles. In Proceedings of the 25th International Symposium on Software Testing and Analysis, pages 235–246. ACM, 2016.

59. Martin D Davis and Elaine J Weyuker. Pseudo-oracles for non-testable programs. In Proceedings of the ACM'81 Conference, pages 254–257. ACM, 1981

60. Spyridon Triantafyllis, Manish Vachharajani, Neil Vachharajani, and David I August. Compiler optimization-space exploration. In Code Generation and Optimization, 2003. CGO 2003. International Symposium on, pages 204–215. IEEE, 2003.

61. Zhelong Pan and Rudolf Eigenmann. Fast and effective orchestration of compiler optimizations for automatic performance tuning. In International Symposium on Code Generation and Optimization (CGO'06), pages 12–pp. IEEE, 2006.

62. Masayo Haneda, Peter MW Knijnenburg, and Harry AG Wijshoff. Automatic selection of compiler options using non-parametric inferential statistics. In Parallel Architectures and Compilation Techniques, 2005. PACT 2005. 14th International Conference on, pages 123–132. IEEE, 2005.

63. Lelac Almagor, Keith D Cooper, Alexander Grosul, Timothy J Harvey, Steven W Reeves, Devika Subramanian, Linda Torczon, and Todd Waterman. Finding effective compilation sequences. ACM SIGPLAN Notices, 9(7):231–239, 2004.

64. GG Fursin, Michael FP OBoyle, and Peter MW Knijnenburg. Evaluating iterative compilation. In International Workshop on Languages and Compilers for Parallel Computing, pages 362–376. Springer, 2002

65. Yang Chen, Yuanjie Huang, Lieven Eeckhout, Grigori Fursin, Liang Peng, Olivier Temam, and Chengyong Wu. Evaluating iterative optimization across 1000 datasets. In ACM Sigplan Notices, volume 45, pages 448–459. ACM, 2010.

66. Kenneth Hoste and Lieven Eeckhout. Cole: compiler optimization level exploration. In Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization, pages 165–174. ACM, 2008.

67. James Pallister, Simon J Hollis, and Jeremy Bennett. Identifying compiler options to minimize energy consumption for embedded platforms. The Computer Journal, 58(1):95–109, 2015.

68. Yang Chen, Shuangde Fang, Yuanjie Huang, Lieven Eeckhout, Grigori Fursin, Olivier Temam, and Chengyong Wu. Deconstructing iterative optimization. ACM Transactions on Architecture and Code Optimization (TACO), 9(3):21, 2012.

69. Luiz GA Martins, Ricardo Nobre, Alexandre CB Delbem, Eduardo Marques, and Jo˜ao MP Cardoso. Exploration of compiler optimization sequences using clustering-based selection. In Proceedings of the 2014 SIGPLAN/SIGBED conference on Languages, compilers and tools for embedded systems, pages 63–72. ACM, 2014.

70. Luiz GA Martins, Ricardo Nobre, Jõao MP Cardoso, Alexandre CB Delbem, and Eduardo Marques. Clustering-based selection for the exploration of compiler optimization sequences. ACM Transactions on Architecture and Code Optimization (TACO), 13(1):8, 2016.

71. San-Chih Lin, Chi-Kuang Chang, and San-Chih Lin. Automatic selection of gcc optimization options using a gene weighted genetic algorithm.In Computer Systems Architecture Conference, 2008. ACSAC 2008. 13th Asia-Pacific, pages 1–8. IEEE, 2008

72. Antonio Mart´ınez- ´Alvarez, Jorge Calvo-Zaragoza, Sergio Cuenca-Asensi, Andr´es Ortiz, and Antonio Jimeno-Morenilla. Multi-objective adaptive evolutionary strategy for tuning compilations. Neurocomputing, 123:381–389, 2014.

73. Dmitry Plotnikov, Dmitry Melnik, Mamikon Vardanyan, Ruben Buchatskiy, Roman Zhuykov, and Je-Hyung Lee. Automatic tuning of compiler optimizations and analysis of their impact. Procedia Computer Science, 18:1312–1321, 2013.

74. Keith D Cooper, Philip J Schielke, and Devika Subramanian. Optimizing for reduced code space using genetic algorithms. In ACM SIGPLAN Notices, volume 34, pages 1–9. ACM, 1999

75. Mark Stephenson, Una-May OReilly, Martin C Martin, and Saman Amarasinghe. Genetic programming applied to compiler heuristic optimization. In European Conference on Genetic Programming, pages 238–253. Springer, 2003.

76. Guy Bashkansky and Yaakov Yaari. Black box approach for selecting optimization options using budget-limited genetic algorithms. In Workshop Proceedings, page 27, 2007.

77. Fran ̧cois Bodin, Toru Kisuki, Peter Knijnenburg, Mike O'Boyle, and Erven Rohou. Iterative compilation in a non-linear optimisation space. In Workshop on Profile and Feedback-Directed Compilation, 1998

78. Keith D Cooper, Alexander Grosul, Timothy J Harvey, Steve Reeves, Devika Subramanian, Linda Torczon, and Todd Waterman. Exploring the structure of the space of compilation sequences using randomized search algorithms. The Journal of Supercomputing, 36(2):135–151, 2006

79. Deborah Whitfield and Mary Lou Soffa. An approach to ordering optimizing transformations. In ACM SIGPLAN Notices, volume 25, pages 137–146. ACM, 1990

80. Prasad A Kulkarni, David B Whalley, Gary S Tyson, and Jack W Davidson. Practical exhaustive optimization phase order exploration and evaluation. ACM Transactions on Architecture and Code Optimization (TACO), 6(1):1, 2009

81. Prasad A Kulkarni, David B Whalley, Gary S Tyson, and Jack W Davidson. Exhaustive optimization phase order space exploration. In Proceedings of the

International Symposium on Code Generation and Optimization, pages 306–318. IEEE Computer Society, 2006.

82. Grigori Fursin, Yuriy Kashnikov, Abdul Wahid Memon, Zbigniew Chamski, Olivier Temam, Mircea Namolaru, Elad Yom-Tov, Bilha Mendelson, Ayal Zaks, Eric Courtois, et al. Milepost gcc: Machine learning enabled self-tuning compiler. International journal of parallel programming, 39(3):296–327, 2011.

83. Paul Lokuciejewski, Sascha Plazar, Heiko Falk, Peter Marwedel, and Lothar Thiele. Multi-objective exploration of compiler optimizations forreal-time systems. In Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC), 2010 13th IEEE International Symposiumon, pages 115–122. IEEE, 2010.

84. Gennady Pekhimenko and Angela Demke Brown. Efficient program compilation through machine learning techniques. In Software Automatic Tuning, pages 335–351. Springer, 2011.

85. Cristian Constantin Spoiala, Alin Calinciuc, Corneliu Octavian Turcu, and Constantin Filote. Performance comparison of a webrtc server on docker versus virtual machine. In 2016 International Conference on Development and Application Systems (DAS), pages 295–298. IEEE, 2016.

86. Dirk Merkel. Docker: lightweight linux containers for consistent development and deployment. Linux Journal, 2014(239):2, 2014

87. Stephen Soltesz, Herbert P¨otzl, Marc E Fiuczynski, Andy Bavier, and Larry Peterson. Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors. In ACM SIGOPS Operating Systems Review, volume 41, pages 275–287. ACM, 2007.

88. Li Li, Tony Tang, and Wu Chou. A rest service framework for fine-grained resource management in container-based cloud. In 2015 IEEE 8th International Conference on Cloud Computing, pages 645–652. IEEE, 2015.

89. Paul Marinescu, Petr Hosek, and Cristian Cadar. Covrig: a framework for the analysis of code, test, and coverage evolution in real software. In Proceedings of the 2014 International Symposium on Software Testing and Analysis, pages 93–104. ACM, 2014.

90. Vincenzo Ferme, Ana Ivanchikj, and Cesare Pautasso. A framework for benchmarking bpmn 2.0 workflow management systems. In International Conference on Business Process Management, pages 251–259. Springer, 2015.

91. Abeer Hamdy, Osman Ibrahim, and Ahmed Hazem. A web based framework for pre-release testing of mobile applications. In MATEC Web of Conferences, volume 76, page 04041. EDP Sciences, 2016.

92. Giuseppe Aceto, Alessio Botta, Walter De Donato, and Antonio Pescap`e. Cloud monitoring: A survey. Computer Networks, 57(9):2093–2115, 2013.

93. Pakorn Kookarinrat and Yaowadee Temtanapat. Analysis of range-basedkey properties for sharded cluster of mongodb. In Information Science and Security (ICISS), 2015 2nd International Conference on, pages 1–4. IEEE, 2015.

94. Yu Sun, Jules White, Sean Eade, and Douglas C Schmidt. Roar: A qosoriented modeling framework for automated cloud resource allocation and optimization. Journal of Systems and Software, 116:146–161, 2016

95. Ren´e Peinl, Florian Holzschuher, and Florian Pfitzer. Docker cluster management for the cloud-survey results and own solution. Journal of Grid Computing, 14(2):265–282, 2016

96. Vıctor Medel, Omer Rana, Unai Arronategui, et al. Modelling performance & resource management in kubernetes. In Proceedings of the 9th International Conference on Utility and Cloud Computing, pages 257–262. ACM, 2016.

97. Wonseok Chae and Matthias Blume. Building a family of compilers. In Software Product Line Conference, 2008. SPLC'08. 12th International, pages 307–316. IEEE, 2008.

98. Juan Jos´e Fumero, Toomas Remmelg, Michel Steuwer, and Christophe Dubach. Runtime code generation and data management for heterogeneous computing in java. In Proceedings of the Principles and Practices of Programming on The Java Platform, pages 16–26. ACM, 2015.

99. Benjamin Dasnois. HaXe 2 Beginner's Guide. Packt Publishing Ltd, 2011.

100. Franck Fleurey, Brice Morin, Arnor Solberg, and Olivier Barais. Mde to manage communications with and between resource-constrained systems. In

International Conference on Model Driven Engineering Languages and Systems, pages 349–363. Springer, 2011.

101. Aseem Rastogi, Nikhil Swamy, C´edric Fournet, Gavin Bierman, and Panagiotis Vekris. Safe & efficient gradual typing for typescript. In ACM SIGPLAN Notices, volume 50, pages 167–180. ACM, 2015.

102. Robert Hundt. Loop recognition in c++/java/go/scala. In Proceedings of Scala Days 2011, June 2011

103. Melina Demertzi, Murali Annavaram, and Mary Hall. Analyzing the effects of compiler optimizations on application reliability. In Workload Characterization (IISWC), 2011 IEEE International Symposium on, pages 184–193. IEEE, 2011.

104. Victor Guana and Eleni Stroulia. Chaintracker, a model-transformation trace analysis tool for code-generation environments. In ICMT, pages 146–153. Springer, 2014.

105. Nelly Delgado, Ann Q Gates, and Steve Roach. A taxonomy and catalog of runtime software-fault monitoring tools. IEEE Transactions on software Engineering, 30(12):859–872, 2004.

106. Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavy weight dynamic binary instrumentation. In ACM Sigplan notices, volume 42, pages 89–100. ACM, 2007

107. Miguel G Xavier, Israel C De Oliveira, Fabio D Rossi, Robson D Dos Passos, Kassiano J Matteussi, and C´esar AF De Rose. A performance isolation analysis of disk-intensive workloads on container-based clouds. In Parallel, Distributed and Network-Based Processing (PDP), 2015 23rd Euromicro International Conference on, pages 253–260. IEEE, 2015.

108. Zhi Quan Zhou, DH Huang, TH Tse, Zongyuan Yang, Haitao Huang, and TY Chen. Metamorphic testing and its applications. In Proceedings of the 8th International Symposium on Future Software Technology (ISFST 2004), pages 346–351, 2004.

109. Tsong Y Chen, Shing C Cheung, and Shiu Ming Yiu. Metamorphic testing: a new approach for generating next test cases. Technical report, Technical Report

HKUST-CS98-01, Department of Computer Science, Hong Kong University of Science and Technology, Hong Kong, 1998.

110. Alastair F Donaldson and Andrei Lascu. Metamorphic testing for (graphics) compilers. In Proceedings of the 1st International Workshop on Metamorphic Testing, pages 44–47. ACM, 2016.

111. Qiuming Tao, Wei Wu, Chen Zhao, and Wuwei Shen. An automatic testing approach for compiler based on metamorphic testing technique. In Software Engineering Conference (APSEC), 2010 17th Asia Pacific, pages 270–279. IEEE, 2010.

112. WK Chan, Tsong Yueh Chen, Heng Lu, TH Tse, and Stephen S Yau. Integration testing of context-sensitive middleware-based applications: a metamorphic approach. International Journal of Software Engineering and Knowledge Engineering, 16(05):677–703, 2006.

113. Haroon Malik, Hadi Hemmati, and Ahmed E Hassan. Automatic detection of performance deviations in the load testing of large scale systems. In Proceedings of the 2013 International Conference on Software Engineering, pages 1012–1021. IEEE Press, 2013.

114. David P Enot, Wanchang Lin, Manfred Beckmann, David Parker, David P Overy, and John Draper. Preprocessing, classification modeling and feature selection using flow injection electrospray mass spectrometry metabolite fingerprint data. Nature Protocols, 3(3):446–470, 2008.

115. Mia Hubert, Peter Rousseeuw, and Tim Verdonck. Robust AOK for skewed data and its outlier map. Computational Statistics & Data Analysis, 53(6):2264–2274, 2009.

116. Mena Nagiub and Wael Farag. Automatic selection of compiler options using genetic techniques for embedded software design. In Computational Intelligence and Informatics (CINTI), 2013 IEEE 14th International Symposium on, pages 69–74. IEEE, 2013.

117. Joel Lehman and Kenneth O Stanley. Exploiting open-endedness to solve problems through the search for novelty. In ALIFE, pages 329–336, 2008.

118. Wolfgang Banzhaf, Frank D Francone, and Peter Nordin. The effect of extensive use of the mutation operator on generalization in genetic programming using sparse data sets. In Parallel Problem Solving from NaturePPSN IV, pages 300–309. Springer, 1996

119. Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and TAMT Meyarivan. A fast and elitist multiobjective genetic algorithm: Nsga-ii. Evolutionary Computation, IEEE Transactions on, 6(2):182–197, 2002

120. Junjie Chen, Wenxiang Hu, Dan Hao, Yingfei Xiong, Hongyu Zhang, Lu Zhang, and Bing Xie. An empirical comparison of compiler testing techniques. In Proceedings of the 38th International Conference on Software Engineering, 2016

121. Eriko Nagai, Atsushi Hashimoto, and Nagisa Ishiura. Scaling up size and number of expressions in random testing of arithmetic optimization of c compilers. In Workshop on Synthesis And System Integration of Mixed Information Technologies (SASIMI 2013), pages 88–93, 2013

122. Grigori Fursin. Collective tuning initiative: automating and accelerating development and optimization of computing systems. In GCC Developers' Summit, 2009

123. Keith D Cooper, Devika Subramanian, and Linda Torczon. Adaptive optimizing compilers for the 21st century. The Journal of Supercomputing, 23(1):7–22, 2002

124. Benjamin Inden, Yaochu Jin, Robert Haschke, Helge Ritter, and Bernhard Sendhoff. An examination of different fitness and novelty based selection methods for the evolution of neural networks. Soft Computing, 17(5):753–767, 2013.

125. Mohamed Boussaa, Olivier Barais, Gerson Suny´e, and Benoit Baudry. A novelty search approach for automatic test data generation. In 8th International Workshop on Search-Based Software Testing SBST@ ICSE 2015, page 4, 2015.

ДОДАТОК А

(Обов’язковий)

КОПІЯ НАУКОВОЇ ПУБЛІКАЦІЇ

Міністерство освіти і науки України
Хмельницький національний університет

ЗБІРНИК НАУКОВИХ ПРАЦЬ

за матеріалами XIV Всеукраїнської науково-практичної конференції
«Актуальні проблеми комп'ютерних наук АПКН-2022»

18-19 листопада 2022

Хмельницький 2022

УДК 004:37:001:62

Збірник наукових праць за матеріалами XIV Всеукраїнської науково-практичної конференції «Актуальні проблеми комп'ютерних наук АПКН-2022». Хмельницький – 2022. – 331с.

У збірнику наукових праць подані перспективні практичні розробки аспірантів, студентів та здобувачів в області сучасних інформаційних технологій. Розглянуто актуальні проблеми комп'ютерних наук, комп'ютерної інженерії, прикладної математики й інженерії програмного забезпечення, приведено ряд робіт по впровадженню інформаційних технологій у виробництво та управління. Висвітлено перспективні розробки сучасних систем пошуку, обробки й захисту інформації, медійних та комунікаційних системи.

УДК 004:37:001:62

Матеріали конференції відтворені з авторських оригіналів. При макетуванні можливі незначні зміни компоновки контенту авторських оригіналів.

Участь у конференції та складові всіх її етапів (розгляд праць, макетування, публікація збірника наукових праць та видача сертифікатів) є безкоштовними для всіх учасників. Оргкомітет конференції висловлює подяку учасникам конференції та сподівається на подальшу співпрацю.

З питань проведення конференції та подальшого обміну інформацією звертатись на e-mail конференції: *apkt.khnu@gmail.com*

АКТУАЛЬНІ ПРОБЛЕМИ КОМП'ЮТЕРНИХ НАУК - 2022

XIV Всеукраїнська науково-практична конференція

Метою конференції є висвітлення актуальних проблем комп'ютерних наук, інформатики та інформаційних технологій.

СЕКЦІЇ КОНФЕРЕНЦІЇ:

1. Комп'ютерні науки та прикладні інформаційні технології.
2. Комп'ютерна інженерія та системи захисту інформації.
3. Математичне моделювання та інженерія програмного забезпечення
4. Телерадіокомунікації, медійні та комунікаційні системи.
5. Проблеми впровадження інформаційних технологій у виробництво та управління.

Робочі мови конференції: українська, англійська

ОРГКОМІТЕТ:

СИНЮК О. М. голова оргкомітету, проректор Хмельницького національного університету з наукової роботи, доктор технічних наук, професор

САВЕНКО О. С. заступник голови оргкомітету, декан факультету Інформаційних технологій ХНУ, доктор технічних наук, професор

БАРМАК О. В. заступник голови оргкомітету, завідувач кафедри Комп'ютерних наук ХНУ, доктор технічних наук, професор

ГОВОРУЩЕНКО Т. О. завідувач кафедри Комп'ютерної інженерії та інформаційних систем ХНУ, доктор технічних наук, професор

ВИСОЦЬКА О. В. доктор технічних наук, завідувач кафедри Радіоелектронних та біомедичних комп'ютеризованих засобів і технологій Національного аерокосмічного університету ім. М. Є. Жуковського «Харківський авіаційний інститут», професор

ЛАВРОВ Є. А. доктор технічних наук, професор (Сумський державний університет)

ТІМОФЄЄВА Л. В. відповідальна за студентську науково-дослідну роботу ХНУ

МАЗУРЕЦЬ О. В. секретар конференції, к.т.н., доцент кафедри Комп'ютерних наук ХНУ

МОЛЧАНОВА М. О. секретар конференції, викладач кафедри Комп'ютерних наук ХНУ

КОНТАКТНА ІНФОРМАЦІЯ:

e-mail для листування: *apkt.khnu@gmail.com*

ЗМІСТ

УДК 004.03

Дьоміна А.І.

Хмельницький національний університет

ВИКОРИСТАННЯ МЕТОДУ ПОШУКУ НОВИЗНИ ДЛЯ АВТОМАТИЧНОЇ ГЕНЕРАЦІЇ ТЕСТОВИХ ДАНИХ

У роботі проводиться ознайомлення з використанням алгоритму пошуку новизни (Novelty Search NS) для задач генерації тестових даних на основі критеріїв, описаних інструкцією. Такий підхід до генерації тестових даних є цікавим, оскільки він дозволяє досліджувати величезний простір тестових даних у вхідній області.

The work introduces the use of the Novelty search algorithm for the tasks of generating test data based on the criteria described in the instructions. This approach to test data generation is interesting because it allows exploring a huge space of test data in the input domain.

Створення тестів вручну для тестування програмного забезпечення займає багато часу та часто породжує купу помилок, що вимагає автоматизації цього процесу. Насправді метаевристичні методи пошуку, такі як генетичні алгоритми (Genetic Algorithms GA), часто використовуються для автоматизації процесу генерації тестових даних і збору відповідних тестових випадків через широкий простір пошуку [1, 2]. Ці методи особливо застосовуються для структурного тестування білого ящика. Для підходів, орієнтованих на покриття, застосування еволюційних алгоритмів для генерації тестових даних було зосереджено на пошуку вхідних даних для конкретного шляху програми відповідно до критерію покриття (наприклад, найдовший шлях). Проблема підходів, орієнтованих на покриття, полягає в тому, що методи, засновані на пошуку, не використовують величезний простір можливих тестових даних. Фактично, деякі структури системи не застосовуються, оскільки вони виконуються лише невеликою частиною вхідної області. Застосування GA для генерації тестових даних полягає в пошуку відповідних тестових даних відповідно до цільової функції, яка намагається, наприклад, максимізувати кількість охоплених операторів або гілок.

У цій статті представлено використання алгоритму пошуку новизни (NS) для проблеми генерації тестових даних. У цьому підході досліджується простір пошуку можливих тестових вхідних значень без урахування будь-яких цілей (немає функції відповідності). Насправді, замість відбору на основі придатності, обираються тестові випадки на основі оцінки новизни, яка показує, наскільки вони відрізняються від усіх інших тестових даних, оцінених до цього часу. Таким чином,

під час еволюційного процесу вибору тестових даних використовуються ті, які залишаються в розріджених регіонах пошукового простору, щоб направляти пошук через новизну обирається показник охоплення заяв як критерій охоплення для генерації тестових даних на основі NS.

Більшість форм автоматичного генерування тестових даних були зосереджені на пошуку конкретних вхідних значень, які відповідають певним критеріям тестування. Під час генерації тестових даних на основі пошуку Гарман і Макмін [5] провели велике емпіричне дослідження, яке порівнює поведінку як глобальної, так і локальної оптимізації на основі пошуку в реальних проблемах. Результати показують, що використання еволюційних алгоритмів підходить у багатьох випадках. Однак його можна перевершити за допомогою простіших методів пошуку.

Крім того, було проведено багато дослідницьких робіт у сфері покриття структурного коду. Наприклад, у роботі Ропера [1] GA використовуються для генерації тестових даних на основі кількості структур, виконаних відповідно до критерію покриття. Тоді як Уоткінс [2] намагається отримати повне покриття.

Щодо стратегій відбору тестових випадків Чен та ін. [6] представили синтез найважливіших результатів адаптивного випадкового тестування (АВТ). Вони підкреслили важливість різноманітності у виборі тестових випадків. Фактично, вони стверджують, що різноманітність серед тестів має бути винагороджена, оскільки невдалі тестові приклади мають тенденцію групуватися в суміжних областях вхідного домену..

У літературі еволюційні алгоритми часто застосовуються до проблеми генерації тестових даних [7]. Ці методи використовують в основному функцію відповідності, щоб керувати пошуком, наприклад, зібрати найпридатніші рішення за покоління. Ці методи хороші, оскільки вони нагороджують індивідів високими балами, але вони не сприяють різноманітності, і пошук може сходитися до багатьох локальних оптимумів [3, 4]. Ідея NS, представлена Леманом та Стенлі у 2008 році [8], є альтернативним рішенням цієї проблеми. Фактично, нові значення в популяції, що розвивається, відбираються на основі того, наскільки вони відрізняються від інших рішень, оцінених до цього часу. Вони також стверджують, що об'єктивні функції відповідності можуть бути оманливими, ведучи еволюційний пошук до локальних максимумів, а не до мети. Навпаки, NS ігнорує мету і просто шукає нову поведінку, і тому не може бути обманутим. Тому в основному NS діє як GA. Однак NS потребує додаткових змін. По-перше, вимірювання поведінки окремих значень. Це залежить від контексту пошуку та способу, яким представлено окремі значення. Потім новий показник новизни, який винагороджує значення з поведінкою, що відрізняється від раніше відкритих рішень. Нарешті, до алгоритму необхідно додати архів, який є свого роду базою даних, яка запам'ятовує значення, які були дуже

новими, коли їх виявили минулі покоління. NS часто оцінювали в оманливих завданнях і застосовували до еволюційної робототехніки (в контексті нейроеволюції) [9, 10].

Представлена система генерації тестових даних має на меті повністю автоматизувати процес генерації тестових даних і уникнути будь-якого впливу тестувальника. У ній використовується нова еволюційна техніка оптимізації, а саме пошук новизни, для проблеми генерації тестових даних.

Для автоматизації генерації тестових даних, враховано, що представлена тестована система схожа на сірий ящик, внутрішня структура якого частково відома. Отже, можна розробити тестові випадки через відкриті інтерфейси та провести аналіз покриття коду із загальної структури цільової тестованої системи (рисунок 1).

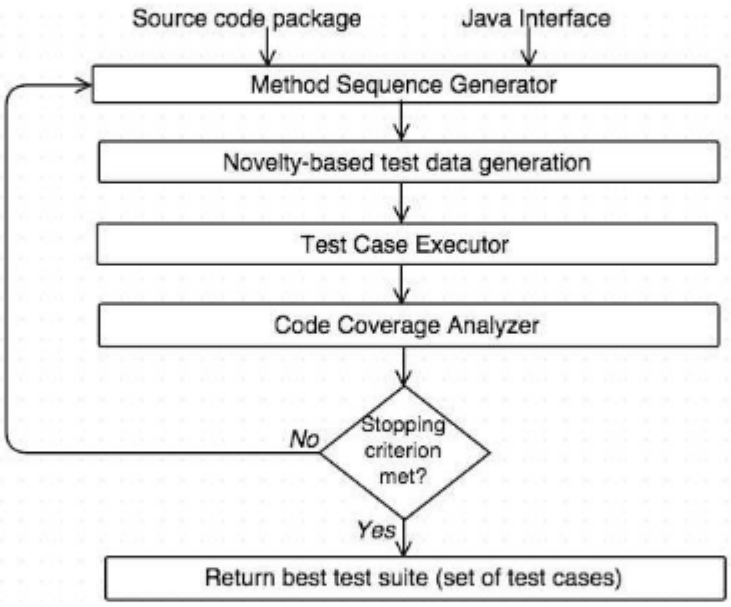


Рисунок 1 – Огляд підходу

Фактично, як показано на рисунку 1, починаючи з інтерфейсу введення та пакета вихідного коду, платформа тестування здатна: (1) автоматично генерувати послідовності виклику методів, (2) генерувати відносні тестові дані, (3) виконувати тестові випадки на цільових тестованих класах , а потім (4) аналізувати покриття коду. Процес повторюється, доки не буде виконано критерій завершення (наприклад, певна кількість ітерацій).

Основним завданням пропонованого алгоритму є використання альтернативного підходу для генерації тестових даних на основі алгоритму NS. Для

цього адаптовано загальну ідею NS, представлену раніше у відповідній роботі, до проблеми генерації тестових даних. Замість використання функції відповідності для оцінки згенерованих тестів, визначається міра новизни для максимізації. Також замінюємо вибір на основі придатності на вибір на основі новизни. Це може сприяти різноманітності згенерованих тестових даних. Крім того, незважаючи на те, що досліджується простір пошуку без жодної мети, зберігається міру охоплення операторів для кожного набору тестових випадків, щоб до кінця еволюційного процесу могли бути зібрана лише тестові випадки з високим значенням охоплення. Також додано архів, для згенерованих раніше тестові приклади. Він використовується для розрахунку метрики новизни. Нижче детально описано основні завдання системи автоматичного генерування тестових даних:

1) Генератор послідовності методів: Генератор послідовності методів використовується для автоматичного створення тестового сценарію. Він приймає як вхідні дані інтерфейс Java і пакет вихідного коду. Згенерована послідовність представляє назви методів і типи їхніх параметрів, оголошені в інтерфейсі Java. Таким чином зберігається та сама кількість і порядок сигнатур методів, як визначено в інтерфейсі. Таким чином тестується кожен метод цільових класів ізольовано, не враховуючи залежності між методами.

2) Генерація тестових даних на основі новизни: для створення тестових випадків створюємо вхідні дані, які будуть передані як аргументи до методів що тестуються. Залежно від типів параметрів генеруємо випадкові вхідні значення для примітивних типів і рядків Java. Насправді ми не визначили жодних обмежень на значення тестових даних і розглянули всі можливі вхідні значення для кожного типу даних. Цей процес застосовує підхід NS для створення тестових даних.

3) Виконавець тестового прикладу: спочатку виконавець тестового прикладу досліджує пакет вихідного коду та перехоплює всі класи, що тестуються, що реалізують інтерфейс введення. Як тільки це буде зроблено, автоматично виконуємо набір тестів поверх цих класів, за винятком абстрактних класів.

4) Аналізатор покриття коду: структура тестування визначає аналізатор, здатний інструментувати й аналізувати всі класи Java у пакеті вихідного коду. Таким чином, після виконання набору тестів інструментальний код аналізується та кожному набору тестів призначається значення покриття. Наприклад, перевіряємо кількість виконаних операторів.

Отже, у представленій адаптації NS було здійснено спробу використовувати великий простір пошуку вхідних значень і виловлювати відповідні тестові випадки, які можуть охоплювати якомога більше виконуваних операторів. Однак обмеження цього підходу полягає у використанні безперервних даних для обчислення оцінки новизни. Якщо ми припустимо, що вхідні значення можуть мати лише певні

значення або категоричні дані, оцінка відстані буде не простою, і можна використовувати інші міри подібності для категоріальних даних..

Перелік посилань:

1. M. Roper, "Computer aided software testing using genetic algorithms," 10th International Quality Week, 1997.
2. A. L. Watkins, "The automatic generation of test data using genetic algorithms," in Proceedings of the 4th Software Quality Conference, vol. 2, 1995, pp. 300–309.
3. W. Banzhaf, F. D. Francone, and P. Nordin, "The effect of extensive use of the mutation operator on generalization in genetic programming using sparse data sets," in Parallel Problem Solving from NaturePPSN IV. Springer, 1996, pp. 300–309.
4. C. Gathercole and P. Ross, "An adverse interaction between crossover and restricted tree depth in genetic programming," in Proceedings of the 1st annual conference on genetic programming. MIT Press, 1996, pp. 291–296.
5. M. Harman and P. McMinn, "A theoretical and empirical study of search-based testing: Local, global, and hybrid search," Software Engineering, IEEE Transactions on, vol. 36, no. 2, pp. 226–247, 2010.
6. T. Y. Chen, F.-C. Kuo, R. G. Merkel, and T. Tse, "Adaptive random testing: The ABT of test case diversity," Journal of Systems and Software, vol. 83, no. 1, pp. 60–66, 2010.
7. M. Harman, L. Hu, R. M. Hierons, A. Baresel, and H. Sthamer, "Improving evolutionary testing by flag removal." in GECCO, 2002, pp. 1359–1366.
8. J. Lehman and K. O. Stanley, "Exploiting open-endedness to solve problems through the search for novelty." in ALIFE, 2008, pp. 329–336.
9. S. Risi, C. E. Hughes, and K. O. Stanley, "Evolving plastic neural networks with novelty search," Adaptive Behavior, vol. 18, no. 6, pp. 470–491, 2010.
10. P. Krcah "Solving deceptive tasks in robot body-brain co-evolution by searching for behavioral novelty," in Advances in Robotics and Virtual Reality. Springer, 2012, pp. 167–186.

ДОДАТОК Б

(Обов’язковий)

ПРЕЗЕНТАЦІЙНІ МАТЕРИАЛИ

Кафедра інженерії програмного забезпечення

Метод нефункційного тестування та автоматичного налаштування генераторів кодів з урахуванням програмно-апаратної платформи

Виконала: студентка ІПЗм-21-1 *Дьоміна А. І.*

Керівник: канд. техн. наук, доцент *Гурман І. В.*

Об’єкт, предмет та мета дослідження

Об’єкт дослідження. Робота генераторів коду в яких можуть бути помилки та неоптимальні рішення.

Предмет дослідження. Методи тестування та оптимізації генераторів коду.

Мета роботи. Удосконалення методів нефункційного тестування та автоматичного налаштування генераторів кодів з урахуванням програмно-апаратного забезпечення

Актуальність

Актуальність теми роботи полягає в необхідності розробки методу нефункційного тестування та автоматичного налаштування генераторів кодів з урахуванням програмно-апаратної платформи, який позитивно впливатиме на якість генерованого коду та сприятиме зменшенню кількості помилок та неоптимальних рішень.

В минулому вже запропоновано ряд різних методів тестування на основі еволюційних методів пошуку, проте всі вони не враховують програмно-апаратне забезпечення мають ряд недоліків ігнорують цілі області тестових значень через недосконалості алгоритму

Завдання дослідження:

Дослідження нефункціонального тестування генераторів коду: існування кількох генераторів коду з порівнянною функціональністю (тобто сімейства генераторів коду) є дуже корисним явищем для автоматичного тестування згенерованого коду. В роботі використано метаморфічне тестування для виявлення нефункціональних невідповідностей у сімействах генераторів коду. У цьому дослідженні акцент зосереджено на тестуванні характеристик продуктивності та використання ресурсів. Невідповідність виявляється, коли згенерований код демонструє неочікувану поведінку з точки зору продуктивності або використання ресурсів порівняно з усіма еквівалентними реалізаціями в тому самому сімействі генераторів коду.

Дослідження автоматичного налаштування генераторів: використано останні досягнення в розробці програмного забезпечення на основі пошуку, щоб забезпечити ефективний підхід до налаштування генераторів (наприклад, компіляторів GCC) за допомогою оптимізації.

Дослідження різноманітності програмних платформ і неоднорідності апаратного забезпечення в тестуванні ПЗ: запуск тестів і оцінка використання ресурсів у гетерогенних середовищах є ресурснозатратним, для вирішення цієї проблеми використано останні досягнення у віртуалізації легких систем, зокрема віртуалізації на основі контейнерів, для ефективної підтримки автоматичного розгортання, виконання та моніторингу коду в гетерогенному середовищі, а також збору нефункціональних метрик (наприклад, споживання пам'яті та ЦП).

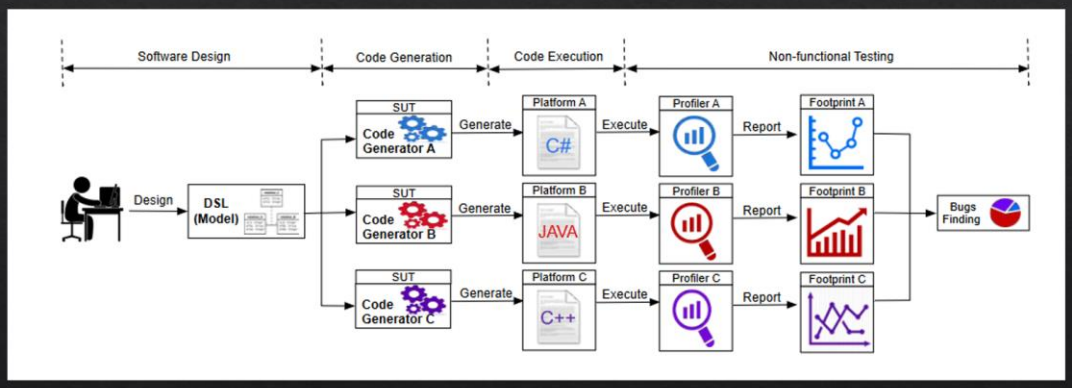
Наукова новизна

Удосконалення методу нефункційного тестування та автоматичного налаштування генераторів кодів завдяки урахуванню програмно-апаратної платформи шляхом використання гібридного методу пошуку заснованого на Novelty search

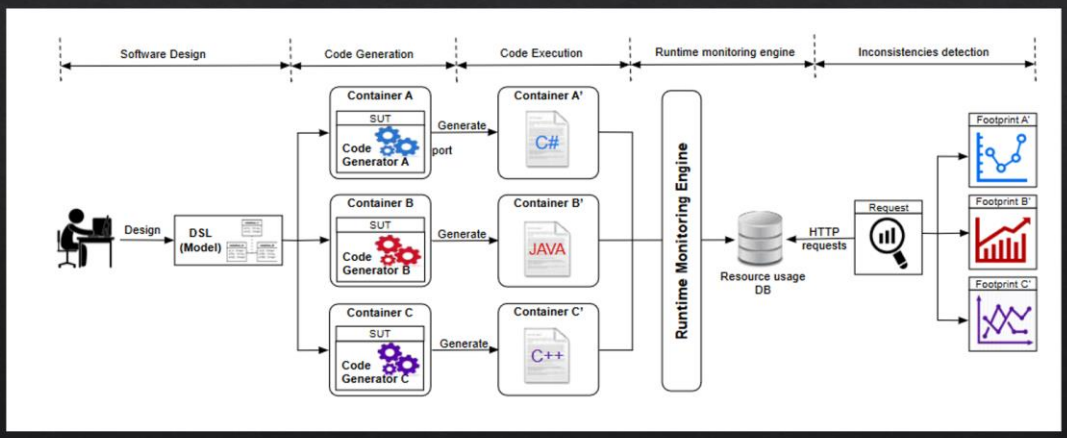
Практичне значення

Сучасні генератори мають широкі можливості конфігурації, що дозволяє користувачеві легко налаштовувати автоматично згенерований код. Тим часом ефективне тестування конфігурованих генераторів створює серйозні проблеми, оскільки вручну або автоматично виконувати та тестувати всі конфігурації занадто дорого. Можна ввімкнути різні категорії параметрів, щоб допомогти розробникам: налагоджувати, оптимізувати та налаштовувати продуктивність програми, вибирати рівні мови та розширення для сумісності, вибирати цільову архітектуру апаратного забезпечення та виконувати багато інших звичайних завдань які налаштовують спосіб генерації коду. Величезна кількість конфігурацій генератора, версій, оптимізацій і утиліт для налагодження робить завдання вибору найкращого набору конфігурацій дуже складним і трудомістким. Крім того, вплив оптимізації сильно залежить від апаратного забезпечення та вхідного вихідного коду. Цей приклад показує, наскільки складно користувачам налаштовувати генератори, такі як компілятори, щоб задовольнити різні нефункціональні властивості, такі як час виконання, час компіляції, розмір коду тощо.

Традиційний процес для нефункціонального тестування сімейства генераторів коду



Процесс нефункціонального тестування за допомогою системи контейнерів



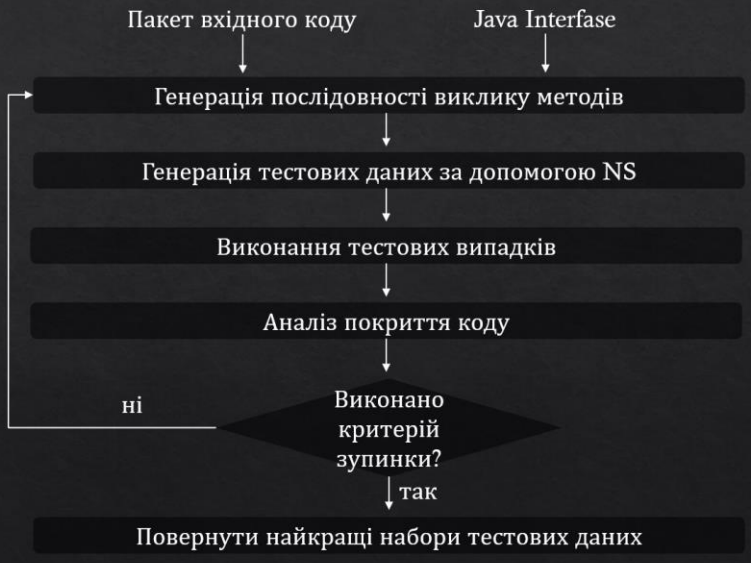
У порівнянні з класичним методом, додано такі функції:

На рівні генерації коду: генератори коду налаштовуються всередині різних контейнерів, щоб генерувати код для цільової платформи.

На рівні виконання коду: бібліотеки, компілятори та різні залежності конфігуруються в різних контейнерах для виконання згенерованого коду. Для кожної цільової платформи створюється новий екземпляр

На нефункціональному рівні тестування: додано механізм моніторингу часу виконання (на основі контейнерів), щоб отримати та зберегти показники використання ресурсів усіх запущених контейнерів. Нарешті, виявлення невідповідностей передбачає аналіз даних про використання ресурсів, отриманих із бази даних, щоб знайти проблеми зі згенерованим кодом.

Використання підходу Novelty search

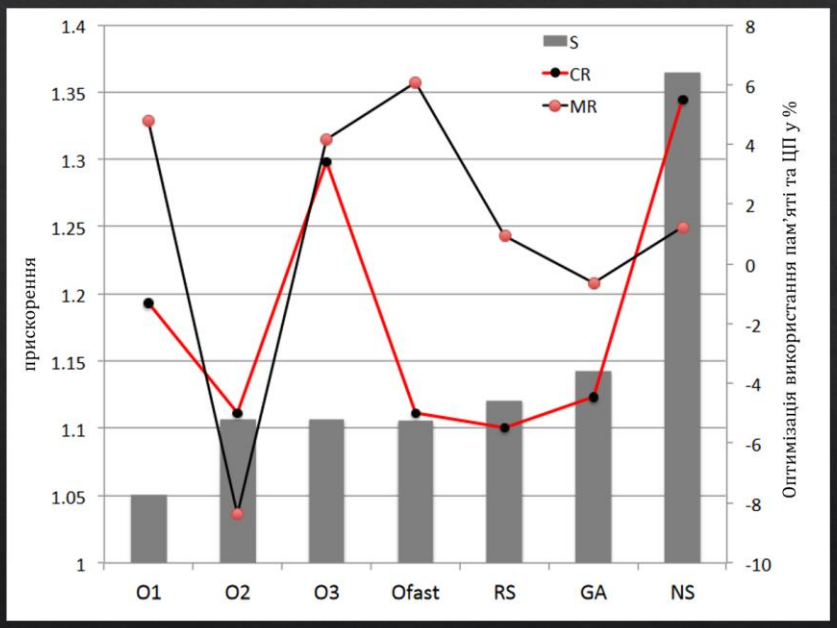


Адаптація методу Novelty search для генерації тестових даних

Require: Java interface I
Require: Source code package Pack
Require: Number of iterations N
Require: Population size PopSize
Require: Coverage threshold minCoverage
Require: Novelty threshold T
Require: Limit L
Require: Nearest neighbors k
Ensure: Set of relevant test cases bestSolutions
1: *targetClasses* ← *loadAllM ethods(I, P ack)*
2: *testCases* ← *generateTestCases(I)*
3: **repeat**
4: *testSuite* ← *generateTestData(testCases)*
5: *P* ← *setOf (testSuite)*
6: **for** *testSuite* ∈ *P* **do**
7: **for** *testCase* ∈ *testSuite* **do**
8: *coverage* ← *execute(testCase, targetClasses)*
9: **end for**
10: *noveltyM etric* ←*distFromkNearest(testSuite, archive, P, k)*
11: **if** *noveltyM etric* > *T* **then**
12: *archive* ← *archive* + *testSuite*
13: *selectedT S* ← *selectedTS* + *testSuite*
14: **end if**
15: **if** *coverage* ≥ *minCoverage* **then**
16: *bestSolutions* ← *bestSolutions* + *testSuite*
17: **end if**
18: **end for**
19: *P* ← *generateNewPopulation(selectedT S)*
20: *generation* ← *generation* + *1*
21: **until** *generation* = *N*
22: **return** *bestSolutions*

Порівняння результатів одноцільових оптимізацій методів на базі пошуку та вбудованих GCC оптимізацій

	O1	O2	O3	Ofast	RS	GA	NS
S	1.051	1.107	1.107	1.103	1.121	1.143	1.365
MR(%)	4.8	-8.4	4.2	6.1	10.70	15.2	15.6
CR(%)	-1.3	-5	3.4	-5	18.2	22.2	23.5



Висновки

В результаті виконання дипломної роботи було проведено системний аналіз в області не функційного тестування генераторів коду.

З точки зору розробки програмного забезпечення, ця робота сприяє підвищенню якості та надійності генераторів. Зокрема, в ній запропоновано алгоритм, який допомагає розробникам/супроводжувачам генераторів ефективно тестувати створені ними генератори коду та таким чином надавати кінцевим користувачам докази якості згенерованого коду, також користувачам генераторів надано засоби для ефективного автоматичного налаштування вихідних генераторів для створення високоякісного коду.

Представлений метод може бути використаний при розробці нових генераторів коду для виявлення помилок та їх усунення.

Публікації

- Дьоміна А. І. Використання методу пошуку новизни (Novelty Search) для автоматичної генерації тестових даних/ А. І. Дьоміна // XIV Всеукраїнська науково-практична конференція “Актуальні проблеми комп’ютерних наук (АПКН – 2022)”/ ХНУ - Хмельницький, 2022
- Дьоміна А. І. Використання методу пошуку новизни (Novelty Search) для автоматичної генерації тестових даних/ А. І. Дьоміна // І Міжнародна науково-практична інтернет-конференція«Проблеми та перспективи розвитку сучасної науки в країнах Європи та Азії»/ Університет Григорія Сковороди в Переяславі - Переяслав-Хмельницький, 2022

Дякую за увагу!

UNICHECK

Ім'я користувача:
Кафедра ІПЗ

ID перевірки:
1013180189

Дата перевірки:
05.12.2022 08:37:10 EET

Тип перевірки:
Doc vs Internet + Library

Дата звіту:
05.12.2022 08:37:59 EET

ID користувача:
100005589

Назва документа: **запискаДьоміна_без дод**

Кількість сторінок: **165** Кількість слів: **35450** Кількість символів: **281464** Розмір файлу: **2.67 MB** ID файлу: **1012944936**

5.12%
Схожість

Найбільша схожість: **0.91%** з джерелом з Бібліотеки (ID файлу: **1012911671**)

4.29% Джерела з Інтернету	222	Сторінка 167
1.18% Джерела з Бібліотеки	143	Сторінка 170

0% Цитат

Вилучення цитат вимкнене

Вилучення списку бібліографічних посилань вимкнене

0%
Вилучень

Немає вилучених джерел

Модифікації

Виявлено модифікації тексту. Детальна інформація доступна в онлайн-звіті.

Замінені символи	8

Mon Dec 05 11:10:59 EET 2022, Хіврич Володимир Русланович, Хмельницький національний університет, ХНУ

Anti-Plagiarism v-15.257

Максимальне співпадіння з одним документом 8.0%

Словники перевірки: en_US, ru_RU, ua_UA. **Помилок в документах: 11%**

ID: 108934 Назва: КРМ на тему: "нефункційного тестування та автоматичного налаштування генераторів кодів з урахуванням програмно-апаратної платформи" Додано в БД: 2022-12-05 Автора: Дьоміна А.І. Керівники: Гурман І.В. Консультанти: Опоненти:	Документ		Сумарний збіг по Базі Даних	
	Символи	Лексеми	Символи	Лексеми
	249341	1918	22381 (9%)	187 (10%)

Джерело плагіату

ID	Опис	Наявність плагіату в документі	
		Символи	Лексеми

РІШЕННЯ ЕКСПЕРНОЇ КОМІСІЇ

КАФЕДРИ ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

ПРО ДОПУСК КВАЛІФІКАЦІЙНОЇ РОБОТИ ДО ЗАХИСТУ

Підтверджуємо ознайомлення з результатом звіту подібності щодо роботи. генерованого системою виявлення текстових збігів/ідентичності/схожості:

Назва: «Метод нефункційного тестування та автоматичного налаштування генераторів кодів з урахуванням програмно-апаратної платформи»

Автор: Дьоміна Анастасія Іванівна

Спеціальність: 121 – Інженерія програмного забезпечення

Освітня програма: освітньо-професійна програма Інженерія програмного забезпечення

Науковий керівник: Гурман Іван Васильович, к.т.н., доц.

Після аналізу звіту подібності зроблено такий висновок:

№	Висновок	Позначка про відповідність
1	Запозичення, виявлені в роботі, є законними і не є плагіатом (далі – зазначаються підстави віднесення запозичень до правомірних). Робота приймається до захисту.	Відповідає
2	Виявлені запозичення не є плагіатом, розміщені в розділах, які не описують безпосередньо авторське дослідження, але кількість цитат перевищує обсяг, виправданий поставленою метою роботи Робота приймається до захисту, але має бути відкоригована. Відкоригований варіант має бути поданий на кафедру за 2 дні до захисту, разом із заявою щодо самостійності виконання письмової роботи та ідентичності друкованої та електронної версії роботи	
3	Виявлені запозичення не є плагіатом, але частково розміщені в розділах, які описують безпосередньо авторське дослідження, а кількість цитат перевищує обсяг, виправданий поставленою метою роботи. В зв'язку з цим мета роботи та поставлені завдання не були досягнені. Робота може бути допущена до захисту (наступного року) після того як буде відкоригована та допрацьована і успішно пройде повторну перевірку на академічний плагіат.	
4	Робота містить навмисні текстові спотворення, передбачувані спроби укриття запозичень або інші прояви академічного плагіату. Робота містить фабрикацію або фальсифікацію даних. Робота не допускається до захисту.	
5	Інше:	

Підтвердження:

Сумарні співпадіння документа складають 5.12%

Серед співпадінь з документами у глобальній мережі максимальне співпадіння з одним документом становить 0,73% та стосуються переліку джерел посилань.

Серед співпадінь з документами з бібліотеки максимальне співпадіння з одним документом становить 0.91% та стосуються стандартних сторінок пояснювальних записок тобто – титульний аркуш, бланк завдання, тощо.

Керівник _____ І. В. Гурман

Гарант ОП _____ О. М. Яшина

Завідувач кафедри _____ Л. П. Бедратюк

ХМЕЛЬНИЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

РЕЦЕНЗІЯ НА ДИПЛОМНУ РОБОТУ
освітнього ступеня «Магістр»

Магістр Дьоміна Анастасія Іванівна

Тема Метод нефункційного тестування та автоматичного налаштування генераторів кодів з урахуванням програмно-апаратної платформи

Спеціальність 121 – Інженерія програмного забезпечення

Обсяг дипломного проекту:

Кількість сторінок записки

1.Короткий зміст пояснювальної записки та прийнятих рішень У дипломній роботі виконано аналіз різних методів тестування та автоналаштування генераторів коду. Визначено функціональні задачі та розроблено ТЗ. Проведено аналіз існуючих інструментів та технологій тестування генераторів коду та розроблено алгоритм пошуку нових тестових випадків на основі методу пошуку новизни. На основі яких розроблено програму для автоналаштування генераторів коду залежно програмно апаратного забезпечення. Після розробки було проведено тестування розробленого методу та його порівняння з уже існуючими рішеннями.

2. Висновок про відповідність проекту поставленому завданню Дипломна робота освітнього ступеня "магістр" у повній мірі відповідає поставленому завданню як у теоретичній, так і в практичній її частині.

3. Характеристика виконання кожного розділу проекту, ступінь використання останніх досягнень науки і техніки та передових методів роботи У вступі описано проблему предметної області, обґрунтовано актуальність теми, сформульовано мету і завдання дипломного проекту. У першому розділі: проведено аналіз функціональних і структурних особливостей та специфіки предметної області У другому розділі виконано порівняння існуючих рішень; визначено функціональні задачі додатку, на основі яких було створено функціональні та нефункціональні вимоги. У третьому розділі розроблено метод тестування оснований на алгоритмі пошуку новизни. У четвертому розділі виконана програмна реалізація запропонованого методу за допомогою сучасних та надійних інструментів. В останньому розділі проведено модульне, функціональне та конфігураційне тестування розробленого методу та його порівняння з уже існуючими рішеннями.

4. Позитивні сторони проекту Дипломна робота містить інноваційні рішення, зокрема зі сторони підходу до тестування з використанням методу пошуку новизни. Розроблена програма має унікальний функціонал та підхід до методу налаштування генераторів коду. Також перевагою розробленого методу є застосування пошуку нових тестових випадків в областях де існуючі методи цього не роблять.

5. Негативні сторони проекту Розроблений метод потребує подальшого вивчення та модернізації для можливості його більш широкого використання. Недолік не є суттєвим, та не зменшує позитивне враження від роботи

6. Оцінка графічного оформлення та пояснювальної записки проекту Графічне оформлення виконано відповідно до теми дипломної роботи з дотриманням всіх вимог та стандартів. У загальному графічне оформлення виконане на хорошому рівні. Пояснювальна записка відповідає усім вимогам стандартів до її оформлення.

7. Відгук про дипломний проект в цілому В цілому дипломна робота заслуговує позитивної оцінки. Матеріал подано структуровано, чітко та послідовно. Описи процесів аналізу, проектування, реалізації і тестування подано відповідно до структури записки. Графічний матеріал ілюструє актуальність інноваційність та практичну цінність рішень, які були прийняті для вирішення поставленої задачі.

8. Інші зауваження

9. Оцінка дипломного проекту Дипломний проект виконаний в повному обсязі. Беручи до уваги всі зазначені переваги і недоліки, можна зробити висновок, що робота заслуговує оцінки «відмінно»(4.75/A).

РЕЦЕНЗЕНТ (прізвище, ім'я, по-батькові, посада, місце роботи) Нічепорук Андрій Олександрович, кандидат технічних наук, доцент кафедри комп'ютерної інженерії та системного програмування (КІСП) ХНУ

"____" _____ 2022 р. _____
(підпис)

Завідувачу кафедри інженерії програмного
забезпечення проф. Бедратюку Л. П.
здобувача вищої освіти
Дьоміної А.І.
факультет ІТ, 2 курс, група ІПЗм-21-1

ЗАЯВА

З правилами чинного Положення «Про дотримання академічної доброчесності в Хмельницькому національному університеті» від 26.09.2020 (зі змінами від 26.11.2020), згідно з яким виявлення плагіату є підставою для відмови в допуску кваліфікаційної роботи до захисту та застосування заходів дисциплінарної та академічної відповідальності, ознайомлена. Про використання програмно-технічних засобів для перевірки кваліфікаційних робіт здобувачів вищої освіти на плагіат оповіщена та надаю свою згоду на обробку та збереження університетом моєї роботи в інституційному репозитарії університету.

Також надаю університету право на передачу моєї роботи для обробки та збереження в базах даних програмно-технічних засобів (Unicheck та Anti-Plagiarism) та використання роботи для виявлення плагіату в інших роботах, які перевіряються програмно-технічними засобами та користувачами, що мають доступ до цих програмно-технічних засобів, виключно в обмежених цілях для виявлення плагіату в текстах робіт.

Робота для перевірки університетом надається в друкованому та електронному варіанті. Електронна версія моєї роботи збігається (ідентична) з друкованою.

23.11.2022
дата

підпис

Завідувачу кафедри
інженерії програмного забезпечення
проф. Бедратюку Л. П.
студента групи ІПЗм-21-1
Дьоміної А.І.
Прізвище, ініціали

ЗАЯВА

Прошу закріпити за мною тему дипломної роботи освітнього ступеня «магістр» за спеціальністю 121 «Інженерія програмного забезпечення»:
Метод нефункційного тестування та автоматичного налаштування генераторів кодів з урахуванням програмно-апаратної платформи

(керівник дипломної роботи – Гурман Іван Васильович)
Прізвище, ім'я, по батькові

30.06.2022
Дата

Підпис студента

СУПРОВІДНИЙ ЛИСТ

Назва роботи	Метод нефункційного тестування та автоматичного налаштування генераторів кодів з урахуванням програмно-апаратної платформи
Автор	Дьоміна Анастасія Іванівна
Тематика	121 Інженерія програмного забезпечення
Ключові слова	Алгоритм пошуку новизни
	Автоматична генерація тестових даних
	Задачі генерації даних
	Генетичні алгоритми
Дата публікації	2022
Видавництво	Хмельницький національний університет
Бібліографічний опис	Дьоміна А.І.. Метод нефункційного тестування та автоматичного налаштування генераторів кодів з урахуванням програмно-апаратної платформи : дипломна робота магістра : 121 Інженерія програмного забезпечення / А.І. Дьоміна; Хмельниц. нац. ун-т. – Хмельницький, 2022. – 155 с.
Короткий огляд (реферат)	У дипломній роботі проаналізовано існуючі підходи та методи для тестування ти автоналаштування генераторів кодів, з урахуванням програмного та апаратного забезпечення. Досліджено придатність методу пошуку новизни, для покращення нефункційного тестування генераторів коду шляхом пошуку невикористаних тесткейсів. Розроблено автотюнер для компіляторів CAT, який знаходить оптимальні налаштування для конкретного програмно-апаратного забезпечення . Проведена оцінка продуктивності методу пошуку новизни для оптимізації компілятора. Виконано оцінювання розробленого методу з точки зору ефективності, порівнюючи його з існуючими рішеннями на базі GCC.
Тип вмісту	Магістерські роботи
Кафедра	Інженерії програмного забезпечення