КВАЛІФІКАЦІЙНА РОБОТА

Група МІПЗс-22

Гончарик Н.І.

2024

ЗВО УНІВЕРСИТЕТ КОРОЛЯ ДАНИЛА

Факультет суспільних та прикладних наук

Кафедра інформаційних технологій

на правах рукопису

Гончарик Назар Іванович

УДК 004.4

Оптимізація моделей та методів рішень на основі віртуальних машин для покращення моніторингу та аналізу додатків

Спеціальність 121 – «Інженерія програмного забезпечення»

Кваліфікаційна робота на здобуття кваліфікації магістра

Нормоконтроль

_____Стисло О.В.

(підпис, дата, розшифрування підпису)

Студент

_____ Гончарик Н.І.

(підпис, дата, розшифрування підпису)

Допускається до захисту

Завідувач кафедри

_____к.т.н., доц. Ващишак С.П.

(підпис, дата, розшифрування підпису)

Керівник роботи

_____ к.т.н., доц. Демчина М.М.

(підпис, дата, розшифрування підпису)

Івано-Франківськ – 2024

ЗВО УНІВЕРСИТЕТ КОРОЛЯ ДАНИЛА

Факультет суспільних та прикладних наук

Кафедра інформаційних технологій

Освітній ступінь: «магістр»

Спеціальність: 121 «Інженерія програмного забезпечення»

ЗАТВЕРДЖУЮ

Завідувач кафедри

_____

« 19 » лютого 2024 року

ЗАВДАННЯ

НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТУ

Гончарик Назару Івановичу

(прізвище, ім'я, по батькові)

1. Тема кваліфікаційної роботи

Оптимізація моделей та методів рішень на основі віртуальних машин для покращення моніторингу та аналізу додатків

керівник роботи:

Демчина Микола Миколайович, кандидат технічних наук, доцент

затверджена наказом вищого навчального закладу від « 26 » червня 2023 року № 32/1 с

2. Термін подання студентом роботи 16.02.2024

3. Вихідні дані роботи: Формальні моделі, методи та алгоритми.

4. Зміст кваліфікаційної роботи (перелік питань, які потрібно розробити)

1. Огляд технології віртуальних машин для моніторингу розробки додатків.
2. Дослідження моделей статичного аналізу типів.
3. Дослідження концепції трасування маркерів при аналізі додатків
4. Реалізація фреймворку трасування маркерів.

5. Дата видачі завдання 29.06.2023

КОНСУЛЬТАНТИ РОЗДІЛІВ КВАЛІФІКАЦІЙНОЇ РОБОТИ

Розділ	Консультант (прізвище, ініціали та посада)	Позначка консультанта про виконання розділу	
		підпис	дата

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Термін виконання етапів роботи	Примітка
1.	Огляд та аналіз технології віртуальних машин для моніторингу розробки додатків	26.09.2023	Виконано
2.	Дослідження моделей статичного аналізу типів	20.10.2023	Виконано
3.	Сутність концепції трасування маркерів при розробці та аналізі програмних додатків	15.11.2023	Виконано
4.	Реалізація фреймворку трасування маркерів	30.11.2023	Виконано
5.	Формування висновків	09.12.2023	Виконано
6.	Оформлення пояснювальної записки	22.12.2023	Виконано
7.	Оформлення графічного матеріалу та підготовка до захисту роботи	11.01.2024	Виконано

Студент _____ Гончарик Н.І.
(підпис) (прізвище та ініціали)

Керівник роботи _____ Демчина М.М.
(підпис) (прізвище та ініціали)

Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень)

Сторінка	Опис графічного матеріалу	Сторінка	Опис графічного матеріалу
15	Відстеження входу та виходу з циклу в програмі	55	Структура дескриптора маркера
21	Огляд архітектури JVM	56	Стек маркерів для кожного потоку
22	Структура файлу класу Java	58	Адаптивний онлайн-аналіз програм: підхід MPS
23	Завантажувачі класів у JVM	59	Адаптивний онлайн-аналіз програм: підхід MDP
24	Спрощений вигляд циклу інструкцій інтерпретатора	60	Потік керування преамбулою відправки для кожного байт-коду маркера

АНОТАЦІЯ

Кваліфікаційна робота присвячена оптимізації моделей та методів рішень на основі віртуальних машин для покращення моніторингу та аналізу додатків шляхом розробки дизайну і реалізації гнучкої динамічної системи трасування, тобто MTF, для програмних додатків на основі віртуальних машин, що надає клієнтам аналізу можливість визначати нові типи подій на основі розташування програм, представлених маркерами.

В першому розділі проаналізовано виконано огляд технології віртуальних машин для моніторингу та аналізу додатків, описані платформи трасування маркерів для аналізу додатків. Виконано дослідження технології сучасних віртуальних машин Java, наведено опис реалізації специфікації HotSpot JVM, інтерфейс віртуальної машини Java та концепці. динамічного трасування для моніторингу поведінки програмних додатків.

В другому розділі проведено дослідження моделей статичного аналізу типів аналізу додатків , виконано аналіз станів типу виконання в об’єктно-орієнтованому програмуванні. Реалізовано моделі процесу Adaptive Online Program Analysis (AOPA) та здійснена оцінка адаптивного аналізу динамічного стану об’єкта.

В третьому розділі проведена оптимізація моделей на основі віртуальних машин для покращення аналізу додатків шляхом реалізації фреймворку трасування маркерів, наведена сутність концепції трасування маркерів при розробці та аналізі програмних додатків. Приведена архітектура фрейворку MFT, реалізовано фреймворк трасування маркерів, здійснена оцінка продуктивності системи трасування маркерів.

КЛЮЧОВІ СЛОВА: ВІРТУАЛЬНА МАШИНА, ДИНАМІЧНИЙ СТАН, ТРАСУВАННЯ МАРКЕРІВ, СПЕЦИФІКАЦІЯ HOTSPOT, АДАПТИВНИЙ АНАЛІЗ, ПРОДУКТИВНІСТЬ СИСТЕМИ, БАЙТ-КОД.

SUMMARY

The qualification work is devoted to the optimization of virtual machine-based decision models and methods to improve application monitoring and analysis by developing the design and implementation of a flexible dynamic tracing system, i.e. MTF, for virtual machine-based software applications, which enables analysis clients to identify new types of events based on location of programs represented by markers.

In the first section, an overview of the technology of virtual machines for application monitoring and analysis is analyzed, and marker tracing platforms for application analysis are described. A study of the technology of modern Java virtual machines is carried out, a description of the implementation of the HotSpot JVM specification, the Java virtual machine interface and concepts are given. dynamic tracing for monitoring the behavior of software applications.

In the second section, a study of models of static analysis of types of application analysis was carried out, an analysis of execution type states in object-oriented programming was performed. Models of the Adaptive Online Program Analysis (AOPA) process were implemented and an assessment of the adaptive analysis of the object's dynamic state was made.

In the third section, the optimization of models based on virtual machines is carried out to improve the analysis of applications by implementing the marker tracing framework, the essence of the concept of marker tracing in the development and analysis of software applications is given. The architecture of the MFT framework is presented, the marker tracing framework is implemented, and the performance of the marker tracing system is evaluated.

KEYWORDS: VIRTUAL MACHINE, DYNAMIC STATE, MARKER TRACKING, HOTSPOT SPECIFICATION, ADAPTIVE ANALYSIS, SYSTEM PERFORMANCE, BYTECODE.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

MTF - marker tracing framework

AOPA - Adaptive Online Program Analysis

SPCC - Selective Probabilistic Calling Context Analysis

VM - Virtual Machine

JVMTI - Java Virtual Machine Tool Interface

API - Application Programming Interface

AOP - Aspect-Oriented Programming

OS - operating system

JIT - Just-in-Time

MTF - marker tracing framework

PCC - Probabilistic Calling Context

GC - garbage collection

ELF - Executable and Linkable Format

COFF - Common Object File Format

oopDesc - Ordinary Object Pointer descriptor

BCI – Bytecode Instrumentation

FSA - finite-state automaton

REGEX - regular expressions

JDI - Java Debugger Interface

ODE - object death event

OSR - on-stack replacement

MPS - Method Pointer Swapping

MDP – Marker Preamble Dispatching

DTrace - Dynamic Tracing Framework

ВСТУП

Актуальність дослідження. Найбільш важливим застосуванням віртуальної машина Java (Jvm) є саме її серверна частина імплементації. Обмеження пам’яті на сервері не такі жорсткі, як на інших платформах. З іншого боку, Jvm для серверів має задовольняти вимоги, як наведені нижче, і які не є такими строгими для клієнта, або вбудованого Jvm зокрема:

1. Експлуатація високопродуктивних процесорів. Поточні компілятори «точно вчасно» (JIT) не застосовують широкі оптимізації для використання сучасних функцій апаратного забезпечення (ієрархія пам’яті, паралелізм на рівні структур, багатопроцесорний паралелізм тощо), які необхідні для отримання продуктивностей порівняних зі статично скомпільованими мовами.

2. Масштабованість SMP — мультипроцесори зі спільною пам'яттю. Конфігурації (SMP) дуже популярні для серверів. Деякі Jvms безпосередньо відображають потоки Java на рівні потоків операційної системи. Це призводить до поганої масштабованості багатопоточної Java програми на SMP рівні, у випадку коли кількість Java ниток збільшується.

3. Обмеження потоків — необхідно багато серверних програм для створення нових потоків для кожного вхідного запиту. Однак через обмеження операційної системи деякі Jvms не можуть створити велику кількість потоків і, отже, можуть працювати лише з обмеженою кількістю одночасних запитів. Ці обмеження сильно обмежують виконувані програми.

4. Постійна доступність — серверні програми повинні мати можливість задовольняти вхідні запити під час роботи постійно, протягом тривалого часу, що є пріоритетом для поточних Jvms імплементацій.

5. Швидке реагування — більшість серверних програм мають строгі вимоги до часу відповіді (наприклад, принаймні 90 відсотків запитів повинні обслуговуватися в менший термін ніж секунда). Однак багато поточних Jvms

на фоні неінкрементного збирання сміття призводять до серйозних збоїв часу відгуку.

6. Використання бібліотеки — для серверних програм, написаних мовою Java, код зазвичай базується на існуючих бібліотеках (біни, фреймворки, компоненти тощо), і не є написаним «з нуля». Однак, оскільки ці бібліотеки написані для обробки загальних випадків, вони часто погано працюють на поточних Jvms.

7. Витончена градація — відповідно до запитів, сервер перенасичений можливостями для їх виконання, це прийнятна основа для продуктивності сервера.

Мета і задачі дослідження. Метою кваліфікаційної роботи є реалізація фреймворку трасування маркерів для архітектури фреймворку MFT та оцінка продуктивності системи динамічного трасування для оптимізації моделей та методів рішень для покращення моніторингу та аналізу додатків.

Для досягнення поставленої мети необхідно розв’язати такі задачі:

1. Виконати огляд технологій віртуальних машин.
2. Проаналізувати моделі статичного аналізу типів.
3. Виконати оптимізацію моделі процесу Adaptive Online Program Analysis;
4. Виконати реалізацію фреймворків трасування маркерів для оптимізації моделі моніторингу розробки програмних додатків.

Об'єктом дослідження є самі моделі статичного аналізу типів та аналіз станів типу виконання в реалізації процесів Adaptive Online Program Analysis.

Предметом дослідження є оптимізація моделі та алгоритмів реалізації специфікації HotSpot JVM, імплементація інтерфейсу віртуальної машини Java в контексті концепції динамічного трасування.

Методи дослідження базуються на використанні методів динамічного програмування, трасування маркерів, засоби алгебри логіки, методи оптимізації та оцінки продуктивності.

Наукова новизна одержаних результатів полягає у тому, що на основі аналізу предметної області було реалізовано структуру Marker Tracing Framework (MTF), для оптимізації моніторингу програмних додатків для мов на основі віртуальних машин, що забезпечує надійну інфраструктуру для розробки детального аналізу динамічних програм на основі трасування.

Практичне значення одержаних результатів полягає в застосуванні клієнта динамічного аналізу станів типів на основі автомата кінцевого стану (FSA), який використовує потужність і корисність MTF та використанні оптимізації Adaptive Online Program Analysis (AOPA) до клієнта, щоб продемонструвати переваги розробки програмного аналізу всередині JVM.

Апробація результатів дослідження. Матеріали дослідження було представлено у матеріалах I Всеукраїнської науково-практичної інтернет конференції “IT екосистема: цифровізація бізнес-процесів в умовах війни”, у тезах доповіді “Концепція хмарних міграцій та консолідацій”.

Структура. Кількість розділів – 3. Загальний обсяг основної частини – 83 сторінок. Список використаних джерел містить – 54 позиції.

РОЗДІЛ 1. ОГЛЯД ТЕХНОЛОГІЇ ВІРТУАЛЬНИХ МАШИН ДЛЯ МОНІТОРИНГУ ТА АНАЛІЗУ ДОДАТКІВ

1.1 Аналіз предметної області використання віртуальних машин в програмуванні

Спостережливість представляє рівень підтримки всередині комп'ютерних систем для точного захоплення, аналізу та представлення внутрішньої інформації, наприклад, структур даних і станів програм, про систему. Інструменти спостереження допомагають розробникам програм зрозуміти код, усунути проблеми, діагностувати вузькі місця продуктивності та виконати оптимізацію. Традиційні рішення для спостереження, включаючи твердження та налагоджувачі, корисні в багатьох випадках. Наприклад, більшість налагоджувачів підтримують перевірку програмних змінних, машинних регістрів і поточних стеків викликів у точках зупинки. Інструкції друку, створені вручну, зручніші там, де стан не можна легко зафіксувати точкою зупину. Однак такі рішення є або дорогими та грубими (налагоджувачі), або статичними та нав’язливими (оператори друку та твердження), що, як наслідок, перешкоджає корисності таких рішень у адаптації сучасних дедалі складніших програмних систем.

Впровадження мов на основі віртуальної машини (VM), наприклад Java і C#, покращило спостережуваність програмного забезпечення, пропонуючи нові інфраструктури. Наприклад, Java Virtual Machine Tool Interface (JVMTI) [2] — це комплексний інтерфейс прикладного програмування (API) для реалізації клієнтів аналізу, які можуть перевіряти внутрішні стани JVM і керують виконанням програм Java. Крім того, підтримка DTrace [1], вбудована у віртуальну машину Java HotSpot , також надає набір зондів подій для клієнтів аналізу з низькими накладними витратами. .NET Profiling API пропонує подібні функції, як і JVMTI для мов .NET. Аспектно-орієнтоване

програмування (AOP) також можна використовувати для систематичного інструментування програм за допомогою спеціальних операцій.

Поширення таких інструментальних API для мов на основі віртуальних машин можна пояснити віртуалізованими середовищами, в яких виконуються програми. Оскільки рідні мови, наприклад, C і Fortran, виконуються безпосередньо на голій машині, для спостереження за цими мовами потрібна підтримка операційної системи (ОС) і апаратного забезпечення, які рідко доступні, якщо взагалі практичні. Навпаки, набагато легше та краще розширити мовну віртуальну машину для спостереження, оскільки такі віртуальні машини не лише надають усі основні служби для програм, але й керують їх повним виконанням. Наприклад, віртуальна машина знає статус кожного використовуваного блокування, оскільки програми покладаються на віртуальні машини для виконання всіх операцій, пов’язаних із блокуванням.

Незважаючи на те, що існуючі інструментальні API виявилися корисними [10, 12] вони все ще обмежені для реалізації детальних і складніших клієнтів аналізу з наступних причин:

- Типи подій. Для кращої модульності та простоти програмування більшість із цих інфраструктур приховують деталі віртуальних машин, надаючи API для написання програмного інструментарію. Незважаючи на вичерпний список підтримуваних типів подій, вони представляють лише підмножину подій, які розробники можуть захотіти спостерігати. Наприклад, розробник може захотіти спостерігати кожен раз, коли програма входить або виходить з певного циклу. Для цього розробнику потрібно створити клієнт аналізу, який відстежує вхід і вихід із циклу в програмі та викликає відповідний зворотний виклик обробників, як це показано на рисунку 1.1.

Однак жодні існуючі інструменти не підтримують таку подію. Крім того, існуючі інструменти не підтримують аналіз, який може вказати, який саме цикл слід відстежувати. Цей тип аналізу потребує інструментальної підтримки на рівні базового блоку, тоді як більшість існуючих API забезпечують інструментальну підтримку на рівні методу.

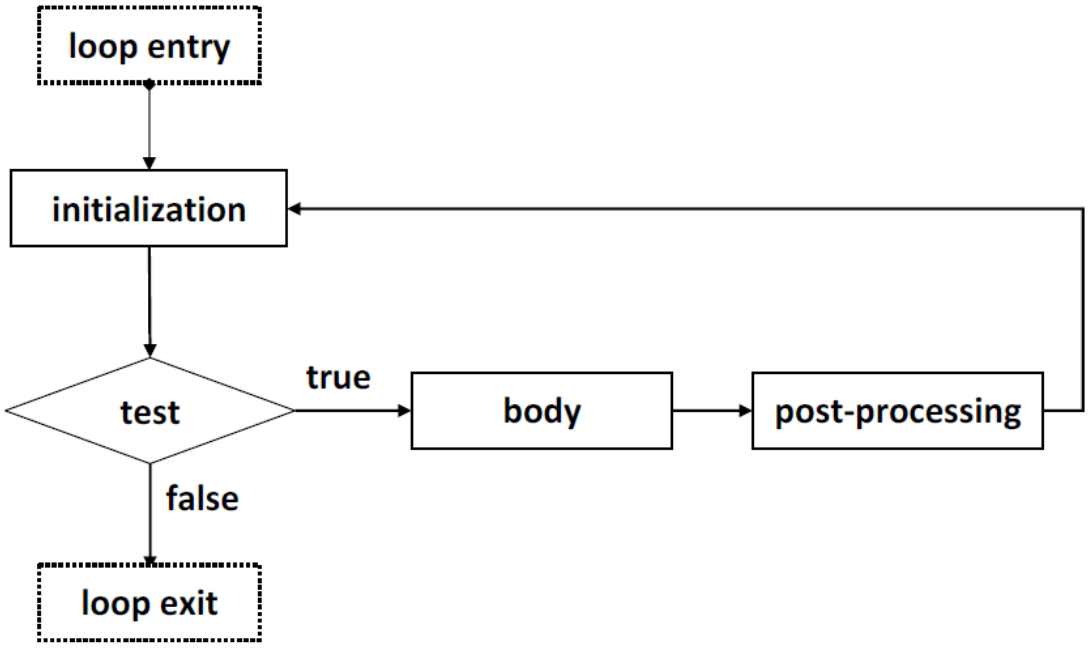


Рисунок 1.1 – Відстеження входу та виходу з циклу в програмі

- Фільтрування подій. Щоб мінімізувати накладні витрати на простір і час, клієнти аналізу вимагають здатності відфільтрувати випадки нецікавих подій від цікавих (наприклад, виклик обробника лише під час виклику методу A або методу B, пропускаючи виклики будь-яких інших методів). Більшість існуючих API дійсно дозволяють відстежувати події входу та виходу методу. Тим не менш, лише кілька методів, наприклад DTrace, підтримують визначені користувачем фільтри для виключення небажаних випадків. Без такої функції інструментарій може викликати надмірну кількість сповіщень про події, оскільки сповіщення надходять про всі події, але фільтрація виконується лише на стороні клієнта. Під час моніторингу дуже частих подій на стороні клієнта фільтрація викликає значні накладні витрати через часті перемикання контексту, де більшість операцій віртуальної машини мають бути зупинені.

JVMTI підтримує інструментарій байт-коду (через інструменти сторонніх розробників), який можна використовувати для вставки спеціального коду навколо циклів для захоплення такої події.

Крім того, DTrace фільтрує небажані події шляхом опитування всіх подій, що призводить до великих накладних витрат на виконання.

- Інформація про контекст. Більшість існуючих API надають різноманітну внутрішню інформацію віртуальної машини за допомогою повного набору службових процедур. Викликаючи такі підпрограми, клієнти аналізу можуть запитувати широкий спектр контекстної інформації про більшість аспектів виконання програми, наприклад, статус потоку, поточне трасування стека, використання монітора об’єктів та доступні об’єкти купи. Тим не менш, все ще неможливо надати певну контекстну інформацію для всіх клієнтів аналізу в реальному світі. Наприклад, розподіл об’єктів і аналіз тривалості життя потребують відстеження сайтів, де виділено кожен об’єкт, тобто сайтів розподілу. Хоча існуючі API підтримують відстеження розподілу об’єктів і подій відновлення, вони не ідентифікують і не відстежують відповідні сайти розподілу.

В даній роботі ми представляємо інфраструктуру, яка знаходиться у віртуальній машині, щоб усунути вищезазначені недоліки в існуючих API інструментів. Мета полягає в тому, щоб забезпечити кращу спостережуваність для мов на основі віртуальних машин за допомогою гнучкої та загальної структури для впровадження більш детальних клієнтів аналізу з низькими витратами.

Спостережливість шляхом використання інформації віртуальної машини. Віртуальні машини на мовах високого рівня (віртуальні машини) — це програмні системи, що підтримують виконання керованих мов, наприклад, Java і C#. Оскільки віртуальні машини служать повним середовищем виконання, вони можуть генерувати багату інформацію під час виконання програми виконання. Крім того, мовна віртуальна машина сама по собі є потужною інфраструктурою з багатьма корисними засобами, наприклад, збирачами сміття та компіляторами Just-in-Time (JIT). Попередні дослідження використовували інформацію про час виконання та засоби віртуальної машини, щоб зробити програми більш ефективними [37]. У цій роботі ми

демонструємо, що така експлуатація також може призвести до покращення спостережуваності програми.

1.2 Опис платформи трасування маркерів для аналізу додатків

Ми будемо розглядати інфраструктуру під назвою: платформа трасування маркерів (MTF - marker tracing framework), яка підтримує детальне трасування програм, дозволяючи клієнтам аналізу визначати спеціальні події та реєструвати обробники.

MTF — це загальна структура, яка повністю не залежить від семантики та процедури обробки кожної визначеної користувачем події. Крім тонкої деталізації специфікацій, MTF також розроблений, щоб бути легким, тобто мати низькі накладні витрати на виконання та ¬ненав’язливий. MTF надає засоби всередині віртуальної машини для виконання важливих операцій для клієнтів аналізу, наприклад, завантаження класів, зворотного виклику та керування контекстом, відправлення подій і компіляції JIT.

Клієнти Analysis визначають спеціальні події шляхом реалізації нових маркерів — спеціальних мовних конструкцій для гнучкого визначення точок інструментів і областей. Маркер складається з ідентифікатора події та метаданих. Кожен маркер інкапсулює довільну область коду в програмі. Клієнти аналізу інструментують програми маркерами під час компіляції. Під час виконання MTF може шукати та викликати відповідний обробник на основі ідентифікатора маркера для кожного входження маркера. Кілька клієнтів аналізу можуть виконуватися паралельно без перешкод, навіть якщо вони реєструють спільні маркери.

Перебуваючи всередині віртуальної машини, клієнти аналізу мають доступ до всієї інформації, доступної віртуальній машині, реалізуючи постачальників контексту — спеціалізовані модулі віртуальної машини для збору необхідної інформації про час виконання клієнтами аналізу. Розробники аналізу можуть повторно використовувати існуючі

постачальники контексту або впроваджувати нові постачальники, коли це необхідно. Постачальники контексту здебільшого загальні та багаторазово використовувані, тому ми можемо комбінувати та поєднувати провайдери контексту для створення складного аналізу часу виконання.

MTF може пом’якшити проблему типу фіксованих подій у існуючих API, дозволяючи клієнтам аналізу визначати спеціальні події, пов’язані з трасуванням. Наприклад, клієнт аналізу може відстежувати кожен цикл у програмі, обернувши його маркером, який потім сигналізує MTF у заголовку циклу та вихід відповідно. Маркери також служать дрібнозернистими фільтрами в програмі, де активовано інструментарій, тоді як решта програми виконується на повній швидкості.

Як приклад, ми можемо реалізувати розширений аналіз ймовірнісного контексту виклику (PCC - Probabilistic Calling Context) [20], який може вибірково обчислювати значення PCC лише для методів, визначених користувачем. Оскільки постачальники аналізу на основі MTF і контексту вбудовані всередину віртуальної машини, вони можуть використовувати служби віртуальної машини для доступу до будь-якої контекстної інформації під час виконання.

Незважаючи на те, що модифікація віртуальної машини є нетривіальною та непереносимою, ми стверджуємо, що переваги цього рішення (наприклад, доступ до інформації про час виконання лише віртуальної машини, ефективне виконання інструментів, менше перемикань контексту) значно переважують недоліки в багатьох сценаріях для покращеної спостережуваності.

Ефективність системи відстеження маркерів. Щоб продемонструвати корисність і універсальність фреймворку, ми будемо розглядати два різних клієнти аналізу, тобто динамічний аналіз стану типів з адаптивним онлайн-аналізом програм (AOPA) [27] і вибірковий імовірнісний аналіз контексту виклику [20]. Ми оцінюємо продуктивність як MTF, так і клієнта typestate.

MTF і клієнти аналізу реалізовані для мови Java на HotSpot JVM. Проте ми вважаємо, що принципи та методології мають однаково добре застосовуватися для інших мов на основі віртуальних машин із порівнянною ефективністю та продуктивністю під час виконання.

1.3 Дослідження технології сучасних віртуальних машин Java (JVM)

Для підвищення надійності, ефективності виконання, портативності та відповідності стандартам більшість сучасних JVM стали дуже складними (наприклад, поточна версія Sun HotSpot VM містить понад півмільйона рядків коду). Крім того, за останні кілька років додатки, що працюють на цих віртуальних машинах, також ускладнилися, оскільки розробники намагаються використовувати паралелізм на рівні потоку, доступний у сучасних багатоядерних процесорах. Таким чином, розробникам було складно спостерігати за роботою внутрішніх механізмів і структур JVM і розуміти, як ці складні системи виконання взаємодіють із їхніми програмами.

Попередні дослідження показали, що використання багатої інформації про час виконання, захованої в JVM, може надати розробникам необхідну інформацію для покращення якості та продуктивності їх програмного забезпечення.

Таким чином, ми вважаємо, що використання такої інформації значно переважує технічні складнощі розробки засобів отримання інформації. Тому ми вдосконалили структуру спостережуваності всередині HotSpot JVM, продуктивної потужної JVM, яка широко розгортається в комерційних умовах. Щоб полегшити розуміння проектних рішень, у цьому розділі розглядаються відповідні технології JVM загалом і HotSpot.

Віртуальна машина Java. Java розроблена як мова високого рівня, що виконується поверх віртуалізованого середовища — віртуальної машини Java. Такий додатковий рівень абстракції є основою філософії мобільності Java —

«Напиши один раз, запусти скрізь». Таким чином, програми Java не перекладаються в залежні від платформи машинні інструкції, як програми, написані рідними мовами, наприклад, C або C++, а в Java Bytecodes [47], віртуальний і нейтральний до платформи набір інструкцій, придатний для виконання на JVM.

Як випливає з назви, кожен байт-код має довжину в один байт із частиною коду операції, яка вказує операцію, яка має бути виконана, за якою слідує нуль або більше операндів, що описують значення, над якими потрібно працювати. Хоча один байт може бути закодований для представлення 256 різних шаблонів, не всі визначені як байт-коди. Подібно до існуючих наборів інструкцій, наприклад, x86 і SPARC, байт-коди Java визначені для виконання звичайних операцій, необхідних для більшості сучасних архітектур, наприклад, завантаження та збереження пам’яті, арифметика, створення об’єктів і маніпулювання ними, передача керування, керування стеком і синхронізація.

Віртуальні машини Java підтримують автоматичне керування пам’яттю на основі концепції збору сміття (GC - garbage collection). Область пам'яті, з якої Java виділяє та звільняє пам'ять, називається купою. Програмісти явно не вивільняють пам'ять, як це роблять за допомогою традиційних мов; тоді як збирачі сміття можуть автоматично шукати об’єкти, які більше не потрібні, і відновлювати пам’ять. Цей механізм ефективно знімає тягар звільнення пам’яті від програмістів, а також пом’якшує безліч помилок пам’яті, наприклад, витік пам’яті.

Сучасні JVM зазвичай включають кілька алгоритмів GC для задоволення різних вимог. Збирачі сміття – це складні системи, які можуть надавати дуже корисну інформацію для програми інструментування, тобто виділення та видалення об'єкта. Рисунок 1.2 представляє огляд архітектури типової сучасної JVM.

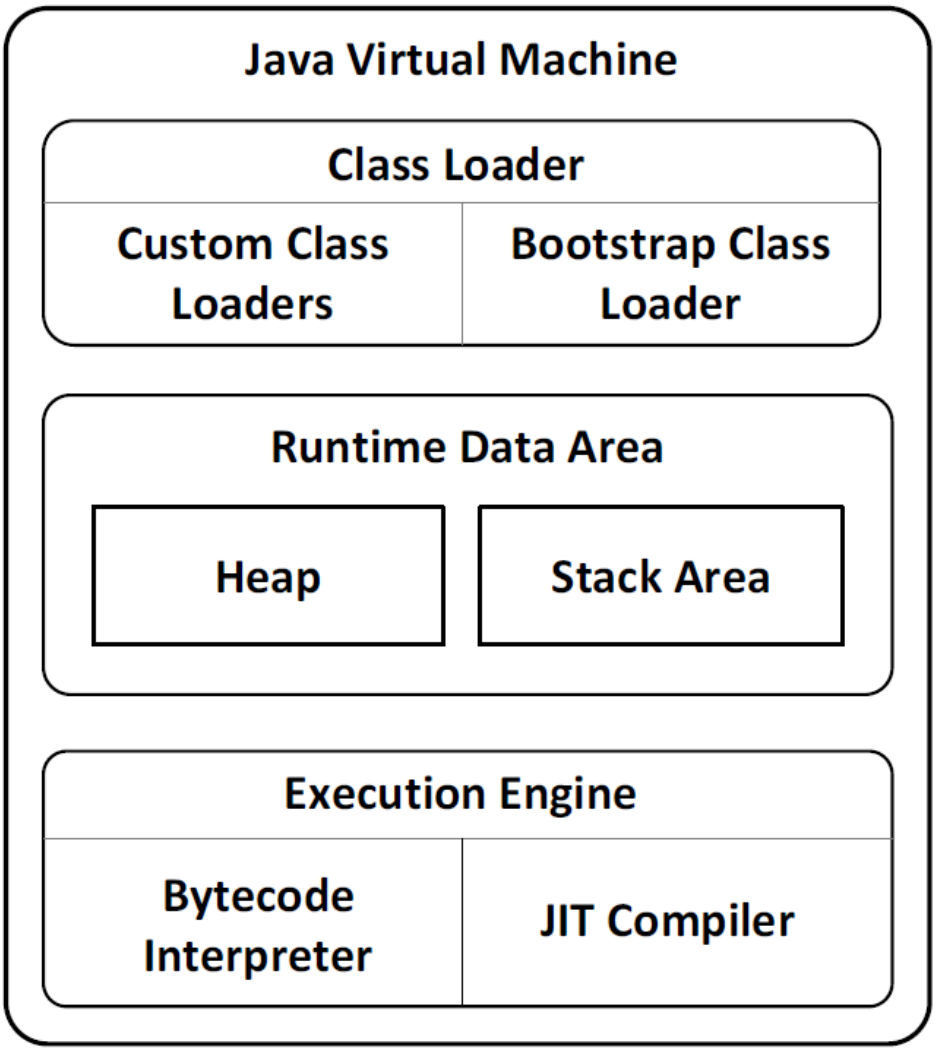


Рисунок 1.2 – Огляд архітектури JVM

Класи Java, скомпільовані в байт-коди, представлені незалежним від апаратного забезпечення та операційної системи двійковим форматом, відомим як формат файлу класу, подібним до форматів рідних об’єктів, таких як Executable and Linkable Format (ELF) і Common Object File Format (COFF) [51]. Файл класу містить не тільки представлення класу в байт-коді, але також містить допоміжну інформацію, таку як постійні значення, таблиці винятків і прапорці доступу, як показано на рисунку 1.3.

Клас завантажувачі відповідають за завантаження та аналіз файлів класів. Вони створюють у пам'яті представлення кожної невід'ємної частини класу, наприклад, методи, поля та постійні значення для JVM, як показано на рисунку 1.4.

Крім початкового завантажувача класів, наданого JVM, програмісти також можуть визначити власні завантажувачі класів, щоб розширити спосіб, яким JVM динамічно завантажує та створює класи.

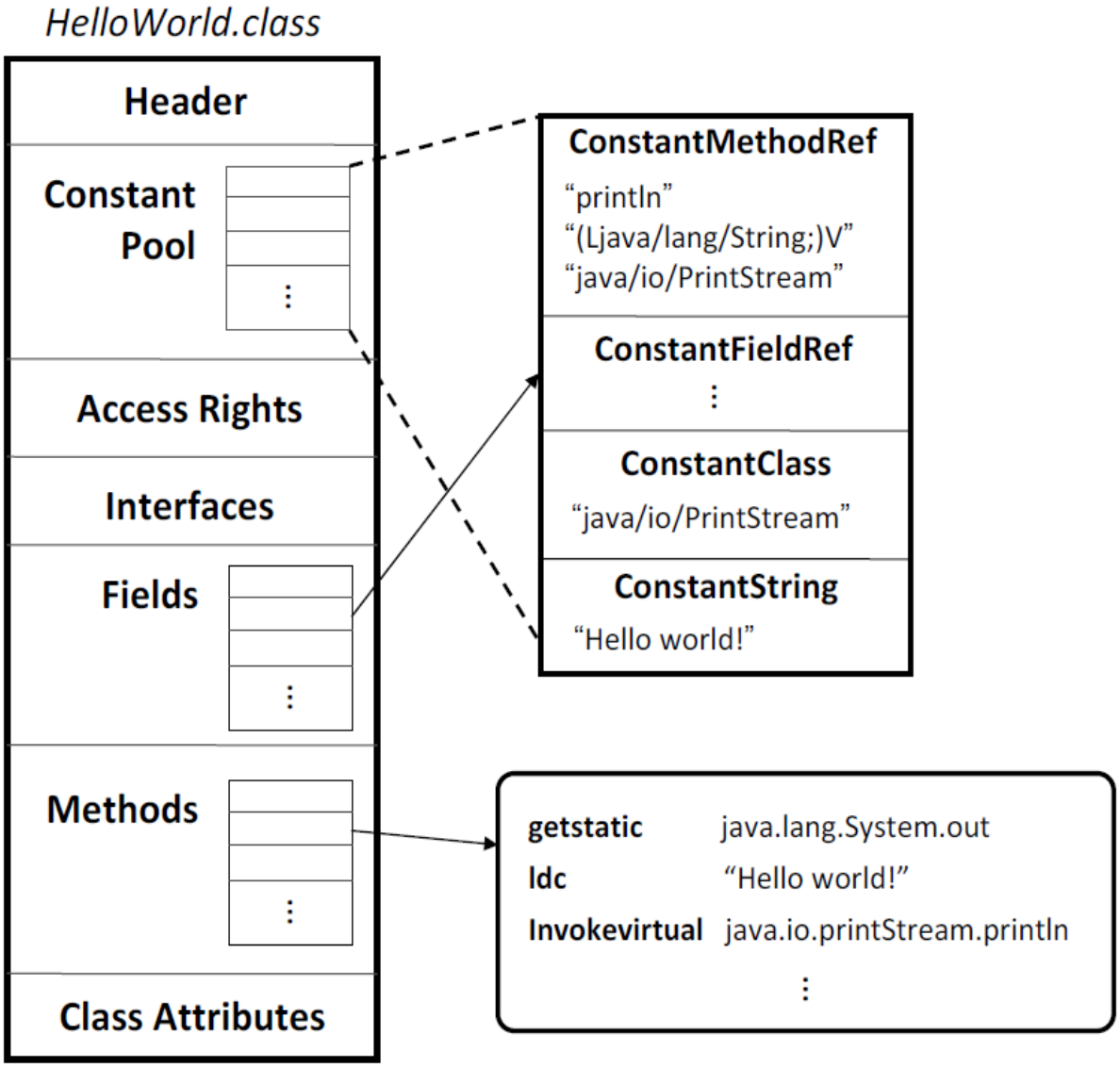


Рисунок 1.3 – Структура файлу класу Java

Наприклад, замість завантаження класу, що зберігається у файлі на диску, ми можемо надати спеціальний завантажувач класів, який генерує клас на льоту безпосередньо в пам’яті. У цій роботі ми розширюємо завантажувач класів початкового завантаження для аналізу інформації, пов’язаної з пропонованою структурою.

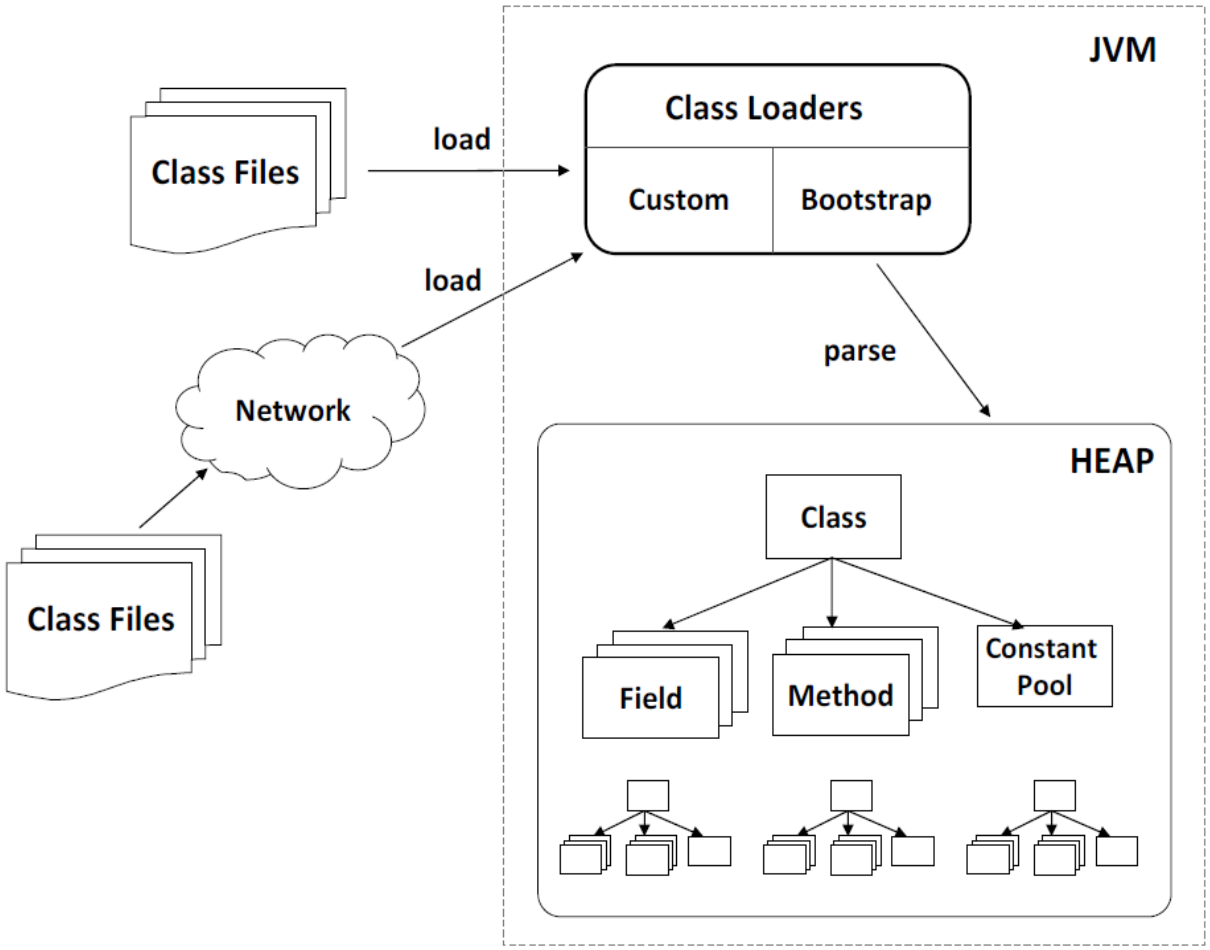


Рисунок 1.4 – Завантажувачі класів у JVM

Модель виконання Java. Оскільки байт-коди Java не залежать від платформи, вони не можуть бути виконані безпосередньо на власних процесорах, а інтерпретуються спеціальним механізмом виконання в JVM, називається інтерпретатором байт-коду. Подібно до мікропроцесора, інтерпретатор байт-коду виконує типовий цикл інструкцій, тобто вибірку, декодування та виконання байт-кодів Java. Рисунок 1.5 надає спрощену ілюстрацію процесу виконання інтерпретатора JVM. Java — це мова, заснована на стеку, так що обчислення виконуються на стеку виразів, тобто байт-коди видаляють операнди й повертають результати назад у стек виразів. Інтерпретатор обробляє лише прості байт-коди, наприклад pushi та iadd, але делегує складні, наприклад new та monitorenter, пов’язаним підсистемам

віртуальної машини, наприклад розподілювачу купи та синхронізатору об’єктів.

```
do {
     fetch an opcode;
     if (operands) fetch operands;
     execute the action for the opcode;
} while (there is more to do);
```

Рисунок 1.5 – Спрощений вигляд циклу інструкцій інтерпретатора

Окрім інтерпретаторів, більшість високопродуктивних JVM включають компілятори Just-in-Time (JIT), які виконують динамічну компіляцію байт-кодів у оптимізовані машинні інструкції, які можуть бути виконані безпосередньо на власних процесорах, що веде до їх прискорення. Метод є загальною одиницею компіляції через його простоту для профілювання, яке є основою для визначення частин коду, що збираються. Інтуїтивно зрозуміло, що методи з найбільшою кількістю викликів або вузьких циклів дають найбільший приріст продуктивності після компіляції JIT-компілятором.

Сучасні JVM зазвичай включають кілька компіляторів JIT і викликають найбільш відповідні версії з огляду на характеристики додатків. Наприклад, інтерактивна програма має бути швидко реагуючою та зазвичай нетривала ¬. Таким чином, компілятор JIT з найкоротшим часом компіляції є бажаним, оскільки він спричиняє мінімальне відволікання на діяльність програми, хоча він генерує менш оптимізований код. У той час як високооптимізований JIT-компілятор підходить для тривалих серверних програм, оскільки одноразова затримка компіляції не викликає занепокоєння.

Мета-дані, у вільному визначенні, означають дані, які можуть описувати аспекти деяких інших даних. У процесі розробки Java є кілька сценаріїв, мета-дані необхідні для різних цілей. Наприклад, JVM очікує, що метадані про клас Java, наприклад, дескриптори полів і методів, будуть присутні в постійному пулі [47] відповідного файлу класу, як визначено специфікацією JVM. Іншим варіантом є зберігання метаданих у «побічних

файлах», які зберігаються осторонь із програмами. Наприклад, програми Java зберігають конфігурації та рядки інтерналізації у властивості файлів. Крім того, програми JavaEE вимагають дескрипторів розгортання на основі XML для інформації про конфігурацію різних ресурсів. З випуском стандартної версії Java 2 Platform Standard Edition (J2SE), Java представляє засіб метаданих анотацій [33] для коментування коду Java з Java, дозволяючи описові метадані безпосередньо біля елемента мови, що описується. Програмісти можуть прикрасити клас, метод, поле, параметр, змінну, конструктор і пакет спеціальними анотаціями.

Залежно від цілей існує кілька способів зберігання метаданих у програмах Java. Використання простих текстових файлів властивостей має перевагу в тому, що їх можна читати людині та редагувати вручну за допомогою текстових редакторів. Однак це вводить додаткові файли в робочу область і підвищує вартість обслуговування. З іншого боку, постійний пул зберігає інформацію низького рівня, необхідну для завантаження класу та виконання програми, подібно до таблиці символів для звичайної мови програмування.

Таким чином, він підходить для зберігання інформації, до якої звертається лише Java Virtual Machine (JVM). Однак завантажувач класів початкового завантаження за замовчуванням має бути змінений для обробки спеціальних метаданих. Анотація Java — це функція мови високого рівня для зберігання метаданих безпосередньо у вихідному коді. Це усуває потребу у «побічних файлах» і його легко редагувати.

Однак оновлення анотацій вимагає повторної компіляції, яка не підходить для прямого використання JVM.

1.4 Опис реалізації специфікації HotSpot JVM

Sun HotSpot JVM — це одна з найпоширеніших у реальному світі якісних віртуальних машин Java. HotSpot — це високопродуктивна та

стандартно-сумісна реалізація специфікації віртуальної машини Java [47]. HotSpot підтримує різноманітні основні платформи, наприклад IA-32, x86-64 і SPARC, а також операційні системи, наприклад Solaris, Linux, Windows і MacOS.

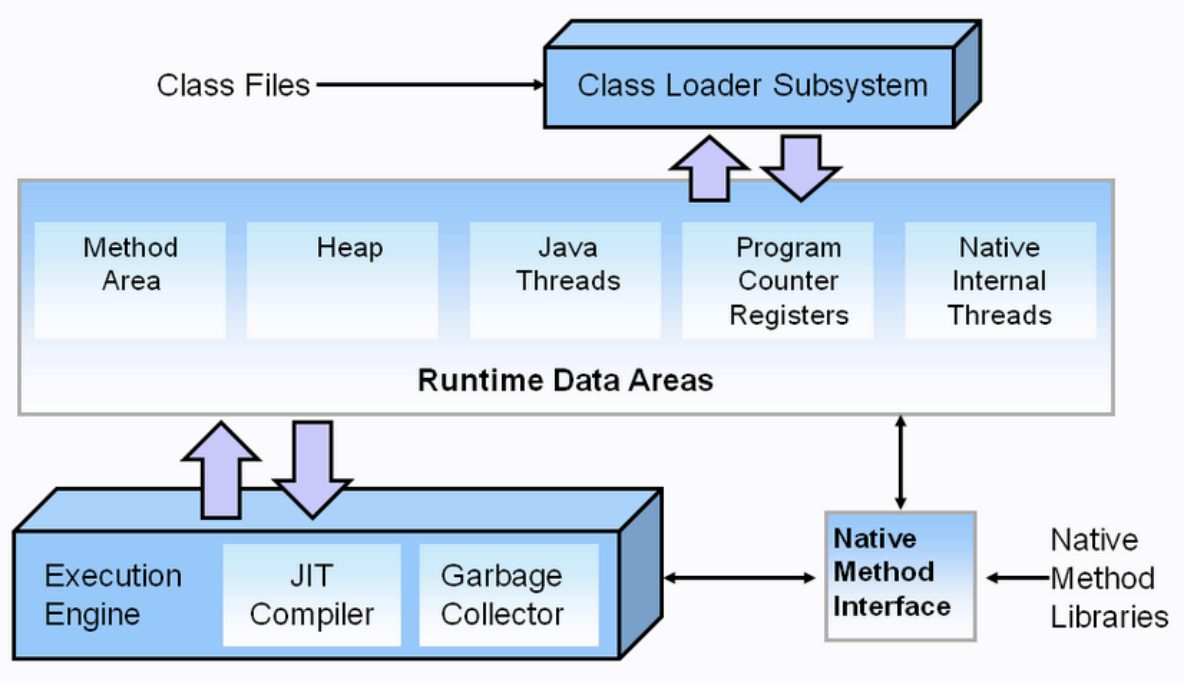


Рисунок 1.6 – Архітектура HotSpot JVM

У HotSpot є два варіанти інтерпретатора, тобто інтерпретатор C++ та інтерпретатор шаблонів. Як випливає з назви, інтерпретатор C++ написаний на C++ з мінімальним кодом складання для низькорівневого керування стеком, який недоступний у C++. Ядром інтерпретатора C++ є гігантська структура, схожа на перемикач, яка відправляє кожен байт-код у відповідну процедуру обробки. З міркувань продуктивності HotSpot наразі за замовчуванням викликає інтерпретатор шаблону. Інтерпретатор шаблону написаний у власних інструкціях зі складання та генерується на льоту під час запуску віртуальної машини [34] для кожного байт-коду. HotSpot зберігає багату інформацію профілювання, наприклад кількість викликів і зворотних розгалужень, для вибору гарячих методів для компіляції JIT.

Компілятори JIT. HotSpot надає два компілятори JIT: компілятор клієнта [44] і компілятор сервера [45]. Клієнтський компілятор має більшу швидкість компіляції з меншою кількістю оптимізацій. Таким чином, він підходить для інтерактивних і короткочасних додатків, де оперативність має високий пріоритет. Компілятор сервера застосовує більш агресивну оптимізацію коду, що призводить до довшої затримки компіляції. Він підходить для тривалих програм, де варто обміняти початкову затримку на більш високу якість коду. У HotSpot компілятори JIT є потоковими, тому вони виконуються паралельно з циклом інтерпретатора. Таким чином, метод, що компілюється, зберігається в інтерпретації до завершення компіляції. Таким чином, кількість інтерпретацій не є детермінованою і може змінюватися серед прогонів.

Представлення об'єкта. HotSpot зберігає сутності Java, наприклад екземпляри, методи, класи, масиви та символи рядків, у купі Java під час виконання. HotSpot представляє їх за допомогою дескриптора звичайного покажчика об’єкта (oopDesc - Ordinary Object Pointer descriptor) і різноманітних підкласів. Наприклад, кожен об’єкт масиву представлений arrayOopDesc, звичайні об'єкти Java представлені instanceOopDesc. HotSpot посилається на ці сутності за допомогою обгорток покажчиків — oop, реалізованих як рідні машинні адреси в пам’яті. Таким чином, instanceOop є вказівником на instanceOopDesc. Окрім даних корисного навантаження, наприклад полів об’єктів, елементів масиву та записів постійного пулу, oopDesc кодує інформацію про кожен об’єкт купи в заголовку об’єкта. Наприклад, заголовок arrayOopDesc записує довжину масиву; заголовок methodOopDesc зберігає позначки доступу методу Java.

Стан потоку та Safepoint. Як і більшість складних систем програмного забезпечення, потоки в HotSpot JVM можуть перебувати в трьох станах виконання, тобто стані Java, стані VM і рідному стані. Стандартним станом є стан Java, коли HotSpot виконує байт-коди Java. У цьому стані GC заборонено, оскільки програми постійно змінюють купу Java. Подібним чином інші операції віртуальної машини, які можуть виділяти пам’ять або

отримувати блокування, також заборонені в стані Java. Для виконання таких операцій HotSpot має перейти в стан віртуальної машини. Таким чином, записи підпрограм середовища виконання захищені прологами переходу стану, наприклад, Java-to-VM і native-to-VM. У стані віртуальної машини HotSpot може викликати GC, коли досягає безпечної точки, де всі потоки в стані Java зупиняються, а корені GC відомі. Більшість сайтів викликів і записів підпрограм виконання кваліфікуються як точки безпеки. Коли HotSpot виконує методи JNI, він переходить у вихідний стан. Слід уникати змін стану потоку, коли це можливо, оскільки деякі напрямки є дорогими, наприклад, нативний до Java.

OpenJDK. OpenJDK – це реалізація специфікації J2SE з відкритим вихідним кодом, випущена Sun. Більшість OpenJDK ліцензовано за публічною ліцензією (GPL), за винятком деяких обтяжених компонентів, які можна поширювати лише як двійкові файли. На сьогоднішній день HotSpot є єдиною реалізацією JVM з відкритим вихідним кодом. Таким чином, ми обрали HotSpot у OpenJDK як нашу дослідницьку JVM у цій роботі через її якість, високу продуктивність, повсюдне розгортання та базу відкритого коду.

Оскільки класи Java скомпільовані в байт-коди та зберігаються у двійкових файлах класів, можна змінити поведінку та структуру класу Java шляхом перетворення байт-кодів і файлів класу відповідно. Ця техніка називається інструментарій байт-коду (BCI – Bytecode Instrumentation). Наприклад, ми можемо ввести нові поля та методи, перейменувати класи та додати нові реалізації інтерфейсу до існуючого класу за допомогою BCI. Використовується багато бібліотек і інструментів BCI, наприклад, BCEL [24], ASM [30], JavaAssist [23] і SOOT.

Нещодавно ASM стала однією з найпопулярніших бібліотек BCI завдяки своїй невеликій площі, швидкій обробці байт-коду та простоті використання. На відміну від попередніх рішень, ASM не покладається на представлення об’єктів для різних типів вузлів у класі деревоподібну структуру або різні типи інструкцій байт-коду. Таким чином, ASM уникає

роздутості і ефективний як у часі, так і в просторі при кодуванні та декодуванні файлів класу.

ASM базується на шаблоні Visitor [32] і підтримує два API для генерації та перетворення файлів класів: основний API забезпечує представлення класів на основі подій, тоді як API дерева забезпечує представлення на основі об’єктів, подібне до BCEL. Core API корисний для контекстно-вільних перетворень, тоді як API дерева підходить для більш складних. Кожна подія в основному API представляє елемент класу, наприклад, заголовок, поле, метод та інструкцію. Перетворення файлів класів за допомогою основного API вимагає заміни віртуальних методів кількох інтерфейсів, наприклад ClassVisitor, FieldVisitor і MethodVisitor.

У цій роботі ми будемо досліджувати інструментальну утиліту на основі ASM на основі основного API для структури MTF, яка покладається на BCI для вбудовування маркерів і пов’язаних метаданих у файли класів. Основного API достатньо для такої мети, оскільки BCI, пов’язаний з маркером, посилається лише на постійний пул і пов’язані методи, тому є контекстно-вільним.

1.5 Інтерфейс віртуальної машини Java

HotSpot JVM реалізує інфраструктуру під назвою: Java Virtual Machine Tool Interface (JVMTI) [2], комплексний інтерфейс програмування, який використовується розробниками та інструментами моніторингу. Численні попередні роботи базуються на JVMTI, наприклад, динамічний аналіз програм [12], налагодження змішаного середовища [46], моніторинг продуктивності [50] та впровадження помилок [38].

JVMTI дозволяє наданому користувачем агенту (клієнту JVMTI) отримувати доступ до внутрішніх станів віртуальної машини та контролювати виконання програм. Агенти можуть отримувати сповіщення про події, коли запускаються зареєстровані події. Крім того, JVMTI передає

аргументи функціям зворотного виклику, щоб надати додаткову інформацію про подію. Приклади подій JVMTI включають, але не обмежуються ними, VMStart , ClassLoad , FieldAccess , MethodEntry та MethodExit. За допомогою функцій JVMTI агенти можуть запитувати різні стани програми, наприклад, трасування стека, стан потоку, локальні змінні, монітори об’єктів і завантажені класи. JVMTI також дає змогу агентам змінювати виконання програм, наприклад призупиняти потоки, встановлювати точки зупину та відкривати кадри стека.

Як фіксований інтерфейс JVMTI обмежує клієнтів існуючими типами подій і функціями доступу. Щоб вирішити цю проблему, JVMTI підтримує інструментарій байт-коду, щоб користувачі могли вставляти додаткові байт-коди в класи для запису недоступних подій. Це можна зробити під час компіляції та під час завантаження або під час виконання програми за допомогою RedefineClasses. Оскільки маніпуляції з байт-кодом безпосередньо не підтримуються JVMTI, ми повинні використовувати інструменти сторонніх виробників, наприклад, ASM або BCEL, щоб трансформувати байт-коди перед передачею в JVMTI. Здатність JVMTI дозволяти такі типи інструментів дозволяє реалізувати певні частини нашої інфраструктури MTF. Однак ми стверджуємо, що таке рішення не таке гнучке чи ефективне, як структура MTF.

По-перше, агенти JVMTI не мають доступу до всіх ресурсів віртуальної машини, як клієнти MTF. По- друге, RedefineClasses — це дорога операція, що включає перезавантаження та повторний аналіз класу, на додаток до накладних витрат на онлайн-перетворення байт-коду. Тоді як за допомогою таких методів, як AOPA, MTF підтримує ефективне перемикання програмних інструментів із значно меншими накладними витратами. Нарешті, це нетривіально та не настільки інтегровано, щоб вручну відтворити функції MTF як агенти JVMTI.

Віртуальна машина якості (QVM) — це спеціалізоване середовище виконання на основі IBM J9 JVM [9]. Метою QVM є забезпечення

інфраструктури для виявлення дефектів програмного забезпечення, які виникають на етапі після розгортання у виробничому середовищі. QVM безперервно, але ефективно відстежує виконання програми щодо визначених користувачем властивостей коректності, наприклад, властивостей стану типу, твердження Java та властивостей купи.

Щоб контролювати накладні витрати, QVM використовує новий диспетчер накладних витрат, щоб забезпечити дотримання визначеного користувачем бюджету накладних витрат.

QVM збирає якомога більше корисної інформації з виконуваної програми, залишаючись у межах зазначеного бюджету за допомогою об’єктоцентричної вибірки, що дозволяє здійснювати вибірку на рівні екземпляра об’єкта. Клієнти аналізу отримують події профілю лише від об’єктів, які позначені як відстежувані, як зазначено бітом у заголовку об’єкта. QVM відбирає об’єкти на основі сайтів розподілу та використовує коротку вбудовану послідовність коду для перевірки біта відстеження перед виконанням будь-яких зворотних викликів QVM. Таким чином, QVM може регулювати розмір вибірки на кожному місці розподілу, щоб контролювати частоту подій, щоб накладні витрати залишалися в межах бюджету.

Для справедливої вибірки сайти з меншою частотою розподілу отримують більші кванти вибірки. QVM підтримує аварійне відключення, тобто відкидання гарячого та довгоживучого об’єкта, щоб уникнути серйозного зниження продуктивності.

Подібно до MTF, QVM підтримує фільтрацію побічних подій віртуальної машини, дозволяючи користувачам визначати методи моніторингу, щоб решта програми працювала на повній швидкості. Однак QVM інструментує програми на рівні методу або поля, а не на рівні основного блоку, як це робить MTF. Таким чином, неможливо відстежити лише підмножину всіх операторів у методі. Тим не менш, підхід QVM до контролю над витратами моніторингу на основі детальної адаптивної вибірки, керованої властивостями, є дуже ефективним. Оскільки MTF також є

рішенням рівня віртуальної машини, він також може реалізувати такий механізм і досягти подібної гарантії продуктивності, як QVM. Крім того, клієнти твердження та властивості купи легко переносяться на MTF. Нарешті, MTF перевершує QVM завдяки підтримці визначених користувачем подій, що значно покращує гнучкість і універсальність програмного інструментарію.

1.6 Концепція динамічного трасування для моніторингу поведінки програмних додатків

Dynamic Tracing Framework (DTrace) — це компонент операційної системи Solaris. DTrace — це потужна інфраструктура для адміністраторів і розробників, яка дозволяє досліджувати довільну поведінку операційної системи та програм користувача з дуже низькими накладними витратами. DTrace підтримує збір показників продуктивності у виробничому середовищі шляхом динамічної зміни ядра операційної системи та процесів користувача в цікавих місцях, тобто зондів, які надають постачальники. Пишучи програми на мові програмування D, користувачі можуть точно й лаконічно вказати зонди, які потрібно ввімкнути, і дії, які потрібно виконати, коли зонди потрапляють. DTrace дозволяє фільтрувати зонди за допомогою предикатів, які оцінюються під час виконання. Усі інструменти в DTrace повністю динамічні, тобто зонди вмикаються лише тоді, коли вони використовуються, а для неактивних зондів інструменти відсутні. Таким чином, решта програм поза зондами працюють на повній швидкості.

Платформа Java, стандартна версія (Java SE) представляє підтримку DTrace у HotSpot JVM за допомогою двох постачальників DTrace: hotspot і hotspot_jni, які побудовані як агенти JVMTI. Постачальник HotSpot підтримує зонди в різних підсистемах HotSpot, наприклад, життєвий цикл віртуальної машини, збір сміття, завантаження класів, JIT-компіляція, розподіл об’єктів і вхід/вихід методу.

Подібно до JVMTI, hotspot і hotspot_jni містять фіксований набір зондів і не підтримують інструментарій байт-коду, який є важливим для розширення існуючого інтерфейсу подій. У той час як MTF розроблено спеціально для того, щоб дозволити користувачам визначати власні події на основі довільних місцезнаходження програми, що їх цікавить, і приєднувати обробники зворотного виклику, які можуть отримати доступ до всіх служб віртуальної машини та потенційно мати складну логіку.

Крім того, DTrace залежить від платформи, оскільки він покладається на підтримку всередині ядра ОС. MTF, з іншого боку, може підтримуватися на всіх платформах незалежно від того, і не покладається на жодні служби ОС. Тим не менш, ми припускаємл, що сценарії D, коли їх розширено, можна використовувати для специфікації динамічних маркерів для MTF, як альтернативу для рішення на основі пулу констант під час компіляції.

Аспектно-орієнтоване програмування. Аспектно-орієнтоване програмування (AOP) також використовувалося для профілювання [7] і динамічного аналізу програм [6]. Розглянемо AspectJ [42] найбільш широко використовувану реалізацію AOP для Java. AspectJ підтримує систематичне програмне інструментування, надаючи мовні конструкції для опису інструментальних точок (точок з’єднання) і додавання спеціальних дій (поради). Пораду можна прикріпити або до однієї точки з’єднання, або до їх набору з точковими розрізами. Приклади точок з’єднання, які підтримує AspectJ, включають: виклики та виконання методів, типи об’єктів відправника та отримувача, обробники винятків і потоки керування. Поради написані стандартною мовою Java і мають доступ до всіх бібліотек Java. Крім того, aspectJ підтримує переплетення порад як під час компіляції, так і під час завантаження за допомогою компілятора AspectJ (ajc) і завантажувачів класів переплетення відповідно.

На відміну від JVMTI і DTrace, інструментарій на основі AspectJ не вимагає спеціальної підтримки з боку JVM або ядра ОС, оскільки цільова програма змінюється безпосередньо. Таким чином, інструменти на основі

AspectJ легше розробити та більш портативні, ніж агенти JVMTI та сценарії DTrace. Однак цей тип інструментів обмежений лише інформацією на рівні користувача, наприклад, об’єктами отримувача та аргументами методу. На такому рівні внутрішня інформація віртуальної машини абсолютно недоступна для AspectJ. Наприклад, інструментарій на основі AspectJ не може взаємодіяти ні зі збирачами сміття, ні з компіляторами класу Just-in-Time (JIT).

Крім того, вплетені аспекти — це повноцінний код Java, який не тільки роздуває цільові програми, але й може створити помітні накладні витрати на виконання. MTF вирішує обидві проблеми, реалізуючи клієнтів як частину віртуальної машини, використовуючи лише один або два байт-коди, хоча клієнти MTF складніше реалізувати.

Незважаючи на недоліки AspectJ для детального низькорівневого програмного забезпечення, його мова та компілятор потенційно можуть бути розширені для користувачів MTF для систематичного визначення маркерів; таке рішення іноді може бути зручнішим, ніж розробка клієнта на основі нашої інструментальної утиліти на основі BCI.

Висновки до розділу 1

В даному розділі проведено огляд технології віртуальних машин, виконано аналіз предметної області використання віртуальних машин в програмуванні для покращення моніторингу та аналізу додатків та досліджено технології сучасних віртуальних машин Java. Здійснено опис реалізації специфікації HotSpot JVM та концепцій і технології, що стосуються розуміння методології Marker Tracing Framework (MTF).

РОЗДІЛ 2. МОДЕЛІ СТАТИЧНОГО АНАЛІЗУ ТИПІВ АНАЛІЗУ ДОДАТКІВ

2.1 Аналіз станів типу виконання в об’єктно-орієнтованому програмуванні

Сучасна практика розробки програмного забезпечення заохочує повторне використання існуючих бібліотек програмного забезпечення, наприклад, Java Runtime Library, .NET Base Class Library та C++ Standard Template Library (STL), для впровадження нових програмних систем. Більшість таких повторно використовуваних компонентів мають визначені обмеження щодо інтерфейсу, яких мають дотримуватися розробники для реалізації програм, що працюють добре. Приклад правила щодо використання класу Iterator у Java вказує, що об’єкт Iterator має запитувати доступність hasNext перед розширеним (next). Порушення такого правила може призвести до виключення виключення під час виконання, яке, якщо воно не буде виявлено, може призвести до збою програми. Хоча більшість таких правил добре задокументовано, розробники додатків не завжди їх дотримуються. Таким чином, помилки програмування через невідповідне використання API все ще трапляються досить часто під час розробки програмного забезпечення.

Властивість typestate описує набір дійсних операцій, які можна виконати над об’єктом, залежно від стану типу об’єкта. Таким чином, властивості typestate підходять для представлення обмежень API. Аналіз стану типу є ефективним методом для перевірки того, чи програма порушує задані властивості typestate. Аналіз стану типу може бути статичним [11, 31,16], динамічним [22 ,17, 27, 9] або гібридним [18, 28].

В об’єктно-орієнтованих мовах програмування тип об’єктів даних визначає набір операцій, які дозволено виконувати над ними як приймачами.

Тоді як typestate визначає підмножину цих операцій, які дозволені в певному контексті. Наприклад, файл можна читати або записувати лише після його відкриття; ресурс операційної системи, пов'язаний з кожним віджетом графічного інтерфейсу користувача (GUI), має бути остаточно звільнений після використання. Такі вимоги зазвичай зустрічаються в програмній документації та можуть бути природним чином представлені у вигляді властивостей типу, які можна перевірити для покращення якості програмного забезпечення.

```
public class File {
  public void open(String name);
  public void close();
  public char read();
  public void write(char c);
  public boolean eof();
}
```

Рисунок 2.1 – Файл API

Кілька формалізмів, наприклад, кінцевий автомат (FSA - finite-state automaton) [27, 9], лінійна часова логіка [15] і мови, засновані на історичних даних [17] використовувалися в минулих дослідженнях. У цій роботі ми використовували FSA як основний формалізм для представлення властивостей типу, оскільки:

1) FSA природним чином моделює динамічні етапи виконання програми;

2) FSA можна легко сконструювати з регулярних виразів (REGEX - regular expressions) і більшість розробників можуть зручно виражати властивості типів у REGEX. Наприклад, файловий API, показаний на рисунку 2.1 можна описати наступним регулярним виразом:

```
(open (read|eof|write)* close)*
```

Моделювання. У нашому аналізі на основі FSA ми зіставляємо кожен стан типу у власності за станом у FSA. Більше того, ми використовуємо два

спеціальні типи станів, джерело та приймач, щоб представити початковий стан перед виконанням програми та стани помилок, відповідно. Перемикання між двома станами представлено як детермінований перехід, позначений подією, яка запускає перехід. Рисунок 2.2 показує FSA для властивості File.

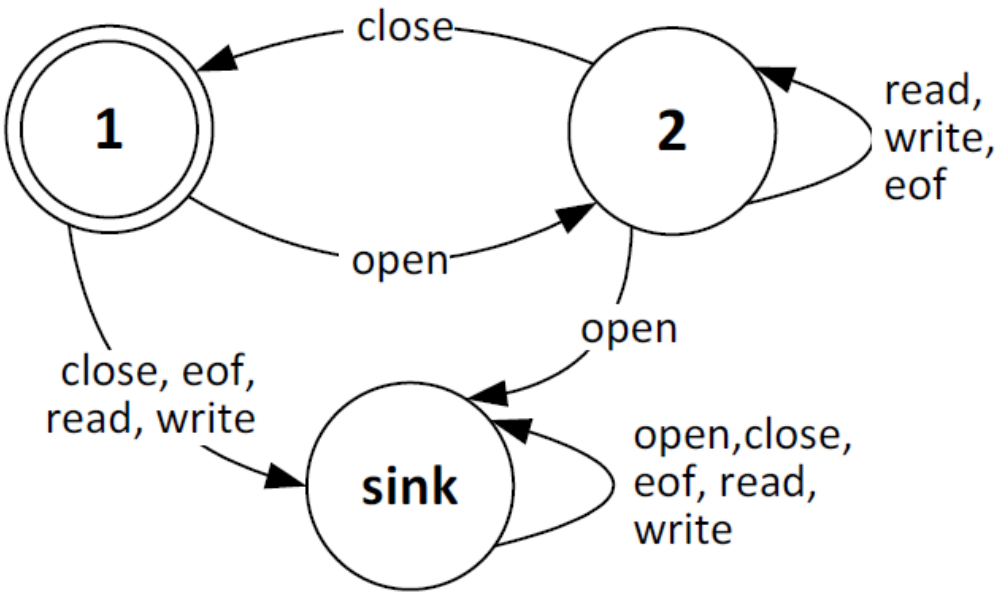


Рисунок 2.2 – FSA для властивості File

Формально властивість стану типу p може бути представлена FSA як

$$(Q, \Sigma, \delta, q_0, F)$$

де Q — множина станів; Σ – набір маркерів (алфавіт); δ – функція переходу, а F – множина потоків. Для прикладу файлу, показаного на рисунку 2.2 , маємо:

1) $Q = \{s_1, s_2, s'\}$; 2) $\Sigma = \{\text{open, read, write, eof, close}\}$.

Трасування. Наш клієнт аналізу стану типів повинен постійно відстежувати потоки програм і записувати стани програм, щоб керувати внутрішнім FSA для онлайн-перевірки властивостей стану типів. Цей

механізм підтримується MTF. Зокрема, ми можемо відстежити кожен стан у FSA, розмістивши маркер навколо відповідної області коду. У прикладі File ми можемо вставляти маркери в записи open, read, write, eof і exit відповідно. Коли виконується програма, яка використовує File, MTF може розпізнавати маркери та викликати події, щоб сповістити клієнт аналізу. Клієнти аналізу можуть включати властивості typestate у цільові класи за допомогою MTF, використовуючи макет, показаний у таблиці 2.1, де:

1) кожен метод представлений переходом з однолітерним символом, наприклад, o для open;

2) маркер 0 є спеціальним маркером для зберігання регулярного виразу властивості File API;

3) кожен предикатний рядок має формат: “ <method-name>, <method-signature> | symbol”.

Таблиця 2.1

Дескриптори маркерів для властивості File API

Marker ID	Predicate String	Description
0	`(o(r|e|w)*c)*`	specifying the regex of the property
1	`open,(Ljava/lang/String;)V|o`	descriptor for the `open` method.
2	`read,()C|r`	descriptor for the `read` method.
3	`write,(C)V|w`	descriptor for the `write` method.
4	`eof,()Z|e`	descriptor for the `eof` method.
5	`close,()V|c`	descriptor for the `close` method.

Перевірка. Клієнт аналізу стану типів визначає обробники для моніторингу подій маркерів, на основі яких підтримується FSA. Крім того, кожен об'єкт пов'язаний зі структурою даних, яка записує поточну інформацію про стан типу, яка оновлюється клієнтом аналізу під час кожного переходу. Кожна маркерна подія означає перехід у FSA відповідно до властивості. Аналіз перевіряє законність переходу щодо властивості перед

фактичним виконанням переходу та подальшим оновленням структури даних для кожного об’єкта.

Перевірка кінцевого стану властивості typestate для кожного відстежуваного об’єкта потребує підтримки з боку збирача сміття, оскільки програми явно не звільняють об’єкти, але GC це робить. Таким чином, лише GC має інформацію про смерть об’єкта. Щоб перевірити, чи помирає об’єкт в одному з кінцевих станів, ми можемо переглянути список мертвих об’єктів і перевірити його поточний стан наприкінці кожної колекції. Перевірка остаточного стану є критичною для багатьох аналізів, наприклад, витоку ресурсів [43, 9] і узгодженості структури даних [45, 26].

Адаптивний онлайн аналіз програм. Оскільки більшість онлайн-аналізу стану типів викликають великі накладні витрати та сповільнюють виконання на порядки, останні дослідження запропонували різні методи оптимізації [22, 9, 27, 16]. AOPA є одним із ефективних методів і базується на спостереженні, що в будь-який момент під час аналізу стану типу лише підмножина всіх переходів може змінити стан програми, тобто вихідні переходи. Інші переходи називаються переходами самоциклу, де джерело та призначення є однаковими станами. Тому події, які є символами до переходу самоциклу можна безпечно ігнорувати, щоб зменшити частоту подій, що призведе до значного зменшення накладних витрат на моніторинг. Крім того, набір ігноруючих символів динамічно оновлюється, коли програми здійснюють вихідні переходи. Аналіз стану типу за допомогою AOPA дає той самий результат, що й неадаптивна версія, але виконується ефективніше.

Наприклад, програма відкрила файл лише для читання (рис 2.2) , який потім залишається в стані 2 незалежно від будь-яких подальших читань, записів або запитів (eof), доки не буде видано остаточне закриття. Таким чином, ми можемо безпечно застосувати AOPA до цього об’єкта File, вимкнувши моніторинг {read, write, eof}, оскільки це події самоциклічних переходів. Оскільки вихідні події {open, close} відстежуються, аналіз все одно може виявити порушення API та оновити набір подій самоциклу, як

показано в таблиці 2.1 . В ідеалі має бути рівно дві події, тобто відкриття та закриття, при цьому всі події читання та eof ігноруються.

Таблиця 2.2

Самоцикл і вихідні символи в File API [27]

State	Self symbols	Out-going symbols
1	{}	Σ
2	{read, write, eof}	{open, close}
sink	Σ	{}

2.2 Реалізація моделі процесу Adaptive Online Program Analysis (AOPA)

Оригінальна реалізація AOPA називається «Sofya» і базується на інтерфейсі Java Debugger Interface (JDI) [47] для перехоплення подій входу/виходу методу шляхом встановлення точок зупину. У цій роботі ми замінюємо JDI на MTF для тієї самої мети, але з набагато меншими накладними витратами, оскільки всі виконання зупиняються, коли Sofya повторно інструментує цільові класи для адаптивного моніторингу. Sofya страждає від накладних витрат у найгіршому випадку, коли виконання передбачає переважно вихідні переходи.

У цьому розділі ми представляємо деталі реалізації клієнта аналізу типових станів на основі FSA за допомогою MTF.

Скінчений автомат. Наш клієнт аналізу станів типів містить Libfa [25], бібліотеку FSA, реалізовану на C, для структур даних FSA, таких як стани, переходи та загальні операції. Найбільш відповідними операціями для нашого аналізу є:

1) розбір регулярного виразу для побудови FSA;

2) мінімізація для перетворення NFA у DFA для зменшення непотрібних станів і переходів.

Таким чином, розробники можуть легко вказати властивості стану типу, використовуючи регулярні вирази як вхідні дані для клієнта аналізу, що робить клієнт загальним для будь-яких властивостей стану типу. Крім того, детермінізм, отриманий в результаті мінімізації, прискорює перевірку властивостей і полегшує аналіз коду.

Щоб заощадити місце, ми зберігаємо в пам’яті лише один екземпляр FSA для кожної властивості typestate. Кожен об’єкт Java зберігає та посилається лише на структуру поточного стану, тобто стани та переходи FSA є повністю спільними. Крім того, кілька класів Java з однаковою властивістю typestate перевіряються одним FSA. Наприклад, усі ітератори в програмі мають бути перевірені за допомогою властивості typestate hasNext, наша JVM кешувала б регулярний вираз і використовувала той самий екземпляр FSA для всіх об’єктів ітератора. Завдяки двом оптимізаціям наше представлення на основі Libfa займає незначний обсяг пам’яті.

Пооб'єктне зберігання. Наш аналіз стану типів виконує перевірку на основі кожного об’єкта. Таким чином, нам потрібно окремо відстежувати та оновлювати стан кожного цільового об’єкта. Є два підходи, централізовані та розподілені залежно від місця, де зберігається інформація про стан типів. У централізованій темі глобальна хеш-таблиця підтримується та індексується ідентифікаторами об’єктів.

Розподілений підхід зберігає інформацію в заголовку об'єкта. Хоча рішення для зберігання для кожного об’єкта займає більше пам’яті завдяки доданим полям, воно економить накладні витрати на пошук і оновлення хеш-таблиці, що може бути значним, коли відстежується велика кількість об’єктів.

У цій роботі ми вирішили реалізувати рішення для зберігання по об’єктах для ефективного пошуку та оновлення інформації про стан типів. Зокрема, ми можемо розширити заголовок об’єкта, представлений структурою instanceOopDesc , додавши додаткове поле покажчика, що вказує на структуру typestate. Ця схема додає 4 або 8 байт до кожного об’єкта на 32- і

64-розрядних платформах відповідно. За допомогою єдиного вказівника ми обмінюємо додаткові розіменування вказівників на простір, коли нам потрібно отримати доступ до полів усередині структури typestate.

Адаптивний онлайн аналіз програм. Щоб динамічно перемикати інструментарій методу на основі стану типу, Sofya використовує JDI для виконання повторного визначення та перезавантаження цільового класу під час виконання, що зазнає великих витрат. У цій роботі ми використовуємо адаптивний механізм виклику маркера, який підтримується MTF, щоб реалізувати AOPA. Зокрема, наш клієнт typestate під час кожного переходу встановлює біти, що відповідають маркерам із вихідними символами, тоді як очищає їх для маркерів із символами самоциклу. Таким чином, інструментування методів, позначених символами самоциклу, автоматично пропускається під час наступних викликів.

Дотримуючись властивості typestate і переходів FSA, такі методи можуть відновити своє інструментування, коли символи, які вони несуть, стануть вихідними символами в майбутньому.

Обробка подій видалення об'єкта. Для певних властивостей typestate важливо перевірити, чи зникають об’єкти в кінцевому стані прийняття. Однак Java використовує автоматичне керування пам’яттю на основі збирання сміття. Таким чином, нам потрібно співпрацювати з підсистемою GC, щоб фіксувати події видалення об’єкта.

HotSpot за замовчуванням використовує збирання сміття за поколіннями з напівпростором для молодого покоління та паралельним компактним колектором для старого покоління. Таким чином, простір мертвих об’єктів просто перезаписується шляхом копіювання живих об’єктів під час відновлення.

Отже, HotSpot не відстежує подію видалення об’єкта. HotSpot підтримує механізм під назвою JNI handle, керований покажчик, який є прозорим для реферера, коли об’єкт, на який посилається, переміщується в пам’ять GC. Коли на об’єкт Java посилається дескриптор JNI, GC не може

відновити об’єкт. На відміну від звичайних дескрипторів JNI, слабкі дескриптори не перешкоджають GC відновлювати об’єкти, на які посилаються, подібно до слабкого посилання. Ми використовуємо евристику, згідно з якою слабкі дескриптори мертвих об’єктів перетворюють на NULL покажчики.

Оскільки для відстеження мертвих об’єктів потрібна спеціальна підтримка з боку HotSpot, для забезпечення такої функції реалізовано постачальник контексту події смерті об’єкта (ODE - object death event). Щоб відстежувати мертві об’єкти після кожного GC, ODE зберігає пов’язаний список структур typestateHandle, тобто пари для кожного об’єкта typestate і слабкого JNI-дескриптора для всіх об’єктів, що контролюються.

ODE реєструє подію GC-end, де він може перевірити живучість об’єктів, багаторазово вирішуючи їх маркери JNI. Для кожного об’єкта він повідомляє обробники, які реєструють подію смерті об’єкта за допомогою структури typestateHandle об’єкта. Наш клієнт аналізу стану типів реєструє подію смерті об’єкта, надану ODE. Таким чином, він може перевірити можливий стан типу, де кожен об’єкт гине.

Щоб мінімізувати накладні витрати від виклику обробника, ми об’єднуємо typestateHandle структури всіх мертвих об’єктів у зв’язаному списку та передають список обробникам як пакет. Щоб ще більше зменшити накладні витрати на створення іншого списку, ми просто вирізаємо живі об’єкти в новий список, оскільки ми спостерігаємо набагато більше смертей, ніж життів під час прогону профілю. Таким чином, typestateHandle мертвих об’єктів залишаються на місці в оригінальному списку, який можна використовувати як пакет. Зрештою, список із живими об’єктами стає новим списком дескрипторів для наступного проходу.

З такою оптимізацією одне виконання GC запускає лише один виклик методу на зареєстрований обробник. Рисунок 2.3 ілюструє механізм ODE і проведену оптимізацію.

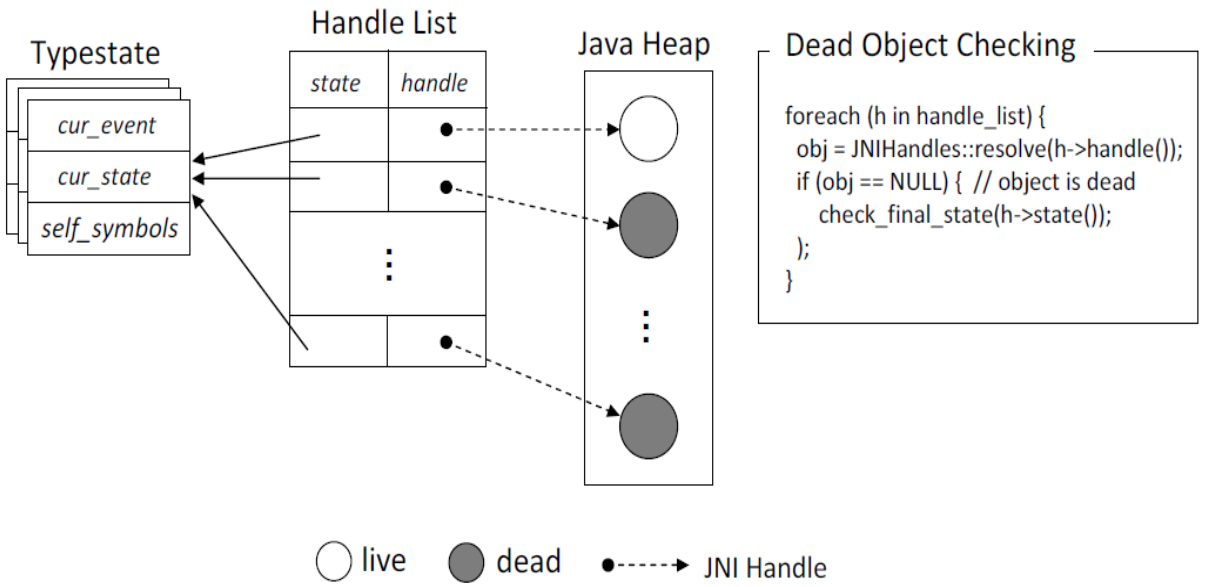


Рисунок 2.3 – Перевірка стану об’єкта

2.3 Оцінка адаптивного аналізу динамічного стану об’єкта

У цьому розділі ми представляємо оцінку нашого адаптивного аналізу динамічного стану типу на основі FSA на базі MTF із набором експериментів.

Файловий API. Ми показуємо ефективність як аналізу, так і оптимізації, перевіряючи файловий API на наборі мікротестів, як показано в таблиці 2.3 . Набір містить відповідні та невідповідні програми з введеними вручну порушеннями для демонстрації ефекту оптимізації та функціональності аналізу типового стану відповідно.

Таблиця 2.3

Мікротести для тестування властивості File-API

Benchmark	Description
Main	Open two files for reading and writting (conforming).
NoEof	Open a file for reading but never call `eof()`.
NoOpen	Read a file that is not opened.
NoClose	Read a file but do not close it in the end.

Крім основного, інші тести певним чином порушують властивість файлового API. NoClose є особливим тестом, оскільки він перевіряє здатність

фреймворка фіксувати подію смерті об’єкта, яка необхідна для перевірки кінцевого стану, коли вмирає кожен цільовий об’єкт. Наприклад, наш аналіз повинен виявити в NoClose, що файл залишається незакритим після завершення програми.

В експерименті клієнт Typestate успішно знайшов і повідомив про всі порушення. Наприклад, повідомила про порушення:

```
Found invalid transition ^ -> r (tests.file.File) <tid=14068>
```

Під час виконання тесту NoOpen , який читає (r) файл (tests.file.File), який не відкривається (~) потоком 14068. За порушення мертвого об'єкта клієнт повідомляє:

```
Dead object (0x01ebcb48) violation @ {r} state
```

Для програми NoClose, вказуючи, що об’єкт за адресою пам’яті 0x01ebcb48 помирає у стані читання та порушує властивість, яка вимагає, щоб кінцевий стан був закритим.

Для Main ми записуємо кількість викликів маркерів до та після застосування AOPA, щоб показати його ефективність. Оскільки еталонний тест короткочасний, ми робимо порівняння не на основі часу виконання, а на основі скорочення викликів маркерів. Main виконано п'ять разів і визначено найкраще, найгірше та середнє геометричне в таблицях 2.4 і 2.5.

Таблиця 2.4

Виклики маркерів тесту File-API

Non-adaptive			Adaptive		
Max	Min	Geomean	Max	Min	Geomean
6,000,008	6,000,008	6,000,008	31,224	29,418	30,439

Таблиця 2.5

Час виконання тесту File-API

Non-adaptive (msec)			Adaptive (msec)		
Max	Min	Geomean	Max	Min	Geomean
378	444	401.0	22	22	22

Ми перевіряємо використання об’єктів-ітераторів у наборі DaCapo, щоб додатково продемонструвати продуктивність і ефективність нашого клієнта. Передбачене використання класів java.util.Iterator вимагає, щоб hasNext передував кожному виклику next, який зазвичай називають властивістю HasNext . У цьому експерименті ми спрощуємо HasNext у нову властивість під назвою HasNextOnce, тобто hasNext потрібно викликати принаймні один раз перед будь-яким наступним next. Ми показуємо кінцеві машини HasNextOnce і HasNext на рисунку 2.4 . Як показано у FSA, HasNextOnce має адаптивну структуру, оскільки ми можемо усунути інструментарій, щойно побачимо hasNext. Таким чином, ми очікуємо, що адаптивний аналіз стану типів добре працюватиме для hasNext . З іншого боку, HasNext представляє найгірший сценарій для AOPA, тому що під час кожного виклику next необхідно відновити моніторинг hasNext, тому жодні інструменти неможливо вимкнути.

Рисунок 2.4 – Скінченні автомати для HasNext (ліворуч) і HasNextOnce (праворуч)

У наших попередніх дослідженнях ми помітили, що програма bloat у має найбільше використання класу ітератора. Таким чином, ми включаємо bloat у цей експеримент разом. Крім того, tomcat, tradebeans, tradesoap і xalan не використовують ітератори у своєму виконанні, тому їх виключено з цього експерименту.

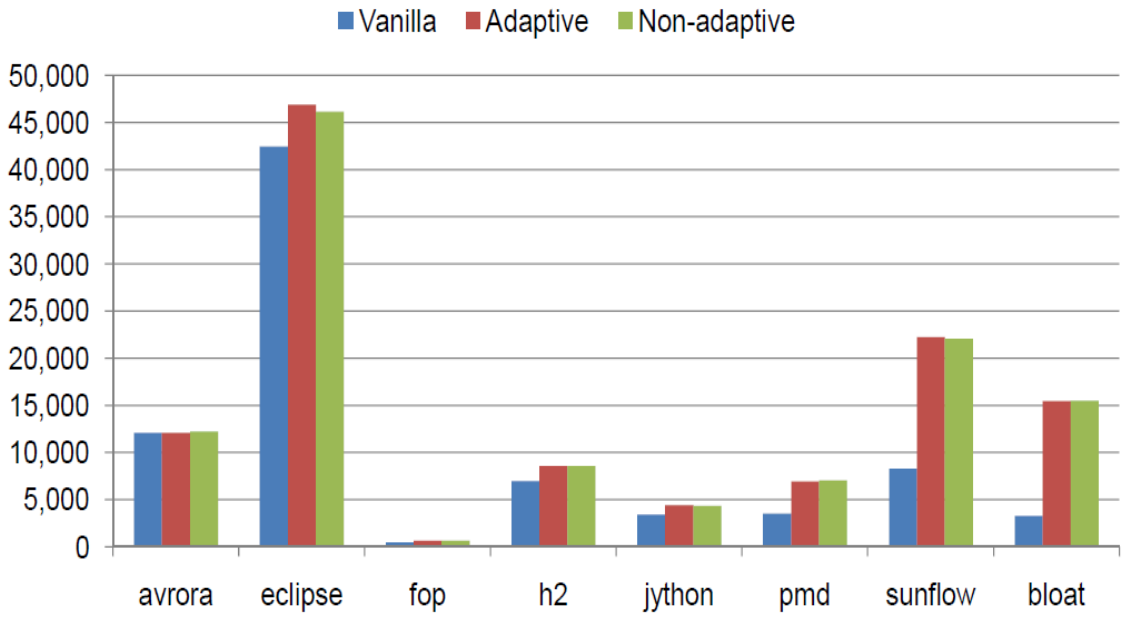


Рисунок 2.5 – Час виконання DaCapo з властивістю HasNext

На рисунках 2.5 і 2.6 ми повідомляємо час виконання кожного тесту DaCapo за допомогою властивостей HasNext і HasNextOnce відповідно. Окрім кількості викликів маркера, таблиці 2.5 і 2.6 також показують скорочення часу виконання та викликів, досягнуте AOPA.

Таблиця 2.6

Час виконання та виклики маркера властивості HasNextOnce

		Adaptive		Non-adaptive		Reduction	
Benchmark	Vanilla	Time (ms)	Count	Time (ms)	Count	Time (%)	Count (%)
avrora	12,066.6	12,308.0	2	11,937.8	26	-3.10%	92.31%
eclipse	42,476.0	47,273.0	3,613	46,714.0	140,774	-1.20%	97.43%
fop	445.6	723.2	16,205	635.6	181,353	-13.78%	91.06%
h2	6,962.1	8,596.7	6,519,560	8,692.3	25,330,249	1.10%	74.26%
jython	3,415.0	4,353.8	26,154	4,255.0	307,590	-2.32%	91.50%
pmd	3,519.0	4,102.2	566,214	4,374.6	1,868,874	6.23%	69.70%
sunflow	8,287.8	9,517.4	1,365,234	9,477.2	3,922,933	-0.42%	65.20%
bloat	3,277.2	6,503.2	941,270	14,920.6	147,604,481	56.41%	99.36%

Таблиця 2.7

Час виконання та виклики маркера властивості HasNext

		Adaptive		Non-adaptive		Reduction	
Benchmark	Vanilla	Time (ms)	Event	Time (ms)	Event	Time (%)	Event (%)
avrora	12,066.60	12,095.60	26	12,230.80	26	1.11%	0.00%
eclipse	42,476.00	46,894.00	99,740	46,158.00	140,768	-1.59%	29.15%
fop	445.60	633.00	176,155	623.40	181,353	-1.54%	2.87%
h2	6,962.10	8,565.30	25,334,946	8,560.40	25,334,340	-0.06%	0.00%
jython	3,415.00	4,366.40	31,552	4,355.20	307,590	-0.26%	89.74%
pmd	3,519.00	6,917.20	1,562,249	6,996.20	1,868,551	1.13%	16.39%
sunflow	8,287.80	22,254.50	3,931,300	22,032.60	3,931,595	-1.01%	0.01%
bloat	3,277.20	15,418.80	148,490,519	15,496.00	149,149,959	0.50%	0.44%

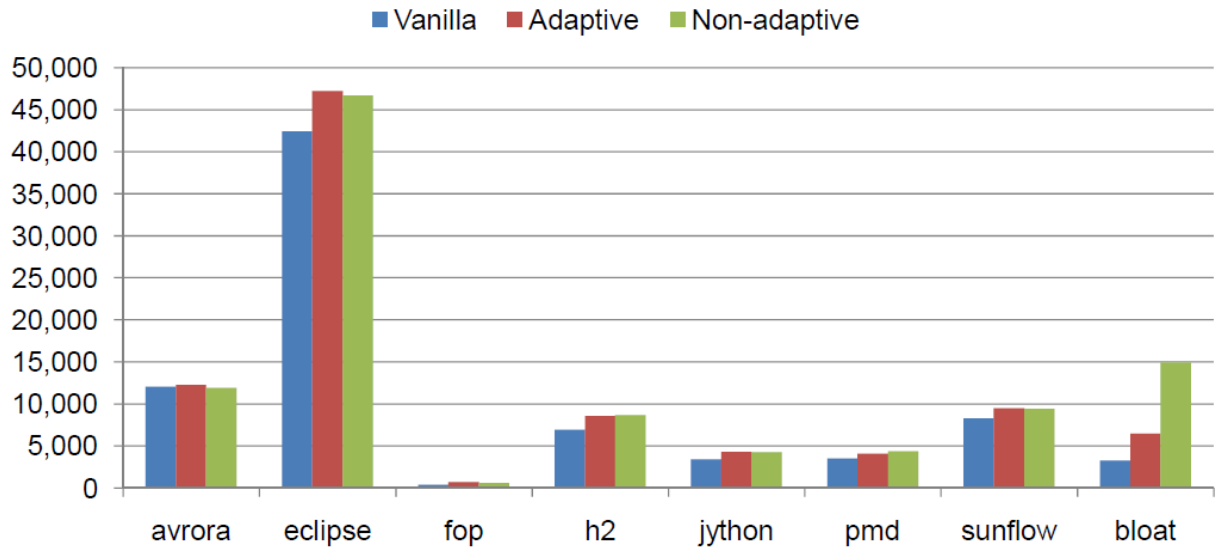


Рисунок 2.6 – Час виконання DaCapo з властивістю HasNextOnce

Потрібно зауважити, що, хоча ми можемо досягти понад 90% скорочення кількості подій, вони безпосередньо не призводять до скорочення часу, як вказують негативні скорочення. Такі результати пов’язані з тим, що накладні витрати на застосування оптимізації переважають витрати на обробку маркерів, коли кількість викликів маркерів недостатньо велика. Наприклад, навіть без оптимізації avrora має лише 26 викликів, тому оптимізація стає надмірною. Тим не менш, вимірювання викликів маркерів в обох таблицях чітко доводять ефективність оптимізації щодо зменшення непотрібних подій маркерів. Найбільш типовим випадком є властивість

HasNextOnce для bloat, яка має найбільше використання ітераторів серед усіх тестів програм із скороченням часу та подій на 60,78% і 99,36%.

Підсумовуючи, ми можемо зробити висновок, що коли кількість маркерних подій достатньо велика, наприклад, понад 1000000, варто застосувати AOPA, оскільки це може значно зменшити накладні витрати на прилади. В іншому випадку така оптимізація не може забезпечити кращу продуктивність, оскільки властиві накладні витрати можуть переважити виграш від зменшення подій.

Висновки до розділу 2

В даному розділі представляно реалізацію клієнта динамічного аналізу стану типів на основі MTF. Клієнт використовує FSA для перевірки властивостей стану типу з оптимізацією AOPA, яка реалізована технікою диспетчеризації преамбули методу. Щоб підтримати перевірку остаточного стану типу, коли об’єкт видаляється, ми використовуємо евристику обробки JNI та фіксуємо подію смерті об’єкта в кінці кожного збору сміття. Результати показують, що пропонований клієнт аналізу стану типів здатний виявляти порушення властивостей і несе прийнятні накладні витрати. Наша реалізація дозволяє включати додаткові методи, наприклад, вибірку та статичний аналіз, щоб ще більше зменшити накладні витрати.

РОЗДІЛ 3. ОПТИМІЗАЦІЯ МОДЕЛЕЙ НА ОСНОВІ ВІРТУАЛЬНИХ МАШИН ДЛЯ ПОКРАЩЕННЯ АНАЛІЗУ ДОДАТКІВ ШЛЯХОМ РЕАЛІЗАЦІЇ ФРЕЙМВОРКУ ТРАСУВАННЯ МАРКЕРІВ

3.1 Сутність концепції трасування маркерів при розробці та аналізі програмних додатків

У розробці програмного забезпечення трасування – це процес безперервного запису та аналізу певних аспектів виконання програми. Корисність методів, заснованих на трасуванні, була продемонстрована великою кількістю попередніх дослідницьких робіт з різними фокусами, наприклад, профілювання [36, 35], налагодження [21] і динамічний аналіз програм [48]. Наївні механізми трасування програм можуть уповільнювати виконання програми на порядки величини та генерувати величезні журнали трасування. Трасування також вносить роздутість коду в цільову програму через додаткову логіку для запису та обробки відстежених даних. У цьому розділі ми представляємо інфраструктуру на основі JVM, ядром якої є легка і ненав’язлива структура трасування – Marker Tracing Framework (MTF).

Трасування є основною технікою ефективного програмного інструментування. Дослідники та розробники додатків використовують методики на основі трасування для вирішення реальних проблем із бажаними результатами. Існують різноманітні техніки для включення трасування логіку в програмах Java. Найпростіший підхід полягає в застосуванні спеціального коду трасування безпосередньо в цільових програмах. Однак ці рішення можуть забруднити вихідний код, і код трасування може бути непридатним для повторного використання в інших проектах. Більш елегантним і гнучким рішенням є використання утиліт на основі AOP для додавання трасування до вихідних програм.

Наприклад, одна з найбільш широко використовуваних реалізацій AOP – AspectJ [42], підтримує дуже складні схеми для точного визначення частин програми, які повинні бути інструментованими. Крім того, оскільки інструментальний код написаний простою мовою Java, програмісти можуть швидко розробити ефективний інструментальний код, використовуючи всі функції мови та стандартні бібліотеки Java. AOP динамічно вплітає аспекти під час компіляції, таким чином інструментальний код повністю відокремлюється від цільової програми.

Такі методи відстеження на рівні програми непридатні для певних типів аналізу, де потрібна інформація низького рівня. Наприклад, виявлення витоків ресурсів і пам’яті для Java потребує відстеження смертей об’єктів, доступ до яких доступний лише в JVM. Щоб задовольнити такі потреби, люди використовують підходи на основі віртуальної машини, наприклад, JVMTI та DTrace, оскільки такі структури забезпечують інтерфейси для доступу та зміни даних лише віртуальної машини. В [9] підсумовуються переваги використання інструментів на основі VM як:

- Інформація лише про VM. Клієнтський аналіз може отримати доступ до наявної інформації про час виконання та записати нову інформацію, яка неможлива на мовному рівні, як-от крадіжка вільних бітів у заголовках об’єктів, кешування даних у локальному сховищі потоку та повторне використання існуючих служб віртуальної машини.
- Продуктивність. Дані профілю від інтерпретаторів і компіляторів JIT можна використовувати для налаштування клієнтів аналізу.
- Динамічне оновлення. Розширені методи, наприклад виправлення коду та заміна в стеку (OSR - on-stack replacement), можна використовувати для динамічного налаштування інструментарію під час виконання.
- Розгортання. Інструменти побудовано всередині JVM, тому програма не має модифікацій, тому будь-які програми можуть використовувати структуру, якщо вони працюють на такій JVM.

Хоча ці інфраструктури виявилися загалом корисними, вони все ще обмежені в певних аспектах і можуть бути покращені в таких аспектах:

- Типи подій. Існуючі рішення використовують інтерфейси програмування для надання послуг, приховуючи деталі віртуальних машин. Незважаючи на те, що інтерфейс є всеохоплюючим, все одно неможливо задовольнити всі вимоги завдань інструментарію в реальному світі. Одним із найбільш обмежувальних факторів є фіксовані типи подій, які обмежують фіксацію точних станів програми.
- Фільтрування подій. Більшість існуючих рішень не підтримують визначені користувачем фільтри, які оцінюються всередині JVM перед викликом функцій зворотного виклику. Таким чином, клієнтам аналізу необхідно прислухатися до всіх подій і відкидати нецікаві. Отримані перемикання контексту можуть спричинити значні витрати на виконання. Тому бажано, щоб інфраструктура дозволяла програмістам вибірково вибирати, які події мають бути доставлені, і навіть динамічно коригувати їх під час виконання.
- Інформація про контекст. Існуючі API надають доступ до інформації віртуальної машини, надаючи набори інтерфейсів програмування. Викликаючи такі підпрограми, клієнти аналізу можуть запитувати широкий спектр контекстної інформації, доступної віртуальній машині. Однак інтерфейси програмування не можуть задовольнити всі можливі потреби аналізу програм у реальному світі. Потрібна здатність структур, які дозволяють таким аналізам отримати необхідну інформацію.

3.2 Архітектура фрейворку MFT

Як показано на рисунку 3.1, структура MTF складається з трьох частин:

1. VM Services, які розширюють існуючі компоненти VM і реалізують байт-коди маркерів і механізми трасування;

2. Analysis Manager, який керує всіма клієнтами аналізу та надає клієнтам основні служби віртуальної машини;

3. Instrumentation Utility, платформа для інструментування програм із байт-кодами маркерів.

В цьому підрозділі ми наведемо основні концепції та малюємо дизайн менеджера аналізу та утиліти інструментарію.

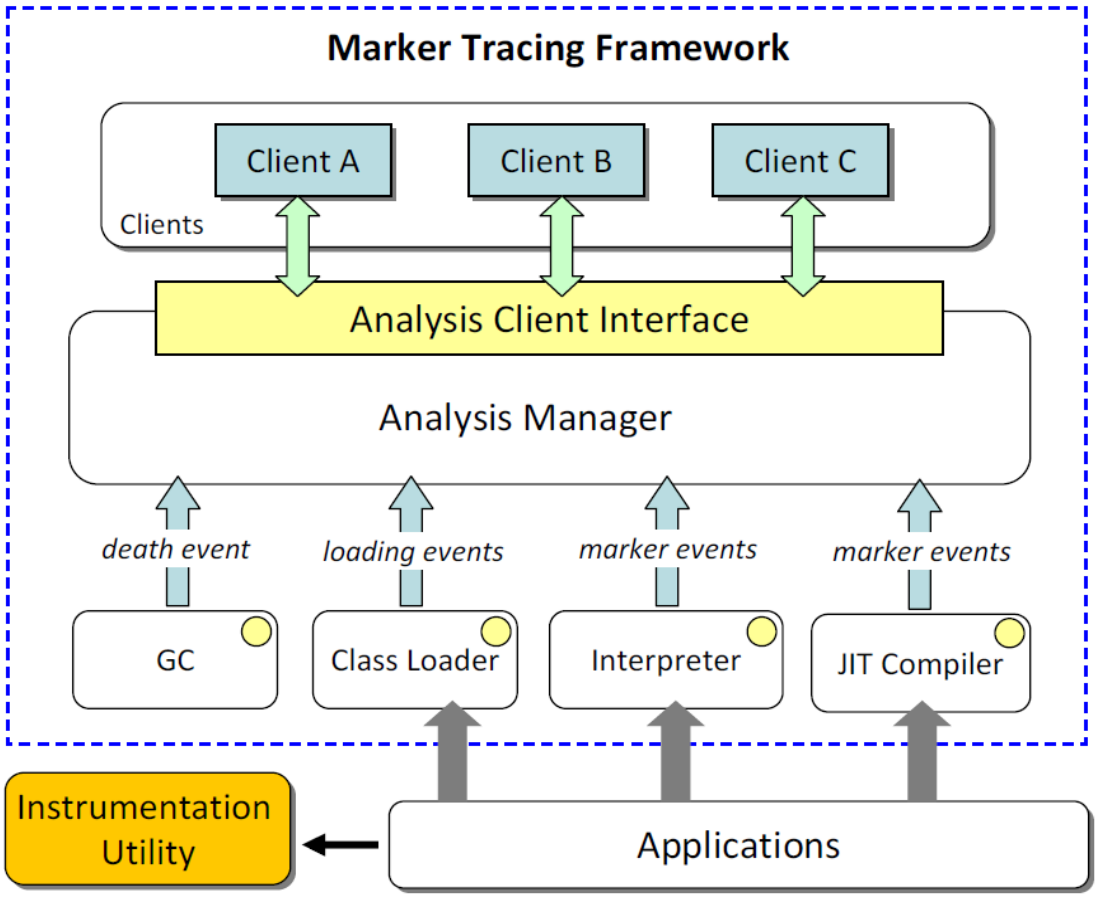


Рисунок 3.1 – Загальна архітектура MTF

Маркер. Основною концепцією фреймворку є маркер, спеціальний «тег», який можна вставити навколо області коду будь-де всередині методу. Кожен маркер представлений і посилається в коді за допомогою цілого ідентифікатора 1, який називається ідентифікатором маркера. Ми розрізняємо вхід і вихід маркера (області коду), винайшовши два нових байт-коди в набір інструкцій Java, тобто markerenter і markerexit, подібно до байт-кодів

монітора Java, які синхронізують байт-коди. Обидва байт-коди приймають цілочисельний ідентифікатор маркера як єдиний операнд. Рисунок 3.2 показує приклад оснащення маркером ідентифікатора 2 у наступному методі. Наступний метод визначається класом ітератора всередині класу EDU.purdue.cs.bloat.util. Graph.

```
public java.lang.Object next();
    Code:
       0: markerenter     2 (mid)
       3: aload_0
       4: getfield        nodeModCount:I
       7: aload_0
       8: getfield        LEDU/purdue/cs/bloat/util/Graph$1;
          ...
          ...
      36: invokeinterface java/util/Iterator.next:()Ljava/lang/Object;
      39: markerenter     1 (mid)
      42: checkcast       java/util/Map$Entry
      44: putfield        last:Ljava/util/Map$Entry;
      47: markerexit      1 (mid)
      50: aload_0
      51: getfield        last:Ljava/util/Map$Entry;
      54: markerexit      2 (mid)
      57: areturn
```

Рисунок 3.2 – Приклад застосування маркерів

Кожен екземпляр маркера пов'язаний з основним набором контекстної інформації, тобто ідентифікатором маркера, ідентифікатором потоку та активним методом і об'єктом приймача. Така інформація необхідна для впровадження складних клієнтів аналізу з потоками та контекстом чутливості. Оскільки інструментальні клієнти реалізовані всередині JVM, більше інформації можна отримати, використовуючи існуючі служби виконання, наприклад, інтерпретатор, JIT і збирачі сміття.

У MTF маркер також може бути без байт-коду виходу; такі маркери називаються односторонніми маркерами, які підходять, коли вони використовуються для позначення події в певному розташуванні програми, де область дії непотрібна. У цьому сценарії ми можемо визначити односторонні

маркери, щоб зменшити накладні витрати, спричинені непотрібними перемиканнями контексту.

Дескриптор маркера. Дескриптор маркера — це специфікація класу маркерів, які мають однакову семантику та процедури обробки, подібно до того, як клас — це шаблон набору об'єктів в об'єктно-орієнтованому програмуванні. Дескриптори маркера складаються з трьох частин, тобто ідентифікатора маркера, предикату та набору обробників зворотного виклику, як показано на рисунку 3.3. Дескриптори маркерів безпосередньо вбудовані в пул констант файлу класу.

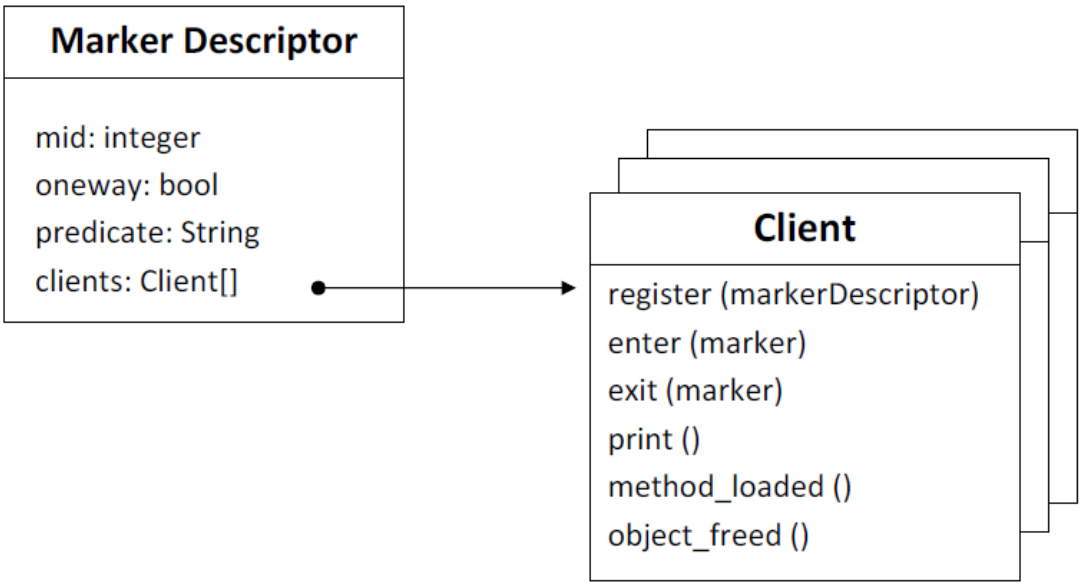


Рисунок 3.3 – Структура дескриптора маркера

Другий слот ділиться двома рядковими полями: маркером і предикатом. Поле маркера вказує на тип маркера, який зацікавлені клієнти аналізу розпізнають і підписують. З іншого боку, поле предикату є необов'язковим і використовується для вказівки допоміжної інформації для роботи обробників маркерів. Обидва поля вільно кодуються клієнтами для належного представлення інформації, необхідної для проведення аналізів.

Стек маркерів. Стек маркерів — це буфер пам'яті для зберігання екземплярів маркерів у порядку « останній прийшов – першим вийшов»

(LIFO - last-in-rst-out). Кожен потік має власний стек маркерів, так що операції натискання та висунення маркера є потокобезпечними, але не блокуються. Рисунок 3.4 ілюструє схему стека маркерів для кожного потоку.

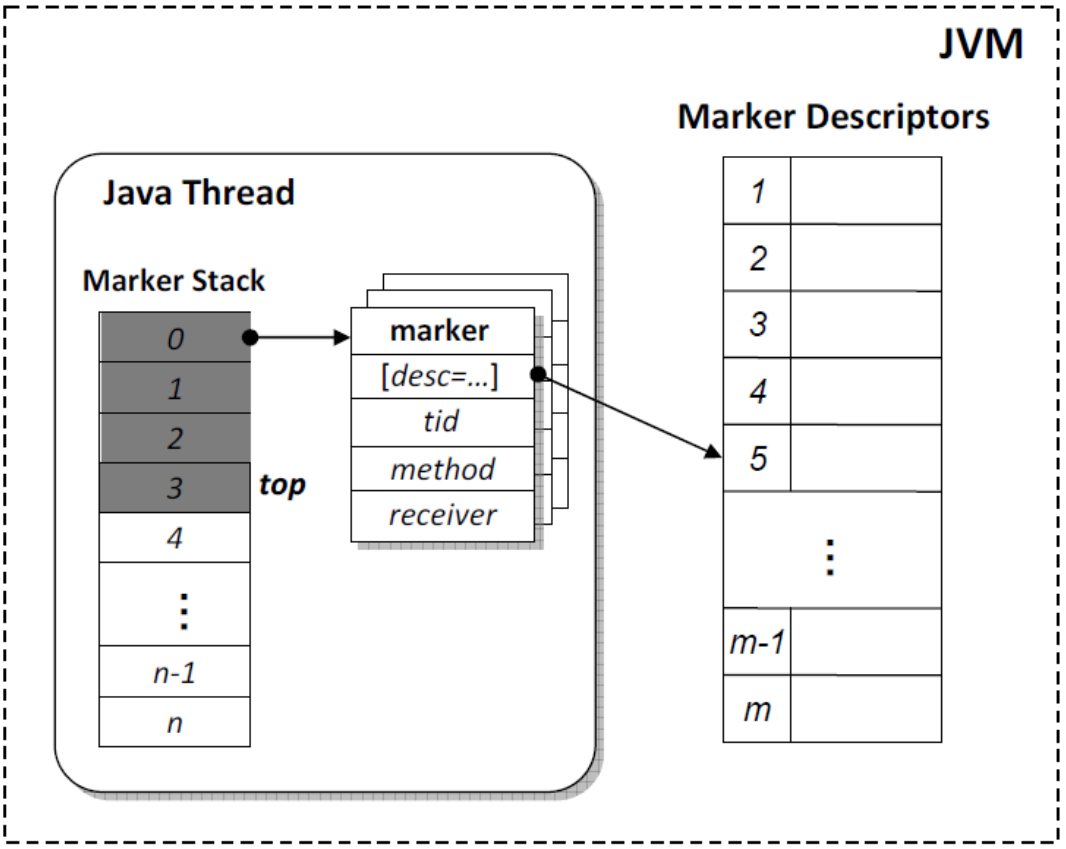


Рисунок 3.4 - Стек маркерів для кожного потоку

Така функція розроблена для клієнтів аналізу, яким потрібна контекстна залежність маркерів. Перегляд стека маркерів має набагато менші накладні витрати, ніж стек викликів, і є більш гнучким, оскільки маркери можна вставляти в будь-які місця програми.

Менеджер аналізу. У MTF менеджер аналізу контролює та взаємодіє з усіма клієнтами аналізу. Для модульності та простоти керування кожен клієнт аналізу повинен реалізувати інтерфейс, визначений фреймворком, члени якого перераховані в таблиці 3.1. Крім того, зв’язок між клієнтами аналізу та фреймворком також базується на цьому інтерфейсі.

Таблиця 3.1

Інтерфейс Analysis client

Method	Description
`register`	Registers an analysis client to the framework
`subscribe`	Decides whether to subscribe to a specific marker.
`marker_enter`	Marker enter event callback
`marker_exit`	Marker exit event callback
`object_death`	Object death event callback
`method_loaded`	Callback for processing loaded methods that have markers
`print`	Callback for printing analysis-specific results

Зокрема, клієнти аналізу реєструються в MTF і залишаються в списку клієнтів аналізу. Під час фази завантаження класу MTF аналізує пули констант у пошуках дескрипторів маркерів. Для кожного знайденого дескриптора MTF опитує зареєстрованих клієнтів із предикатним рядком як аргументом. Очікується , що підпрограма підписки кожного клієнта повертає логічне значення, яке вказує, чи хоче він підписатися на події маркера, як зазначено дескриптором. Коли MTF отримує правдиву відповідь, він приєднує клієнта до списку кожного дескриптора, який містить усіх підписників. Під час виконання програми, коли MTF спостерігає за появою маркера, він індексує список дескрипторів з ідентифікатором маркера та повторює список, викликаючи кожен обробник. MTF створює структуру маркера з дескриптором і поточною контекстною інформацією, такою як ідентифікатор потоку та покажчик стека, і передає маркер обробникам як єдиний аргумент.

Виклик адаптивного маркера. Щоб дозволити клієнтам аналізу адаптивно вмикати та вимикати інструментарій маркерів під час виконання, ми досліджуємо два альтернативні рішення на основі віртуальних машин у MTF, зосереджуючись на низьких витратах на комутацію.

Заміна вказівників методів. Як описано раніше, сучасні JVM компілюють байт-коди Java у власні інструкції для часто виконуваних методів за допомогою компілятора JIT. Як правило, скомпільовані методи

зберігаються в буферах коду і на них посилаються покажчики. Після завершення компіляції наступний виклик кожного такого методу слідує за покажчиком на відповідний буфер коду та починає виконання.

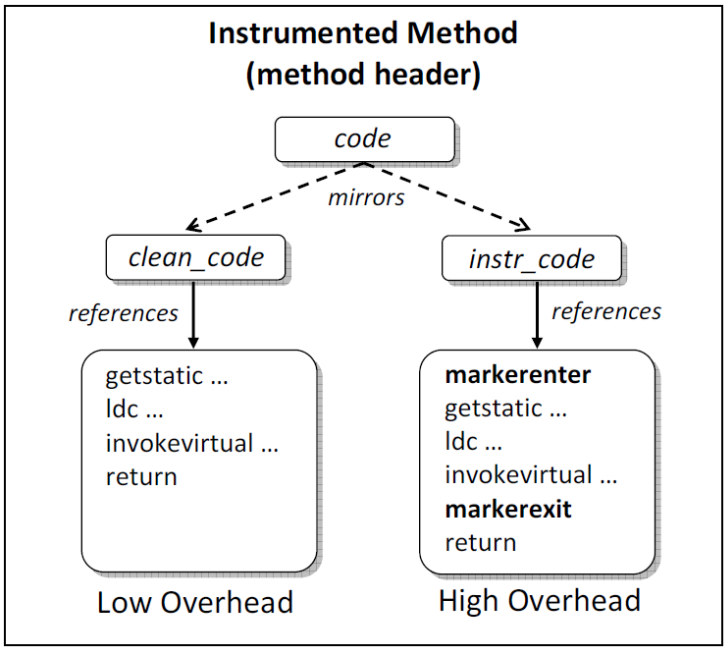


Рисунок 3.5 – Адаптивний онлайн-аналіз програм: підхід MPS

Базуючись на цьому дизайні, Method Pointer Swapping (MPS) додає два нових покажчики, instr_code та clean_code в заголовок методу поряд з оригінальним кодом код покажчика. Крім того, MPS розширює компілятор JIT для виконання двох компіляцій, включаючи та виключаючи інструментарій маркера. Отримані буфери коду посилаються на instr_code та clean_code відповідно. Під час аналізу MPS може адаптивно перемикатися між інструментальним і неінструментованим кодом, призначаючи або instr_code або clean_code коду, який є глобальною точкою входу, що використовується у всій віртуальній машині. MTF виконує таке перемикання від імені кожного клієнта аналізу за запитом. Рисунок 3.5 показано підхід до заміни вказівника методу.

Ця схема схожа на версію інструментарію з повним дублюванням, представлена в [8]. Однією з переваг є те, що неінструментований код працює

на повній швидкості, коли інструменти не потрібні. Однак він також має кілька серйозних недоліків. По-перше, кожен скомпільований метод вимагає двох компіляцій і двох буферів коду, таким чином витрачаючи час і простір. Продуктивність ще більше погіршується, коли JIT-компілятори виконують кілька повторних компіляцій, щоб отримати найкращу оптимізацію, яка є звичайною оптимізацією в сучасних JVM. По-друге, ця техніка вимагає суттєвих модифікацій віртуальної машини, щоб підтримувати синхронізацію заголовка методу та двох буферів. Через ці недоліки ми не використовували підхід MPS у нашому остаточному рішенні MTF.

Для динамічного перемикання інструментального коду функція диспетчеризації преамбули маркера (MDP – Marker Preamble Dispatching) додає крихітний фрагмент преамбули коду перевірки та відправлення для кожного скомпільованого маркера. На відміну від MPS, MDP генерує лише один буфер коду на метод, а байт-коди маркерів завжди компілюються та інтегруються в остаточний буфер, як показано на рисунку 3.6 . Щоб досягти такої ж адаптивності, MDP перевіряє додатковий прапорець у заголовку об’єкта, який доступний кожному клієнту аналізу. Прапор являє собою бітову маску з полями для всіх маркерів.

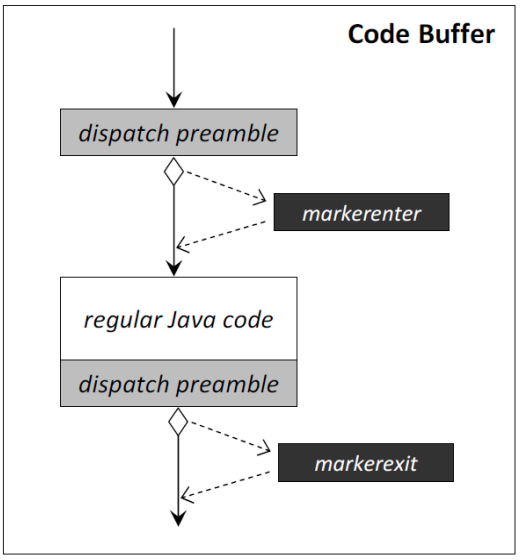


Рисунок 3.6 – Адаптивний онлайн-аналіз програм: підхід MDP

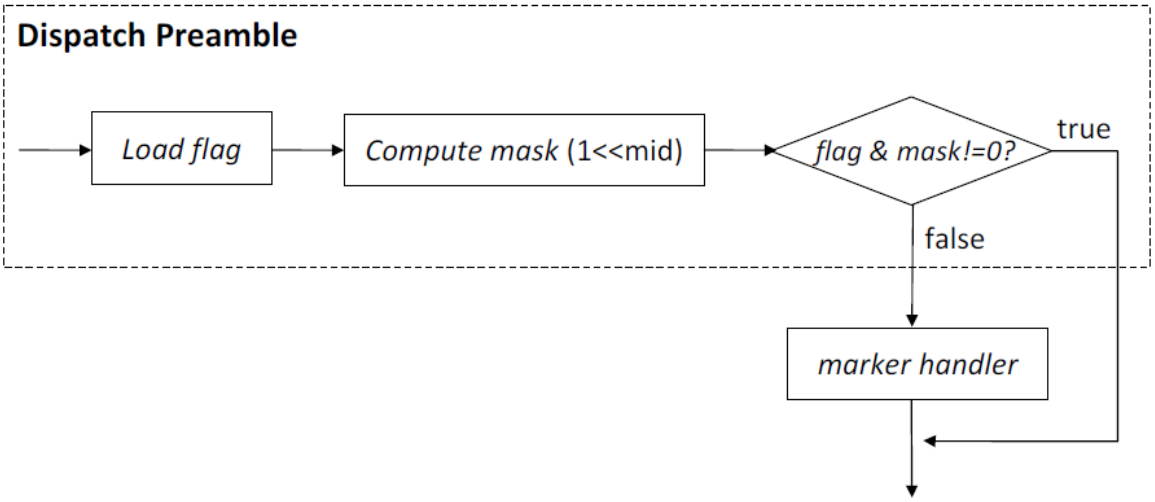


Рисунок 3.7 – Потік керування преамбулою відправки для кожного байт-коду маркера

Залежно від стану біта, MDP або виконує, або перескакує певний інструментарій маркера. рисунок 3.7 показує потік керування кодом перевірки преамбули, вставленим для кожного байт-коду маркера.

MDP має подібний дизайн, як і підхід без дублювання, представлений у [8]. На відміну від MPS, MDP не потребує додаткової компіляції чи буфера коду. Крім того, код перевірки складається лише з кількох машинних інструкцій, тому додає лише незначні накладні витрати. Тим не менш, масивний потік подій може спричинити помітні накладні витрати, хоча ми вважаємо, що такий сценарій є дуже рідкісним на практиці. Крім того, реалізація бітового вектора не масштабується, оскільки складність аналізу зростає, коли для простоти та ефективності використовуються вектори фіксованої довжини. Насправді ми очікуємо, що кількість усіх активних маркерів у сеансі аналізу буде меншою за 32 або 64, для чого потрібні лише вектори довжиною слова або подвійного слова.

Незважаючи на обмеження MPD, ми реалізували підхід MPD в остаточному рішенні через його концептуальну та технічну простоту.

Інструментальна утиліта. Оскільки стандарт Java не вказує байт-коди маркерів, тобто markerenter і markerexit, жодна з існуючих інструментальних

утиліт не може бути використана для аналізу на основі MTF без змін. У цій роботі ми пропонуємо прототип утиліти на основі інструментарію байт-коду для анотування програм байт-кодами маркерів. Утиліта розроблена таким чином, що її можна розширювати, щоб можна було легко розробляти та додавати нові клієнтські інструменти для підтримки нових аналізів. Утиліта працює з вхідним файлом класу та виводить інструментальну копію з наявними байт-кодами, збереженими незмінними, щоб зберегти оригінальну семантику.

Утиліта використовує існуючу структуру BCI, ASM [30], для маніпулювання байт-кодами, що зберігаються у двійковому файлі класу. Кожен клієнт аналізу повинен реалізувати інтерфейс InstrumentationClient (табл 3.2) , що визначається корисністю. В ASM ClassVisitor — це інтерфейс верхнього рівня, який може проходити та перетворювати пов’язані з класом елементи, такі як прапорці доступу, ім’я класу та пул констант. Спеціальні відвідувачі, наприклад MethodVisitor і FieldVisitor, створюються ClassVisitor для обробки методів і полів відповідно. Кожен InstrumentationClient визначає настроюваний ClassVisitor для вбудовування дескрипторів маркерів у постійний пул. Спеціальний MethodVisitor також потрібен для додавання байт-кодів маркерів у відповідні методи.

Таблиця 3.2

Інтерфейс instrumentation client

Method	Description
`getClassVisitor`	Returns the custom class visitor for annotating the bytecodes.
`postProcess`	Optional post-processing step for some instrumentation clients.

Основна утиліта виконує введення/виведення файлів класу та створює екземпляр інструментального клієнта, як зазначено користувачем із відображенням. Потім утиліта проходить файл класу за допомогою ClassVisitor, повернутого клієнтом.

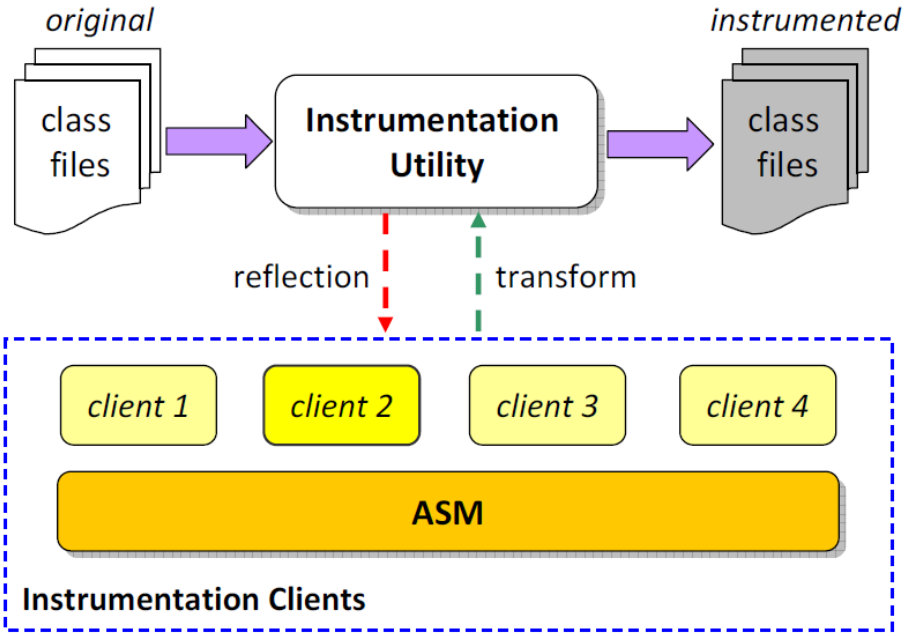


Рисунок 3.8 – Представлення робочого процесу

Отже, відповідні інструментальні завдання виконуються вибраним клієнтом. Після виконання завдань утиліта зберігає інструментований клас у призначений файл класу. Рисунок 3.8 ілюструє робочий процес інструментів для маркерів, які виконуються цією утилітою. MTF базується на гібридному підході, тобто на клієнтах аналізу на основі віртуальної машини під час виконання в поєднанні з клієнтами інструментів під час компіляції, щоб забезпечити функціональність трасування на основі детального аналізу. рисунок 3.1 показує загальну архітектуру структури трасування маркерів на високому рівні. Аналіз програм за допомогою MTF передбачає кроки:

- охарактеризувати місця розташування програми, наприклад, запис методу, заголовок циклу, куди слід повідомляти аналіз;
- визначити формат предикатного рядка дескриптора маркера;
- вставте маркери в ці місця та вбудуйте дескриптори маркерів у постійний пул;
- реалізувати та підключити клієнт аналізу до віртуальної машини для обробки маркерних подій;

- запустіть інструментальну програму на віртуальній машині з підтримкою MTF за допомогою клієнта аналізу.

3.3 Реалізація фреймворку трасування маркерів

Оскільки MTF розширює набір інструкцій Java і процес завантаження класів, кілька компонентів усередині віртуальної машини потрібно змінити для реалізації підтримки MTF. На рисунку 3.1 показано огляд архітектури реалізації MTF. У цьому підрозділі описано деталі кожного основного компонента нашого впровадження в HotSpot JVM.

Розширення HotSpot JVM. Sun реалізувала HotSpot JVM для підтримки двох архітектур: x86 і SPARC для 32-розрядної та 64-розрядної моделей пам’яті. Більшість віртуальної машини не залежить від платформи. Частини системи середовища виконання, наприклад, інтерпретатор, генератор рідного коду та менеджер кадрів, однак залежать від платформи. Хоча наша реалізація MTF є нейтральною для ОС, вона виконується лише на x86_32 та x86_64 , оскільки інтерпретатор шаблону та JIT потрібно модифікувати для MTF. Однак реалізація має бути легко переносимою на інші архітектури з однаковою ефективністю:

```
bool has_markers = marker_index() > 0 ? true : false;
if (has_markers) {
  for (int i=marker_index(); i<length; i+=2){
    // parse marker id
    symbolOop sym = cp->symbol_at(i);
    {
      ResourceMark rm(THREAD);
      buf = sym->as_C_string();
      int len = strlen(buf);
      if (strchr(buf, '*')) { // one-way marker
        oneway = true;
        buf[len-1] = '\0';
      }
      mid = atoi(buf);
```

```
        // parse predicates
        sym = cp->symbol_at(i+1);
        buf = sym->as_C_string();
        copy_str(buf, preds);
      }

      markerDescriptor* md;
      if (mid != 0) // regular marker
        md = markerDescManager::add(mid, class_name, preds, oneway);
      else // psuedo marker
        md = new markerDescriptor(mid, class_name, preds, oneway);

      analysisManager::accept(md);
    }
  }

  Klass* clazz = cp->pool_holder()->klass_part();
  clazz->set_has_code_markers(has_markers);
```

Рисунок 3.9 – Процедура аналізу дескриптора маркера

Завантажувач класів початкового завантаження в HotSpot розширено для аналізу дескрипторів маркерів із постійного пулу кожного файлу класу. Зокрема, підпрограма аналізу постійного пулу модифікована для запису індексу роздільника, представленого 32-бітним цілим числом, тобто Oxbabecafe, у нашій реалізації. Процедура синтаксичного аналізу дескриптора маркера працює над записами, починаючи з індексу, і виконує операції, як показано на рисунку 3.9 .

Підпрограма перевіряє, чи має поточний клас маркери. Якщо така умова виконується, він отримує ідентифікатор маркера, напрямок (one-way) і рядок предикату (preds) із записів пулу. Потім створюється дескриптор маркера з назвою класу та зібраними значеннями. Після цього він викликає метод accept у analyManager , який виконує ітерацію по всіх зареєстрованих клієнтах аналізу з дескриптором маркера. Клієнти можуть повертати справжнє значення, вказуючи на те, що вони хочуть отримувати повідомлення про випадки появи маркера, як описано в дескрипторе. Після обробки всіх дескрипторів підпрограма кешує наявність маркерів коду в об’єкті Klass

поточного класу Java. Цей прапорець консультується структурою для спеціалізованої обробки в інтерпретаторі та JIT-компіляторі. Клієнти аналізу також можуть переглянути позначку, якщо необхідно.

У HotSpot складні операції середовища виконання, такі як виділення повільного шляху, упереджене блокування та вирішення віртуальних методів, обробляються двома базовими системами виконання: InterpreterRuntime і SharedRuntime. InterpreterRuntime обслуговує лише інтерпретатор байт-коду, тоді як SharedRuntime надає послуги як для інтерпретованого, так і для скомпільованого коду. Нетривіально реалізувати підпрограми маркерів шляхом генерації шаблонів складання (interpreter) або конструювання проміжних представлень (JIT). Таким чином, ми реалізуємо підпрограми маркерів у SharedRuntime таким чином, що як інтерпретований, так і скомпільований код можуть обробляти маркери, запитуючи такі служби у SharedRuntime. Рисунок 3.10 показує реалізацію markerenter як методу SharedRuntime.

```
void SharedRuntime::markerenter(JavaThread* thread, int mid) {
  pid_t tid = thread->tid();
  intptr_t *sp = thread->last_Java_sp();
  markerStack *st = thread->mkstack();

  // notify predicate watchers
  analysisManager::marker_enter(st->push(mid, tid, sp));
}
```

Рисунок 3.10 – Обробник markerenter у SharedRuntime

Як інформацію про контекст маркера, markerenter збирає поточний ідентифікатор потоку та вказівник стека та поміщає маркер у стек маркерів для кожного потоку. Потім він повідомляє AnalysisManager, який, у свою чергу, передає подію маркера клієнтам-передплатникам. Інші клієнти, які скасували підписку, пропускаються, будучи поза списком дескрипторів поточного маркера.

Ми розширюємо інтерпретатор шаблонів для підтримки байт-кодів маркерів у HotSpot. В інтерпретаторі шаблонів кожен байт-код інтерпретується шаблоном коду складання, створеним під час запуску віртуальної машини. За винятком розгалуження та повернення байт-кодів, епілог кожного шаблону отримує наступний байт-код і надсилає до відповідного шаблону. Кожен шаблон може визначати атрибути середовища виконання, наприклад, формат операнда, тип результату, ефект стека та викидання винятку.

Таблиця 3.3 показує визначення наших маркерних байт-кодів, де «bcc» означає один байт-код, за яким слідують два байти (один короткий); VOID означає, що значення не повертається; 0 означає, що байт-код не змінює стек; і true вказує на те, що байт-коди можуть викликати винятки.

Таблиця 3.3

Визначення байт-кодів маркерів

Bytecode	Format	Result	Stack	Exception
markerenter	bcc	VOID	0	true
markerexit	bcc	VOID	0	true

Генератором кожного шаблону коду є метод C++ у класі TemplateTable, який реалізовано спеціально для кожної архітектури, тобто x86_32, x86_64 та SPARC. У HotSpot низькорівнева генерація коду виконується асемблером макросів, який не тільки реалізує базовий набір інструкцій, наприклад MOV , CMP , XOR , але також надає основні макроси для типових завдань, таких як виклик під час виконання, доступ до байт-коду та перевірка нульового покажчика. За допомогою InterpreterMacroAssembler ми реалізували байт-коди маркерів як два методи в TemplateTable для x86_32 і x86_64. Рисунок 3.11 надає реалізацію markerenter на x86_64. Шаблон починається із завантаження двобайтового цілого числа без знаку (ідентифікатора маркера) у

поточний вказівник байт -коду (bcp) і збереження значення для реєстрації rbx. Потім він викликає SharedRuntime::markerenter з ідентифікатором маркера в rbx як аргумент.

```
void TemplateTable::markerenter() {
  transition(vtos, vtos);
  _masm->get_unsigned_2_byte_index_at_bcp(rbx, 1);
  _masm->call_VM(noreg, CAST_FROM_FN_PTR(address,
                 SharedRuntime::markerenter), rbx);
}
```

Рисунок 3.11 – Генератор шаблону коду markerenter на x86_64

У HotSpot метод Java компілюється за допомогою JIT, якщо виявлено, що він часто виконується або містить жорсткий цикл. Щоб підтримувати маркери в таких методах, ми розширюємо серверний компілятор JIT (C2) у HotSpot для компіляції байт-кодів маркерів у власні інструкції, такі ж, як і звичайні байт-коди.

C2 базується на графі програмних залежностей («море вузлів»), як проміжне представлення з кількома фазами, наприклад, розбір, оптимізація та генерація коду. Щоб підтримувати MTF, ми модифікували метод do_one_bytecode класу Parse, який має комплексний оператор switch для кожного байт-коду Java. Ми представили дві нові мітки для аналізу байт-кодів маркерів у операторі switch. Подібно до шаблонів маркерного коду в інтерпретаторі, ми генеруємо еквівалентну послідовність дій за допомогою IR компілятора. Однак нам не потрібно завантажувати ідентифікатор маркера під поточний bcp, як у шаблонах інтерпретатора, які є загальними підпрограмами для всіх викликів маркерів, оскільки ідентифікатор маркера можна отримати статично, досліджуючи потік байт-коду, що компілюється. Таким чином, компілятор може просто згенерувати цілочисельний постійний вузол як операнд для виклику VM, наприклад, SharedRuntime::markerenter .

C2 компілює виклики середовища виконання, використовуючи угоду про виклики Java, яка відрізняється від угоди C++, що використовується методами середовища виконання. Таким чином, C2 генерує код адаптера для кожного методу виконання з належною специфікацією підпису.

Код адаптера з'єднує дві угоди про виклики та, зрештою, виконує виклик під час виконання. Зокрема, підпис визначається методом "{method}_Type()", де "{method}" – це ім'я методу середовища виконання. Подібним чином код адаптера називається «{method}_Java», а покажчик методу C++ — «{method}_C». Таким чином, ми реалізували маркер-обробник _Type, щоб забезпечити однаковий підпис для markerenter і markerexit .

Інструментальна утиліта. Рисунок 3.12 схематично описує алгоритм обробки вхідного файлу класу за допомогою зазначеного інструментального клієнта (обробка помилок опущена для стислості). Підпрограма getBytesFromFile читає файл класу в байтовий буфер.

```
public byte[] instrument(File classFile, String className) {
  // Start processing the class file
  byte[] input = getBytesFromFile(classFile);
  ClassReader reader = new ClassReader(input);
  ClassWriter writer = new ClassWriter(reader, 0);

  InstrumentationClient client = (InstrumentationClient)
    Class.forName(className).newInstance();
  ClassVisitor visitor = client.getClassVisitor(writer);
  reader.accept(visitor, 0);
  client.postProcess();

  return writer.toByteArray();
}
```

Рисунок 3.12 – Алгоритм інструментування байт-коду маркера

Щоб зробити інструментальну процедуру загальною, ми створюємо екземпляр інструментального клієнта з назви класу за допомогою відображення Java. Наприклад, «клієнт 2» створено як інструмент, який буде

виконано в програмі на рисунку 3.8 . У нашій утиліті шлях до файлу класу, ім’я класу клієнта та вихідний шлях аналізуються з аргументів командного рядка. Утиліта також підтримує інструментування «розібраних» файлів JAR шляхом рекурсивного переходу до каталогів і обробки кожного файлу класу. Ця функція особливо корисна для пакетного інструментування бібліотек класів зі складною структурою пакетів і великою кількістю класів.

Дескриптори маркерів. Як описано раніше, постійний пул використовується для зберігання дескрипторів маркерів інструментованого класу. В ASM клієнт може перевизначити метод відвідування в ClassAdapter , щоб додати дескриптори маркерів до постійного пулу. Оригінальні записи відокремлюються від записів маркера роздільником — магічним числом (0xbabecafe). Кожен дескриптор маркера представлений двома записами, тобто ідентифікатором маркера (ціле число) та предикатний рядок (UTF8), створений newConst і newUTF8 об’єкта ClassWriter в ASM. На рисунку 3.13 показано схему постійного пулу з двома вбудованими дескрипторами маркерів, причому другий є одностороннім маркером, як зазначено зірочкою.

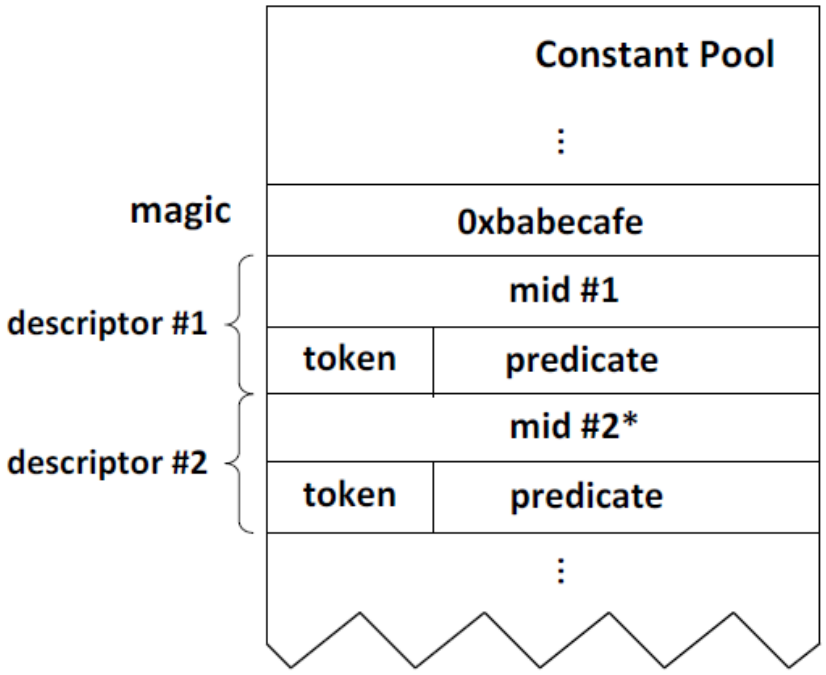


Рисунок 3.13 – Розташування постійного об’єму після вимірювання

Маркери безпосередньо додаються до методів, які слід відстежувати під час виконання. У ASM цього можна досягти шляхом перевизначення відповідних методів відвідувачів у MethodAdapter. Наприклад, клієнт може змінити метод visitCode, щоб додати маркери до записів методу. Оскільки обидва байт-коди маркерів приймають цілочисельний операнд, visitIntInsn в інтерфейсі MethodVisitor можна використовувати для вставки байт-кодів маркерів. Наприклад, visitIntInsn(markerenter,1) вставляє маркер маркера №1 у поточному місці, яке відвідується. На рисунку 3.14 показано, як додати вихід маркера до всіх операторів повернення в поточному методі, де mv є поточним об’єктом MethodVisitor.

```
public void visitInsn(int opcode){
  if (opcode >= Opcodes.IRETURN && opcode <= Opcodes.RETURN){
    mv.visitIntInsn(markerexit, <mid>);
  }

  mv.visitInsn(opcode);
}
```

Рисунок 3.14 – Оператори повернення інструменту в методі з байт -кодом markerexit

Подальша обробка. У режимі пакетної обробки клієнт, як правило, генерує ідентифікатори маркерів у міру проходження інструментарію. Таким чином, кількість контрольних точок може бути недоступною до початку. Однак постійний пул відвідує ClassVisitor перед тим, як інструкції відвідує MethodVisitor. Крім того, дескриптори маркерів повинні вказувати ідентифікатори маркерів у пулі констант під час відвідування. Тому наша утиліта надає метод інтерфейсу postProcess для виправлення пулу констант після відвідування всіх методів. Крім того, postProcess може повертати логічне значення, яке перевіряється на корисність. Якщо повернуто true, утиліта переходить до наступного файлу класу; інакше весь процес вимірювання припиняється. Ця функція корисна, коли клієнт визначає, що він

завершив інструментування всіх цільових класів і хоче взагалі припинити всі наступні обходи.

3.4 Оцінка продуктивності системи трасування маркерів

У цьому підрозділі ми оцінюємо продуктивність системи трасування маркерів (MTF). Ми починаємо з опису методології експерименту. Потім описуємо результати експерименту.

Методологія. Обираємо пакет тестів DaCapo [13], оскільки DaCapo створює програми реального світу з різноманітною поведінкою. Таблиця 3.4 відображає основні характеристики кожного тесту DaCapo.

Таблиця 3.4

Основні характеристики кожного тесту в пакеті DaCapo

Benchmark	Description	Executed Methods
avrora	Parallel discrete event simulator	1,437
batik	Vector graphics renderer	N/A
eclipse	Eclipse JDT performance tests.	575,590
fop	XSL-FO to PDF convertor.	2,921
h2	Banking application benchmark.	1,501
jython	pybench Python benchmark.	4,610
luindex	lucene-based index generation.	874
lusearch	lucene-based text search.	469
pmd	Java code analyzer.	2,516
sunflow	Rendering system for image synthesis.	583
tomcat	Webpages retrieval and verification against Tomcat.	602
tradebeans	Daytrader benchmark via Jave Beans.	15
tradesoap	Daytrader benchmark via SOAP.	15
xalan	XML to HTML transformation.	2,064

Стовпець «Виконані методи» повідомляє про загальну кількість методів програми (за винятком методів бібліотеки), які фактично виконуються та записуються MTF під час прогону профілювання. В експериментах ми виключаємо тест батика , оскільки для нього потрібен API кодеку JPEG , який вилучено з бази коду OpenJDK. З цієї причини наша JVM не може виконати batik.

Ми оцінюємо накладні витрати часу виконання MTF, запускаючи програми з різними рівнями інструментарію на клієнті «Nop», чиї методи ¬markerenter і markerexit виконують нульові операції. Оскільки накладні витрати MTF пропорційні кількості подій, ми випадково відбираємо 10%, 25%, 50% і 75% усіх виконаних методів, які отримані шляхом профілювання за допомогою MTF. Для кожного розміру вибірки ми виконуємо тести 10 разів і беремо середнє арифметичне як кінцевий результат. Щоб показати найгіршу продуктивність, ми також інструментуємо 100% виконуваних методів. Для порівняння ми запускаємо неінструментовані тести DaCapo на немодифікованому OpenJDK з тією ж версією, що й наша JVM з підтримкою MTF (ми називаємо цю немодифіковану JVM ванільною). Ми встановлюємо опцію -converge пакета DaCapo, який запускає кожен тест кілька разів, щоб зареєстрований час виконання знаходився в межах довірчого інтервалу 3%.

Під час експериментів ми лише інструментуємо методи додатків, тому що ми вважаємо, що ці методи є більш цікавими та актуальними для перевірки кінцевими користувачами, отже, є більш репрезентативними, ніж методи з бібліотеки середовища виконання Java. Результати та їх обговорення. У цьому розділі ми подаємо інформацію про продуктивність тесту DaCapo трасування MTF за допомогою клієнта аналізу nop у таблиці 3,5, 3.6 та рис 3.15.

Таблиця 3.5

Час виконання тестів DaCapo на MTF

Benchmark	Vanilla	10%	25%	50%	75%	100%
avrora	13,087	**8,642**	16,819	24,892	33,454	43,976
eclipse	41,364	47,547	98,392	153,362	212,275	275,831
fop	446	469	845	1,033	1,452	1,768
h2	7,159	7,345	15,130	23,222	31,842	41,861
jython	3,360	3,521	7,250	10,811	14,487	18,513
luindex	1,039	1,235	2,652	4,108	5,596	7,345
lusearch	3,940	4,679	9,964	15,680	22,561	30,193
pmd	3,808	**3,735**	7,819	12,026	16,619	21,575
sunflow	7,558	9,118	19,504	30,936	43,328	55,111
tomcat	5,448	5,473	10,909	16,510	22,926	29,033
tradebeans	7,639	9,102	19,194	31,479	46,423	61,839
tradesoap	17,088	19,709	42,154	67,216	94,924	123,100
xalan	2,727	3,282	7,250	11,686	17,880	25,037

Таблиця 3.6

Виклики маркерів під час виконання тестів DaCapo на MTF

Benchmark	10%	25%	50%	75%	100%
avrora	201	13,420,619	42,872,995	90,676,427	543,934,712
eclipse	338	47,255	104,674	211,352	358,208
fop	1,775	183,711	714,998	1,925,066	10,755,787
h2	68,335,094	155,329,573	251,138,862	350,215,459	1,325,200,083
jython	223	244,681	5,867,601	21,013,345	356,096,651
luindex	159	9,845	22,598	45,680	62,876,224
lusearch	23	20,420,789	81,029,620	284,642,316	886,577,763
pmd	809,858	1,948,717	3,152,651	6,948,307	91,662,920
sunflow	300	1,263	2,885	2,842,068	239,384,047
tomcat	29	136	7,152	803,893	3,795,574
tradebeans	1	4	11	22	39
tradesoap	1	5	11	23	45
xalan	84,693	15,457,628	63,626,175	147,161,490	367,424,093

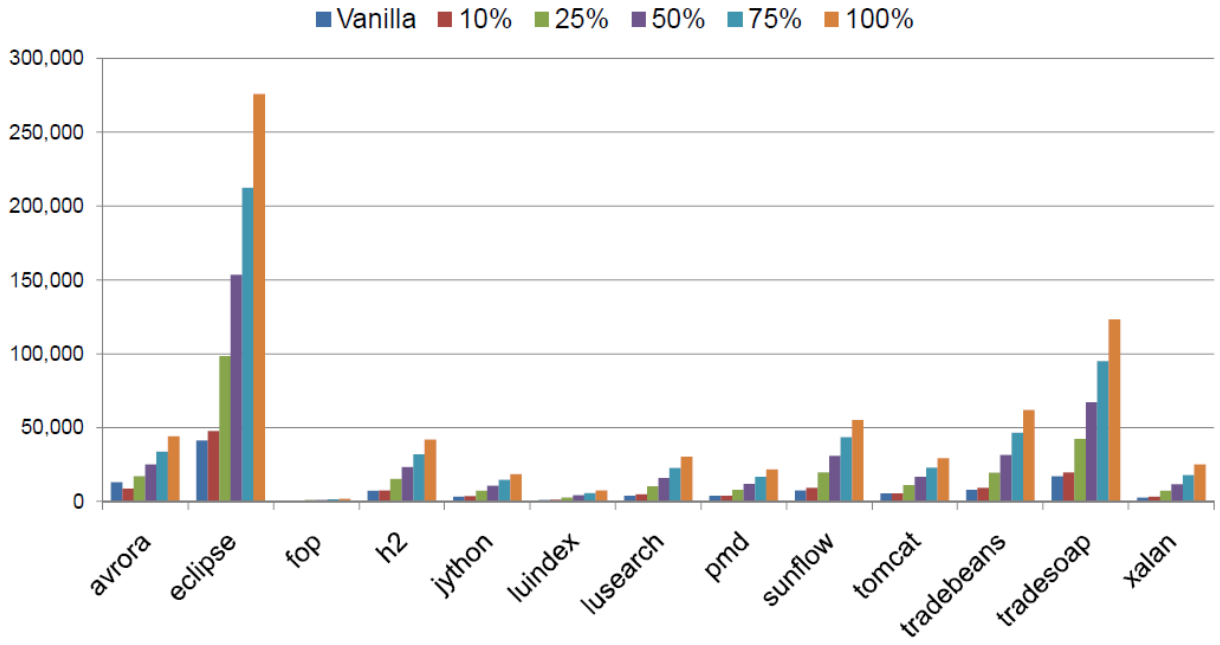


Рисунок 3.15 – Час виконання інструментальних тестів DaCapo на MTF

Таблиця 3.6 показує кількість викликів маркерів, зафіксованих MTF під час виконання інструментальних тестів DaCapo.

Як показано на рисунку 3.15, накладні витрати MTF є прийнятними для малих розмірів вибірки, наприклад 10% і 25%. Це підтверджує інтуїцію про те, що базові накладні витрати є низькими, коли цільова програма має мало

або зовсім не має маркерних інструментів. Однак із зростанням розміру вибірки накладні витрати стають все більшими. Найбільші накладні витрати, тобто уповільнення у 8 разів, виникають під час виконання повністю інструментованого тесту xalan . Великі зразки показують накладні витрати на часті перемикання контексту, викликані надмірними викликами обробників маркерів, як показано в таблиці 3.6 .

Слід зазначити, що цифри, виділені жирним шрифтом 3,5 прогонів 10% інструментів avorora та pmd менші, ніж у ванільних прогонів. Це не означає, що MTF працює швидше, ніж звичайна віртуальна машина, але це пов’язано зі складними реакціями віртуальної машини з присутністю MTF, які можуть дещо змінити поведінку деяких віртуальних машин, наприклад, JIT і кеш.

Висновки до розділу 3

Отже, в цьому розділі запропоновано дизайн і реалізацію гнучкої динамічної системи трасування, тобто MTF, для мов на основі віртуальних машин. MTF надає клієнтам аналізу можливість визначати нові типи подій на основі розташування програм, представлених маркерами. Крім того, показано, що розширену семантику можна прикріпити до кожного класу маркерів за допомогою дескриптора маркера, який можна використовувати для кодування специфічних для аналізу даних, пов’язаних із маркером. Інфраструктура дозволяє гнучко керувати кожним викликом маркера, який також може використовуватися кількома клієнтами. Щоб продемонструвати здійсненність і ефективність інфраструктури, реалізовано MTF для Java у SUN HotSpot JVM, застосувавши лише невеликий набір змін. Ми також оцінюємо продуктивність фреймворку, інструментуючи та аналізуючи тести DaCapo. Показано, що MTF пропонує велику гнучкість для розробки клієнтів програмного аналізу з низькими накладними витратами в більшості програм.

ВИСНОВКИ

В кваліфікаційній роботі виконана оптимізація моделей та методів рішень на основі віртуальних машин для покращення моніторингу та аналізу додатків. Для цього запропонована структура Marker Tracing Framework (MTF), для покращення моніторингу програмних додатків для мов на основі віртуальних машин. MTF забезпечує надійну інфраструктуру для розробки детального аналізу динамічних програм на основі трасування. MTF дозволяє користувачам точно вказувати області коду за допомогою спеціальних маркерів, які можуть викликати події під час виконання, які отримуються та передаються кожному клієнту аналізу. Семантика подій незалежно визначається кожним аналізом, щоб кілька клієнтів аналізу могли обробляти події одночасно з різними логіками. Також розроблено розширювану утиліту для додавання інструментів маркерів у програми. Крім того, новий клієнт інструментів можна легко додати до утиліти для підтримки нових аналізів.

На основі MTF досліджено два клієнти аналізу, тобто аналіз станів типів і вибірковий імовірнісний аналіз контексту виклику (SPCC). Аналіз стану типу використовує автомат із кінцевим станом для представлення властивості стану типу та використовує MTF для відстеження переходів станів під час виконання. Виконуючи аналіз усередині віртуальної машини, він може розширити заголовок об’єкта, а також співпрацювати зі збирачем сміття для ефективної і точної перевірка стану типу. SPCC узагальнює існуючий аналіз PCC із можливістю вибіркового обчислення значень PCC для набору методів, які також можуть адаптивно оновлюватися клієнтом SPCC на основі результатів аналізу. Таким чином, ми стверджуємо, що MTF може запропонувати велику гнучкість і адаптивність, які разом підвищують точність і ефективність таких існуючих програмних аналізів.

Було застосовано фреймворк, так як клієнт аналізу у галузі JVM HotSpot на платформі x86. Результати експерименту показують, що JVM з

підтримкою MTF пропонує достатню підтримку часу виконання для обох клієнтів аналізу з прийнятними накладними витратами. У майбутній роботі ми плануємо ще більше зменшити накладні витрати, використовуючи існуючі методи, наприклад, статичний аналіз і вибірку. Крім того, оцінюється кілька вдосконалень, щоб підвищити корисність інфраструктури та клієнтів.

Також реалізовано клієнт динамічного аналізу станів типів на основі автомата кінцевого стану (FSA), який використовує потужність і корисність MTF. Крім того, застосовано оптимізацію Adaptive Online Program Analysis (AOPA) до клієнта, щоб продемонструвати переваги розробки програмного аналізу всередині JVM.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Ole Agesen and Alex Garthwaite. Efficient object sampling via weak references. In ISMM ’00: Proceedings of the 2nd international symposium on Memory management, pages 121–126, New York, NY, USA, 2000.

2. Chris Allan, Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Ondˇrej Lhota´k, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. Adding trace matching with free variables to aspectj. SIGPLAN Not., 40(10):345–364, 2005.

3. Bowen Alpern, C. R. Attanasio, Anthony Cocchi, Derek Lieber, Stephen Smith,Ton Ngo, John J. Barton, Susan Flynn Hummel, Janice C. Sheperd, and Mark Mergen. Implementing jalapeno in java. SIGPLAN Not., 34(10):314–324, 1999.

4. Danilo Ansaloni, Walter Binder, Alex Villaz´on, and Philippe Moret. Parallel dynamic analysis on multicores with aspect-oriented programming. In AOSD’10: Proceedings of the 9th International Conference on Aspect-Oriented Software Development, pages 1–12, New York, NY, USA, 2010.

5. Danilo Ansaloni, Walter Binder, Alex Villaz´on, and Philippe Moret. Rapid development of extensible profilers for the java virtual machine with aspectoriented programming. In WOSP/SIPEW ’10: Proceedings of the first joint WOSP/SIPEW international conference on Performance engineering, pages 57–62, New York, NY, USA, 2010.

6. Matthew Arnold and Barbara G. Ryder. A framework for reducing the cost of instrumented code. SIGPLAN Not., 36(5):168–179, 2001.

7. Kevin Bierhoff and Jonathan Aldrich. Modular typestate checking of aliased objects. In OOPSLA ’07: Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications, pages 301–320, New York, NY, USA, 2007.

8. Walter Binder, Jarle Hulaas, and Philippe Moret. Advanced java bytecode instrumentation. In PPPJ ’07: Proceedings of the 5th international symposium on Principles and practice of programming in Java, pages 135–144, New York, NY, USA, 2007.

9. Stephen M. Blackburn, Matthew Hertz, Kathryn S. Mckinley, J. Eliot B. Moss, and Ting Yang. Profile-based pretenuring. ACM Trans. Program. Lang. Syst., 29(1):2, 2007.

10. Eric Bodden. Efficient and Expressive Runtime Verification for Java. In Grand Finals of the ACM Student Research Competition 2005, 2005.

11. Eric Bodden. Efficient Hybrid Typestate Analysis by Determining ContinuationEquivalent States. In International Conference of Software Engineering (ICSE). ACM Press, 2010.

12. Eric Bodden, Laurie Hendren, Patrick Lam, Ondrej Lhotak, and Nomair A. Naeem. Collaborative Runtime Verification with Tracematches. Oxford Journal of Logics and Computation, 2008.

13. Eric Bodden, Patrick Lam, and Laurie Hendren. Finding programming errors earlier by evaluating runtime monitors ahead-of-time. In SIGSOFT ’08/FSE-16: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering, pages 36–47, New York, NY, USA, 2008.

14. Michael D. Bond and Kathryn S. McKinley. Bell: bit-encoding online memory leak detection. SIGOPS Oper. Syst. Rev., 40(5):61–72, 2006.

15. Michael D. Bond and Kathryn S. McKinley. Probabilistic calling context. In OOPSLA ’07: Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications, pages 97–112, New York, NY, USA, 2007.

16. S. Boroday, A. Petrenko, J. Singh, and H. Hallal. Dynamic analysis of java applications for multithreaded antipatterns. In WODA ’05: Proceedings of the third international workshop on Dynamic analysis, pages 1–7, New York, NY, USA, 2005.

17. Feng Chen and Grigore Ro¸su. Mop: an efficient and generic runtime verification framework. In OOPSLA ’07: Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications, pages 569– 588, New York, NY, USA, 2007.

18. Shigeru Chiba and Muga Nishizawa. An easy-to-use toolkit for efficient java bytecode translators. In GPCE ’03: Proceedings of the 2nd international conference on Generative programming and component engineering, pages 364–376, New York, NY, USA, 2003.

19. Markus Dahm. Byte Code Engineering with the BCEL API. Technical report, Freie University Berlin, April 2001.

20. Brian Demsky and Martin Rinard. Automatic detection and repair of errors in data structures. In OOPSLA ’03: Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications, pages 78–95, New York, NY, USA, 2003.

21. Matthew B. Dwyer, Alex Kinneer, and Sebastian Elbaum. Adaptive Online Program Analysis. In ICSE ’07: Proceedings of the 29th international conference on Software Engineering, pages 220–229, Washington, DC, USA, 2007.

22. Matthew B. Dwyer and Rahul Purandare. Residual dynamic typestate analysis exploiting static analysis: results to reformulate and reduce the cost of dynamic analysis. In ASE ’07: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering, pages 124–133, New York, NY, USA, 2007.

23. Matthew B. Dwyer and Rahul Purandare. Residual dynamic typestate analysis exploiting static analysis: results to reformulate and reduce the cost of dynamic analysis. In ASE ’07: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering, pages 124–133, New York, NY, USA, 2007.

24. Romain Lenglet Eric Bruneton and Thierry Coupaye. ASM: A code manipulation tool to implement adaptable systems. In In Proceedings of the ASF

(ACM SIGOPS France) Journees Composants 2002: Adaptable and extensible component systems, November 2002.

25. Stephen Fink, Eran Yahav, Nurit Dor, G. Ramalingam, and Emmanuel Geay. Effective typestate verification in the presence of aliasing. In ISSTA ’06: Proceedings of the 2006 international symposium on Software testing and analysis, pages 133–144, New York, NY, USA, 2006.

26. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design patterns: elements of reusable object-oriented software. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

27. James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. Java(TM) Language Specification, The (3rd Edition) (Java (Addison-Wesley)). Addison-Wesley Professional, 2005.

28. Robert Griesemer. Generation of virtual machine code at startup. In OOPSLA ’99 Workshop on Simplicity, Performance, and Portability in Virtual Machine Design, 1999.

29. Clemens Hammacher, Kevin Streit, Sebastian Hack, and Andreas Zeller. Profiling java programs for parallelism. In IWMSE ’09: Proceedings of the 2009 ICSE Workshop on Multicore Software Engineering, pages 49–55, Washington, DC, USA, 2009.

30. Matthias Hauswirth, Amer Diwan, Peter F. Sweeney, and Michael C. Mozer.

31. Automating vertical profiling. In OOPSLA ’05: Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, pages 281–296, New York, NY, USA, 2005.

32. Matthew Hertz, Yi Feng, and Emery D. Berger. Garbage collection without paging. SIGPLAN Not., 40(6):143–153, 2005.

33. Network Applications. In NCA ’04: Proceedings of the Network Computing and Applications, Third IEEE International Symposium, pages 303–308, Washington, DC, USA, 2004.

34. Richard E. Jones and Chris Ryder. A study of java object demographics. In ISMM ’08: Proceedings of the 7th international symposium on Memory management, pages 121–130, New York, NY, USA, 2008.

35. Maria Jump, Stephen M. Blackburn, and Kathryn S. McKinley. Dynamic object sampling for pretenuring. In ISMM ’04: Proceedings of the 4th international symposium on Memory management, pages 152–162, New York, NY, USA, 2004.

36. Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An Overview of AspectJ. In ECOOP ’01: Proceedings of the 15th European Conference on Object-Oriented Programming, pages 327–353, London, UK, 2001.

37. Goh Kondoh and Tamiya Onodera. Finding bugs in java native interface programs. In ISSTA ’08: Proceedings of the 2008 international symposium on Software testing and analysis, pages 109–118, New York, NY, USA, 2008.

38. Thomas Kotzmann, Christian Wimmer, Hanspeter M¨ossenb¨ock, Thomas Rodriguez, Kenneth Russell, and David Cox. Design of the java hotspotTMclient compiler for java 6. ACM Trans. Archit. Code Optim., 5(1):1–32, 2008.

39. Patrick Lam, Viktor Kuncak, and Martin Rinard. Generalized typestate checking for data structure consistency. In In 6th International Conference on Verification, Model Checking and Abstract Interpretation, 2005.

40. Byeongcheol Lee, Martin Hirzel, Robert Grimm, and Kathryn S. McKinley. Debug all your code: portable mixed-environment debugging. In OOPSLA ’09: Proceeding of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications, pages 207–226, New York, NY, USA, 2009.

41. Tim Lindholm and Frank Yellin. Java Virtual Machine Specification. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

42. David Lo and Shahar Maoz. Hierarchical inter-object traces for specification mining. In OOPSLA Companion ’08: Companion to the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications, pages 761–762, New York, NY, USA, 2008.

43. Allen Davis Malony. Performance observability. PhD thesis, Champaign, IL, USA, 1990. Adviser-Reed, D. Marcus Meyerh¨ofer. TestEJB: response time measurement and call dependency tracing for ejbs. In MAI ’07: Proceedings of the 1st workshop on Middlewareapplication interaction, pages 55–60, New York, NY, USA, 2007.

44. Nomair A. Naeem and Ondrej Lhotak. Typestate-like analysis of multiple interacting objects. In OOPSLA ’08: Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications, pages 347–366, New York, NY, USA, 2008.

45. Michael Paleczny, Christopher Vick, and Cliff Click. The java hotspottm server compiler. In JVM’01: Proceedings of the 2001 Symposium on JavaTM Virtual Machine Research and Technology Symposium, pages 1–1, Berkeley, CA, USA, 2001.

46. Michael Paleczny, Christopher Vick, and Cliff Click. The java hotspottm server compiler. In JVM’01: Proceedings of the 2001 Symposium on JavaTM Virtual Machine Research and Technology Symposium, pages 1–1, Berkeley, CA, USA, 2001.

47. Alexandar Pantaleev and Atanas Rountev. Identifying Data Transfer Objects in EJB Applications. In WODA ’07: Proceedings of the 5th International Workshop on Dynamic Analysis, page 5, Washington, DC, USA, 2007.

48. Society. David J. Pearce, Matthew Webster, Robert Berry, and Paul H. J. Kelly. Profiling with aspectj. Softw. Pract. Exper., 37(7):747–777, 2007.

49. Matt Pietrek. The .NET Profiling API and the DNProfiler Tool. MSDN Magazine, December 2001.

50. Guillaume Pothier, Eric Tanter, and Jos´e Piquer. Scalable omniscient debugging.´ In OOPSLA ’07: Proceedings of the 22nd annual ACM SIGPLAN

conference on Object-oriented programming systems and applications, pages 535–552, New York, NY, USA, 2007.

51. Steven P. Reiss. Controlled dynamic performance analysis. In WOSP ’08: Proceedings of the 7th international workshop on Software and performance, pages 43–54, New York, NY, USA, 2008.

52. Ohad Shacham, Martin Vechev, and Eran Yahav. Chameleon: adaptive selection of collections. In PLDI ’09: Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation, pages 408–418, New York, NY, USA, 2009.

53. Antonio Da Silva, Alberto Gonzalez-Calero, Jos´e Ferna´n Martinez, Lourdes Lopez, Ana Belen Garcia, and Vicente Hernandez. Design and Implementation of a Java Fault Injector for Exhaustif R SWIFI Tool. In DEPCOS-RELCOMEX ’09: Proceedings of the 2009 Fourth International Conference on Dependability of Computer Systems, pages 77–83, Washington, DC, USA, 2009.

54. Paramvir Singh and Hardeep Singh. Dynametrics: a runtime metric-based analysis tool for object-oriented software systems. SIGSOFT Softw. Eng. Notes, 33(6):1–6, 2008.

метадані

Заголовок

Оптимізація моделей та методів рішень на основі віртуальних машин для покращення моніторингу та аналізу додатків

Автор

Гончарик Н.І. Науковий керівник / Експерт

підрозділ

King Danylo University

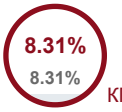
Тривога

У цьому розділі ви знайдете інформацію щодо текстових спотворень. Ці спотворення в тексті можуть говорити про МОЖЛИВІ маніпуляції в тексті. Спотворення в тексті можуть мати навмисний характер, але частіше характер технічних помилок при конвертації документа та його збереженні, тому ми рекомендуємо вам підходити до аналізу цього модуля відповідально. У разі виникнення запитань, просимо звертатися до нашої служби підтримки.

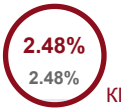
Заміна букв	ß	1
Інтервали	A→	0
Мікропробіли	·	0
Білі знаки	ß	0
Парафрази (SmartMarks)	a	83

Обсяг знайдених подібностей

Коефіцієнт подібності визначає, який відсоток тексту по відношенню до загального обсягу тексту було знайдено в різних джерелах. Зверніть увагу, що високі значення коефіцієнта не автоматично означають плагіат. Звіт має аналізувати компетентна / уповноважена особа.



КП 1



КП 2

25
Довжина фрази для коефіцієнта подібності 2

15628
Кількість слів

116196
Кількість символів

Подібності за списком джерел

Нижче наведений список джерел. В цьому списку є джерела із різних баз даних. Колір тексту означає в якому джерелі він був знайдений. Ці джерела і значення Коефіцієнту Подібності не відображають прямого плагіату. Необхідно відкрити кожне джерело і проаналізувати зміст і правильність оформлення джерела.

10 найдовших фраз Колір тексту

ПОРЯДКОВИЙ НОМЕР	НАЗВА ТА АДРЕСА ДЖЕРЕЛА URL (НАЗВА БАЗИ)	КІЛЬКІСТЬ ІДЕНТИЧНИХ СЛІВ (ФРАГМЕНТІВ)	
1	http://repository.ukd.edu.ua/bitstream/handle/123456789/391/%D0%9F%D0%B0%D1%85%D0%BE%D0%BB%D1%8C%D1%87%D1%83%D0%BA%20%D0%9E.%D0%A0.%20%D0%B4%D0%B8%D0%BF%D0%BB%D0%BE%D0%BC%D0%BD%D0%B0.pdf?sequence=1	44	0.28 %
2	https://api.crossref.org/works/10.1145%2F1735997.1736016	39	0.25 %
3	http://repository.ukd.edu.ua/bitstream/handle/123456789/391/%D0%9F%D0%B0%D1%85%D0%BE%D0%BB%D1%8C%D1%87%D1%83%D0%BA%20%D0%9E.%D0%A0.%20%D0%B4%D0%B8%D0%BF%D0%BB%D0%BE%D0%BC%D0%BD%D0%B0.pdf?sequence=1	38	0.24 %
4	https://www.cs.cmu.edu/~ckaestne/pdf/thesispusch.pdf	36	0.23 %