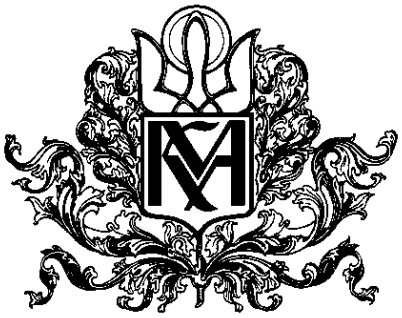
Міністерство освіти і науки України



НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
«КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»
Кафедра інформатики

Особливості використання GraalVM в застосунках JVM екосистеми

Текстова частина до курсової роботи
за спеціальністю 122 «Комп’ютерні науки»

Керівник курсової роботи
Андрощук М.В.
_____
(підпис)
“____” _____ 2024 р.

Виконав студент
Анісімов Є.О.
“____” _____ 2024 р.

Київ 2024

Міністерство освіти і науки України

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»

Кафедра інформатики Факультету інформатики

ЗАТВЕРДЖУЮ

Зав.кафедри інформатики,

Доцент., к. ф.-м. н. С.С.Гороховський

(підпис)

“____”_____ 2023 р.

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ

на курсову роботу

студенту Анісімову Євгену Олександровичу факультету інформатики 3-го курсу

ТЕМА Особливості використання GraalVM в застосунках JVM екосистеми

Зміст ТЧ до курсової роботи:

1. Індивідуальне завдання
2. Календарний план
3. Анотація
4. Теоретичні аспекти GraalVM
5. Покращення продуктивності з GraalVM
6. Практичне порівняння продуктивності GraalVM
7. Висновки
8. Список використаної джерел

Дата видачі “___” _____ 2023 р. Керівник _____

Завдання отримав _____

Тема: Особливості використання GraalVM в застосунках JVM екосистеми

Календарний план виконання роботи:

№	Назва етапу	Термін виконання	Примітка
1.	Отримання теми курсової роботи	27.09.2023	
2.	Пошук тематичної літератури	09.11.2023	
3.	Ознайомлення з літературою	31.12.2023	
4.	Написання текстової частини курсової роботи	24.02.2024	
5.	Реалізація практичного порівняння	15.03.2024	
6.	Внесення змін до курсової роботи відповідно до зауважень наукового керівника	27.04.2024	
7.	Створення презентації	12.05.2024	
8.	Захист роботи	20.05.2024	

Студент Анісімов Є.О.

Керівник Андрощук М.В.

“____” _____

Зміст

Анотація

Ця курсова робота надає детальний аналіз особливостей застосування GraalVM у додатках, розроблених для JVM екосистеми. У роботі розглянуто ключові аспекти GraalVM, такі як JIT-компіляція, AOT-компіляція та підтримка поліглотності, щоб зрозуміти, чому вона є надійним вибором для розробки високопродуктивних додатків у сучасному програмному ландшафті. Робота зосереджується на технічних аспектах, структурі GraalVM, та проводить порівняння його продуктивності з традиційною JVM на основі серії практичних завдань різного масштабу. Особлива увага приділяється аналізу поліпшень у виконанні коду, оптимізації застосунків, а також можливостям GraalVM щодо підтримки мов програмування, що не є частиною стандартної JVM екосистеми.

Вступ

У сучасному світі розробки програмного забезпечення, проєкти неперервно зростають у масштабі та стають більш вимогливими до технологічної інфраструктури. Водночас компанії прагнуть якомога менше фінансів витрачати на її функціонування. В такий складний для людства час питання оптимізації ефективності технічних рішень є як ніколи актуальним.

Не дивлячись на стрімкий розвиток мов програмування за останні роки, Java досі займає передові позиції в сфері backend розробки, залишаючись однією з найпопулярніших мов програмування у світі. [1]

Вибір віртуальної машини для виконання Java додатків є критично важливим для забезпечення оптимальної продуктивності та ефективності розробки. Традиційно, Java виконується на JVM, проте метою курсової роботи є дослідження відносно нової віртуальної машини, а саме GraalVM, яка призначена для прискорення виконання програм, написаних на Java та інших мовах JVM, за допомогою власного компілятора Graal. [2]

У теоретичній частині курсової роботи ми ознайомимося з архітектурою GraalVM і дослідимо її додаткові можливості. Зокрема, розглянемо можливість створення нативних файлів застосунків та підтримку виконання багатьох мов програмування в рамках одного застосунку. Також розглянемо додаткові інструменти, які GraalVM підтримує нативно для спрощення процесу розробки та налагодження програмного коду.

В практичній частині курсової роботи ми порівняємо продуктивність GraalVM з аналогом у вигляді JVM за допомогою тестування. Практично продемонструємо додаткові можливості GraalVM та дослідимо їх вплив на

продуктивність програмних застосунків, вимірявши час запуску програми та кількість пам’яті, яку вони використовують при роботі на різних віртуальних машинах.

Розділ 1. Теоретичні аспекти GraalVM

1.1 Вступ до GraalVM

Історія GraalVM починається з проєкту Maxine VM, амбітною open source віртуальною машиною, розробленою дослідниками з лабораторії Sun Microsystems. Maxine VM слугувала експериментальною платформою для дослідження нових підходів до структури та внутрішньої реалізації віртуальних машин. Особливістю Maxine було те, що вона написана повністю на Java, що дозволяло внести ясність та спростити розуміння архітектури віртуальної машини.

Цей підхід дав можливість не тільки легше розуміти архітектури віртуальної машини, але й значно спростити процес дебагінгу та розробки нових функцій. Maxine VM продемонструвала, що компілятор написаний мовою Java може ефективно виконувати оптимізацію та генерацію машинного коду без шкоди для швидкості компіляції. Такий підхід згодом надихнув на створення проєкту GraalVM. [3]

З 2012 року почалася розробка компілятора Graal, який мав стати інноваційним JIT компілятором, написаним повністю мовою Java. Graal був спроектований для того, щоб замінити існуючі компілятори C1 та C2 в HotSpot VM і був включений до OpenJDK 9 як AOT компілятор, а в версії OpenJDK 10 як експериментальний JIT компілятор. [4]

Завдяки підтримці Oracle, GraalVM швидко переріс в самостійний проєкт, здатний ефективно виконувати Java код та інтегруватися з іншими мовами програмування через Truffle Framework. Офіційний реліз GraalVM відбувся в квітні 2019 року, отримавши широке визнання в спільноті розробників, і

продовжує вносити значний внесок у розвиток віртуальних машин.

1.2 Приклади використання GraalVM

Серед компаній, які використовують GraalVM, можна виділити таких технічних гігантів як:

1) **Facebook:** Одна з найбільш відвідуваних соціальних платформ, активно використовує Java для роботи з великими даними та бекенд-сервісами. Перехід на GraalVM дозволив компанії без змін до коду значно прискорити обробку даних у Spark на 10%-42%, а також зменшити витрати пам'яті та процесорного часу.

2) **X** (колишній Twitter): Використовує GraalVM для оптимізації своїх численних JVM у різних дата-центрах. Завдяки GraalVM компанія знизила використання CPU на 8-11% та скоротила кількість необхідних серверів на 18%, що суттєво знизило вартість утримання інфраструктури.

3) **Alibaba:** Застосовує технологію Native Image від GraalVM для статичної компіляції мікросервісних додатків у виконувальні файли ELF, що забезпечує швидкий старт Java-додатків і поліпшує ефективність розгортання служб.

4) **Nvidia**: Використовує можливості GraalVM для інтеграції GPU-прискорених бібліотек у свої програмні рішення. Зокрема, з допомогою grCUDA, розробники ефективно запускають ядра GPU і обмінюються даними між GPU та мовами, що підтримуються GraalVM, такими як Python, R, Ruby та JavaScript. [25]

GraalVM знаходить широке застосування в різних галузях, від фінансових послуг до здоров'я та роздрібної торгівлі. Цей рушій використовується для оптимізації хмарних додатків, створення ефективних мікросервісних архітектур і навіть в розробці безсерверних додатків. GraalVM покращує час запуску додатків, знижує використання ресурсів і забезпечує підтримку поліглотого програмування, що дозволяє компаніям створювати більш гнучкі та адаптивні системи. Впровадження GraalVM сприяє швидкій інтеграції новітніх технологій, зниженню витрат і забезпеченню більшої надійності програмних рішень.

1.3 Архітектура GraalVM

GraalVM складається з 5 основних компонентів: Java HotSpot VM, JVM Compiler Interface, Graal Compiler, Truffle Framework, Sulong (LLVM), поговоримо про них детальніше.

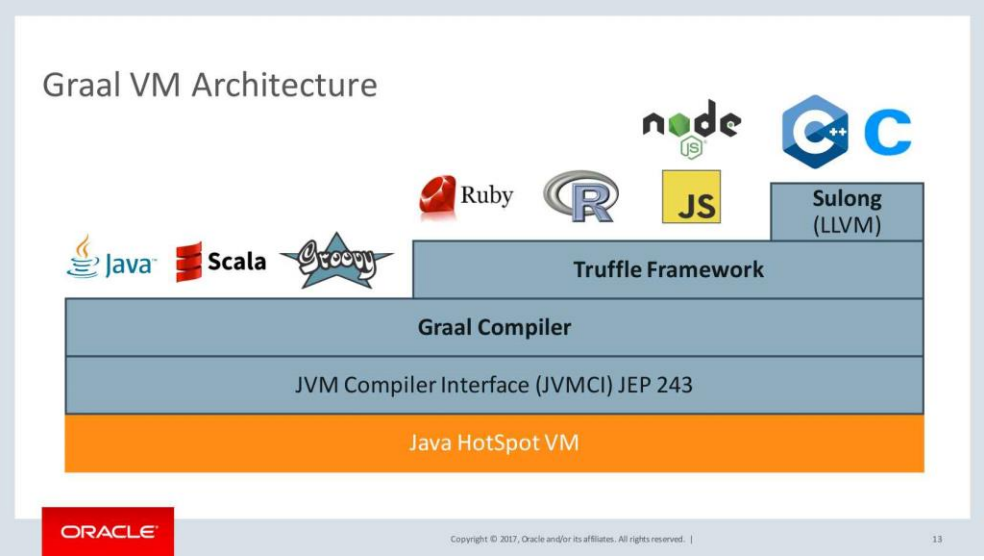


Рисунок 1.1 Архітектура GraalVM [2]

1.3.1 Java HotSpot Virtual Machine

Java HotSpot Virtual Machine є імплементацією JVM яка почала використовуватись за замовчуванням з версії Java 1.3. Вона є високопродуктивною віртуальною машиною, проте здатна працювати тільки з тими мовами, які підтримують виконання на JVM, а саме: Scala, Kotlin, Groovy, Clojure.

Однією з ключових особливостей Java HotSpot Virtual Machine є виявлення та оптимізація “гарячих точок” - частин коду, які виконуються найчастіше. HotSpot використовує адаптивну оптимізацію та JIT-компіляцію для перетворення цих ділянок із байт-коду (проміжного представленням Java коду, який компілюється з вихідного коду компілятором javac.) в машинний код (набір інструкцій які виконуються безпосередньо центральним процесором). Цей процес не тільки значно збільшує продуктивність, але й мінімізує час, необхідний для інтерпретації коду.

Адаптивна оптимізація HotSpot використовує факт, що більшість програм витрачають переважну більшість часу, виконуючи невелику частину свого коду. VM аналізує виконання програми в реальному часі, швидко ідентифікує ці критичні ділянки та оптимізує їх. Замість компіляції всього коду, HotSpot фокусується на “гарячих точках”, що дозволяє ефективніше використовувати ресурси компіляції та зменшувати загальний обсяг використаної пам'яті.

Цей процес включає в себе не тільки компіляцію в машинний код, але й збір даних про використання коду, що дозволяє здійснювати більш точну оптимізацію на основі реального використання додатка. Окрім збору даних про “гарячі точки”, збираються й інші типи інформації, такі як взаємозв'язки

між викликами методів, що допомагає оптимізувати віртуальні виклики.

Такий підхід не тільки підвищує, але й адаптує продуктивність програми до змінних умов виконання, реагуючи на нові “гарячі точки”, які можуть з'явитися внаслідок зміни поведінки користувача або даних. Така динамічність забезпечує оптимальне використання ресурсів та гарантує, що програма завжди виконується з максимальною ефективністю. [5]

HotSpot VM містить два JIT компілятора: клієнтський C1 та серверний C2. Різниця між ними полягає в тому, що клієнтський компілятор фокусується на високій швидкості компіляції, тоді як серверний - на максимальній продуктивності. Клієнтський компілятор виконує лише обмежений набір оптимізацій і найкраще підходить для короткочасних клієнтських додатків. Серверний компілятор потребує більше часу для компіляції, але створює більш оптимізований машинний код, тому скомпільовані методи Java будуть виконуватися швидше. Тому він найкраще підходить для довготривалих серверних додатків. Наразі існують спроби дозволити багаторівневу компіляцію. Це означає, що методи спочатку компілюються за допомогою клієнтського компілятора і лише дуже важливі методи Java-програми пізніше перекомпілюються за допомогою серверного компілятора.

Всередині клієнтський компілятор використовує представлення Java коду на основі потоку управління для виконання оптимізацій. Інструкції групуються в блоки, де всі інструкції виконуються послідовно, якщо не виникає винятків. Серверний компілятор, навпаки, використовує граф залежностей програми. Це дозволяє проводити більш складні оптимізації, що охоплюють більші області методу, але структура графу також стає

складнішою. [6]

1.3.2 JVM Compiler Interface

JVM Compiler Interface (JVMCI) — це частина сучасних JVM, яка дозволяє інтеграцію власних JIT компіляторів, написаних на Java. Вперше введений у Java 9, JVMCI відкрив нові можливості для розробників з оптимізації виконання Java-програм.

Перед введенням JVMCI, JIT компілятори для JVM традиційно створювались на мовах нижчого рівня, зокрема C++. Це створювало бар'єри для розробників Java, оскільки внесення змін та розширення функціональності компіляторів вимагало глибоких знань інших мов та складних систем. JVMCI спрямований на розв'язання цих проблем, дозволяючи реалізовувати JIT компілятори безпосередньо на Java, що спрощує їх розробку та підтримку.

Щоб використовувати JVMCI, розробник повинен реалізувати інтерфейс JVMCICompiler, основний метод якого — compileMethod. Цей метод відповідає за перетворення байт-коду Java в машинний код.

```
interface JVMCICompiler {
    byte[] compileMethod(byte[] bytecode);
}
```

Рисунок 1.2 Інтерфейс для імплементації власного компілятора

Перевизначивши цей метод [7], GraalVM за замовчуванням підключає свій інноваційний компілятор Graal як JIT компілятор вищого рівня, про який поговоримо далі.

1.3.3 Graal, the JIT compiler

Найважливішою частиною GraalVM є однойменний компілятор Graal, високопродуктивний компілятор, який зазвичай використовується для JIT та AOT компіляції. Завдяки модульній архітектурі GraalVM її компілятор є відокремленим від VM і замінює компілятори C1 та C2 які використовуються в HotSpot.

Ця архітектура забезпечує можливість повторного використання всіх компонентів VM які не заважають процесу компіляції, такі як інтерпретатор, garbage collector, завантажувач класів та інші.

Оскільки компіляція є складним процесом, у GraalVM вона розділена на дві частини (рис. 1.3), перша з яких є специфічною для вихідного коду, де виконується більшість високорівневих оптимізацій. Ця частина включає в себе фазу оптимізації, яка не залежать від платформи і називається Graal Intermediate Representation (Graal IR). Друга, специфічна для цільового коду частина, представлена як Back End, відповідає за перетворення високорівневого Graal IR в низькорівневий IR (LIR).

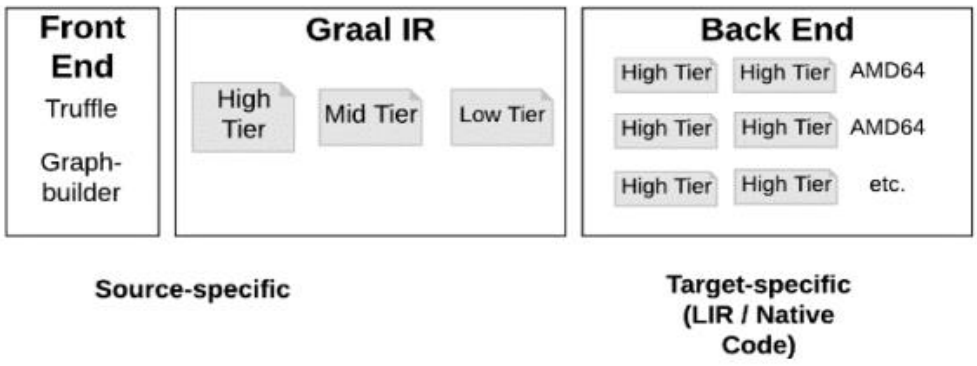


Рисунок 1.3 Процес компіляції Graal [8]

Front end відповідає за перетворення вихідного представлення у вигляді байт-коду в Graal IR, використовуючи фреймворк Truffle або Graphbuilder. Graphbuilder розбирає масив байт як байт-код JVM у граф Graal і комбінує зворотній зв'язок від інтерпретатора з профілюванням. З іншого боку, якщо ми використовуємо фреймворк Truffle, Graal розпізнає це і агресивно оптимізує свій інтерпретатор, дозволяючи розробляти середовище виконання для різних мов програмування.

Після того, як байт-код інтерпретовано у високорівневе проміжне представлення Graal IR, можна починати процес оптимізації. Це загальна частина компіляції, де виконуються всі незалежні від платформи оптимізації. Graal IR - це складна гібридна структура, яка контролює залежності потоку і даних, яка складається з двох спрямованих ациклічних графів і розділяє процес оптимізації на три рівні: High Tier, Mid Tier та Low Tier. Рівні деконструюються на незалежні фази, і в кінці кожного рівня підготовка, яка називається фазою зниження, деконструює операції, готуючи їх до наступного рівня.

High Tier фокусується на високорівневій оптимізації, де використовуються загальні оптимізації компілятора, але акцент робиться на видатних методах Graal, таких як інлайнінг методів та Partial escape analysis. Partial escape перевіряє, чи можна використовувати виділений об'єкт за межами потоку або методу, що його виділив. Об'єкт виділяється у стеку поточного потоку або безпосередньо виштовхується у регістри, щоб уникнути виділення. Mid Tier в основному займається оптимізацією пам'яті, а Low Tier виконує остаточне очищення, невеликі оптимізації та підготовку графів для перетворення в низькорівневе представлення.

Нарешті, Back End, який займається перекладом Graal IR в LIR, виконує розподіл регістрів і пересилає байт-кодові інструкції процесору. [8]

1.3.4 Truffle Language Implementation Framework

При масштабній розробці програмного забезпечення програмісти часто пишуть програми кількома мовами, а не однією. Поєднання декількох мов дозволяє їм використовувати найбільш придатну мову для вирішення певної проблеми, поступово мігрувати існуючі проєкти з однієї мови на іншу або повторно використовувати існуючий вихідний код. Для вирішення цих задач і існує Truffle.

Truffle Language Implementation Framework – це фреймворк з відкритим вихідним кодом який дозволяє написати простий інтерпретатор для власної мови програмування мовою Java.

Truffle використовує концепцію абстрактного синтаксичного дерева (представлення структури програми у вигляді дерева, де кожен вузол є виразом) для імплементації інтерпретатора, модифікуючи його під час інтерпретації за допомогою компілятора Graal, використовуючи інформацію гостьової мови програмування. В кінцевому результаті ми отримаємо продуктивність на тому ж рівні, як при написанні власного компілятора, проте з меншими витратами часу на розробку. [8]

```
// Клас Node наданий Truffle
public abstract class MyNode extends Node {
    public abstract int executeInt();
}

public class IntLiteralNode extends MyNode {
    private final int value;

    public IntLiteralNode(int value) {
        this.value = value;
    }

    @Override
    public int executeInt() {
        return this.value;
    }
}

public class IntAddNode extends MyNode {
    private final MyNode left, right;

    public IntAddNode(MyNode left, MyNode right) {
        this.left = left;
        this.right = right;
    }

    @Override
    public int executeInt() {
        int leftResult = this.left.executeInt();
        int rightResult = this.right.executeInt();
        return leftResult + rightResult;
    }
}
```

Рисунок 1.4 Приклад створення інтерпретатора для мови яка підтримує додавання цілих чисел [9]

Компілятор Graal для оптимізації інтерпретатора використовує техніку часткового оцінювання (Partial evaluation), суть якої полягає в зменшенні кількості обчислень на етапі компіляції програми, для швидшого виконання процесором. Для цих цілей використовуються такі методи оптимізації як:

1) **Інлайнінг (Inlining)**: Замість того, щоб викликати функцію, компілятор або інша оптимізаційна система вставляє тіло функції безпосередньо у місце

її виклику. Це дозволяє уникнути накладних витрат на виклик функції і може призвести до покращення продуктивності програми.

```
# Функція, яку ми хочемо оптимізувати
def add(a, b):
    return a + b

# Виклик функції
result = add(3, 4)

# Після інлайнінгу
result = 3 + 4
```

Рисунок 1.5 Приклад інлайнінгу функції

2) **Складання констант (Constant folding)**: При частковому оцінюванні, коли відомі деякі константи або вирази з відомими значеннями, ці вирази можуть бути частково обчислені на етапі компіляції. Це може включати обчислення математичних виразів або логічних операцій, що залежать від сталих значень, що дозволяє зменшити обсяг обчислень під час виконання програми. З попереднього прикладу код буде виглядати ось так:

```
# До constant folding
result = 3 + 4

# Після constant folding
result = 7
```

Рисунок 1.6 Приклад складання констант

3) **Видалення мертвого коду (Dead code elimination):** Під час часткового оцінювання, коли аналізується структура програми, може виявитися, що деякі частини коду ніколи не використовуються або не мають впливу на результат виконання програми. Ці непотрібні частини коду можуть бути видалені, що сприяє зменшенню обсягу програми та покращенню її продуктивності.

Не менш важливою функцією Truffle є підтримка поліглотного

програмування за допомогою Polyglot API. Це дозволяє GraalVM підтримувати сумісність між будь-якими мовами без потреби заглиблення в реалізації кожної мови. Наразі Polyglot API підтримує такі відомі мови програмування як: JavaScript, R, Ruby, та Python

1.3.5 Sulong (LLVM)

LLVM — це модульний фреймворк, призначений переважно для статично типізованих мов програмування, що дозволяє виконувати код на основі LLVM. Фреймворк працює з мовами, такими як C, C++ та Fortran, спершу транслюючи код у LLVM IR (Intermediate Representation). LLVM IR є проміжним представленням коду, яке служить для глибокого аналізу та оптимізації перед фінальною компіляцією у машинний код. Ця низькорівнева, мовонезалежна форма коду є доступною для зрозуміння, оскільки вона зберігається у формі, що зрозуміла для людини. [10]

У рамках GraalVM, Sulong додає можливість виконання LLVM IR у середовищі Java Virtual Machine (JVM). Розроблений на базі фреймворку Truffle, Sulong надає платформу для ефективного виконання коду, написаного на мовах, які компілюються до LLVM IR, включаючи C і C++. Це включає в себе використання всіх оптимізацій і вдосконалень, які пропонує компілятор Graal, забезпечуючи високу продуктивність та оптимізацію коду на рівні JVM. [11][12]

```
int add(int x, int y) {
return x + y;
}

# Функція add у форматі LLVM IR

define i32 @add(i32 %x, i32 %y) #0 {
entry:
%1 = add i32 %x, %y
ret i32 %1
}
```

Рисунок 1.7 Приклад проміжного представлення функції add у форматі LLVM IR

Після створення LLVM IR, Truffle оптимізує його, за допомогою методів згаданих в розділі 1.2.3 і виконає його.

Розділ 2. Покращення продуктивності з GraalVM

2.1 Ahead-of-time компіляція

У сучасному світі розробки програмного забезпечення, компанії активно застосовують підхід пакування додатків. Цей процес є важливим для ефективного розповсюдження програм на різні пристрої. Завдяки пакуванню, ІТ-відділи мають значно менше клопоту, оскільки кожен пакет вже містить усе необхідне для коректної роботи програми, включаючи налаштування та залежності. Це не тільки сприяє швидшому встановленню та підвищує надійність роботи додатків, але й створює уніфіковане програмне середовище. Таке середовище знижує ризик технічних помилок і конфліктів між програмами, забезпечуючи високу сумісність і стабільність їхньої роботи. [13]

Нативно, JVM підтримує пакування програм у форматі JAR. JAR - це формат, який зазвичай використовується для об'єднання класів Java та пов'язаних з ними метаданих і ресурсів в один файл для розповсюдження прикладного програмного забезпечення або бібліотек на платформі Java. Простими словами, JAR-файл - це файл, який містить стиснуту версію файлів .class. Ми можемо уявити файл .jar як заархівований файл zip. Оскільки JAR файл містить в собі скомпільовані класи Java.class, для його запуску потрібне середовище виконння Java додатків - JRE. Важливо, що версія Java в середовищі виконання відповідала версії, в якій був скомпільований JAR файл. [14]

На відміну від JVM, GraalVM підтримує можливість AOT компіляції. GraalVM виконує AOT компіляцію завдяки Substrate VM (SVM). Це

вбудована віртуальна машина з інтегрованими інструментами для розробки, написаними на Java та оптимізованими для виконання на Truffle-сумісних мовах. SVM має функції, подібні до звичайної віртуальної машини, але попередньо скомпільована разом із Graal. Java додаток попередньо компілюється разом з SVM, що створює нативний образ всього додатка. Внутрішньо це двійковий файл який містить віртуальну машину, упаковану разом з Java-програмою. Використання нативних образів може мати значні переваги у продуктивності у певних ситуаціях, коли JVM не потрібна для запуску, завантаження та ініціалізації класів. Такі виконувані файли не демонструють пікової продуктивності, але швидкий запуск і низький час виконання можуть мати велике значення в сучасному хмарному або безсерверному сучасному хмарному або безсерверному виробничому середовищі. [8]

Процес створення нативного образу програми включає компіляцію Java байт-коду, який містить код застосунку, бібліотеки, елементи JDK та віртуальної машини (VM). Аналіз досяжності (points-to analysis) визначає, які класи, методи та поля будуть використовуватися під час виконання програми. Це робиться на основі точок входу, наприклад, головного методу додатка. Аналіз виконується ітеративно до досягнення стану, коли подальші аналізи не виявляють нових досяжних елементів. Аналіз формує граф потоку керування, який допомагає визначити, які об'єкти та їхні типи доступні, та слідкує за їх використанням всередині програми.

Після досягнення фіксованої точки в аналізі досяжності, виконується ініціалізаційний код: викликаються ініціалізатори класів та можуть виконуватися специфічні для додатка функції, зазначені розробником. Це

дає змогу ініціалізувати та конфігурувати програму до початку її виконання.

Також застосовується алгоритм створення знімку купи (heap snapshotting), який будує об'єктний граф на основі визначених досяжних статичних полів та інших значень, які були відомі під час компіляції. Ці об'єкти серіалізуються і зберігаються у секції даних нативного образу.

Нарешті, методи, позначені як досяжні, компілюються в машинний код і розміщуються у текстовій секції нативного образу. Використовується компілятор Graal, який оптимізує код, застосовуючи вже згадані оптимізації.

При виконанні програми вона стартує з уже попередньо заповненою купою, що містить об'єкти, які були ініціалізовані під час створення образу. Це сприяє швидкому запуску і зменшенню використання пам'яті. [15]

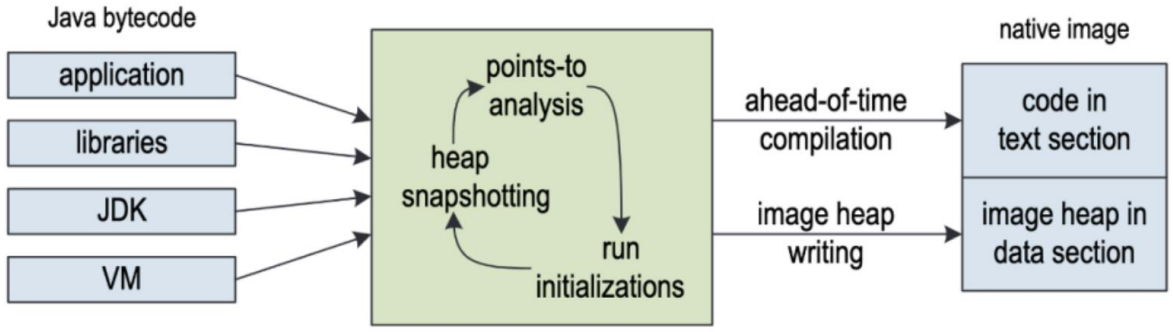


Рисунок 2.1 Процес компіляції в нативний образ [15]

2.2 Варіанти реалізації GraalVM: Community та Enterprise Editions.

GraalVM пропонується в двох редакціях: Community Edition (CE) та Enterprise Edition (EE).

GraalVM Community Edition є відкритим та безкоштовним рішенням, що надається розробникам. Воно підтримує велике розмаїття мов, включно з Java, JavaScript, Python, Ruby, та R, і пропонує можливості JIT компіляції та створення нативних образів, що дозволяє розробникам оптимізувати продуктивність своїх застосунків. CE має широку підтримкою спільноти, що призводить до його неперервного удосконалення та надання підтримки користувачам, а також є популярним вибором для індивідуальних розробників та невеликих команд, які оцінюють гнучкість та відкритість.

Enterprise Edition, натомість, є комерційним продуктом, призначеним для задоволення потреб бізнесу корпоративного рівня. EE містить усі можливості CE з додатковими оптимізаціями та інструментами, що призначені для більш масштабних, виробничих застосунків. Рішення при виборі між CE та EE залежить від специфіки проєкту, бюджету, технічних потреб, довгострокової стратегії підтримки та розширення.

Вибір CE є економічно ефективним і надає сильні інструменти для розробки, тоді як EE пропонує додаткові функції на рівні підприємства, забезпечуючи підтримку та безпеку для важливих додатків. Таким чином, ретельне вивчення потреб проєкту дозволить вибрати між відкритою та доступною платформою CE та більш спеціалізованою, комерційно-орієнтованою EE, кожна з яких має свої унікальні переваги та призначена для задоволення конкретних потреб розробки та впровадження.

2.3 Інструменти розробки та налагодження

GraalVM нативно підтримує низку інструментів, для налагодження та

моніторингу розгорнутих додатків, а також самої платформи GraalVM, таких як VisualVM, GraalVM Insight, Profiling Command Line Tools та інші.

2.3.1 VisualVM

VisualVM представляє собою інтегрований інструментальний комплекс, розроблений для надання глибокого інсайту в роботу Java додатків, що виконуються на JVM, включаючи ті, що базуються на GraalVM. Як універсальний інструмент, VisualVM об'єднує кілька командних утиліт і легко розширюваних плагінів для забезпечення неперевершеного моніторингу та профілювання додатків у реальному часі. VisualVM дозволяє відстежувати використання процесору, пам'яті та інших критичних ресурсів системи в реальному часі, що надає розробникам інформацію необхідну для оптимізації продуктивності їхніх додатків.

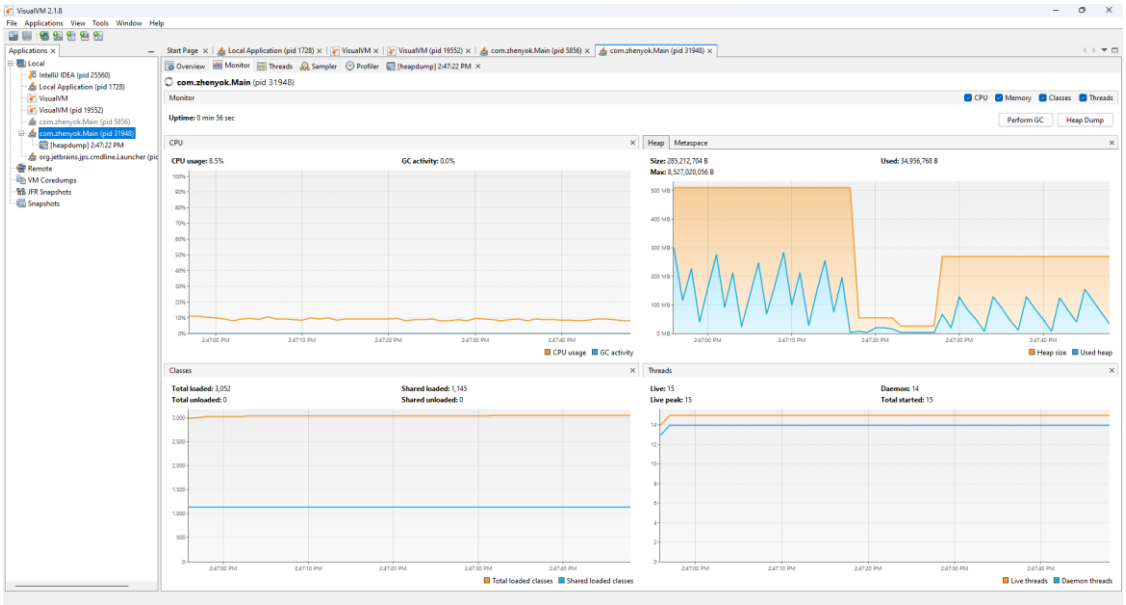


Рисунок 2.2 Відстеження основної інформації програми

Visual VM Забезпечує можливість аналізувати дампи пам'яті та потоків, що допомагає в ідентифікації витоків пам'яті та блокувань потоків, а також в розумінні загальної поведінки додатка.

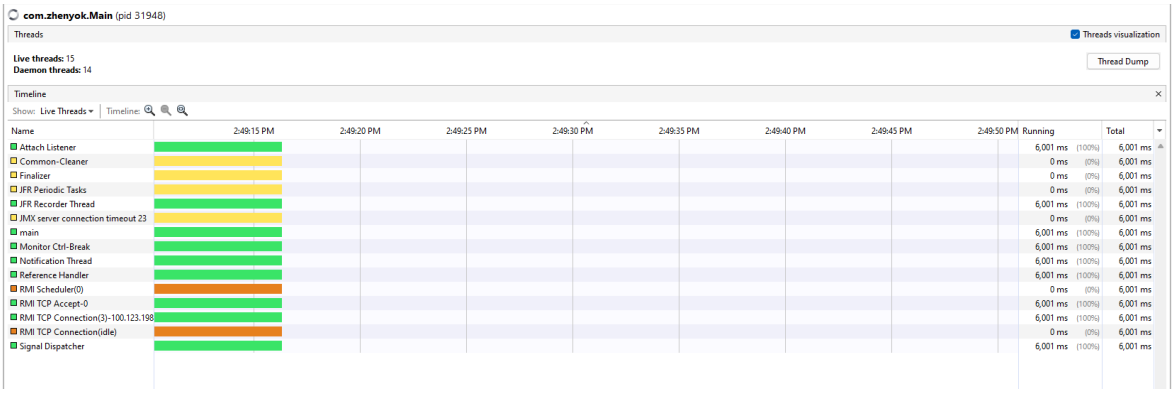


Рисунок 2.3 Аналіз потоків програми

Вбудовані можливості профілювання дозволяють точно визначати “вузькі місця” в коді додатка, аналізуючи час виконання та частоту викликів методів.

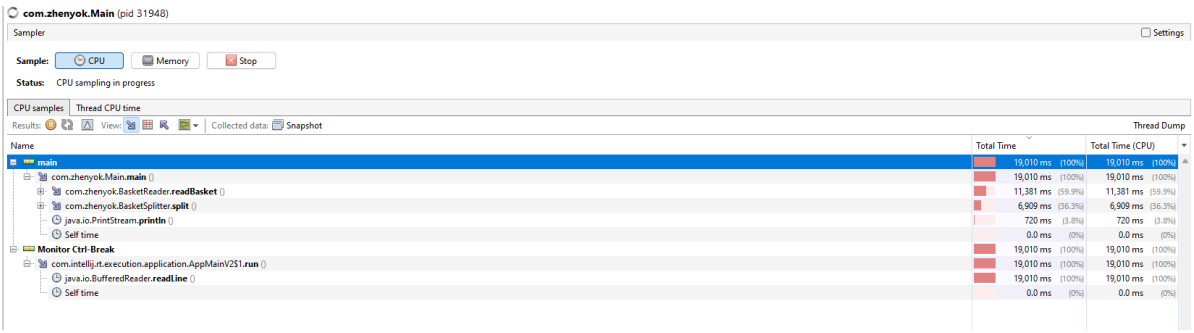


Рисунок 2.4 Аналіз часу виконання методів

VisualVM може бути розширений за допомогою сторонніх плагінів, що дозволяє додавати нові функції або покращувати існуючі можливості інструменту.

Всі популярні середовищі розробки, такі як IntelliJ IDEA, VS Code, Eclipse та NetBeans підтримують можливість інтеграції Visual VM за допомогою плагінів та сторонніх сервісів для спрощення процесу профілювання Java аплікацій.

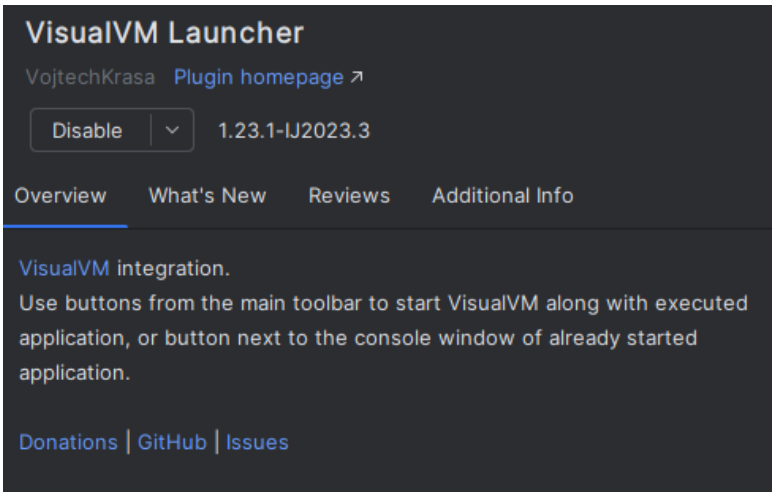


Рисунок 2.5 Плагін VisualVM в IDE IntelliJ IDEA

Ключовою перевагою VisualVM є його нативна інтеграція з GraalVM, яка надає розширені можливості для моніторингу та профілювання поліглотних додатків. Завдяки підтримці GraalVM, VisualVM може використовуватися для аналізу додатків, написаних на різних мовах програмування, що виконуються на одній VM. Це робить його незамінним інструментом для розробників, які прагнуть оптимізувати продуктивність і стабільність своїх поліглотних додатків. [16]

2.3.2 GraalVM Insight

GraalVM Insight представляє собою інструмент, вбудований у середовище GraalVM, який надає розробникам унікальну можливість динамічного аналізу виконання коду. Він розроблений для забезпечення глибокого розуміння поведінки додатків на рівні виконання коду, надаючи можливість вставляти “точки спостереження” без необхідності зупиняти додаток або модифікувати його код.

GraalVM Insight дозволяє розробникам вставляти точки спостереження в код додатка в режимі реального часу. Це надає можливість збирати детальні дані про виконання коду, відслідковувати змінні, виклики функцій та інші важливі аспекти без зупинки або перезавантаження додатка.

Завдяки підтримці багатомовності GraalVM, Insight ефективно працює з додатками, написаними на різних мовах програмування. Це робить його незамінним інструментом для аналізу комплексних поліглотних систем, де різні компоненти можуть взаємодіяти та виконуватися на одній платформі. Інструмент надає розширені можливості для налаштування точок спостереження, дозволяючи розробникам визначати, які саме аспекти виконання їх цікавлять. Це дозволяє проводити цілеспрямований аналіз та оптимізацію, зосереджуючись на критично важливих частинах додатка. [16]

Крім VisualVM та GraalVM Insight, GraalVM надає розробникам цілий арсенал інших потужних інструментів, призначених для оптимізації продуктивності та спрощення процесу розробки поліглотних додатків. До них належать командні інструменти для профілювання, які дозволяють здійснювати глибокий аналіз продуктивності додатків, інструменти для

аналізу покриття коду, які допомагають розробникам визначати, які частини їхнього коду були виконані під час тестування, та Ideal Graph Visualizer для візуалізації внутрішньої структури програм на низькому рівні. Ці інструменти забезпечують додаткові можливості для детального розуміння та оптимізації коду. Також, підтримка Chrome Debugger та інтеграція з протоколами Language Server Protocol та Debug Adapter Protocol розширюють можливості розробників у контексті розробки та налагодження сучасних веб-додатків та сервісів, роблячи GraalVM універсальною платформою, здатною задовольнити різноманітні потреби в розробці програмного забезпечення. [16]

Розділ 3. Практичне порівняння продуктивності GraalVM

3.1 Аналіз продуктивності GraalVM

Після аналізу доступних бенчмарків для оцінки продуктивності віртуальних машин було обрано набір тестів DaCapo Benchmarks версії 9.12 MR1-bach. Цей набір розроблений спеціально для бенчмаркінгу VM, оскільки він включає в себе тестування компіляторів та управління пам'яттю. DaCapo складається з набору реальних додатків з відкритим вихідним кодом, що дозволяє демонструвати середовище, наближене до виробничого. Важливість вибору саме DaCapo полягає в його здатності до точного відтворення реального навантаження на VM, що забезпечує високу релевантність результатів тестування. Зокрема, для детального аналізу продуктивності обрані тести, такі як sunflow, який є індикатором обчислювальної потужності в умовах інтенсивної роботи з графікою та пам'яттю, що є критичним для оцінки властивостей сучасних VM. [23]

Для оцінки продуктивності було обрані наступні тести:

- h2 - виконує JDBC подібний бенчмарк в пам'яті, виконуючи ряд транзакцій на моделі банківського додатка.
- lusearch-fix - Використовує бібліотеку Apache Lucene для текстового пошуку за ключовими словами в масиві даних, що включає твори Шекспіра та Біблію короля Якова.
- xalan - перетворює XML-документи у формат HTML.
- pmd - аналізує набір Java класів на предмет наявності проблем у програмному коді.

- sunflow - рендерить набір зображень з використанням трасування променів.
- jython - інтерпретація бенчмарку pybench на Python. [24]

Методологія тестування базувалася на виконанні кожного з тестів за фіксованою кількістю ітерацій, а саме 20. Перші 15 ітерацій були призначені для “розігріву” віртуальної машини, під час якого відбувається активізація та стабілізація роботи компонентів, таких як JIT-компілятор (Just-In-Time) та системи управління пам'яттю (Garbage Collector). Це дозволяє зменшити вплив початкових варіацій на продуктивність та отримати більш стабільні результати. Останні 5 ітерацій використовувались для збору та аналізу результатів тестування, забезпечуючи точне відображення оптимізованої продуктивності віртуальних машин.

Для проведення дослідження було обрано використання версій Java Development Kit (JDK) та GraalVM, які базуються на Java 17. Вибір JDK 17 обумовлений її статусом як “Long Term Support” (LTS) версії. Версії LTS забезпечують довготривалу підтримку та стабільність, що є критично важливим для виробничих середовищ. Це дозволяє користувачам розраховувати на постійні оновлення та підтримку безпеки протягом тривалого періоду часу, що знижує ризики та сприяє надійності виробничих систем.

На момент дослідження новіші версії Java недостатньо широко поширені в проєктах, оскільки організації часто віддають перевагу перевіреним та стабільним версіям. Отже, застосування Java 17 дозволяє провести

дослідження в умовах, які будуть релевантні більшості проєктів.

Для JDK було обрано версію Oracle jdk-17.0.10. Вибір GraalVM також припав на версію, що відповідає JDK 17, а саме graalvm-jdk-17.0.10+11.1 версії 23.0-b27. Вибір саме цієї версії GraalVM диктується її сумісністю та оптимізацією під JDK 17, що забезпечує можливість коректного порівняння продуктивності між стандартним JDK та GraalVM на однаковій технологічній основі.

Таблиця 3.1 Результати тестування GraalVM та JDK 17 за допомогою DaCapo в мілісекундах з поданим стандартним відхиленням

Бенчмарки	**GraalVM**	**JDK 17**
h2 avg	13243.4 ± 494.4	14880.9 ± 631.6
lusearch avg	546.2 ± 27.06	590.5 ± 22.7
xalan avg	2819.4 ± 49.13	2927.4 ± 59.3
pmd avg	1561.4 ± 20.3	1646.0 ± 10.2
sunflow avg	1263.7 ± 41.4	1584.8 ± 79.2
jython avg	3996.0 ± 38.0	5002.0 ± 37.9

В рамках проведених замірів ми бачимо вагомі відмінності в продуктивності між GraalVM та JDK 17. В цілому, було помічено, що GraalVM демонструє кращі середні показники часу виконання (avg) та більшу стабільність (нижче стандартне відхилення, stdev) порівняно з JDK 17 у більшості тестів.

Зокрема, значні переваги GraalVM були виявлені у таких тестах, як jython і h2. У тесті jython, який оцінює продуктивність інтерпретації скриптової мови Python, середній час виконання в GraalVM був майже на 20% кращий від JDK 17, що вказує на ефективність GraalVM у оптимізації високорівневого коду. Це є результатом розширеної підтримки поліглотного виконання в GraalVM, яке оптимізовано для кращої взаємодії між різними мовами програмування.

Тест h2 також продемонстрував значну перевагу GraalVM, з більш ніж на 10% кращим середнім часом виконання у порівнянні з JDK 17. Цей результат обумовлений вдосконаленою обробкою внутрішньої пам'яті та більш ефективною роботою з базами даних завдяки оптимізаціям, що вбудовані в GraalVM.

Ці результати вказують на те, що GraalVM може надавати кращу продуктивність для застосунків, що вимагають інтенсивної роботи з пам'яттю та високої обчислювальної потужності, а також тих, що використовують міжмовну інтеграцію. Хоча загальна тенденція свідчить про перевагу GraalVM, детальніше дослідження індивідуальних тестів може дати більш глибоке розуміння оптимального використання обох JVM залежно від конкретних вимог застосунку.

3.2 Вибір проєкту для порівняння продуктивності

Проаналізувавши список популярних фреймворків мовою Java було обрано Spring. Його справедливо називають найкращим фреймворком Java для веб-розробки. Це надійний, простий у використанні та найпопулярніший

фреймворк для прикладної Java, що впливає на підвищення швидкості, простоти і безпеки використання Java. Spring користується популярністю серед великого корпоративного сегменту, неодноразово стаючи вибором номер один серед професійних розробників. [17]

Spring підходить для створення корпоративних Java-додатків. Він дозволяє розробникам зосередитися на бізнес-логіці додатка. Поєднання Spring MVC та Spring Cloud перетворює Java на сучасний, реактивний та хмарний інструмент для створення високопродуктивних, складних веб-додатків. [17]

В рамках курсової роботи для порівняння продуктивності було взято існуючий веб-додаток для ресторану “Spring Бутери”. Застосунок має всі основні функції для функціонування магазину.

Система має можливість реєстрації, авторизації та автентифікації, розділяючи доступний функціонал між користувачем та адміністратором. Користувач може замовляти страви та переглядати свої замовлення, адміністратор, в свою чергу, може переглядати всіх користувачів, доступних страв для замовлення, категорій страв та замовлень.

Даний додаток реалізує базовий функціонал повноцінного веб додатка а саме: хешування паролів, використання бази даних, реалізацію CRUD операцій та виклик API для відображення погоди. Для бази даних використовується виділений сервер PostgreSQL.

Проєкт містить в собі 59 Java класів та 10 HTML шаблонів, які мають 2057 та 518 рядків коду відповідно. До проєкту через конфігураційний файл pom.xml підключено 26 залежностей, в тому числі spring-boot-starters , що забезпечують підтримку основних модулів Spring, включно з безпекою, веб-

розробкою, доступом до даних та управлінням транзакціями. Ці залежності сприяють інтеграції зовнішніх служб та оптимізації робочих процесів, забезпечуючи високий рівень абстракції та зниження складності програмного забезпечення. [18]

Крім того, завдяки використанню Spring Security проєкт впроваджує розширені засоби автентифікації та авторизації, що дозволяє забезпечити захист даних та конфіденційність користувачів. [19] Spring Data JPA спрощує роботу з базою даних, надаючи потужні абстракції для реалізації складних запитів і транзакцій без безпосереднього написання SQL-коду. [20]

Проєкт також включає модуль Spring Boot Actuator для моніторингу та управління застосунком у режимі реального часу, що важливо для виявлення та вирішення проблем у процесі експлуатації додатка.

3.3 Тестування з обраним проєктом

3.3.1 Підготовка до тестування

Методологія тестування полягала в запуску проєкту 10 разів підряд без пауз та замірів середнього значення часу запуску та використання оперативної пам’яті за допомогою платформи Docker, яка дозволяє “обгорнути” проєкт в цілісний образ для запуску в вигляді контейнера. Завдяки графічному інтерфейсу, Docker дозволяє моніторити всю потрібну інформацію про запущений контейнер. Порівняння проводилось для згаданих у розділі 3.1 версій JDK та Graalvm, також у тестуванні брав участь нативний образ проєкту, створений завдяки компоненту Native Image.

Створення нативного образу проєкту вимагав додаткової конфігурації, а саме врахування такої важливої складової Java, як рефлексія. Рефлексія дозволяє коду Java перевіряти власні класи, методи, поля та їхні властивості під час виконання. Вивчення та доступ до елементів програми за допомогою рефлексії або завантаження класів час виконання вимагає підготовки додаткових метаданих для цих елементів програми в зображенні. Ці метадані мають зберігатися в зображенні вже під час його створення.

Для уникання завантаження класів під час виконання програми, потрібно створити конфігурацій файл “reflect-config.json” і занести туди інформацію про елементи програми, до яких відбувається рефлексивний доступ у форматі:

```
[
  {
    "name" : "java.lang.Class",
    "queryAllDeclaredMethods" : true,
    "queryAllPublicMethods" : true,
    "allDeclaredClasses" : true,
    "allPublicClasses" : true
  }
]
```

Для коректної роботи потрібно вказати назву класу включно з пакетом, та конфігурації доступності елементів класу під час виконання програми. Через високу абстрактність Java, вручну прописати конфігурацію для всіх елементів які використовують рефлексію буде неможливо. Для цього в GraalVM існує функціонал автоматичного визначення елементів, які потребують декларування в конфігураційному файлі а саме Tracing Agent. [21]

Tracing Agent використовується для легкого збору метаданих та підготовки конфігураційних файлів. Агент відстежує всі використання динамічних функцій під час виконання програми на звичайній Java VM. Під час запуску агент шукає класи, методи, поля, ресурси, для яких інструменту Native Image потрібна додаткова інформація. Після завершення роботи програми і виходу з JVM агент записує метадані у JSON-файли у вказаний каталог виводу, вказаний під час запуску агенту:

java -agentlib:native-image-agent=config dir=/path/to/config-dir/ ...

Згенеровані конфігураційні файли можна надати інструменту native-image, розмістивши їх у каталозі META-INF/native-image/. У цьому каталозі (або будь-якому з його підкаталогів) виконується пошук файлів з іменами jni-config.json, reflect-config.json, proxy-config.json, resource-config.json, predefined-classes-config.json, serialization-config.json, які потім автоматично включаються у процес збірки в нативний образ. [22]

Створення нативного образу для запуску в Docker середовищі вимагало додаткових конфігурацій процесу створення, оскільки Native Image не підтримує кросплатформну компіляцію і створює образ для цільової системи, на якій він запущений. Це вимагало створення нативного образу одразу в Docker з середовищем Linux всередині. Оскільки Docker більш пристосований до роботи в Unix середовищах, то і образ потрібно створити для роботи в Unix подібних системах.

Для створення нативного образу, потрібно додати плагін “native-maven-plugin” у конфігураційний файл pom.xml та додати аргументи для збірки:

- -H:ConfigurationFileDirectories – для вказання директорії з файлами,

які були створені за допомогою Tracing Agent

- --static – для створення статично зв’язаних образів, це означає, що всі необхідні бібліотеки, які зазвичай динамічно підвантажуються під час виконання програми, будуть вбудовані безпосередньо в сам виконуваний файл.
- --libc=musl – musl — це легка та швидка альтернатива стандартній бібліотеці C (glibc), яка використовується для розгортання в Docker. [22]

Після чого, додати команду в Dockerfile яка створить нативне зображення: native:compile -Pnative, створити Docker образ та запустити контейнер.

Для демонстрації ефективності різних конфігурацій виконання проєкту, порівняння було зосереджене на двох ключових параметрах: часі запуску та використанні оперативної пам'яті. Нижче наведено аналіз результатів тестування трьох різних варіантів: проєкту, запущеного на традиційній JVM, проєкту, запущеного на GraalVM, та проєкту в нативному образі, створеному за допомогою GraalVM:

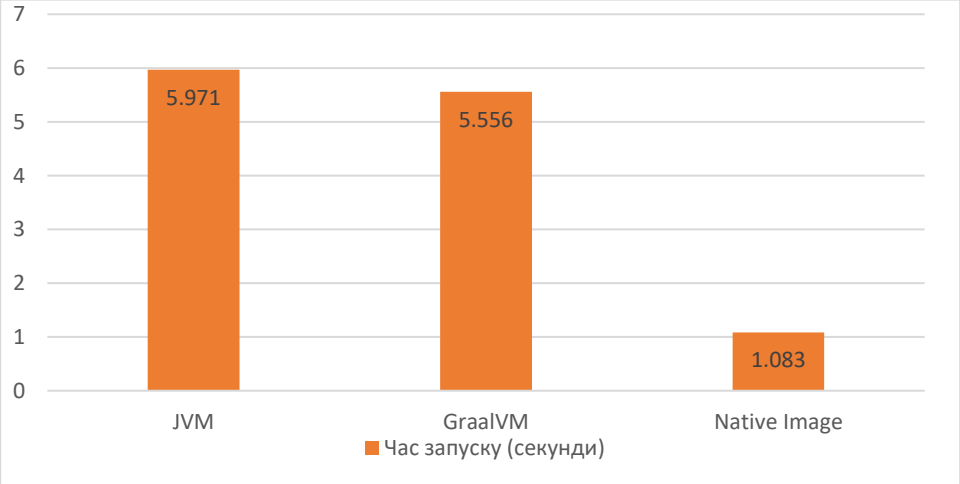


Рисунок 3.1 Порівняння часу запуску в секундах

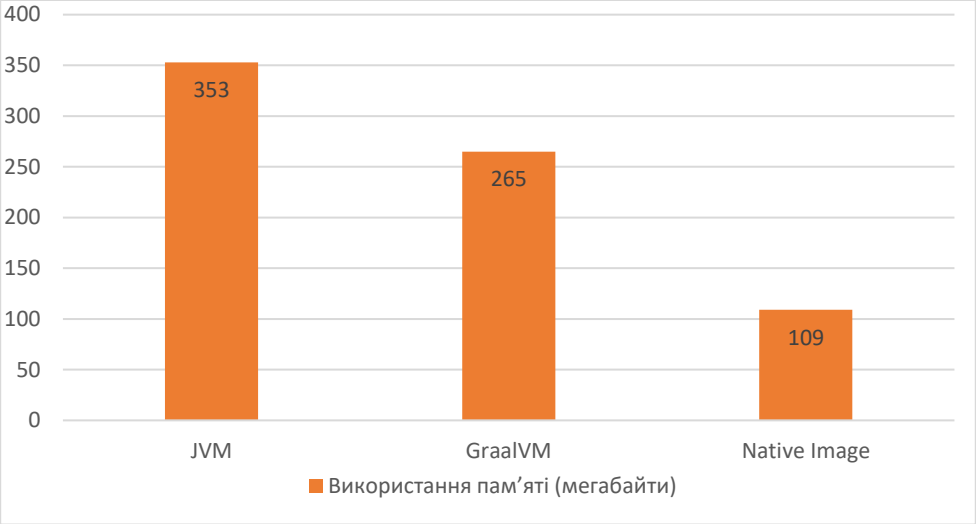


Рисунок 3.2 Порівняння використання пам’яті в мегабайтах

3.3.2 Результати тестування

Перший ключовий аспект аналізу — це час запуску. Тестування (рис. 3.1) показало, що проєкт у нативному образі має значно швидший час запуску порівняно з JVM і GraalVM. Середній час запуску для нативного образу становив 1.083 секунди, тоді як для GraalVM і традиційного JVM цей показник був 5.556 і 5.971 секунд відповідно. Це підкреслює переваги компіляції в нативний код, яка забезпечує менший час завантаження за рахунок відсутності необхідності в інтерпретації та компіляції в байт-код під час запуску.

Що стосується використання пам'яті, результати (рис 3.2) також були на користь нативного образу. Проєкт у нативному образі споживав приблизно на 30% менше оперативної пам'яті порівняно з JVM і GraalVM. JVM використовувала близько 353 МБ пам'яті, в той час як GraalVM споживала 285 МБ, а нативний образ вимагав лише близько 109 МБ. Це вказує на більш

ефективне управління пам'яттю у випадку нативних образів, яке обумовлено відсутністю необхідності використання JRE для запуску. Нативний образ компілюється безпосередньо в машинний код, який виконується безпосередньо на цільовій системі, що значно зменшує споживання ресурсів, зокрема оперативної пам'яті. Така оптимізація є особливо важливою для ресурсообмежених середовищ або при масштабних розгортаннях, де кожен мегабайт пам'яті на рахунку.

Висновки

У цій курсовій роботі було здійснено глибокий аналіз можливостей GraalVM, сучасної віртуальної машини, яка значно покращує продуктивність застосунків, розроблених у JVM екосистемі. Значну увагу було приділено JIT-компіляції та AOT-компіляції, що робить GraalVM надійним інструментом для розробки високопродуктивних програмних рішень.

За результатами практичного порівняння, проведеного в третьому розділі, GraalVM продемонструвала значні переваги порівняно з традиційною JVM. Наприклад, у тестах DaCapo GraalVM показала зменшення часу виконання на 20-30% порівняно з JVM. Особливо вражаючі результати були отримані при запуску тесту 'jython', де GraalVM показала на 20% швидше виконання порівняно зі стандартною JVM.

Крім того, застосування GraalVM в реальному веб-проєкті на базі Spring підтвердило її переваги у зниженні часу запуску та використанні оперативної пам'яті. Нативний образ, створений за допомогою GraalVM, запускався на 75% швидше порівняно з традиційною JVM. Також нативний образ вимагав на 70% менше оперативної пам'яті, що є критично важливим для виробничих середовищ із обмеженими ресурсами.

Завдяки адаптивності до поліглотного програмування та високій продуктивності, GraalVM виявилася ефективним рішенням для компаній, які прагнуть інтегрувати новітні технології та оптимізувати виконання своїх застосунків. Використання цієї технології компаніями, такими як Twitter,

Alibaba та Nvidia, підтверджує її ефективність та надійність, забезпечуючи їм конкурентні переваги у виробництві надійних та масштабованих рішень.

В результаті, GraalVM демонструє значний потенціал для використання в різних сферах розробки, від веб-додатків до мікросервісних архітектур, і є ключовим інструментом для компаній, що прагнуть максимально ефективно використовувати можливості сучасних технологій.

Список використаної літератури

1. Рейтинг мов програмування 2024 [Електронний ресурс]. – Режим доступу до ресурсу: https://dou.ua/lenta/articles/language-rating-2024.

2. Introduction to GraalVM [Електронний ресурс]. – Режим доступу до ресурсу: https://www.graalvm.org/22.2/docs/introduction/.

3. Maxine Research VM [Електронний ресурс]. – Режим доступу до ресурсу: https://labs.oracle.com/pls/apex/f?p=94065:12:17236785846387:9.

4. Graal Project [Електронний ресурс]. – Режим доступу до ресурсу: https://openjdk.org/projects/graal/.

5. The Java HotSpot Performance Engine Architecture [Електронний ресурс]. – Режим доступу до ресурсу: https://www.oracle.com/java/technologies/whitepaper.html.

6. Visualization of Program Dependence Graphs [Електронний ресурс] / [Thomas Wuerthinger, Hanspeter Mössenböck, Christian Wimmer]. – 2008. – Режим доступу до ресурсу: https://www.researchgate.net/publication/221302634_Visualization_of_Program_Dependence_Graphs.

7. Github graal [Електронний ресурс]. – Режим доступу до ресурсу: https://github.com/oracle/graal/blob/407ca89d90b9a37f0aded86e334c2711d16b411e/compiler/src/jdk.graal.compiler/src/jdk/graal/compiler/hotspot/HotSpotGraalCompiler.java#L107

8. Exploring Aspects of Polyglot High-Performance Virtual Machine GraalVM [Електронний ресурс] / [M. Šipek, B. Mihaljević, A. Radovan]. – Режим доступу до ресурсу: https://arxiv.org/ftp/arxiv/papers/2112/2112.14716.pdf

9. End of Line blog [Електронний ресурс] / [Adam Ruka]. – 2024. – Режим доступу до ресурсу: https://www.endoflineblog.com/graal-truffle-tutorial-part-0-what-is-truffle

10. Project Sulong: an LLVM bitcode interpreter on the Graal VM with Matthias Grimmer [Електронний ресурс]. – 2018. – Режим доступу до ресурсу: https://www.youtube.com/watch?v=yyDD_KRdQQU

11. Polyglot API docs [Електронний ресурс] / [Oracle]. – 2022. – Режим доступу до ресурсу: https://docs.oracle.com/en/graalvm/enterprise/20/docs/reference-manual/embed-languages/

12. The LLVM Compiler Infrastructure [Електронний ресурс]. – Режим доступу до ресурсу: https://llvm.org/

13. Application Packaging: What is Software Packaging and Why is it Important? [Електронний ресурс]. – Режим доступу до ресурсу: https://cpl.thalesgroup.com/software-monetization/application-packaging

14. JAR files in Java [Електронний ресурс]. – Режим доступу до ресурсу: https://www.geeksforgeeks.org/jar-files-java/

15. Christian Wimmer, Codrut Stancu, Peter Hofer, Vojin Jovanovic, Paul Wögerer, Peter B. Kessler, Oleg Pliss, Thomas Würthinger. 2019. Initialize once, start fast: application initialization at build time. Proc. ACM Program. Lang. 3, OOPSLA, Article 184 (October 2019), 29 pages. https://doi.org/10.1145/3360610

16. Tools for GraalVM Languages / [Електронний ресурс]. – Режим доступу до ресурсу: https://www.graalvm.org/latest/tools/

17. 10 Best Java Frameworks for Web Development in 2024/ [Електронний

ресурс]. – Режим доступу до ресурсу: https://www.aimprosoft.com/blog/java-framework-for-web-development/

18. Intro to Spring Boot Starters [Електронний ресурс] / Режим доступу до ресурсу https://www.baeldung.com/spring-boot-starters

19. Spring Security [Електронний ресурс] / Режим доступу до ресурсу: https://spring.io/projects/spring-security

20. Spring Data JPA [Електронний ресурс] / Режим доступу до ресурсу: https://spring.io/projects/spring-data-jpa

21. Collect Metadata with the Tracing Agent [Електронний ресурс] / Режим доступу до ресурсу: https://www.graalvm.org/latest/reference-manual/native-image/metadata/AutomaticMetadataCollection/

22. Static and Mostly Static Images [Електронний ресурс] / Режим доступу до ресурсу: https://www.graalvm.org/22.0/reference-manual/native-image/StaticImages/

23. A Multifaceted Memory Analysis of Java Benchmarks [Електронний ресурс] / [O. Papadakis, A. Andronikakis, N. Foutris та ін.]. – 2023. – Режим доступу до ресурсу: https://dl.acm.org/doi/10.1145/3617651.3622978.

24. DaCapo Benchmarks [Електронний ресурс] / Режим доступу до ресурсу: https://dacapobench.sourceforge.net/benchmarks.html

25. GraalVM Adoption [Електронний ресурс] / Режим доступу до ресурсу: https://www.graalvm.org/use-cases/