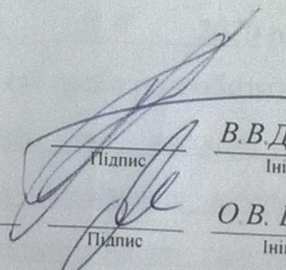
Хмельницький національний університет
Факультет програмування та комп’ютерних і телекомунікаційних систем
Кафедра комп'ютерних наук та інформаційних технологій

ДИПЛОМНА РОБОТА
МАГІСТРА

Галузь знань *12 – Інформаційні технології*
Шифр і назва галузі знань

Спеціальність *122 – Комп’ютерні науки*
Шифр і назва спеціальності

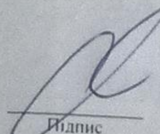
на тему *Модель, метод та засоби оптимізації продуктивності кластерних java-систем*



Виконав: студент *2* курсу, група *КНм-19-1* ______ *В.В.Доброловський*
Підпис Ініціали, прізвище

Керівник: *д.т.н., професор кафедри КНІТ* ______ *О.В. Бармак*
Підпис Ініціали, прізвище

До захисту допускаю:
Зав. кафедри КНІТ, д.т.н., професор ______ *О.В. Бармак*
Підпис Ініціали, прізвище



7 *12* 2020 р.

Хмельницький 2020

ХМЕЛЬНИЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

Факультет *програмування та комп'ютерних і телекомунікаційних систем*
Кафедра *комп'ютерних наук та інформаційних технологій*
Освітній ступінь *магістр*
Галузь знань *12 – Інформаційні технології*
Спеціальність *122 – Комп'ютерні науки*

ЗАТВЕРДЖУЮ
Завідувач кафедри *комп'ютерних наук та інформаційних технологій*

____
(підпис)

д.т.н., професор О.В. Бармак

«7» 09 2020 року

ЗАВДАННЯ
НА ДИПЛОМНУ РОБОТУ МАГІСТРА

1. Тема дипломної роботи магістра: *«Модель, метод та засоби оптимізації продуктивності кластерних java-систем»*

2. Завдання видано студент *Доброловському Володимиру Володимировичу*
(прізвище, ім'я, по батькові)

3. Керівник роботи *д.т.н., професор Бармак О. В.*
(прізвище, ім'я, по батькові)

4. Затверджені наказом університету від «9» 09 2020 р. № 22

5. Зміст пояснювальної записки (перелік питань, які треба розробити):
Мета роботи – розробка моделі, методу та засобів оптимізації продуктивності кластерних java-систем. Для досягнення мети необхідно дослідити існуючі підходи до оптимізації продуктивності кластерних java-систем, а також створити відповідну систему й дослідити її ефективність.

Студент 2 курсу, група КНм-19-1 ________ (підпис) В.В. Доброловський (ініціали, прізвище)

Керівник дипломної роботи магістра ________ (підпис) О.В. Бармак (ініціали, прізвище)

Консультант з нормоконтролю ________ (підпис) Богрій Ф.О. (ініціали, прізвище)

Реферат

Дипломна робота магістра присвячена розробці моделі, методу та засобів оптимізації продуктивності кластерних java-систем.

Актуальність теми. Кластерні середовища зазвичай використовуються в програмах на рівні підприємства, для досягнення більш швидкого часу відгуку та більшої пропускної здатності, ніж середовища окремих машин. Однак цей перехід від монолітної архітектури до розподіленої посилив складність цих застосувань, значно ускладнивши всі заходи, пов'язані з оптимізацією продуктивності таких кластерних систем. Отже, автоматичні методи необхідні для полегшення цих видів діяльності, пов'язаних з продуктивністю, є дуже схильними до помилок і забирають багато часу. Ця робота сприяє оптимізації продуктивності кластерних систем в Java, особливо спрямованих на широкомасштабне середовище. У цій роботі пропонуються дві методики для вирішення проблем ефективного виявлення проблем, пов'язаних з робочим навантаженням, та ефективного уникнення впливу на продуктивність великого збору данних - двох проблем, які є типовими для кластерних систем на Java. Зокрема, ця робота вводить адаптивну основу для автоматизації використання засобів діагностики продуктивності при тестуванні, продуктивності кластерних систем. Мета полягає в тому, щоб полегшити виявлення проблем з ефективністю роботи, зменшивши зусилля та знання, необхідні для ефективного використання таких інструментів. Крім того, запроваджена адаптивна стратегія збалансування навантаження, яка використовує основні прогнози збору данних, щоб визначити найкращий спосіб збалансувати навантаження між наявними вузлами.

Критерій, пов’язаний з продуктивністю, є критичним аспектом якості та головним завданням будь-якого програмного проекту. Однак часті випадки, коли проблеми з продуктивністю виникають і втілюються в серйозні проблеми у значному відсотку застосувань (наприклад, перебої у

виробничому середовищі або навіть скасування програмних проектів). Наприклад, опитування 2007 року, проведене з керівниками інформаційних технологій [69], показало, що 50% з них стикалися з проблемами продуктивності щонайменше в 20% розгорнутих програмах. Ця ситуація частково пояснюється всеосяжним характером продуктивності, що ускладнює оцінку, оскільки на продуктивність практично впливають усі аспекти дизайну, коду та середовища виконання програми.

В останні роки кластерні обчислення набули популярності як потужне та економічно ефективне рішення для паралельної та розподіленої обробки [73]. Таким чином, використання кластерів стає повсюдним: сучасні високоякісні системи та додатки на рівні підприємства, які, як правило, вимагають як швидкого часу відгуку, так і великої пропускної здатності на постійній основі, зазвичай застосовуються в кластерних екземплярах для виконання таких суворих вимог до продуктивності. І це перехід від монолітної архітектури до розподіленої.

Також, збільшено складність цих застосунків, додатково ускладнюючи всі заходи, пов'язані з ефективністю. Особливою проблемою, задокументованою багатьма авторами [50, 17, 57], є те, що сучасні засоби діагностики ефективності значною мірою покладаються на експертів-людей, які мають правильне розуміння проблеми та інтерпретацію результатів. Також, для діагностики проблем із продуктивністю зазвичай потрібні кілька джерел, особливо у високо розподілених середовищах. Наприклад, у Java: дампи потоків, журнали збору данних, кучі дампів, використання центрального процесора та використання пам'яті. І це кілька прикладів інформації, необхідної тестувальнику, щоб зрозуміти ефективність програми. Ця проблема збільшує знання, необхідні для аналізу результативності, котрим зазвичай займається лише невелика кількість експертів в організації [45]. Тому це питання потенційно може призвести до вузьких місць, коли певна діяльність може здійснюватися лише цими експертами,

Доспрощуючи аналіз та діагностику ефективності, багато дослідників розробляють інструменти із вбудованим досвідом [46, 47, 50]. Однак у цих інструментів діагностики існують обмеження, які перешкоджають їх ефективному використанню у високо розподілених середовищах. По-перше, ці інструменти все ще повинні бути налаштовані вручну відповідно до різних чутливих параметрів, які потрібно налаштувати, щоб уникнути поганого впливу на точність результатів роботи інструментів. Якщо використовується невідповідна конфігурація, інструменти можуть не отримати бажані результати, що призведе до значної витрати часу. Крім того, для використання цих інструментів тестувальникам потрібно вручну здійснювати збір даних. У кластерному середовищі, де потрібно контролювати та координувати кілька вузлів, такий ручний процес може зайняти багато часу та спричинити помилки через величезний обсяг даних для збору та консолідації. У тривалому сценарії перевірки продуктивності таке ручне використання засобів діагностики є складнішим через безліч періодичних процесів збору даних. Подібним чином надмірна кількість результатів, вироблених інструментами, може переповнити тестер через час, необхідний для співвіднесення та аналізу результатів. Ця проблема спричинена численними звітами, які зазвичай створюються для кожного моніторингового вузла програми, інформація, яку потрібно корелювати та аналізувати вручну. Надмірна кількість результатів, вироблених інструментами, може перевантажити тестер через час, необхідний для співвіднесення та аналізу результатів. Ця проблема спричинена численними звітами, які зазвичай створюються для кожного моніторингового вузла програми, інформація, яку потрібно корелювати та аналізувати вручну.

Навіть, якщо найбільш критичні помилки продуктивності виправлені до випуску програми у виробниче середовище, інші фактори також можуть погіршити продуктивність кластерної програми. Серед них ще однією ключовою проблемою в кластерних обчисленнях є те, як ефективно розподілити робоче навантаження між ними (оскільки

дисбаланс навантаження може призвести до неефективності обробки [52]). Щоб вирішити цю проблему, численні дослідницькі зусилля спрямовані на розробку більш ефективних алгоритмів та стратегій балансування навантаження, заснованих на різних критеріях та евристиках [64, 77, 79].

Оцінюючи вплив на бізнес в розмірі ста мільярдів доларів на рік, Java є головним продуктом на рівні підприємств [81, 41]. Тому ця технологія зазвичай використовується для побудови кластерних систем. Особливою проблемою щодо продуктивності в Java є збір данних (GC). Незважаючи на те, що це ключова особливість мови Java, яка автоматизує більшість завдань, пов'язаних з управлінням пам'яттю, GC також несе витрати: щоразу, коли це спрацьовує, GC впливає на продуктивність системи, призупиняючи задіяні програми. Хоча паузи в мілісекунди, як правило, не є проблемою, довші паузи GC можуть суттєво вплинути на продуктивність системи, впливаючи на задіяні бізнес-функції та загальний досвід користувачів. Це особливо справедливо для програм, які потребують швидкого часу відгуку або великої пропускної здатності. Крім того, багато дослідницьких робіт засвідчили витрати на показники GC. Дослідження також показали, що неможливо створити єдину стратегію GC, яка найкраще підходить для всіх, оскільки поведінка GC залежить від вхідних даних програми та конфігурації системи [82]. Наприклад, автори [57] показали, що GC особливо чутливий до розміру купи, і навіть незначні зміни, які можуть здатися тривіальними, можуть вплинути на її поведінку. Через численні фактори (наприклад, збільшення обсягу роботи, використання величезних куп чи неідеальних налаштувань), які можуть спровокувати тривалі паузи MaGC (ймовірно, сотні мілісекунд або довше), прийнято вважати, що GC відіграє важливу роль у роботі систем Java.

Мета і задачі роботи. Метою є покращення продуктивності кластерної системи, уникаючи впливів на роботу кластера через значний збір данних, що відбувається на окремих вузлах. Представлені експериментальні результати застосування цих методів до набору реальних програм,

демонструють переваги, які ці методи приносять кластерній системі Java. Для досягнення поставленої мети визначені наступні задачі дослідження:

– здійснити аналіз кластерних сиситем побудованих на java, засоби їх діагностики;

– удосконалити метод, що базується на адаптивній системі автоматизації, яки розглядає загальні обмеження використання інструмента діагностики для ефективного використання при тестуванні продуктивності кластерних програм;

– удосконалити метод адаптивного балансування навантаження для збору даних GC;

– здійснити подальший розвиток технології з визначеним набором метрик, орієнтованих на GC, що характеризують поведінку програм Java з точки зору колекцій данних;

– розробити відповідні засоби діагностики та провести експериментальні дослідження.

Об’єкт дослідження – кластерні сиситеми побудовані на java.

Предмет дослідження – методи визначення продуктивності кластерних систем.

Методи дослідження, застосовані для вирішення поставлених завдань: метод струкутрного проектування «зверху-вниз», метод класифікації, методи аналізу даних.

Наукова новизна одержаних результатів. В результаті проведеної роботи були отримані такі результати:

1. Удосконалено метод, що базується на адаптивній системі автоматизації (M1_JAVA), яки розглядає загальні обмеження використання інструмента діагностики для ефективного використання при тестуванні продуктивності кластерних програм. Фреймворк виконується одночасно з тестом продуктивності, повністю захищаючи тестер від складностей правильної конфігурації та використання діагностичного інструменту, так що тестувальнику потрібно лише взаємодіяти з інструментом тестування навантаження. Це дозволяє підвищити продуктивність тестера за рахунок

зменшення зусиль та досвіду, необхідних для використання діагностичного інструменту. Внутрішньо фреймворк використовує набір політик для автоматичного управління різними ручними процесами, які зазвичай беруть участь у конфігурації та використанні інструменту діагностики продуктивності.

2. Удосконалено метод адаптивного балансування навантаження для збору даних GC (M2_JAVA). Цей вдосконалений балансир навантаження обирає вузли програми, для яких не очікується, що в найближчому майбутньому відбуватиметься подія MaGC, як оптимальні вузли для вхідного робочого навантаження. Отже, ця стратегія може допомогти уникнути впливу на роботу кластера через появу подій MaGC в окремих вузлах. Внутрішньо M2_JAVA використовує алгоритм прогнозування MaGC, щоб визначити найкращий спосіб збалансувати навантаження між вузлами програми.

3. Здійснено подальший розвиток технології з визначеним набором метрик, орієнтованих на GC, що характеризують поведінку програм Java з точки зору колекцій данних. Крім того, на основі їхніх характеристик GC представлена класифікація програм у оцінених тестах Java.

Практичне значення одержаних результатів. Фреймворк виконується одночасно з тестом продуктивності, повністю захищаючи тестер від складностей правильної конфігурації та використання діагностичного інструменту, так що тестувальнику потрібно лише взаємодіяти з інструментом тестування навантаження. Це дозволяє підвищити продуктивність тестера за рахунок зменшення зусиль та досвіду, необхідних для використання діагностичного інструменту. Внутрішньо фреймворк використовує набір політик для автоматичного управління різними ручними процесами, які зазвичай беруть участь у конфігурації та використанні інструменту діагностики продуктивності.

Цей вдосконалений балансир навантаження обирає вузли програми, для яких не очікується, що в найближчому майбутньому відбуватиметься подія MaGC, як оптимальні вузли для вхідного робочого навантаження. Отже, ця стратегія може допомогти уникнути впливу на

роботу кластера через появу подій MaGC в окремих вузлах. Внутрішньо M2_JAVA використовує алгоритм прогнозування MaGC, щоб визначити найкращий спосіб збалансувати навантаження між вузлами програми.

Апробація результатів дипломної роботи магістра та публікації. Основні наукові та практичні результати *доповідалися* на конференціях:

– доповідь на тему «Прогнозування завантаженості ресторану з використанням штучного інтелекту» на Всеукраїнській конференції молодих вчених «Актуальні проблеми комп'ютерних наук» (Хмельницький, 9-10 листопада 2020 року, Хмельницький національний університет).

За темою кваліфікаційної роботи магістра автором виконано одну *наукову публікацію* [83].

Структура та обсяг роботи. Кваліфікаційна робота магістра складається з завдання, реферату, змісту, переліку скорочень, вступу, 4 розділів, висновків, переліку посилань із 83 найменувань. Загальний обсяг дипломної роботи магістра становить 110 сторінок, з них 84 сторінки основного тексту та 18 сторінок застосунків. У роботі наведено 38 рисунків та 10 таблиць,

Ключові слова: кластерна система, діагностична система, клачифікація, java-система.

Зміст

Перелік скорочень 10
Вступ 11
Розділ 1 ....
Аналіз методів і засобів предметної області 17
1.1 Java Віртуальна машина (JVM) 17
1.2 Стратегії GC 20
1.3 Балансування навантаження 22
1.4 Методи прогнозування 23
1.5 Автономні обчислення 25
1.6 Тестування продуктивності 26
1.7 Аналіз відомих рішень 27
1.8 Прогнозування пам'яті 29
1.9 Оптимізація розподілених систем 29
1.10 Моніторинг та управління ефективністю 31
1.11 Автоматизація в тестуванні 32
1.12 Постановка задачі 35
Висновки до розділу 1 35
Розділ 2 ....
Проєктування M1_JAVA та дослідження характеристик 36
2.1 Архітектура 36
2.2 Політика збору даних щодо точності та цільових показників 39
2.3 Політика ефективності та цільового завантаження 40
2.4 Політика консолідації Multi-View 42
2.5 Експериментальне оцінювання 45
Висновки до розділу 2 62
Розділ 3 ....
Стратегія балансування навантаження з урахуванням GC 64
3.1 Використання стратегії балансування навантаження 64

3.2 Алгоритми реалізації стратегії балансування навантаження 66
3.3 Алгоритми балансування навантаження GC-Aware 70
Висновки до розділу 3 72
Розділ 4 ....
Експериментальне оцінювання балансування навантаження 73
4.1 Постановка експериментів 73
4.2 Алгоритми балансування навантаження 80
Висновки до розділу 4 90
Загальні висновки 91
Перелік посилань 94
Додатки

Перелік скорочень

Скорочення, термін, позначення	Пояснення
GC	Сборка мусора (англ. Garbage Collection) – одна з форм автоматичного управління памяттю
JVM	Java Virtual Machine
RTSJ	Специфікація реального часу для Java
SLA	Угоди про рівень обслуговування
EJB	Enterprise JavaBean
ДРМ	Дипломна робота магістра

Вступ

Дипломна робота магістра присвячена розробці моделі, методу та засобів оптимізації продуктивності кластерних java-систем.

Актуальність теми. Кластерні середовища зазвичай використовуються в програмах на рівні підприємства, для досягнення більш швидкого часу відгуку та більшої пропускної здатності, ніж середовища окремих машин. Однак цей перехід від монолітної архітектури до розподіленої посилив складність цих застосувань, значно ускладнивши всі заходи, пов'язані з оптимізацією продуктивності таких кластерних систем. Отже, автоматичні методи необхідні для полегшення цих видів діяльності, пов'язаних з продуктивністю, є дуже схильними до помилок і забирають багато часу. Ця робота сприяє оптимізації продуктивності кластерних систем в Java, особливо спрямованих на широкомасштабне середовище. У цій роботі пропонуються дві методики для вирішення проблем ефективного виявлення проблем, пов'язаних з робочим навантаженням, та ефективного уникнення впливу на продуктивність великого збору данних - двох проблем, які є типовими для кластерних систем на Java. Зокрема, ця робота вводить адаптивну основу для автоматизації використання засобів діагностики продуктивності при тестуванні, продуктивності кластерних систем. Мета полягає в тому, щоб полегшити виявлення проблем з ефективністю роботи, зменшивши зусилля та знання, необхідні для ефективного використання таких інструментів. Крім того, запроваджена адаптивна стратегія збалансування навантаження, яка використовує основні прогнози збору данних, щоб визначити найкращий спосіб збалансувати навантаження між наявними вузлами.

Критерій, пов’язаний з продуктивністю, є критичним аспектом якості та головним завданням будь-якого програмного проекту. Однак часті випадки, коли проблеми з продуктивністю виникають і втілюються в

серйозні проблеми у значному відсотку застосувань (наприклад, перебої у виробничому середовищі або навіть скасування програмних проектів). Наприклад, опитування 2007 року, проведене з керівниками інформаційних технологій [69], показало, що 50% з них стикалися з проблемами продуктивності щонайменше в 20% розгорнутих програмах. Ця ситуація частково пояснюється всеосяжним характером продуктивності, що ускладнює оцінку, оскільки на продуктивність практично впливають усі аспекти дизайну, коду та середовища виконання програми.

В останні роки кластерні обчислення набули популярності як потужне та економічно ефективне рішення для паралельної та розподіленої обробки [73]. Таким чином, використання кластерів стає повсюдним: сучасні високоякісні системи та додатки на рівні підприємства, які, як правило, вимагають як швидкого часу відгуку, так і великої пропускної здатності на постійній основі, зазвичай застосовуються в кластерних екземплярах для виконання таких суворих вимог до продуктивності. І це перехід від монолітної архітектури до розподіленої.

Також, збільшено складність цих застосунків, додатково ускладнюючи всі заходи, пов'язані з ефективністю. Особливою проблемою, задокументованою багатьма авторами [50, 17, 57], є те, що сучасні засоби діагностики ефективності значною мірою покладаються на експертів-людей, які мають правильне розуміння проблеми та інтерпретацію результатів. Також, для діагностики проблем із продуктивністю зазвичай потрібні кілька джерел, особливо у високо розподілених середовищах. Наприклад, у Java: дампи потоків, журнали збору данних, кучі дампів, використання центрального процесора та використання пам'яті. І це кілька прикладів інформації, необхідної тестувальнику, щоб зрозуміти ефективність програми. Ця проблема збільшує знання, необхідні для аналізу результативності, котрим зазвичай займається лише невелика кількість експертів в організації [45]. Тому це питання потенційно може

призвести до вузьких місць, коли певна діяльність може здійснюватися лише цими експертами,

Доспрощуючи аналіз та діагностику ефективності, багато дослідників розробляють інструменти із вбудованим досвідом [46, 47, 50]. Однак у цих інструментів діагностики існують обмеження, які перешкоджають їх ефективному використанню у високо розподілених середовищах. По-перше, ці інструменти все ще повинні бути налаштовані вручну відповідно до різних чутливих параметрів, які потрібно налаштувати, щоб уникнути поганого впливу на точність результатів роботи інструментів. Якщо використовується невідповідна конфігурація, інструменти можуть не отримати бажані результати, що призведе до значної витрати часу. Крім того, для використання цих інструментів тестувальникам потрібно вручну здійснювати збір даних. У кластерному середовищі, де потрібно контролювати та координувати кілька вузлів, такий ручний процес може зайняти багато часу та спричинити помилки через величезний обсяг даних для збору та консолідації. У тривалому сценарії перевірки продуктивності таке ручне використання засобів діагностики є складнішим через безліч періодичних процесів збору даних. Подібним чином надмірна кількість результатів, вироблених інструментами, може переповнити тестер через час, необхідний для співвіднесення та аналізу результатів. Ця проблема спричинена численними звітами, які зазвичай створюються для кожного моніторингового вузла програми, інформація, яку потрібно корелювати та аналізувати вручну. Надмірна кількість результатів, вироблених інструментами, може перевантажити тестер через час, необхідний для співвіднесення та аналізу результатів. Ця проблема спричинена численними звітами, які зазвичай створюються для кожного моніторингового вузла програми, інформація, яку потрібно корелювати та аналізувати вручну.

Навіть, якщо найбільш критичні помилки продуктивності виправлені до випуску програми у виробниче середовище, інші фактори

також можуть погіршити продуктивність кластерної програми. Серед них ще однією ключовою проблемою в кластерних обчисленнях є те, як ефективно розподілити робоче навантаження між ними (оскільки дисбаланс навантаження може призвести до неефективності обробки [52]). Щоб вирішити цю проблему, численні дослідницькі зусилля спрямовані на розробку більш ефективних алгоритмів та стратегій балансування навантаження, заснованих на різних критеріях та евристиках [64, 77, 79].

Оцінюючи вплив на бізнес в розмірі ста мільярдів доларів на рік, Java є головним продуктом на рівні підприємств [81, 41]. Тому ця технологія зазвичай використовується для побудови кластерних систем. Особливою проблемою щодо продуктивності в Java є збір данних (GC). Незважаючи на те, що це ключова особливість мови Java, яка автоматизує більшість завдань, пов’язаних з управлінням пам’яттю, GC також несе витрати: щоразу, коли це спрацьовує, GC впливає на продуктивність системи, призупиняючи задіяні програми. Хоча паузи в мілісекунди, як правило, не є проблемою, довші паузи GC можуть суттєво вплинути на продуктивність системи, впливаючи на задіяні бізнес-функції та загальний досвід користувачів. Це особливо справедливо для програм, які потребують швидкого часу відгуку або великої пропускної здатності. Крім того, багато дослідницьких робіт засвідчили витрати на показники GC. Дослідження також показали, що неможливо створити єдину стратегію GC, яка найкраще підходить для всіх, оскільки поведінка GC залежить від вхідних даних програми та конфігурації системи [82]. Наприклад, автори [57] показали, що GC особливо чутливий до розміру купи, і навіть незначні зміни, які можуть здатися тривіальними, можуть вплинути на її поведінку. Через численні фактори (наприклад, збільшення обсягу роботи, використання величезних куп чи неідеальних налаштувань), які можуть спровокувати тривалі паузи MaGC (ймовірно, сотні мілісекунд або довше), прийнято вважати, що GC відіграє важливу роль у роботі систем Java.

Мета і задачі роботи. Метою є покращення продуктивності кластерної системи, уникаючи впливів на роботу кластера через значний збір данних, що відбувається на окремих вузлах. Представлені експериментальні результати застосування цих методів до набору реальних програм, демонструють переваги, які ці методи приносять кластерній системі Java. Для досягнення поставленої мети визначені наступні задачі дослідження:

– здійснити аналіз кластерних сиситем побудованих на java, засоби їх діагностики;

– удосконалити метод, що базується на адаптивній системі автоматизації, яки розглядає загальні обмеження використання інструмента діагностики для ефективного використання при тестуванні продуктивності кластерних програм;

– удосконалити метод адаптивного балансування навантаження для збору даних GC;

– здійснити подальший розвиток технології з визначеним набором метрик, орієнтованих на GC, що характеризують поведінку програм Java з точки зору колекцій данних;

– розробити відповідні засоби діагностики та провести експериментальні дослідження.

Об’єкт дослідження – кластерні сиситеми побудовані на java.

Предмет дослідження – методи визначення продуктивності кластерних систем.

Методи дослідження, застосовані для вирішення поставлених завдань: метод струкутрного проектування «зверху-вниз», метод класифікації, методи аналізу даних.

Наукова новизна одержаних результатів. В результаті проведеної роботи були отримані такі результати:

1. Удосконалено метод, що базується на адаптивній системі автоматизації (M1_JAVA), яки розглядає загальні обмеження використання

інструмента діагностики для ефективного використання при тестуванні продуктивності кластерних програм. Фреймворк виконується одночасно з тестом продуктивності, повністю захищаючи тестер від складностей правильної конфігурації та використання діагностичного інструменту, так що тестувальнику потрібно лише взаємодіяти з інструментом тестування навантаження. Це дозволяє підвищити продуктивність тестера за рахунок зменшення зусиль та досвіду, необхідних для використання діагностичного інструменту. Внутрішньо фреймворк використовує набір політик для автоматичного управління різними ручними процесами, які зазвичай беруть участь у конфігурації та використанні інструменту діагностики продуктивності.

2. Удосконалено метод адаптивного балансування навантаження для збору даних GC (M2_JAVA). Цей вдосконалений балансир навантаження обирає вузли програми, для яких не очікується, що в найближчому майбутньому відбуватиметься подія MaGC, як оптимальні вузли для вхідного робочого навантаження. Отже, ця стратегія може допомогти уникнути впливу на роботу кластера через появу подій MaGC в окремих вузлах. Внутрішньо M2_JAVA використовує алгоритм прогнозування MaGC, щоб визначити найкращий спосіб збалансувати навантаження між вузлами програми.

3. Здійснено подальший розвиток технології з визначеним набором метрик, орієнтованих на GC, що характеризують поведінку програм Java з точки зору колекцій данних. Крім того, на основі їхніх характеристик GC представлена класифікація програм у оцінених тестах Java.

Практичне значення одержаних результатів. Фреймворк виконується одночасно з тестом продуктивності, повністю захищаючи тестер від складностей правильної конфігурації та використання діагностичного інструменту, так що тестувальнику потрібно лише взаємодіяти з інструментом тестування навантаження. Це дозволяє підвищити

продуктивність тестера за рахунок зменшення зусиль та досвіду, необхідних для використання діагностичного інструменту. Внутрішньо фреймворк використовує набір політик для автоматичного управління різними ручними процесами, які зазвичай беруть участь у конфігурації та використанні інструменту діагностики продуктивності.

Цей вдосконалений балансир навантаження обирає вузли програми, для яких не очікується, що в найближчому майбутньому відбуватиметься подія MaGC, як оптимальні вузли для вхідного робочого навантаження. Отже, ця стратегія може допомогти уникнути впливу на роботу кластера через появу подій MaGC в окремих вузлах. Внутрішньо M2_JAVA використовує алгоритм прогнозування MaGC, щоб визначити найкращий спосіб збалансувати навантаження між вузлами програми.

Апробація результатів дипломної роботи магістра та публікації. Основні наукові та практичні результати *доповідалися* на конференціях:

– доповідь на тему «Прогонозування завантаженості ресторану з використанням штучного інтелекту» на Всеукраїнській конференції молодих вчених «Актуальні проблеми комп'ютерних наук» (Хмельницький, 9-10 листопада 2020 року, Хмельницький національний університет).

За темою кваліфікаційної роботи магістра автором виконано одну *наукову публікацію* [84].

Структура та обсяг роботи. Кваліфікаційна робота магістра складається з завдання, реферату, змісту, переліку скорочень, вступу, 4 розділів, висновків, переліку посилань із 84 найменувань. Загальний обсяг дипломної роботи магістра становить 110 сторінок, з них 84 сторінки основного тексту та 18 сторінок застосунків. У роботі наведено 38 рисунків та 10 таблиць,

Ключові слова: кластерна система, діагностична система, клачифікація, java-система.

Розділ 1
Аналіз методів і засобів предметної області

Розглянемо відповідний стан техніки з точки зору застосовної довідкової інформації та супутніх робіт. У цьому розділі представлені основні особливості та характеристики Java, автоматичні обчислення, а також типовий процес балансування навантаження та тестування продуктивності, які необхідні для розуміння решти роботи. У цьому розділі також описуються методи прогнозування, засоби діагностики продуктивності та тести Java, що використовуються в цій роботі.

1.1 Аналіз предметної області

Це середовище виконання, на якому можуть працювати програми, розроблені на Java. Це тому, що JVM може інтерпретувати двійковий формат (тобто байт-код), в якому компілюється програма, розроблена на Java. Оскільки JVM доступні для більшості сучасних операційних систем, складена програма Java є надзвичайно портативною. Більше того, JVM може бути налаштований під час запуску за допомогою декількох параметрів конфігурації, включаючи кілька параметрів, пов'язаних з управлінням пам'яттю. У таблиці 1.1 описані деякі найбільш часто використовувані варіанти пам'яті.

Таблиця 1.1. Параметри JVM, пов’язані з пам’яттю

Параметр JVM	Опис
-XX: + DisableExplicitGC	Вимкнути явні запити на виконання GC
-XX: PermSize	Ініціалізуйте розмір PermGen
-XX: MaxPermSize	Визначте максимальний розмір PermGen

-Xms	Ініціалізуйте розмір купи
-Xmx	Визначте максимальний розмір купи

Ця форма автоматичного управління пам'яттю пропонує значні переваги в програмній інженерії перед явним управлінням пам'яттю. Наприклад, це звільняє розробників від тягаря ручного управління пам'яттю, уникаючи найпоширеніших джерел перезапису та витоків пам'яті, а також збільшуючи продуктивність розробників. Такі зусилля, як створення специфікації реального часу для Java (RTSJ) [39], яка дозволяє програмам повністю пропустити GC, щоб запропонувати можливості в реальному часі, також можна розглядати як якісний доказ цієї ситуації.

Крім того, неможливо програмно примусити виконати GC. Найближчою дією, яку може зробити розробник, є виклик методу Runtime.getRuntime (). Gc () (або його еквівалентний метод System.gc ()), щоб запропонувати JVM виконати MaGC. Тим не менше, JVM не змушений виконувати цей запит і може ігнорувати його. Використання цих методів також не рекомендується постачальниками JVM [14], оскільки JVM зазвичай робить кращу роботу, вирішуючи, коли робити GC. Не дивно, що це одна з основних мотивацій використання автоматичного управління пам’яттю. Область пам'яті в Java відома як купа. В даний час одним із найбільш часто використовуваних типів купи є генераційна купи [35], де об’єкти розділені за віком на регіони пам'яті (зазвичай обмежені двома-трьома), відомі як покоління.

Нові предмети створюються у наймолодших покоління, оскільки показники виживання молодих поколінь зазвичай нижчі, ніж показники старших поколінь. Тобто молоді покоління частіше містять данних і їх можна збирати частіше, ніж старші. GC у молодих поколінь відомий як Minor GC (MiGC). Зазвичай він недорогий і рідко викликає занепокоєння щодо продуктивності.

MiGC також відповідає за переміщення живих об'єктів, які стали досить дорослими, до старших поколінь. Це означає, що MiGC відіграє ключову роль у розподілі пам'яті старших поколінь. GC у старших поколіннях відомий як MaGC, і він загальновизнаний як найдорожчий тип GC через його ефективність. Нарешті, закінчення вільної пам’яті в одному поколінні викликає відповідний тип події GC. У цій купі молоде покоління (YoungGen) складається з 3 під-областей: один основний, де створюються об'єкти, і два вижили, де МіГК ітеративно розміщує живі об'єкти доти, поки вони не стануть достатніми для переходу до старого покоління (OldGen).

1.2 Стратегії GC

Купа управляється стратегією GC, обраною під час запуску JVM. Їх наявність зазвичай пов’язана з типом купи. Наприклад, три найпоширеніші стратегії GC в галузі [30] працюють виключно на генераторних стратегіях: послідовний GC (який виконує всю свою роботу з використанням одного потоку і є кращим для клієнтських JVM), Parallel GC (який використовує кілька потоків, і переважно для серверних JVM, коли пропускна здатність важливіша за час відгуку), та паралельний GC (який більшість своїх робіт виконує одночасно з потоками програм, і переважно для серверних JVM, коли час відгуку важливіший, ніж пропускна здатність).

Кожна стратегія GC включає два алгоритми, один застосовний до молодого покоління, а інший до старого покоління. Таблиця 1.2 узагальнює їх. Крім того, вони описані в наступних параграфах.

Таблиця 1.2. Алгоритми стратегії GC

Стратегія GC	Алгоритм YoungGen	Алгоритм OldGen
Послідовний	Копіювати	Mark Sweep Compact

Паралельно	PS Scavenge	PS Mark Sweep
Паралельно	PS Scavenge	Одночасна розгортка

Серійні алгоритми GC. Serial GC використовує алгоритм копіювання для YoungGen і Mark Sweep Compact для OldGen. Алгоритм копіювання працює, перевіряючи вік живих об’єктів (з точки зору того, скільки колекцій молодого покоління вони пережили). Потім ті «досить старі» об’єкти копіюються в OldGen, тоді як решта об’єктів копіюються у невикористаний простір вижилих. Якщо цей простір стає повним, живі об'єкти, які перевищили його потужність, також копіюються в OldGen, незалежно від їх віку. Тим часом алгоритм Mark Sweep Compact складається з трьох фаз: у фазі позначки ідентифікуються об'єкти, що перебувають під напругою. Потім у фазі розгортки данних визначається. Далі алгоритм ковзає живі об'єкти до початку простору OldGen,

Паралельні алгоритми GC. Паралельний GC використовує алгоритм PS Scavenge для YoungGen та PS Mark Sweep для OldGen. Ці алгоритми є паралельними версіями алгоритмів, що використовуються послідовним GC. Вони все ще залишаються світовими алгоритмами, але більшість своїх операцій виконують паралельно. Це робиться за рахунок використання наявності декількох процесорів.

Одночасні алгоритми GC. Паралельний GC використовує алгоритм PS Scavenge в YoungGen (той самий алгоритм, що використовується паралельним GC), і паралельний розгортковий марк для OldGen. Алгоритм одночасного розгортки розгортки робить більшу частину своєї роботи одночасно з виконанням програми. Починається з короткої паузи (фаза, відома як початкова позначка), щоб визначити набір об’єктів, що діють безпосередньо, з коду програми. Потім усі живі об'єкти, до яких можна перейти з цього набору, також позначаються (фаза, відома як паралельна

розмітка). Потім настає друга пауза (фаза, відома як зауваження) для завершення маркування шляхом перегляду будь-яких об’єктів, які були змінені програмою під час паралельної фази маркування. Цей крок необхідний, щоб гарантувати, що всі об’єкти, що живуть, були правильно позначені (оскільки програма працювала під час паралельної фази маркування, а деякі об’єкти, що живуть, могли бути пропущені). Завершальною фазою алгоритму є паралельна розгортка, яка відновлює всі данні, які було виявлено.

1.3 Балансування навантаження

Завданням стратегії балансування навантаження є оптимізація продуктивності програми, що працює в кластері, що складається з набору вузлів застосунків (як правило, розміщених в центрі обробки даних), кожен з яких має ідентичне кодове зображення програми. Кластеризована програма також повинна бути розподілена на завдання менших розмірів (наприклад, традиційне веб-застосунок, яке зазвичай складається з атомарних операцій, таких як вхід, вихід, пошук тощо). Діапазон існуючих алгоритмів балансування навантаження є широким [56, 85]. Сьогодні в галузі часто використовуються чотири алгоритми: кругові, випадковий, зважений круговий і зважений випадковий. Round Robin ітеративно вибирає вузли, з часом розподіляючи навантаження рівномірно по доступних вузлах. У випадку випадкового, кожен вузол вибирається навмання серед доступних. Нарешті, у зважених версіях цих алгоритмів кількість випадків, коли буде обрано вузол (відповідно до їх логіки прийняття рішень), регулюється з використанням ваги, визначеної для кожного вузла.

1.4 Методи прогнозування

Завдання методики прогнозування полягає у прогнозуванні майбутньої події на основі наявних даних та аналізу її тенденцій. Крім того, фактори, які стосуються того, що прогнозується, повинні бути відомі та добре зрозумілі для досягнення надійного прогнозу. Спектр існуючих методів прогнозування є широким [99, 73]. Зазвичай їх класифікують на дві основні категорії: якісні та кількісні методи. Якісні прийоми носять суб’єктивний характер, засновані на оцінці певного виду предметних експертів. Ці методи доречні, коли минулі дані відсутні. Прикладом якісної методики є метод Дельфі [80], який використовує поєднання експертного судження та історичної інформації для вироблення індивідуальних оцінок (на одного експерта), які згодом порівнюються серед групи учасників. Цей процес повторюється повторно до досягнення консенсусу.

Методи кількісного прогнозування використовуються для прогнозування майбутніх даних як функції минулих даних. Це досягається шляхом формального відображення співвідношень між задіяними змінними у вигляді математичних рівнянь. Ці методи доречні, коли наявні чисельні дані доступні та коли доцільно припустити, що деякі закономірності в даних очікуються в майбутньому. Три кількісні методи, які найчастіше використовуються, це ковзне середнє, експоненціальне згладжування та регресія. Вони описані в наступних параграфах.

Ковзне середнє. Ця методика включає обчислення середнього значення часового ряду (спостереження з однаковою віддаленістю у часі) від декількох послідовних періодів. Це називається "переміщенням", оскільки воно постійно перераховується, коли нові дані стають доступними. Він прогресує, скидаючи найдавніше значення та додаючи останнє значення.

Трьома найбільш часто використовуваними варіаціями є: (1) проста ковзна середня, у якій значення для даного періоду часу замінюється середнім значенням цього значення на основі значень для деякої кількості попередніх періодів часу; (2) наївна ковзна середня, що є частковим випадком простої ковзної середньої, в якій кількість періодів, що використовуються для згладжування, дорівнює одиниці; та (3) зважена ковзаюча середня, в якій значення для даного періоду часу замінюється середньозваженим значенням цього значення на основі значень для деякої кількості попередніх періодів часу.

Експоненціальне згладжування. Ця техніка може виявити значні зміни в даних, ігноруючи коливання, що не мають значення для цільової мети. На відміну від ковзної середньої, старим даним надається поступово-менша відносна вага (тобто важливість), тоді як новішим дається поступово-більша вага.

Три найбільш часто використовувані його варіації: (1) просте експоненціальне згладжування зважує минулі спостереження з експоненціально зменшуваним ваги для прогнозування майбутніх значень; (2) подвійне експоненціальне згладжування (також відоме як метод Холта) - це вдосконалення простої експоненціальної техніки згладжування, яке додає інший компонент для врахування будь-яких тенденцій у даних; та (3) потрійне експоненціальне згладжування (також відоме як метод Вінтера) є вдосконаленням методу подвійного експоненціального згладжування, що додає інший компонент для врахування будь-якої сезонності (або періодичності) у даних.

Регресія. Ця методика допомагає оцінити значення залежної змінної на основі наявних історичних даних набору незалежних змінних. Це робиться шляхом розробки регресійної моделі, яка складається з незалежних змінних, залежної змінної та будь-яких невідомих (постійних) параметрів.

Три найбільш часто використовувані варіації: (1) проста лінійна регресія включає одну незалежну змінну; (2) множинна лінійна регресія, яка є продовженням єдиної лінійної регресії, включає множинні та / або векторні незалежні змінні; та (3) поліноміальна регресія є формою регресії, при якій зв'язок між незалежною змінною та залежною змінною моделюється як поліном n-го ступеня. Ця властивість дозволяє моделі відповідати нелінійному відношенню між змінними.

1.5 Автономні обчислення

Автономні обчислення [77] - це самокерована обчислювальна модель, яка визначає набір характеристик, необхідних комп'ютерній системі, щоб мати можливість адаптуватися до непередбачуваних змін у своєму середовищі виконання. Його мета - створити системи, які можуть працювати самі по собі. Це означає, що не тільки здатна функціонувати на високому рівні, але й зберігає складність системи невидимою для операторів та користувачів. Мета полягає в тому, що цей тип систем може допомогти подолати стрімко зростаючу складність управління обчислювальними системами, а також зменшити бар'єр, який ускладнення створює для подальшого зростання систем [81]. Модель визначає чотири основні властивості, які повинна мати адаптивна система: (1) самоконфігурування з метою адаптації до динамічно мінливих середовищ (отже, підвищення її чуйності); 2) самовилікування з метою виявлення, діагностування та дії для запобігання зривам (для досягнення стійкості бізнесу); (3) самооптимізація для того, щоб налаштувати ресурси та збалансувати навантаження, щоб максимізувати використання ресурсів (для операційної ефективності); та (4) самозахист з метою передбачення, ідентифікації та захисту від атак (отже, забезпечення інформації та ресурсів). З точки зору архітектури, петлі управління [72] були широко вивчені і визначені ключовим механізмом для

досягнення самоадаптації в програмних системах [61]. На їх основі запропоновано багато рішень для побудови самоадаптивних систем. Наприклад, схема реактивної адаптації [24] описує механізм побудови реактивних компонентів, здатних змінювати свою поведінку у відповідь на зовнішню подію, без потреби в будь-якому контурі управління. В якості альтернативи внутрішня подача задня петля pattern [34] пропонує механізм збагачення стандартної логіки системи шляхом реалізації циклу управління, який відстежує контекст його виконання, визначає зміни, що підлягають застосуванню, і вводить їх у дію. Нарешті, найпоширенішим адаптивним шаблоном є MAPE-K [82]. Він базується на системі управління зворотним зв'язком, що використовується в галузі теорії управління. MAPE-K складається з п’яти елементів, зображених на рисунку 1.4: елемент моніторингу для отримання інформації з керованих систем (за допомогою системних індикаторів); елемент аналізу для оцінки, чи потрібна якась адаптація; елемент для планування адаптації; елемент для його виконання (за допомогою адаптацій); та елемент знання для підтримки інших у виконанні відповідних завдань.

1.6 Тестування продуктивності

Коли застосунок тестується під час розробки, важливо не лише перевірити його здатність виконувати бажані ділові функції за допомогою функціонального тестування, але й оцінити, наскільки застосунок виконує ці функції, коли до нього мають доступ одночасні користувачі. Це мета тестування продуктивності, яке спрямоване на оцінку поведінки застосунка за певного навантаження [81]. У традиційному процесі розробки програмного забезпечення (тобто водоспаду) тестування продуктивності зазвичай проводиться в напрямку закінчення процесу та повторюється для кожної нової побудованої версії

програми. Тим часом у гнучкому процесі розробки програмного забезпечення тестування продуктивності, як правило, проводиться в кінці кожної ітерації (або групи ітерацій).

Тестовий запуск продуктивності передбачає вплив програми на робоче навантаження, яке нагадує очікувані реальні умови. Це досягається використанням генератора навантаження (наприклад, IBM Rational Performance Tester [26], Apache JMeter [2] або HP LoadRunner [22]) протягом тривалого періоду часу (наприклад, 24-годинної або навіть більшої тривалості) для імітації бажаного набору одночасних користувачів, які взаємодіють із додатком.

Під час виконання тестування продуктивності тестувальники зазвичай збирають лічильники, пов’язані з продуктивністю. Зазвичай ці лічильники бувають двох основних типів [43]: метрики продуктивності (тобто часу відгуку та пропускної здатності) та метрики ресурсів (наприклад, використання центрального процесора або пам'яті). Мета полягає в аналізі поведінки контрольованих лічильників у часі, а також у порівнянні лічильників із визначеною базовою лінією продуктивності (наприклад, цільовою угодою про рівень обслуговування), з метою виявлення аномалій продуктивності. Крім того, тестувальники зазвичай використовують якийсь інструмент діагностики ефективності для подальшого дослідження зібраних лічильників ефективності, щоб шукати аномальну поведінку та їх потенційні основні причини.

1.7 Аналіз відомих рішень

Багато дослідницьких зусиль були зосереджені на поліпшенні результатів ГХ. Наприклад, у кількох роботах було запропоновано нові одночасні [11, 13] та паралельні алгоритми [53, 12], які мають менший вплив

на продуктивність програм. Інші роботи спрямовані на розробку алгоритмів, які можуть мати передбачувані показники ГХ [8, 68]. Однак ця передбачуваність виражається в плані м'яких вимог, що означає, що GC все ще може прийнятибільше часу, ніж спочатку передбачалося. Застосовність цих підходів також обмежена, оскільки для них потрібні нестандартні JVM, що також зазвичай передбачає більші витрати на ліцензію. Іншим дослідженим підходом була розробка алгоритмів для конкретних сценаріїв використання. Наприклад, [19] описує алгоритм, придатний для серверів застосунків Java, який використовує різну природу локальних та віддалених об'єктів. Подібним чином [70] представляє перший алгоритм данних, який розбиває купу на кілька областей, щоб повернути пам'ять лише з тих областей, які в основному заселені даннихм. Робота над [48] представляє GC, придатний для багатоядерних процесорів, який врівноважує проблеми місцевості даних з використанням купи та проблемами фрагментації для досягнення хороших показників, підтримуючи абстракцію однієї великої купи незалежно від кількості ядер. Незважаючи на те, що всі ці роботи допомогли зменшити частоту та вплив ГК. Нарешті, інші дослідження зосереджені на вирішенні обмежень зручності використання специфікації реального часу для Java (RTSJ), яка дозволяє програмістам взагалі перейти на GC, ціною вимагаючи значних змін у вихідному коді та втрати переваг автоматичного управління пам’яттю. Наприклад, робота, описана в [26], зосереджена на автоматизації змін коду, необхідних для використання RTSJ, тоді як автори [58] представляють аналіз проблем проектування та програмування роботи з RTSJ. Оскільки для використання RTSJ все ще потрібні структурні зміни у вихідному коді, це обмеження залишається основним перешкодою для впровадження цієї технології.

1.8 Прогнозування пам'яті

Прогноз пам’яті, який тісно пов’язаний з прогнозом GC, є ще однією активною областю досліджень у спільноті GC, яка фокусується на самовдосконаленні JVM, шукаючи способи викликати GC лише тоді, коли це варто. Наприклад, робота, представлена в [33], використовує спостереження про те, що мертві предмети, як правило, об’єднуються в групи, щоб оцінити, скільки місця буде відведено для MaGC, щоб уникнути низькопродуктивних GC. Тим часом підхід, застосований у [62], спрямований на максимізацію переваг GC шляхом профілювання коду в автономному режимі для виявлення сприятливих точок збору, які згодом використовуються під час виконання програми для запуску MaGC (або MiGC), коли співвідношення живих об'єктів низький. Нарешті, автори [140] представляють підхід щоб оцінити кількість загиблих об’єктів у будь-який час, інформація, яку JVM може використовувати для вирішення, коли запускати MaGC.

У всіх цих випадках прогнози пам’яті допомагають визначити, чи сприятливий час (з точки зору потенційного збільшення пам’яті) для виконання GC. Однак лише ці прогнози пам’яті не дають достатньо інформації, щоб знати, коли відбудеться наступний MaGC. На відміну від цього, моя робота спрямована на прогнозування подій MaGC, а також надання цієї інформації за межами JVM, щоб інші учасники (наприклад, балансир навантаження) могли використовувати її та приймати більш обґрунтовані рішення.

1.9 Оптимізація розподілених систем

Дослідження також були зосереджені на оптимізації розподілених архітектур, вдосконалюючи їх з різних точок зору. Наприклад, автори [13] представили метод, що полегшує міграцію монолітного застосунка Java до

розподіленої архітектури за допомогою автоматичного введення залежностей вихідного коду. У випадку [44] в цій роботі був описаний механізм досягнення високої надійності кластерних веб-служб, який базувався на його здатності пропонувати прозору відмовостійкість до різних типів транзакцій. Крім того, в роботі над [81] було запропоновано рішення щодо управління ресурсами для розподілених систем, що пропонує такі можливості, як автоматичне виявлення перевантажених ресурсів.

Завдяки своїй важливості балансування навантаження є добре вивченою проблемою в області паралельних та розподілених систем, де існує значна кількість літератури [65]. Наприклад, автори [37] запропонували методику оцінки загального навантаження на балансир навантаження, щоб використовувати цю інформацію для балансування нового навантаження. Тим часом, у роботі над [59] була запропонована адаптивна стратегія балансування навантаження, яка спрямована на виконання угод про рівень обслуговування (SLA) на основі набору пріоритетів клієнтів. Аналогічним чином, автори [54] представили рішення на основі агента для забезпечення можливостей динамічного збалансування навантаження хмарним службам та ресурсам. Тим часом автори [60] представили схему балансування навантаження, яка враховує статистику минулих страт для розподілу нового робочого навантаження. Нарешті, JVM-куча, потоки та центральний процесор, щоб вирішити, як розподілити навантаження. Подібним чином, робота, представлена в [96], пропонує функцію для обчислення використання Enterprise JavaBean (EJB), а потім використовує цю інформацію для розподілу вхідного навантаження між доступними екземплярами EJB.

На відміну від усіх раніше обговорених робіт, моя дослідницька робота зосереджена на вдосконаленні балансування навантаження шляхом розгляду прогнозів MaGC на рівні прийняття рішень. У такому випадку балансир навантаження може отримати додаткові знання про JVM, щоб

контролювати робоче навантаження системи, на застосунок до інших існуючих політик балансування навантаження, які можуть бути застосовними.

1.10 Моніторинг та управління ефективністю

Дослідження також зосереджувались на вдосконаленні моніторингу ефективності та управління кластерними системами, вдосконалюючи ці процеси з різних перспектив. Наприклад, автори [72] запропонували структуру MonALISA, яка забезпечує можливості моніторингу, управління, контролю та оптимізації розподіленої системи. MonALISA спирається на архітектуру, засновану на агентах, в якій набір агентів, використовуючи інформацію моніторингу в режимі реального часу, співпрацює для виконання підтримуваних завдань. Тим часом у роботі [41] описана NetHAM-nano, яка є платформою для розподіленого моніторингу на основі хмарної архітектури. Обробляючи потреби моніторингу як послуги, NetHAM-nano використовує парадигму публікації-передплати, щоб забезпечити високу гнучкість управління ресурсами (оскільки завдання моніторингу можуть бути призначені будь-яким вузлам, що мають доступну потужність). Подібним чином автори [52] представили Ganglia - розподілену систему моніторингу, спеціально розроблену для високопродуктивних систем. Ganglia покладається на протокол багатоадресного прослуховування / оголошення для моніторингу станів кожного кластера, а також на ієрархічну структуру (у формі федерацій кластерів), що дозволяє аналізувати та представляти свої результати на різних рівнях абстракції.

Більше того, в роботі над [11] була представлена Astrolabe, яка є розподіленою системою управління інформацією, яка здатна збирати великомасштабні стани систем та агрегувати інформацію на ходу. Виходячи зі своїх можливостей, Astrolabe може запропонувати масштабований спосіб

відстежувати стан системи в часі та виконувати самокорекційні дії на основі цієї інформації (наприклад, знаходити ресурси або виконувати розподілену синхронізацію у великій системі). Подібним чином автори [10] запропонували CoMon, який є масштабована система моніторингу для PlanetLab. За допомогою гібридної стратегії пасивного / активного збору набору корисних показників CoMon може спростити процес адміністрування та налагодження своїх експериментів адміністраторами та користувачами, а також виявлення проблем (наприклад, проблемних машин) на глобальному тестовому стенді PlanetLab. Нарешті, робота над [49] представила GridICE, що є службою моніторингу, спеціально розробленою для мережевих систем. Він пропонує можливість мати справу з різними вимірами, які зазвичай потрібні в системі мереж (наприклад, віртуальні організації та сайти), а також просту інтеграцію (за допомогою модульної архітектури) із проміжним програмним забезпеченням, яке зазвичай використовується для моніторингу типової системи мереж.

На відміну від раніше обговорюваних робіт, які були розроблені для сприяння управлінню кластерною системою, запропонована структура розроблена для задоволення конкретних потреб тестувальника при тестуванні продуктивності. Це не тільки ізолює тестер від складності налаштування інструмента діагностики ефективності, але також дозволяє ефективно використовувати такі інструменти в області тестування продуктивності.

1.11 Автоматизація в тестуванні

Велика частина досліджень зосереджена на автоматизації генерації наборів для випробувань навантаження [45, 67]. Наприклад, автори [7] пропонують підхід до автоматизації генерації тестових кейсів на основі

заданих рівнів навантаження та комбінацій ресурсів. Подібно [67] представляє структуру автоматизації, яка відокремлює логіку програми від сценаріїв тестування продуктивності, щоб збільшити повторну використання тестових скриптів. Тим часом [14] представляє структуру, розроблену для автоматизації тестування продуктивності веб-застосунків, яка внутрішньо використовує дві моделі використання для більш реального моделювання поведінки користувачів.

Інші дослідницькі зусилля були зосереджені на автоматизації конкретних методів аналізу. Наприклад, [46] представляє комбінацію методів аналізу охоплення та налагодження для автоматичної ізоляції змін, що викликають несправності. Подібним чином автори [5] розробили методику зменшення кількості помилкових попереджень про витік пам'яті, що генеруються методами статичного аналізу, шляхом автоматичної перевірки та класифікації цих попереджень.

Нарешті, інші дослідники запропонували основи для підтримки різних процесів програмної інженерії. Наприклад, автори [54, 76] представлені фреймворки для моніторингу програмних послуг. Обидва механізми контролюють використання ресурсів та взаємодію компонентів у системі. Один зосереджений на Java [76], а інший - на технологіях Microsoft [54]. На відміну від цих робіт, які були розроблені для надання допомоги в оперативно-допоміжних заходах, запропонована мною структура була розроблена для задоволення конкретних потреб тестувальника у тестуванні продуктивності, ізолюючи тестер від складності використання та налаштування інструмента діагностики ефективності.

Багато дослідницьких зусиль були спрямовані на вдосконалення процесів аналізу ефективності. Основна тенденція досліджень була зосереджена на виявленні помилок ефективності та їх основних причин. Наприклад, робота в [1] пропонує підхід для прогнозування вузьких місць продуктивності, що залежать від навантаження (WDPB), за допомогою

моделей складності, які визначають кількість ітерацій тих потенційних WDPB. Подібним чином, робота в [77] представляє методику виявлення процесів, що здійснюють доступ до спільного ресурсу без належної синхронізації, і які є загальною причиною проблем; в той час як автори [55] проаналізували купи пам'яті кількох реальних об'єктно-орієнтованих програм та надали уявлення щодо вдосконалення методів розподілу пам'яті та аналізу програм.

Високий відсоток запропонованих методів аналізу продуктивності вимагає певного типу приладів. Наприклад, автори в [82] інструментують вихідний код відстежуваних застосунків для видобування послідовностей графіків викликів при нормальній роботі, інформація, яка згодом використовується для виведення будь-яких відповідних моделей помилок. Подібний випадок трапляється з роботами, представленими в [63], які спираються на контрольно-вимірювальні прилади для динамічного виведення інваріантів у застосунках та виявлення помилок програмування; або підхід, запропонований [66], який використовує контрольно-вимірювальні прилади для фіксації шляхів виконання для визначення розподілів нормальних шляхів та пошуку будь-яких значних відхилень для виявлення помилок. У всіх цих випадках контрольно-вимірювальні прилади затьмарюють продуктивність програми під час тестування продуктивності, отже, перешкоджаючи їх використанню.

Крім того, автори [8] представляють непроникливий підхід, який автоматично аналізує журнали виконання тесту навантаження для виявлення проблем продуктивності. Оскільки цей підхід спирається лише на результати випробувань на навантаження, він не може визначити першопричини. Подібний підхід представлений у [7], який має на меті запропонувати інформацію про причини цих проблем. Однак він може забезпечити лише підсистему, відповідальну за відхилення продуктивності. Навпаки, запропонована мною структура дозволяє ефективно використовувати засоби

діагностики продуктивності в області тестування продуктивності за допомогою автоматизації, щоб пришвидшити процес виявлення проблем продуктивності та їх основних причин. Крім того, методи, представлені в [8, 74], вимагають інформації з попередніх циклів для базового аналізу, яка може бути не завжди доступною.

1.12 Постановка задачі

Для досягнення поставленої мети визначені наступні задачі дослідження:

– здійснити аналіз кластерних сиситем побудованих на java, засоби їх діагностики;

– удосконалити метод, що базується на адаптивній системі автоматизації, яки розглядає загальні обмеження використання інструмента діагностики для ефективного використання при тестуванні продуктивності кластерних програм;

– удосконалити метод адаптивного балансування навантаження для збору даних GC;

– здійснити подальший розвиток технології з визначеним набором метрик, орієнтованих на GC, що характеризують поведінку програм Java з точки зору колекцій данних;

– розробити відповідні засоби діагностики та провести експериментальні дослідження.

Висновки до розділу 1

Проведено дослідження предметної області, здійснено аналіз кластерних систем та засобів їх діагностики.

Розділ 2
Проєктування M1_JAVA та дослідження характеристик

2.1 Адаптивна політика та політика аналізу даних

Метою цієї роботи було розробити структуру (M1_JAVA) для автоматизації процесів, задіяних у використанні інструменту діагностики ефективності під час тестування продуктивності кластерного застосунка. Такі рамки покращили б продуктивність тестувальників, а також процес тестування продуктивності, зменшивши зусилля та досвід, необхідні для використання інструменту діагностики. Самоадаптаційна система зазвичай складається з керованої системи та автономного менеджера [8]. У цьому контексті M1_JAVA відіграє роль автономного менеджера. Тому він контролює цикл зворотного зв'язку, який адаптує керовану систему відповідно до набору цілей. Тим часом інструмент діагностики та вузли програми відіграють роль керованих систем.

Як визначено багатьма авторами [15, 14], самоадаптація наділяє систему автономною адаптацією до внутрішніх та зовнішніх змін для досягнення певних цілей якості в умовах невизначеності. У контексті M1_JAVA це означає збалансування різних компромісів, які існують при використанні інструмента діагностики (наприклад, кількість накладних витрат, що вводяться у вузли програми щодо точності помилок, досягнутої інструментом, оскільки обидва фактори залежать про частоту вибірки та обсяг вибіркових даних). Щоб включити самоадаптацію до M1_JAVA, наслідувано відому адаптивну модель MAPE-K [82]. Ця модель була обрана, оскільки вона дозволяє відокремити адаптивний рівень від бізнес-логіки, отже, збільшуючи модульність рішення.

Ключовим аспектом M1_JAVA є його політична база, яка виконує роль елемента «Знання» (в рамках моделі MAPE-K) та визначає пул доступних політик. Інкапсуляція знань у політику дозволяє M1_JAVA легко розширюватись та включати безліч політик, які можуть бути придатними для різних сценаріїв та інструментів діагностики. У цьому контексті політика визначає набір взаємопов’язаних завдань, необхідних для виконання одного із процесів, пов’язаних із використанням інструменту діагностики в рамках

тестового циклу виконання. Кожен інструмент діагностики вимагає трьох політик: політика збору даних (для контролю збору зразків у вузлах програми), політика завантаження (для контролю, коли зразки надсилаються на інструмент діагностики для обробки), і політика консолідації (для інтеграції всіх результатів, отриманих за допомогою діагностичного інструменту для різних вузлів програми). Крім того, інструмент діагностики може мати інші доступні політики (наприклад, резервне копіювання отриманих зразків або активація дій на основі результатів роботи інструменту). Ці політики можуть також використовувати наявний набір помічників аналітики даних (підтримуюча логіка, яка надає різні послуги для подальшої настройки поведінки політики). Наприклад, політика може зосередитись на оцінці серйозності помилок, виявлених інструментом. У цьому прикладі можна визначити декілька помічників, щоб запропонувати різні набори рівнів серйозності для класифікації помилок (оскільки відповідні рівні серйозності можуть змінюватися залежно від сценарію використання). Наприклад, політика завантаження може вимагати часового інтервалу, щоб знати, коли відправляти зразки на обробку, або політика збору даних може використовувати інтервал вибірки, щоб знати частоту збору зразків. Подібним чином, політика консолідації може вимагати в якості вхідних даних топологію кластерної системи, щоб вона могла диференціювати вузли програми, що складають кластер.

M1_JAVA має основний процес, який координує елементи MAPE-K. Він спрацьовує, коли починається тестовий запуск продуктивності. На початковому етапі він отримує новий ідентифікатор контрольного тесту, значення, яке однозначно ідентифікує тестовий запуск та зібрані дані. Це значення поширюється на всі вузли. Далі всі вузли програми запускають (паралельно) цикл, зазначений у моніторі та аналізують фази, поки не завершиться тестовий запуск. Новий набір зразків даних збирається відповідно до політики збору даних. Після завершення збору процес аналізатора перевіряє правила завантаження. Якщо будь-яка політика завантаження була виконана, дані надсилаються в інструмент діагностики (маркування даних за допомогою ідентифікатора контрольного тесту, щоб інформація з різних вузлів могла бути ідентифікована як частина одного і

того ж запуску тесту). Так само, оновлені результати отримуються з інструменту діагностики для консолідації. Додаткові політики також можуть бути виконані залежно від вхідної конфігурації користувача. Наприклад, як можуть деякі колекції даних бути дорого (наприклад, створення єдиного дампа пам'яті в Java може зайняти хвилини і зажадати сотні мегабайт жорсткого диска), політика резервного копіювання може бути використана для подальшого аналізу зібраних даних у режимі офлайн. Цей основний процес продовжується ітеративно, поки не завершиться тестовий запуск продуктивності (або не буде виконана альтернативна умова виходу, визначена політикою). Коли це трапляється, усі застосовні політики оцінюються останнє до завершення процесу. Крім того, будь-які винятки обробляються та повідомляються внутрішньо.

M1_JAVA реалізовано з мультиагентною архітектурою як M1_JAVA складається з трьох типів агентів. Агент управління відповідає за взаємодію з інструментом для перевірки навантаження, щоб знати, коли тест починається і закінчується. Він також відповідає за оцінку політики та розповсюдження рішень на інші вузли. Тим часом агент вузла програми відповідає за виконання необхідних завдань у кожному вузлі програми (наприклад, збір вибірки або надсилання зібраних зразків на інструмент діагностики). Нарешті, засіб діагностики відповідає за взаємодію з інструментом діагностування (наприклад, його подача або подальша обробка його згенерованих звітів).

Внутрішньо кожен агент складається з різних компонентів. Агент має три основні компоненти: загальний компонент містить логіку управління та всю допоміжну функціональність, яка не залежить від цільової діагностики та тестування навантаження інструменти (наприклад, завдання аналізу та планування політики). Що стосується логіки, яка взаємодіє з цільовими інструментами, вона повинна бути налаштована для кожного інструменту. Тому ця логіка інкапсульована у відповідні компоненти, щоб мінімізувати необхідні зміни коду. Для доповнення цієї стратегії дизайну доступ до компонентів здійснюється лише через інтерфейси. Структура на високому рівні компонента діагностичного інструменту. Він містить основний інтерфейс IDiagnosisTool, щоб виставити всі

необхідні дії та абстрактний клас для всіх загальних функціональних можливостей. Потім цю ієрархію можна розширити для підтримки конкретних інструментів діагностики (наприклад, WAIT) для різних операційних систем. Внутрішньо автоматично вибирається необхідний клас (підтримує певний інструмент і, можливо, конкретну операційну систему).

Нарешті, агенти спілкуються за допомогою команд, дотримуючись шаблону проектування команди [7]. Агент управління викликає команди, тоді як інші агенти реалізують логіку, відповідальну за виконання кожної конкретної команди. Як тільки тестер розпочав тестовий запуск продуктивності (крок 1), агент управління розповсюджує цю дію на всі агенти вузла програми (кроки 2-4). Потім кожен агент вузла програми виконує свої періодичні завдання (кроки з 5 по 9), поки не буде виконана будь-яка з налаштованих політик завантаження, а дані надіслані до інструменту діагностики для обробки (кроки 10 та 11). Ці кроки продовжуються ітеративно, поки тест не закінчиться. У цей момент контрольний агент поширює дію зупинки (кроки 21, 22 та 24). У будь-який час тестувальник може вибрати перегляд проміжних результатів інструменту діагностики (кроки 12-14) до отримання остаточних результатів (кроки 25-27).

2.2 Політика збору даних щодо точності та цільових показників

Ця адаптивна політика була розроблена, щоб збалансувати компроміс між достовірністю результатів інструменту діагностики та накладними витратами, введеними в тестоване застосунок. Це пояснюється тим, що обидва фактори впливають на вибір інтервалу вибірки (SI). Ця політика вимагає двох введень користувача. Поріг часу відгуку, який є максимально допустимим впливом на час відгуку (виражений у відсотках) і період розминки. Подібно до його використання в тестуванні продуктивності [34], період розминки - це час, після якого всі транзакції були виконані принаймні один раз (отже, це сприяє середньому часу відгуку тестового запуску). Два

додаткові параметри отримуються з бази знань, оскільки їх значення специфічні для кожного інструменту діагностики. Мінімальний SI, який слід використовувати для збору та ΔSI, що вказує на те, наскільки повинен змінитися SI у разі коригування. Процес розпочинається з очікування налаштованого періоду розігріву. Потім він отримує середній час відгуку з інструменту тестування навантаження. Це значення стає базовим часом відгуку. Після цього процес ініціалізує вузли програми з мінімальним SI. Ця стратегія дозволяє збирати якомога більше зразків, якщо продуктивність не погіршується нижче бажаного порогу, порушуючи тим самим прийнятну угоду про рівень обслуговування (SLA). Далі починається процес ітераційного моніторингу (який триває до завершення тестування продуктивності): Спочатку процес очікує поточного SI (оскільки ніякого впливу на продуктивність, спричиненого інструментом діагностики, може не відбутися, поки не відбудеться збір даних). Потім отримується новий час і порівнюється з часом відгуку, щоб перевірити, чи не перевищено порогове значення. Якщо так, це означає, що поточний SI занадто малий, щоб утримувати накладні витрати нижче встановленого порогу. У цьому випадку SI збільшується на значення, налаштоване як ΔSI. Нарешті, новий SI поширюється на всі вузли програми, які починають використовувати його з наступної ітерації збору даних.

Ця політика аналогічна сценарію, де група випробувачів прагне мінімізувати кількість необхідних тестових запусків через обмеження бюджету або графіку. Цього можна досягти, дозволивши певний рівень відомих накладних витрат у тестованій програмі, щоб ідентифікувати якомога більше помилок продуктивності, на застосунок до звичайних результатів, отриманих під час тестового запуску.

2.3 Політика ефективності та цільового завантаження

Ця адаптивна політика була розроблена для збалансування компромісу між рівнями використання ресурсів, необхідних інструменту діагностики, та кількістю вибіркових даних, які одночасно обробляються інструментом. Це пов’язано з тим, що на обидва фактори впливає вибір

інтервалу завантаження (UI).

Кожен крок зіставляється з відповідним елементом MAPE-K. Це вимагає двох входів користувача. Використовувати початковий інтерфейс; та ΔUI, який вказує на те, наскільки UI повинен змінюватися, коли потрібне коригування. Додатковий параметр буде отримано з бази знань (оскільки його значення специфічне для кожного типу ресурсу): Ціль максимального використання (UMAX).

Процес розпочинається з ініціалізації вузлів програми початковим інтерфейсом користувача. Потім процес чекає, поки всі вузли програми не завантажать свої зразки один раз. Цей крок робиться для того, щоб переконатися, що інтерфейс користувача змінюється лише за потреби. Після того, як усі вузли завантажили свої результати, процес отримує середнє використання ресурсів (RESAV tt) спільної служби (наприклад, сервер WAIT) під час обробки цих зразків, а також середню тривалість використання ресурсів (DRESAV tt). Потім RESAV tt порівнюється з UMAX. Якщо UMAX перевищено, обчислюються нові інтерфейси. В якості першої стратегії для кожного вузла розраховується різний інтерфейс, щоб запобігти завантаженню всіх вузлів своїх результатів одночасно. Щоб максимально поважати поточний інтерфейс користувача, DRESAV tt від поточного інтерфейсу, поки всі вузли не мають різного інтерфейсу). Наприклад, якщо ми маємо 5 вузлів, поточний інтерфейс 30 хвилин і DRESAV tt 1 хвилину, нові інтерфейси вузлів будуть розподілятися як 28, 29, 30, 31 і 32. Нарешті, нові інтерфейси розповсюджуються до відповідних вузлів програми, які починають їх використовувати з наступної ітерації завантаження. Якщо потрібно подальше коригування (це означає, що лише розділення інтерфейсу було недостатньо, щоб RESAV tt було нижче UMAX), поточний інтерфейс зменшується (на значення, налаштоване як ΔUI), перед тим, як обчислювати нові інтерфейси. зроблено. Це робиться для зменшення кількості відправлених зразків (на вузол) при кожному завантаженні. Ця політика орієнтована на сценарій, де кількість доступних ліцензій на певний інструмент обмежена. У цьому сценарії надзвичайно бажана можливість ефективного розподілу наявних інструментів між різними групами та проектами, щоб максимізувати повернення інвестицій [37]

інструменту.

2.4 Політика консолідації Multi-View

Ця політика була розроблена, щоб мінімізувати зусилля та досвід, необхідні тестеру для аналізу результатів використання діагностичного інструменту в кластерній програмі. Це робиться шляхом надання чотирьох різних поглядів на результати: Основний погляд - це зведений вигляд на рівні системи. Це дозволяє тестувальнику легко відстежувати хід проведеної діагностики в загальному кластері та перевіряти, чи не сталася якась відповідна проблема системного рівня. Також доступні три додаткові подання: Індивідуальний системний рівень дозволяє побачити результати, отримані в результаті окремого циклу обробки (як контролюється користувацьким інтерфейсом). Крім того, консолідовані подання на рівні вузла та окремі рівні на рівні вузла ведуть себе подібно до своїх аналогів на рівні системи, але фокусуючись на конкретному вузлі. Разом цей набір поглядів пропонує тестувальник різних рівнів деталізації,

Процес розпочинається із створення набору порожніх звітів. Далі процес чекає, поки засіб діагностики не сформує новий набір результатів. Як тільки це відбувається, відповідний звіт на рівні вузла оновлюється. Цей крок включає аналіз усіх отриманих результатів (оскільки один процес завантаження може генерувати кілька звітів про вихід) з метою отримання відповідної кваліфікаційної інформації (тобто інформації про виявлені проблеми, включаючи їх категорії та підкатегорії). Внутрішній аналіз спирається на набір правил, визначених у базі знань (оскільки вони різняться залежно від інструментів діагностики). Після вилучення нові випуски включаються до звітів на рівні вузлів: створюється новий звіт на індивідуальному рівні шляхом консолідації всіх результатів інструментів. Подібним чином зведений рівень звіту вузла оновлюється новими випусками. Нещодавно додана інформація також позначається тегами, так що наступні завдання (тобто оцінка подібності та серйозності) можуть легко її ідентифікувати. Крім того, відстежуються оновлення на вузол. Це робиться для того, щоб після отримання результатів з усіх вузлів отримували звіти на рівні системи також оновлений (дотримуючись логіки, подібної до звітів на

рівні вузлів, про які раніше йшлося).

Після оновлення звітів оцінюється схожість нових результатів. Це робиться для подальшої консолідації результатів (оскільки результати діагностичного інструменту, як правило, перекриваються при моніторингу декількох вузлів протягом тривалого періоду часу). Правила для оцінки подібності отримуються з бази даних знань (оскільки вони залежатимуть від інструменту діагностики). Набір стандартних статистичних показників (тобто середнє, стандартне відхилення та коефіцієнт варіації [116]) також розраховується для поступового побудови історичної тенденції для кожного типу звіту. Далі проводиться оцінка тяжкості. Це робиться з метою допомогти тестувальнику зосередитись на наборі питань ефективності, які варто вивчити (оскільки інструмент діагностики може дати значну кількість результатів, і, як правило, справді актуальним є лише підмножина). Як початковий крок, ступінь тяжкості нових результатів обчислюється, дотримуючись правил, що застосовуються до інструменту діагностики (наприклад, на основі частоти виявлених проблем). Далі результати класифікуються відповідно до стилю тяжкості та порогових значень, налаштованих тестувальником. Нарешті, стандартна статистика (подібна до попередньо обговореної) обчислюється за типом тяжкості. Після цього нова версія зведених звітів готова та доступна для тестувальників.

Додатковим (і необов’язковим) етапом у цьому процесі є оцінка оцінки “не ходи”. Метою цього кроку є запропонувати альтернативний критерій виходу (крім тривалості тестового циклу) тестового запуску. У разі, якщо тестувальник увімкнув цю опцію, буде проведена відповідна оцінка "Не ходи". Якщо його критерії виходу виконані, буде спрацьовано зупинка. Ця дія поводиться подібно до команди stop, описаної в розділі 3.3.

Ця політика орієнтована на сценарій, коли тестувальник повинен контролювати кластеризовану програму (як правило, складається з декількох вузлів застосунків) протягом тривалих періодів часу (можливо, один або кілька днів). За цих умов обсяг вихідних даних, що генеруються інструментом діагностики, може бути величезним, що легко перевантажує тестер через значну кількість зусиль та досвіду, необхідних для консолідації та аналізу отриманих результатів.

Оцінка подібності визначає логіку, яка використовується для ідентифікації тих проблем ефективності, про які повідомляє інструмент діагностики, і які слід об'єднати, оскільки їх симптоматика свідчить про те, що вони є екземплярами однієї проблеми. Цей тип допоміжної логіки фіксується як помічник аналітики даних у M1_JAVA.

На основі результатів оцінки компромісів, а також після аналізу результатів досліджуваних інструментів діагностики, розглянемо здійснення наступних двох оцінок подібності. Оцінка рівності, застосовна до тих інструментів, які генерують виключно якісні описи проблем (наприклад, IBM WAIT). У цьому випадку опис випусків можна безпосередньо порівняти. Оцінка семантичної подібності, застосовна до тих інструментів, які породжують якісні описи проблем із деякими включеними кількісними даними (наприклад, IBM Health Center). У цьому випадку переважно порівнювати описи питань з точки зору їх семантичної подібності. Для цієї роботи було обрано відстань JaroWinkler [144] (спочатку розроблену в галузі зв'язку записів для виявлення дублікатів рядків). Це пов'язано з тим, що цей показник широко використовується в літературі, і він також пропонує нормалізований бал (де 0 означає відсутність подібності, а 1 означає точну відповідність). Оскільки, як правило, змінюється лише кількісна інформація опису випуску (серед випадків одного випуску), порогу несхожості 0,1 було достатньо для виявлення подібних проблем.

Стиль суворості визначає набір категорій важкості, за якими може бути класифікована проблема продуктивності. Цей тип допоміжної логіки фіксується як помічник аналітики даних у M1_JAVA. Крім того, стиль оцінки може бути використаний в рамках оцінки, щоб налаштувати її поведінку. Класифікація проблеми ефективності в межах категорії важкості базується на частоті проблеми (як визначено для інструменту діагностики) щодо порогових значень, налаштованих для відповідних категорій тяжкості. Ця класифікація проводиться незалежно від обраного стилю суворості.

На підставі результатів, отриманих в результаті оцінки компромісів (де було помічено, що кількість проблем, виявлених інструментом діагностики, як правило, стабілізується з часом), оцінка "Відмовитись" на основі коефіцієнта варіації (CV) [36] була розроблена. Резюме було обрано,

оскільки ця статистична метрика дозволяє безрозмірно вимірювати мінливість у кількості виявлених проблем. Наприклад, якщо резюме 0,1 отримано у наборі різних версій звітів, це значення вказує на практично стабільну кількість виявлених проблем серед звітів (незалежно від фактичної кількості помилок або кількості звірів, що порівнюються). Отже, цей показник може охоплювати, коли процес ідентифікації помилок вичерпався, даючи нові результати. Як тільки ця точка досягнута, перевірку продуктивності можна зупинити, щоб запобігти марній витраті цінних людських та обчислювальних ресурсів. Ця оцінка Go No-Go вимагає трьох введень користувача. Оцінювані категорії тяжкості, що визначає підмножину категорій тяжкості (у межах обраного стилю тяжкості), які будуть оцінюватися (оскільки не всі категорії тяжкості можуть представляти інтерес - наприклад, некритичні або косметичні проблеми). Кількість зведених звітів на системному рівні для оцінки, що дозволить обмежити кількість версій цього звіту (починаючи з останньої), які будуть оцінені. Побічно цей параметр також впливає на мінімальну тривалість тесту (оскільки перед першим оцінюванням має бути достатньо історичних даних). Набір порогових значень CV (по одному для кожної категорії тяжкості для оцінки), який буде використаний при оцінці, щоб визначити, чи розрахований CV входить в допустимий діапазон для певної категорії тяжкості.

Від з точки зору процесу, оцінка починається з перевірки наявності достатньої кількості історичної інформації (у формі зведених звітів на рівні системи) для обчислення необхідних значень CV. Якщо це не так, оцінка не вдається. В іншому випадку CV розраховується для кожної категорії тяжкості. Потім отримані значення порівнюють із відповідними пороговими значеннями. Тільки якщо значення CV нижче (або дорівнює) відповідному порогу для всіх категорій тяжкості, оцінка вважається виконаною.

2.5 Експериментальне оцінювання

У цьому розділі представлені експерименти, проведені для оцінки M1_JAVA. Щоб зрозуміти, яка політика буде найкращою, я почав із оцінки виявлених компромісів. У ньому брали участь три експерименти. По-перше, я оцінив точність кожного інструменту діагностики щодо накладних витрат,

що вводяться процесами вибірки даних, що подають інструмент. По-друге, було оцінено зусилля, необхідні тестеру, щоб проаналізувати результати кожного інструменту діагностики щодо кількості результатів, створених інструментом. По-третє, оцінено ресурси, необхідні кожному інструменту діагностики, з урахуванням кількості зразків, оброблених інструментом.

Після розробки запропонованих політик (описаних у розділі 3.4), було проведено два додаткові експерименти для оцінки переваг та витрат від використання M1_JAVA: По-перше, я оцінив точність реалізованої політики. Тут метою було оцінити потенційний компроміс між точністю результатів, отриманих інструментом діагностики, та накладними витратами, що вводяться у вузли програми процесами вибірки даних, що подають інструмент. У наступних параграфах представлено розроблений прототип, тестове середовище та параметри, що визначали оцінені експериментальні конфігурації: Вибрані інструменти діагностики, тести Java, вибіркові інтервали (SI) та інтервали завантаження (UI).

Прототип був розроблений спільно з нашим промисловим партнером. Агент управління був реалізований поверх Apache JMeter 2.9 [2], який є провідним інструментом з відкритим кодом, що використовується для програмиперевірка якості. Тим часом Агент вузла застосунків та Агент діагностики були реалізовані як окремі програми Java. Внутрішньо кожен агент має вбудований контейнер веб-сервлетів Jetty [15] (популярне рішення з відкритим кодом, що використовується для забезпечення можливості зв'язку машин). Це дозволяє різним агентам спілкуватися через HTTP-запити.

Крім того, було реалізовано дві початкові політики. Політика збору даних, яка використовує постійну CI під час повного виконання тесту; і політика завантаження, яка використовує постійний інтерфейс. Оскільки SI контролює частоту збору зразків із контрольованої програми (що є основною потенційною причиною накладних витрат, представлених інструментом діагностики), було перевірено широкий діапазон значень (0,125, 0,25, 0,5, 1, 2, 4, 8 і 16 хвилин). Найменше значення в діапазоні (0,125 хвилин) було навмисно обрано меншим ніж мінімальне рекомендоване значення для обраних засобів діагностики (0,5 хвилини). Подібним чином, найбільшим значенням в діапазоні (16 хвилин) було обрано більше 8 хвилин (CI, який зазвичай використовується в галузі). Нарешті, оскільки користувальницький

інтерфейс не бере участь у процесі збору даних, було використано постійне значення 30 хвилин.

Всі експерименти проводились в ізольованому тестовому середовищі, так що контролювалося все навантаження. Це середовище складалося з восьми віртуальних машин: кластера з п’яти вузлів застосунків з одним балансиром навантаження, одним сервером інструмента діагностики та одним вузлом тестера навантаження. Усі віртуальні машини мали такі характеристики: 4 віртуальні процесори на частоті 2,20 ГГц, 3 ГБ оперативної пам'яті та 50 ГБ HD; під управлінням Linux Ubuntu 12.04L 64-розрядної версії та OpenJDK JVM 7u25-2.3.10 з купою 1,6 Гб. Вузол тестування навантаження також використовував Apache JMeter 2.9 [2] (провідний інструмент з відкритим кодом, що використовується для тестування продуктивності застосунків), а на вузлах застосунків працював Apache Tomcat 6.0.35 [3] (популярний сервер веб-застосунків з відкритим кодом для Java). Віртуальні машини розташовувались на сервері Dell PowerEdge T420 [12], оснащеному 2-ма процесорами Intel Xeon з частотою 2,20 ГГц (12 ядер / 24 потоки), під управлінням Linux Ubuntu 12.04L 64-біт, 96 ГБ оперативної пам'яті, 2 ТБ HD і з KVM [32] для віртуалізації.

Було використано п’ять інструментів діагностики продуктивності, обговорені в розділі 2.1.9: Eclipse Memory Analyzer (EMAT), IBM Garbage Collection Lite (GCLITE), IBM Garbage Collection and Memory Visualiser (GCMV), IBM Health Center (HC) та IBM Whole Analysis Час очікування (ЗАЧЕКАЙТЕ). Це рішення було прийнято для диверсифікації більш оціненої поведінки.

Тест DaCapo 9.12 [40] був обраний набором програм, оскільки він пропонує широкий спектр різних способів поведінки застосунків для тестування. Для виклику програм Dacapo з тестового скрипту було розроблено JSP-обгортку та встановлено в екземплярі Tomcat кожного вузла програми. Це дозволило виконувати будь-яку програму DaCapo через вхідний параметр. Кожен окремий виклик програми вважався транзакцією. Нарешті, для вибору більш реалістичних умов тесту було обрано 24-годинну тривалість тесту.

Тестувальники-учасники. Усі вони мали ступінь бакалавра з інформатики та мали професійний досвід (у розробці та тестуванні програмного забезпечення) понад 10 років. Дотримуючись порогового рівня досвіду, використаного в інших роботах [32], учасників вважали досвідченими тестувальниками. Крім того, вони мали попередній практичний досвід використання залучених інструментів діагностики (отже, не було кривої навчання, яка могла б вплинути на результати). Подібним чином пояснювалось використання M1_JAVA перед початком експерименту.

Критерії оцінки. Що стосується продуктивності, основними показниками були пропускна здатність в секунду (tps) і час відгуку (мс). Що стосується часу відгуку, то нижчі значення кращі; тоді як для пропускної здатності вищі значення кращі. Ці показники були зібрані за допомогою JMeter. З точки зору продуктивності тестування, основними показниками були кількість виявлених помилок та кількість знайдених критичних помилок. В обох випадках вищі значення кращі. Ці показники були отримані із звітів, сформованих використовуваними інструментами діагностики.

Для всіх інструментів на основі вибірки (тобто WAIT, HC, EMAT), отримані результати показали, що існує чітка залежність між вибором SI та вартістю продуктивності використання інструментів. Ці наслідки для продуктивності в основному спричинені залученими процесами генерації вибірки. Наприклад, у випадку з WAIT, генерація Javacore включає тимчасово призупиняючи виконання прикладних процесів, що працюють у JVM [20]. Незважаючи на те, що витрати були мінімальними при використанні вищих CI, вони поступово ставали помітними (особливо при використанні CI менше 0,5 хвилин). Навпаки, кількість виявлених помилок збільшується, коли CI зменшується. Цей позитивний вплив є прямим наслідком подачі більшої кількості зразків на інструмент діагностики, який змушений зробити більш детальний аналіз моніторингової програми. Варто зазначити, що найбільші наслідки для продуктивності зазнав EMAT. Це пояснюється тим, що для цього потрібно генерувати heapdumps, процес,

який, як відомо, займає багато часу і який може мати серйозний вплив на продуктивність програми, що контролюється [19].

Для інструментів на основі слідів (тобто GCLITE та GCMV), отримані результати показали, що не існує взаємозв'язку між вибором SI та вартістю продуктивності використання цих інструментів ‘Там можна помітити, як різниця в продуктивності була мінімальною, відносно постійною і незалежною від використовуваної CI. Це пояснюється тим, що кількість сформованого багатослівного GC, залежить лише від виконуваної функціональності програми.

Проведено другий раунд аналізу, зосереджений на найбільш критичних питаннях, визначених інструментами на основі вибірки. Метою було оцінити, чи спостерігались там також описані раніше способи поведінки (щодо існуючого компромісу між вибором CI та точністю результатів інструментів). Як показано на рисунку 2.1, спостерігалася подібна поведінка, що підтверджує актуальність компромісу.

Додатковим спостереженням цього експерименту було те, що кількість виявлених некритичних помилок була значно вищою, ніж кількість критичних помилок. Це пояснювалось тим, що засоби діагностики, як правило, повідомляли про потенційні проблеми з продуктивністю, навіть коли їх частота була дуже низькою. Цей сценарій частіше трапляється при використанні невеликої CI (наприклад, 0,5 хвилини або менше) або інструменту діагностики, який аналізує зразки окремо (наприклад, HC та EMAT). Наприклад, більшість некритичних помилок, про які повідомляв WAIT, мали частоту нижче 1%, що означає, що дуже ймовірно, що підозрювані помилки були лише нормальною логікою, що обробляється. Ця поведінка наочно зображується. Так само кількість нових виявлених помилок зменшилась під час виконання тестового запуску. Це було пов’язано з тим, що більшість помилок, виявлених на пізніх етапах тестового запуску, були лише екземплярами попередньо виявлених критичних помилок. Така поведінка припускає, що тестовий запуск вже вичерпав свої переваги (з точки зору знайдених помилок) у якийсь момент під час його виконання.

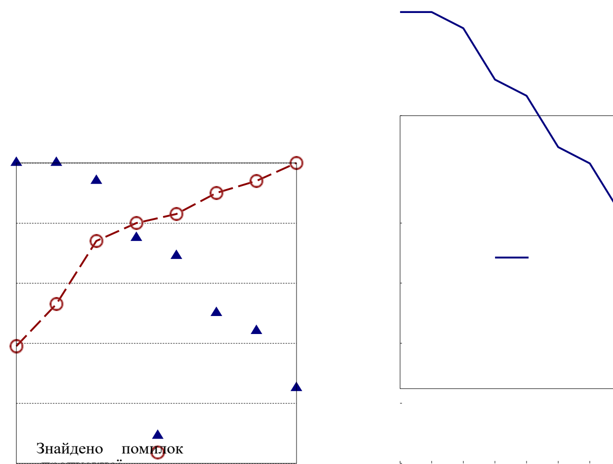


Рисунок 2.1. Критичні помилки проти пропускної здатності

Результати показали, як вибір CI впливає на накладні витрати, які інструменти, що базуються на вибірці, вводять у моніторингову програму. Це зробило так, що автоматичний вибір параметра SI був визначений як відповідна політика в рамках M1_JAVA. Для інструментів діагностики на основі слідів (які не чутливі до вибору CI) – константа SI може бути більш підходящим. Також було помічено, що кількість некритичних помилок (рис. 2.2), як правило, значно більша за критичні, а кількість нових виявлених помилок має тенденцію до зменшення під час виконання тестового запуску.

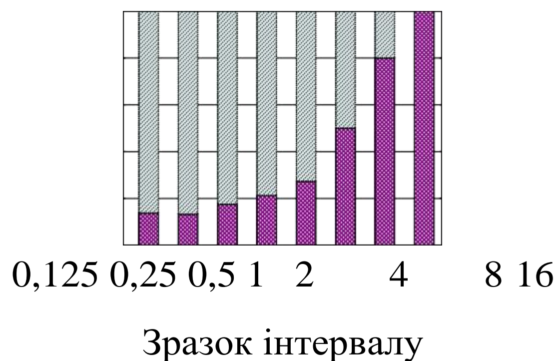


Рисунок 2.2. Поширення помилок

В другому експерименті було оцінено компромісні зусилля. Тут метою було оцінити зусилля, необхідні для аналізу результатів роботи обраних інструментів діагностики, а також зрозуміти причини цього.

Метою було оцінити потенційні зусилля, яких може досягти M1_JAVA. У наступних розділах описується цей експеримент та його результати.

Ця оцінка повторно використала деякі результати, отримані в результаті експерименту №1. В першу чергу, це звіти, створені інструментами діагностики, та зусилля, вкладені тестувальниками в аналіз звітів. Через величезну кількість звітів (понад 24000 на комбінацію інструмент / CI), створених за допомогою інструментів діагностики з невеликим CI (тобто 0,125 або 0,25 хвилин), було неможливо зробити ручний аналіз для цих експериментальних конфігурацій (через обмежену доступність тестерів). У цих випадках зусилля були оцінені. Для екстраполяції цих зусиль були використані середні зусилля (за звіт), отримані з інших експериментальних конфігурацій (тобто тих, що використовують CI вище 0,5 хвилин). Винятком було, оскільки цей інструмент не давав величезної кількості звітів з жодним SI. У цьому експерименті аналіз зосереджувався на оцінці зусиль, вкладених в аналіз результатів роботи інструментів діагностики. Метою було визначити базову лінію, з якою можна порівняти M1_JAVA.

Зусилля з аналізу. З якісної точки зору, цей експеримент дозволив розглянути процес, за яким зазвичай стежить тестер, щоб проаналізувати результати роботи діагностичного інструменту: після збору та обробки зразків, тестери ітеративно переглядали результати нових звітів, потім виявляли / об’єднували будь-які помилки, які були зразками раніше виявлених помилок, оцінювали їх важкість та шукали будь-яку відповідну помилку тенденції. З кількісної точки зору загальні результати показали, що аналіз звітів, сформованих за допомогою діагностичного інструменту, зазвичай є трудомістким процесом і є значні потенційні вигоди (з точки зору економії зусиль). Можна помітити, як зусилля з аналізу, як правило, значно збільшуються при використанні менших CI (наприклад, 0,125). Це відображає поведінку інструментів: Усі інструменти діагностики (крім WAIT) генерують один звіт на оброблену вибірку. Отже, кількість звітів безпосередньо пов’язана з обраною CI. На навпаки, генерує один звіт за цикл обробки (тобто інтерфейс користувача), незалежно від вибіркових даних. Варто зазначити, що навіть у цьому відносно контрольованому /

стабільному сценарії (де WAIT створював 2 звіти на вузол / годину), зусилля, що вимагалися, не було незначним (близько 20 годин). Як обговорювалося в результатах, отриманих в експерименті №1, засоби діагностики на основі слідів (наприклад, GCLITE та GCMA) не отримують вигоди (від ефективності пошуку помилок) від використання невеликого CI (наприклад, 0,125 хвилини). Подібність, з точки зору зусиль, краще використовувати великий CI (наприклад, 8 хвилин), оскільки це зменшує кількість звітів для аналізу, не втрачаючи точності (з точки зору виявлених помилок).

Наступний раунд аналізу був зосереджений на розумінні основних факторів, що обумовлювали кількість зусиль, необхідних для аналізу результатів, отриманих інструментами діагностики. На початковому етапі я проаналізував різницю в складності звітів, сформованих інструментами. Цей якісний аналіз показав, що звіти, створені EMAT, HC та WAIT, мають однакову складність (через їх структуру та обсяг представленої інформації), що вимагає аналогічної кількості часу для перегляду. Навпаки, звіти, створені GCLITE та GCMA, зайняли значно менше часу через відносно вузький обсяг інструментів (тобто проблеми з GC). Незважаючи на те, що спостережувані відмінності в зусиллях можуть бути суб'єктивними для досвіду тестувальника, основним зауваженням цього аналізу було те, що складність інструменту була другорядним фактором щодо загальних зусиль, необхідних для аналізу. Це пов’язано з тим, що основним фактором, що впливає на кількість зусиль, необхідних для проведення аналізу, була кількість створених звітів. Подібним чином це було зумовлено кількома факторами: по-перше, поведінкою інструменту (чи створює він звіт за зразком - тобто SI - або за цикл обробки - тобто UI -). Крім того, кількість звітів безпосередньо пов'язана з кількістю вузлів програми та тривалістю тесту продуктивності. Тобто, чим більша кількість вузлів застосунків (або чим довший тест), тим більша кількість генеруються звітів.

Результати цього експерименту показали, наскільки потенційні заощадження зусиль, на які може звернутися M1_JAVA (особливо враховуючи досвід залучених тестувальників та відносний скромний розмір

кластера - 5 вузлів -). Це пов’язано з тим, що аналіз звітів, сформованих інструментом діагностики, займає багато часу. Незважаючи на те, що складність звітів, сформованих кожним інструментом, може відрізнятися, зусилля, задіяні в аналізі, в основному визначаються кількістю звітів. Оскільки кількість звітів безпосередньо пов’язана з тривалістю тесту та кількістю вузлів застосунків у відстежуваному середовищі, потенційні вигоди будуть досить високими при тривалих прогонах, які є загальними при тестуванні продуктивності та зазвичай тривають кілька днів.

Метою наступного експерименту було оцінити потенційний компроміс між кількістю зразків, одночасно оброблених інструментом діагностики, та кількістю ресурсів, необхідних для обробки зразків. Налаштування було подібним до того, що було використано в експерименті №1, з двома відмінностями: По-перше, оскільки і SI, і UI впливають на кількість зразків, які надсилаються на діагностичний інструмент для обробки, діапазон був обраний для кожного параметра. Для СІ використовувались наступні 3 значення: 0,5, 4 та 8 хвилин. Для інтерфейсу користувача використовувались наступні 3 значення: 5, 30 та 60 хвилин. По-друге, основними показниками були використання процесора (%) та пам'яті (%) у вузлі діагностичного інструменту під час обробки зразків. Ці показники були зібрані за допомогою команди “top”.
Загалом, результати показали, як використання ресурсів у вузлі діагностичного інструменту пов’язане з кількістю паралельно оброблених зразків; що є функцією як SI, так і UI. Наприклад, експериментальні конфігурації (на інструмент діагностики), які використовували SI 4 хвилини та UI 30 хвилин, повідомляли про подібне використання ресурсів, ніж конфігурація, яка використовувала SI 8 хвилин та UI 60 хвилин. Це пояснюється тим, що обидві комбінації подавали однакову кількість зразків (за одну ітерацію завантаження) на інструмент діагностики. Незважаючи на те, що використання центрального процесора та пам'яті демонструвало схожі тенденції (у тому сенсі, що вони мали тенденцію до зростання щодо оброблених зразків), кожен інструмент діагностики мав різну поведінку процесора / пам'яті: наприклад, WAIT виявився значно більшим CPU- інтенсивний (з його CPUAV tt, що перевищує 90%

використання у більшості перевірених конфігурацій). Навпаки, функція WAIT мала значно меншу пам’ять (з найвищим значенням MEMAV tt менше 5% у всіх тестованих конфігураціях). Нарешті, варто зауважити, як кілька експериментальних конфігурацій перевищили 90% використання центрального процесора та / або пам'яті (ціль використання часто пропонується для того, щоб зберегти деяку невикористану ємність і забезпечити м'яку впевненість у якості обслуговування [45]). , незважаючи на відносний скромний розмір кластера (5 вузлів).

Проведені тести продемонстрували, як вибір користувацького інтерфейсу впливає на використання ресурсів в інструменті діагностики. Незважаючи на те, що кожен діагностичний інструмент мав різні рівні інтенсивності процесора / пам'яті, отримані результати дозволили визначити, що автоматичний вибір параметра UI був визначений як інша відповідна політика в M1_JAVA.

Метою цього експерименту було оцінити поведінку M1_JAVA, а також набір запропонованих політик, щоб оцінити, наскільки добре вони виконали свою мету щодо вирішення виявлених компромісів без необхідності ручного втручання з боку тестера. Налаштування було подібним до того, що було використано в першому експерименті, з наступними відмінностями. По-перше, адаптивні політики замінили ручну онфігурацію параметрів SI та UI. Крім того, політики використовували такі конфігурації: Для політики точності було визначено поріг часу відгуку 20%. Це значення було запропоновано IBM для відображення реальних умов. Крім того, виявилося, що періоду розминки 5 хвилин достатньо для того, щоб усі тестові транзакції були виконані принаймні однією. Нарешті, мінімальний SI та ΔSI були встановлені на 30 секунд. Що стосується політики ефективності, початковий інтерфейс користувача був встановлений на 60 хвилин (діапазон часу, який зазвичай використовується в галузі для моніторингу тестових запусків); і IUI було встановлено на 15 хвилин. Нарешті, порогові значення максимального використання центрального процесора та пам’яті були встановлені на 90%, щоб уникнути насичення цих ресурсів (сценарій, якого слід уникати для оптимальної роботи. Що стосується політики консолідації, то дворівневий

стиль суворості був використаний. Цей параметр запропонував IBM для спрощення класифікації помилок продуктивності (між критичними та некритичними), оскільки, як правило, лише критичні цікавлять тестування продуктивності. Поріг тяжкості був встановлений на частоту 50% (щоб критична категорія концептуально перекривала критичну та основні категорії ISTQB). Врешті-решт, було включено оцінку CV No-Go. Він був налаштований на оцінку результатів останніх двох годин для всіх категорій тяжкості. Для порогового значення CV встановлено значення Поріг тяжкості був встановлений на частоті 50% (щоб критична категорія концептуально перекривала критичну та основні категорії ISTQB). Врешті-решт, було включено оцінку «без ходу» на основі резюме. Він був налаштований на оцінку результатів останніх двох годин для всіх категорій тяжкості. Для порогового значення CV встановлено значення Поріг тяжкості був встановлений на частоті 50% (щоб критична категорія концептуально перекривала критичну та основні категорії ISTQB). Врешті-решт, було включено оцінку «без ходу» на основі резюме (описану в розділі 3.5.3). Він був налаштований на оцінку результатів останніх двох годин для всіх категорій тяжкості. Для порогового значення CV встановлено значення 0,1 для критичної категорії, тоді як 0,3 для некритичної. У цьому експерименті аналіз зосередився на двох основних аспектах: оцінці точності реалізованої політики та оцінці приросту продуктивності, який M1_JAVA приніс у процес тестування продуктивності. Отже, отримані результати порівнювали з результатами попередньо проведених оцінок компромісів.

Політика збору даних щодо точності та цільових показників. Перша частина аналізу була зосереджена на оцінці політики точності. Щодо накладних витрат на результати, результати продемонстрували, що політика точності працювала добре, оскільки можна було закінчити тест із накладними витратами, спричиненими інструментами діагностики, у межах бажаного порогу. Це було результатом збільшення CI, коли порогове значення було перевищено для зменшення впливу на ефективність. Наприклад, це коригування для WAIT передбачало збільшення CI вдвічі, перехід від початкового значення 30 секунд до 60 секунд, а потім до остаточного значення 90 секунд. Щодо охоплення

помилок, кількість помилок, виявлених за допомогою адаптивного правила, завжди була вищою, ніж кількість помилок, знайдених із відповідною статичною CI (наприклад, 90 секунд у випадку WAIT). Це було результатом використання інших (менших) CI під час тесту, ситуація, яка спровокувала, що охоплення помилками було вищим (порівняно з відповідною статичною CI) протягом певних періодів тесту. Крім того, той же аналіз був проведений з урахуванням лише критичних помилок, і спостерігалася подібна поведінка. Отже, SI для них не змінився (залишившись у вихідному значенні 0,5 хвилини).

Політика ефективності та цільового завантаження. Друга частина аналізу була зосереджена на оцінці політики ефективності. Отримані результати показали, що політика ефективності досягла своєї мети - зменшити використання у спільних послугах (тобто використані інструменти діагностики у цьому сценарії). Це було результатом зменшення інтерфейсу користувача, коли порогове значення було перевищено з метою зменшення використання ресурсів. Наприклад, у випадку WAIT, CPUAV tt першого раунду завантажень (який відбувся до будь-якого коригування) становив 90,7%. Оскільки це значення перевищило мету максимального використання (UMAX), політика ефективності коригувала інтерфейси користувачів вузлів після першого раунду завантажень. Після налаштування інтерфейсу користувача, CPUAV tt зменшився до 65,7%, залишаючись нижче UMAX протягом решти тесту. Так само, користувальницькі інтерфейси, якими користувався GCLITE і HC під час тестових пробігів, були скориговані. Для GCLITE коригування було викликано використанням процесора, тоді як для HC це було викликано використанням пам'яті. Тим часом GCMA та EMAT не вимагали коригування інтерфейсу користувача. Це було тому, що ці інструменти ніколи не перевищували UMAX будь-якого відстежуваного ресурсу під час їх виконання. Ця поведінка показана на рисунку 3.30, де представлено середнє використання центрального процесора та пам'яті, досягнуте кожним інструментом діагностики. На рисунку UMAX зображений сірою горизонтальною лінією. Ця поведінка показана на рисунку 3.30, де представлено середнє використання центрального процесора та пам'яті, досягнуте кожним інструментом

діагностики. На рисунку UMAX зображений сірою горизонтальною лінією. Ця поведінка показана на рисунку 3.30, де представлено середнє використання центрального процесора та пам'яті, досягнуте кожним інструментом діагностики. На рисунку UMAX зображений сірою горизонтальною лінією.

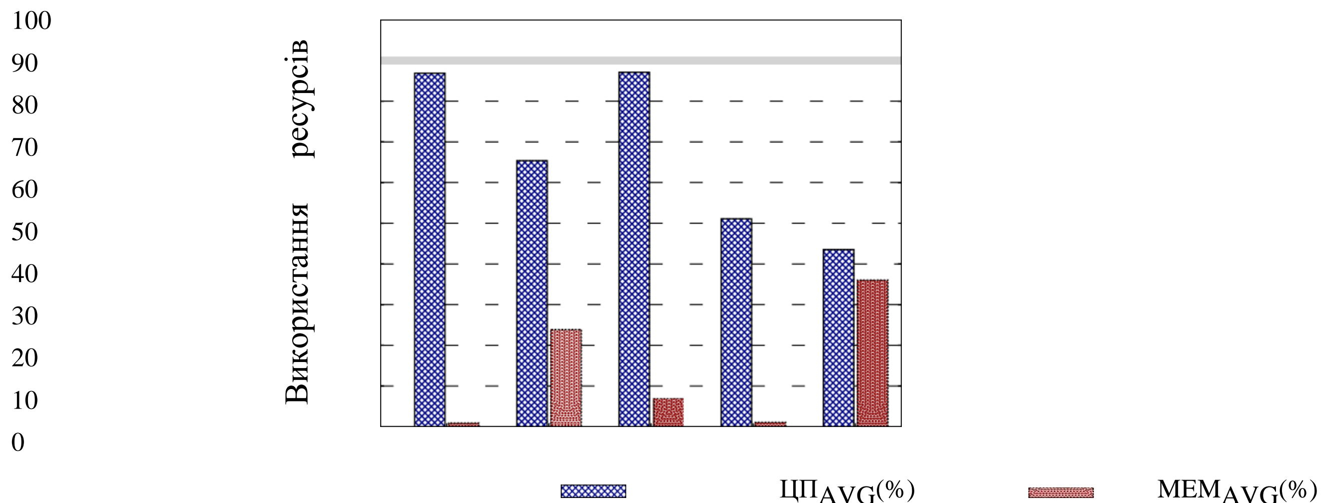


Рисунок 2.3. Ресурси проти конфігурації - всі інструменти.

Політика консолідації Multi-View. Третя частина аналізу була зосереджена на оцінці обсягу економії зусиль, отриманого за допомогою M1_JAVA. Цей аналіз визначив два основних типи економії: ті, що виконуються в задачах аналізу ефективності, та ті, що виконуються в завданнях тестування продуктивності. Щодо завдань аналізу продуктивності, оскільки M1_JAVA може самоконфігуруватись, щоб утримувати тестовий запуск у межах бажаних обмежень (наприклад, пороговий рівень), тестер більше не ризикує вибрати невідповідну конфігурацію (серед набору кандидатів конфігурації). Тому тестувальнику потрібно виконати лише один тестовий запуск продуктивності. Наприклад, припускаючи, що існує вісім наборів конфігурації кандидатів (таких як ті, що оцінені в експерименті №1), тестер заощадить 87,5% часу, необхідного для тестування повного спектру конфігурації (оскільки потрібен лише один тестовий запуск, а не вісім). Додоповнюючи аналіз, пропонуючи більш консервативну перспективу отриманих прибутків, адаптивні пробіги також порівнювали з найкращими та найгіршими показниками. Це порівняння також показало, що M1_JAVA працював добре, оскільки досягнуті

вдосконалення також були дуже значними. Значні зменшення зусиль були результатом автоматизації більшості завдань аналізу, які раніше виконувались вручну.

M1_JAVA зміг заощадити додатковий час на процес тестування продуктивності. Це було зумовлено двома основними причинами. Як вже було обговорено, тестувальнику більше не потрібно випробовувати різні набори конфігурації. Така поведінка безпосередньо перетворюється на економію часу (з точки зору часу виконання тестових запусків). Наприклад, у цьому експерименті ці скорочення становили від 95% до 98% (із середнім показником 97% та стандартним відхиленням 0,8%). Використання запропонованої оцінки "без ходу" на основі резюме (обговорено в розділі 3.5.3) дозволило M1_JAVA визначити момент (під час виконання тестового циклу), коли тестовий цикл вичерпав свої переваги (з точки зору виявлених помилки). Щоразу, коли це відбувалося, тестовий запуск зупинявся, що призводило до додаткової економії часу в процесі. Точний час варіювався серед інструментів діагностики (оскільки на це впливали темп та кількість помилок, виявлених кожним інструментом), але у всіх випадках економився значний час: Порівняно з будь-яким із їхніх статичних аналогів, економія часу коливалась між 66% та 81% (із середнім показником 75% та стандартним відхиленням 6,6%). Результати зведені на рисунку 3.32, який порівнює результати, отримані в результаті адаптивних тестових запусків (на діагностичний інструмент), та їх статичні аналоги.

Результати цього експерименту продемонстрували, як M1_JAVA, завдяки набору запропонованих політик, досягав запланованих цілей: Політика точності утримувала накладні витрати, введені у відстежувані вузли застосунків, в межах бажаного порогу, максимізуючи при цьому кількість виявлених помилок в обмежених умовах. Так само, політика ефективності зменшила використання ресурсів спільної послуги (тобто інструментів діагностики), мінімізуючи можливість її насичення. Нарешті, використання M1_JAVA довів, як він може різко зменшити кількість зусиль / часу, витраченого тестерами під час процесів тестування та аналізу продуктивності.

Подальша оцінка переваг, які приносить M1_JAVA для тестера продуктивності (з точки зору зменшення зусиль та часу), порівнюючи автоматичне використання діагностичного інструмента через M1_JAVA із чисто ручним використанням діагностичного інструменту. Експериментальна установка була подібною до тієї, що використовувалась в четвертому експерименті, з наступними відмінностями. По-перше, використовувалось застосунок iBatis JPetStore 4.0 [23] (з робочим навантаженням 2000 паралельно - користувачі оренди). JPetStore було обрано, оскільки це добре задокументована програма з відкритим кодом, яка також проста у використанні. По-друге, кількість вузлів програми була збільшена (до 10 вузлів) для тестування M1_JAVA в більшому тестовому середовищі. Цей експеримент також передбачав модифікацію вихідного коду JPetStore, щоб ін’єктувати п’ять проблем із продуктивністю (два обмеження блокування, два блокування та одна помилка затримки вводу-виводу). Оскільки експеримент №4 довів, що M1_JAVA добре працює незалежно від інструменту діагностики, я зосередився на. Цей інструмент було обрано за пропозицією нашого промислового партнера (який вважав WAIT найцікавішим інструментом, серед оцінюваних, завдяки його потужним аналітичним можливостям). Оскільки WAIT також показав досягнення «найнижчих» потенційних прибутків, його використання дозволило мені визначити базовий рівень вдосконалення (оскільки покращення в інших інструментах діагностики буде вищим). Нарешті, жодна оцінка GoNoGo не була налаштована для тестування M1_JAVA у тестовому запуску продуктивності типу надійності.

Два виконувались типи запусків. Перший тип включав тестувальника, який намагався ідентифікувати ін’єктованих помилок за допомогою WAIT вручну (M-WAIT). Другий тип запуску передбачав використання WAIT через систему автоматизації (A-WAIT). В обох випадках випробувач не знав кількості або характеристик введених помилок. Результати цього експерименту зведені в таблицю 2.1.

Таблиця 2.1. Порівняння M-WAIT та A-WAIT.

Метричний	М-очікування (год)	Очікування (год)	М-очікування проти Очікування (%)
a. Тривалість виконавець діяльність з випробування манса	32.8	24.1	-27%
b. Тривалість тестування манса	24,0	24,0	0%
c. Зусилля продуктивності аналіз (d + e)	8.8	4.2	-52%
d. Зусилля щодо виявлення помилок фікація	6.8	2.2	-68%
e. Зусилля першопричини аналіз	2.0	2.0	0%

Після порівняння результатів обох запусків було задокументовано дві економії часу при використанні автоматизованого WAIT: По-перше, зусилля, необхідні для ідентифікації помилок, значно зменшились (на 68% менше, ніж WAIT в ручному режимі). Ця економія часу стала результатом спрощення аналізу звітів WAIT: замість того, щоб мати декілька звітів (по одному на вузол / годину), які потрібно було проаналізувати та скорегувати вручну, тестувальник, використовуючи автоматизовану функцію WAIT, потребував лише моніторингу єдиний звіт, який поступово розвивався. Друга економія включала час, необхідний тестувальнику для ідентифікації всіх ін'єкційних помилок. Використовуючи M1_JAVA, можна було поступово подавати WAIT під час виконання тестового запуску (на відміну від ручного WAIT, коли тестеру потрібно було почекати до кінця тестового запуску продуктивності). Така поведінка дозволила тестувальнику за допомогою автоматизованого очікування легко отримати проміжні результати під час пробного запуску. У цьому експерименті всі помилки були ідентифіковані тестером за допомогою автоматизованого очікування після першої години виконання тесту. Таким чином, тестувальник зміг розпочати аналіз цих помилок паралельно до решти виконання тестового запуску (який тестер постійно контролював). Прямим наслідком цієї другої економії часу стало те,

що загальна тривалість тестування продуктивності зменшилась на 27%. Для тестувальника, що використовує автоматизовану функцію WAIT, діяльність практично тривала лише заплановану 24-годинну тривалість тестового запуску, плюс деякий додатковий час, необхідний для перегляду остаточного зведеного звіту WAIT. Це також варто всі помилки були виявлені тестером за допомогою автоматизованого очікування після першої години виконання тесту. Таким чином, тестувальник зміг розпочати аналіз цих помилок паралельно до решти виконання тестового запуску (який тестер постійно контролював). Прямим наслідком цієї другої економії часу стало те, що загальна тривалість тестування продуктивності зменшилась на 27%. Для тестувальника, що використовує автоматизовану функцію WAIT, діяльність практично тривала лише заплановану 24-годинну тривалість тестового запуску, плюс деякий додатковий час, необхідний для перегляду остаточного зведеного звіту WAIT. Це також варто всі помилки були виявлені тестером за допомогою автоматизованого очікування після першої години виконання тесту. Таким чином, тестувальник зміг розпочати аналіз цих помилок паралельно до решти виконання тестового запуску (за яким тестер постійно контролював). Прямим наслідком цієї другої економії часу стало те, що загальна тривалість тестування продуктивності зменшилась на 27%. Для тестувальника, що використовує автоматизовану функцію WAIT, діяльність практично тривала лише заплановану 24-годинну тривалість тестового запуску, плюс деякий додатковий час, необхідний для перегляду остаточного зведеного звіту WAIT. Це також варто тестер міг розпочати аналіз цих помилок паралельно до решти виконання тестового запуску (який тестер постійно контролював). Прямим наслідком цієї другої економії часу стало те, що загальна тривалість тестування продуктивності зменшилась на 27%. Для тестувальника, що використовує автоматизовану функцію WAIT, діяльність практично тривала лише заплановану 24-годинну тривалість тестового запуску, плюс деякий додатковий час, необхідний для перегляду остаточного зведеного звіту WAIT. Це також варто тестер міг розпочати аналіз цих помилок паралельно до решти виконання тестового запуску (який тестер постійно контролював). Прямим наслідком цієї другої економії часу стало те, що загальна тривалість тестування продуктивності зменшилась на 27%. Для

тестувальника, що використовує автоматизовану функцію WAIT, діяльність практично тривала лише заплановану 24-годинну тривалість тестового запуску, плюс деякий додатковий час, необхідний для перегляду остаточного зведеного звіту WAIT. Це також варто згадавши, що обидва тестери змогли ідентифікувати всі ін'єктовані помилки за допомогою звітів WAIT.

Щоб підсумувати експериментальні результати, вони дозволили додатково виміряти переваги продуктивності, які тестер може отримати, використовуючи інструмент діагностики за допомогою M1_JAVA. Зокрема, було задокументовано дві економії часу: зусилля, необхідні для виявлення помилок, були значно зменшені (у цьому випадку - 68%), а також загальна тривалість тестування (у цьому випадку - 27%). Прямим наслідком такої економії часу є зменшення залежності від людських експертних знань та зменшення зусиль, необхідних тестувальнику для виявлення проблем з продуктивністю, а отже і підвищення продуктивності праці.

Висновок до розділу 2

Представлено M1_JAVA, нову адаптивну структуру, яка автоматизує конфігурацію та загальне використання інструменту діагностики в середовищі кластерного тестування. Метою було покращити продуктивність тестера, зменшивши зусилля та знання, необхідні для використання інструментів діагностики. Також було представлено детальне обговорення різних компонентів та політик, які доповнюють можливості M1_JAVA. Крім того, було обговорено всебічну експериментальну оцінку M1_JAVA. Спочатку були оцінені різні компроміси, які зазвичай виникають при використанні інструменту діагностики (з точки зору точності пошуку помилок, зусиль тестувальників та використання ресурсів). Отримані результати показали, що потенційна економія часу / зусиль була значною. Отримані експериментальні результати також продемонстрували, що M1_JAVA може різко скоротити час / зусилля, необхідні тестеру для проведення процесів тестування та аналізу продуктивності. Нарешті, результати також довели, що M1_JAVA здатний спростити конфігурацію діагностичного інструменту. Ця мета була досягнута шляхом вирішення виявлених компромісів без необхідності ручного втручання тестера. Таким

чином, було показано, що M1_JAVA спрощує використання інструменту діагностики та скорочує час, необхідний для аналізу проблем продуктивності, тим самим зменшуючи витрати, пов'язані з тестуванням продуктивності. Можна зробити висновок, що ці результати пропонують практикам цінну довідку щодо переваг, які система автоматизації, зосереджена на ефективному вирішенні загальних обмежень використання, які зазнає інструмент діагностики, може принести тестуванню продуктивності кластерних програм.

Розділ 3
Стратегія балансування навантаження з урахуванням GC

3.1 Використання стратегії балансування навантаження

Метою цієї дослідницької роботи було визначити стратегію балансування навантаження, відому GC, M2_JAVA (показану на рисунку 3.1), яка може динамічно пристосовуватися до конкретних характеристик GC кластерного застосунка (як правило, розміщеного в центрі обробки даних). Ця стратегія дозволить балансирувальнику навантаження прогнозувати появу подій MaGC з достатньою точністю, щоб використовувати цю інформацію для покращення продуктивності кластера.

Концептуальний вигляд рішення зображено на рисунку 3.2. M2_JAVA періодично отримує інформацію з вузлів програми для її характеристики. Потім він визначає найбільш підходящу політику на основі характеристик GC програми, що працює на кожному вузлі (називається сімейством програм). Нарешті, обрана політика використовується для прогнозування подій MaGC та збалансування вхідного навантаження між доступними вузлами застосунків.

Як визначено багатьма авторами [14, 15], самоадаптація забезпечує систему з можливістю самостійно пристосовуватися до змін у навколишньому середовищі для досягнення певних цілей якості в умовах невизначеності.

У контексті M2_JAVA це означає мінімізацію впливу GC на кластер на продуктивність. Щоб включити самоадаптацію до M2_JAVA, я наслідував відому адаптивну модель MAPE-K [82]. Ця модель була обрана, оскільки вона дозволяє акуратно відокремити адаптивний рівень від бізнес-логіки, отже, збільшуючи модульність рішення.

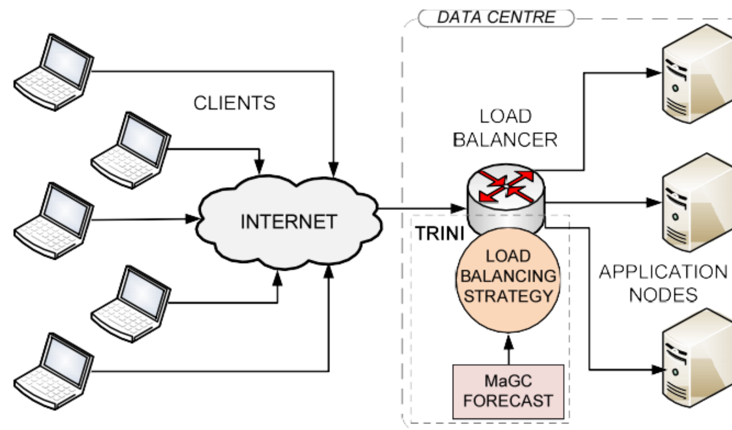


Рисунок 3.1 - M2_JAVA - стратегія LB, яка знає про GC.

У M2_JAVA елемент Знання виконується набором ідентифікованих сімейств програм. Інкапсуляція знань у сім'ї дозволяє M2_JAVA легко розширюватися і мати можливість включати багато політик балансування навантаження, які можуть бути придатними для різних сценаріїв та поведінки в застосуванні. У цьому контексті сімейство програм охоплює набір програми, до яких можна поводитися подібним чином, оскільки вони мають деякі загальні характеристики GC.

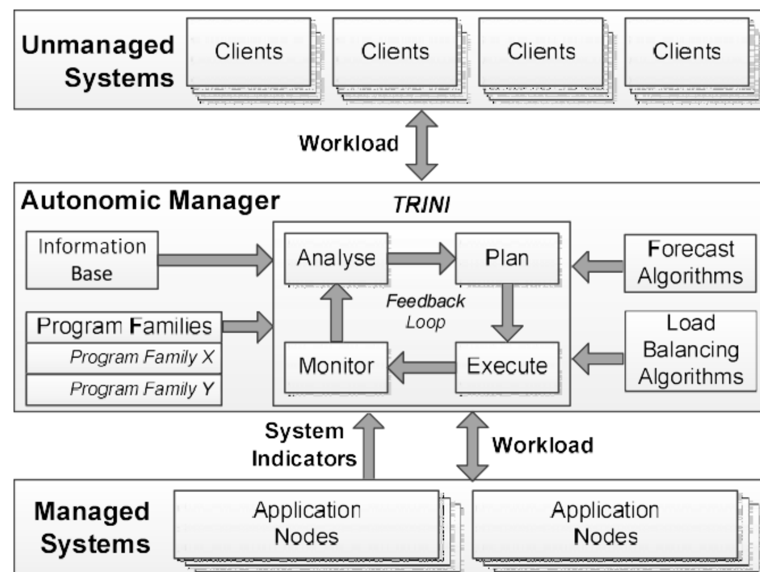


Рисунок 3.2 - M2_JAVA - концептуальний вигляд.

Наприклад, набір сімейств програм може бути визначений відповідно до тривалості MaGC. Можна визначити одну родину для тих програм, які, як

правило, страждають від MaGC невеликої тривалості (наприклад, кілька сотень мілісекунд). Це пов'язано з тим, що ці MaGC зазвичай не представляють серйозної проблеми з продуктивністю. Навпаки, іншу родину можна визначити для тих програм, які, як правило, страждають від MaGCs більшої тривалості. Кожне сімейство програм має дві властивості: (1) Критерії оцінки, щоб визначити, чи відповідає поведінка GC програми цій групі. У попередньому прикладі можливим критерієм оцінки може бути порівняння тривалості MaGC моніторингової програми з діапазонами тривалості кожної визначеної сімейства програм. (2) Політика, яка визначає правила для виконання прогнозування MaGC та балансування навантаження. Наслідуючи попередній приклад, можливою політикою може бути вибір різних діапазонів історичних даних (на сім'ю), які будуть враховані у прогнозі МГК. Ці політики також використовують набір доступних алгоритмів балансування прогнозу та навантаження.

3.2 Алгоритми реалізації стратегії балансування навантаження

M2_JAVA має основний процес, який координує свої елементи MAPE-K. Цей процес (зображений на рисунку 3.3) запускається, коли запускається балансир навантаження. На початковому етапі він використовує політику за замовчуванням (наприклад, усі наявні історії MiGC можуть бути використані для прогнозування MaGC). Ця початкова політика враховує будь-яку додаткову конфігурацію, надану під час запуску (наприклад, інформаційну базу, таку як використаний алгоритм балансування навантаження), і вона використовується для всіх вузлів програми. Далі цикл, зазначений у фазах моніторингу та аналізу, запускається для всіх вузлів програми (паралельно), поки не закінчиться балансування навантаження: Збирається новий набір вибірок даних на основі програмних характеристик GC, що використовуються для визначення набору доступних сімейств програм (наприклад, знімки GC та пам'яті). Після того, як відбувається збір, процес аналізатора перевіряє, чи відповідає поточне сімейство програм GC характеристикам базової програми. Якщо це не так, оцінюються критерії оцінки інших сімейств програм, щоб визначити нове сімейство програм, яке

потім використовується до наступного етапу оцінки. Ці дії отримують їх конфігурації з бази даних сімейств програм (представлені пунктирними стрілками на рисунку 3.3). Цей основний процес продовжується ітеративно до закінчення балансування навантаження. Крім того, будь-які винятки обробляються та повідомляються внутрішньо.

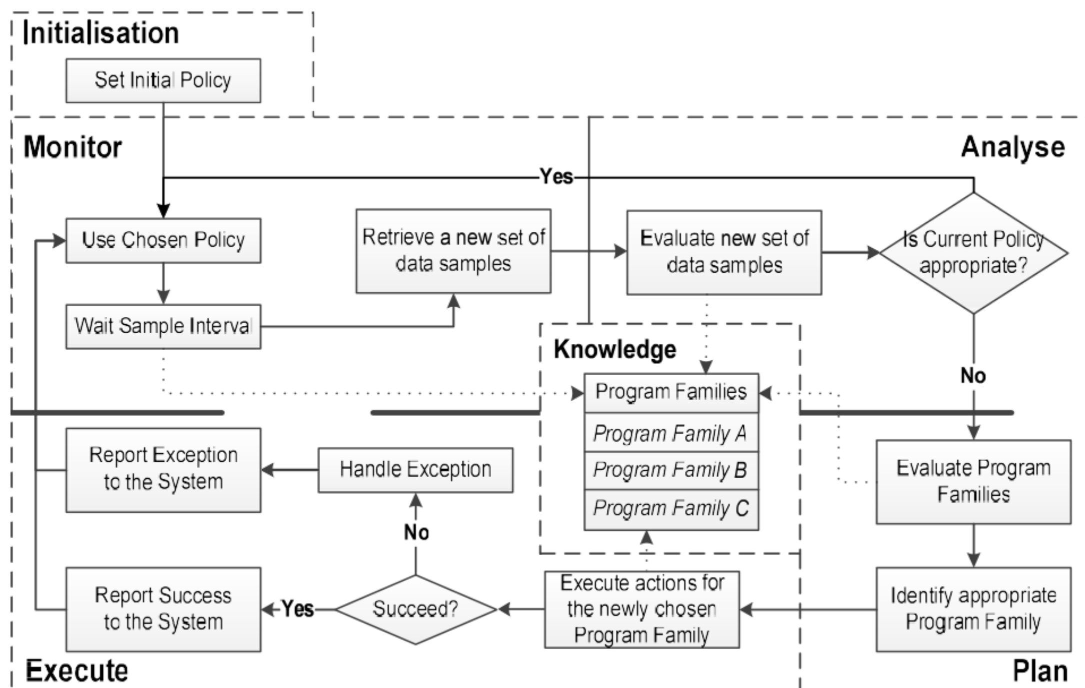


Рисунок 4.3: M2_JAVA - основний процес.

Основною можливістю, необхідною M2_JAVA, є здатність точно прогнозувати, коли відбудуться MaGC. Для задоволення цієї потреби я розробив MaGA, який є алгоритмом прогнозування подій MaGC у кучах поколінь. У наступних розділах описуються внутрішні елементи алгоритму. Крім того, для обговорення алгоритму будуть використані наведені нижче визначення. Час завжди виражається як кількість мілісекунд, які пройшли з моменту запуску програми. Зразки молодого / старого поколінняскладаються з позначки часу та використання відповідної генерації пам'яті. Зразок MiGC складається з часу початку, часу закінчення та використання пам'яті до та після останньої події MiGC. Спостереження використовуються в статистичному контексті і складаються з одного незалежного та одного залежного значень. Коли залежне значення не містить історичних даних, спостереження називається передбачуваним. Стабільний станце стан, до якого застосунок переходить після того, як JVM закінчує завантаження всіх

своїх класів. Передбачається, що цього стану було досягнуто, якщо кількість завантажених класів залишається незмінною для певної кількості послідовних зразків.

На рисунку 3.4 зображено огляд алгоритму, який складається з п'яти фаз. Спочатку ініціалізація, яка встановлює параметри, необхідні алгоритму. Після цього інші етапи ітеративно виконуються для безперервного отримання прогнозів MaGC: Нові вибірки отримуються з контрольованого JVM на етапі збору даних. Потім генеруються нові спостереження за допомогою нових зразків на етапі Асамблеї спостережень. Далі відбувається Розрахунок Прогнозу. Нарешті, логіка очікує інтервал вибірки перед початком нової ітерації. Цей цикл триває, поки відстежувана програма не закінчиться або прогноз більше не потрібен.

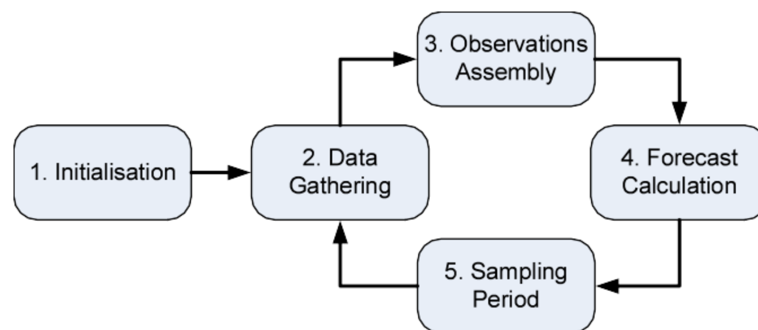


Рисунок 3.4 - MaGa - Огляд процесу прогнозування.

Алгоритм розроблений для роботи на кучах поколінь, оскільки це найпоширеніший тип купи Java. Крім того, він використовує лише стандартні дані, які можна отримати з будь-якої JVM (наприклад, пам'яті або даних GC), щоб полегшити їх впровадження як у JVM, так і поза ним. Якщо алгоритм буде застосовано в JVM, взаємодія з потенційними споживачами буде спрощена. Якщо воно впроваджується поза JVM, реалізація буде працювати з будь-яким JVM, що є на даний момент, полегшуючи прийняття.

Варто зауважити, що LRM був обраний методом прогнозування, що використовується для прогнозування появи наступного MaGC після проведення всебічної оцінки найбільш широко використовуваних методів прогнозування. Метою було визначити, яка техніка може краще моделювати

ГХ. У цій оцінці поведінка GC усіх 23 застосунків, що належать до двох найбільш широко використовуваних тестів Java, була зафіксована під час 1-годинних запусків (з використанням 5 різних розмірів купи). Далі кожна техніка прогнозування отримувала отриману інформацію. Потім були розраховані точність отриманих прогнозів та порівняно між різними методами. Для цього використовувались два найпопулярніші показники точності прогнозу: Середня абсолютна похибка [58] (яка вимірює, наскільки близькі прогнози до кінцевих результатів шляхом обчислення середнього значення похибок абсолютного прогнозу), і середня квадратична помилка квадрата [58] (що представляє стандартне відхилення різниці між передбачуваними значеннями та спостережуваними значеннями). Результати цього аналізу показали, що LRM перевершив інші методики в обох показниках. Це пояснювалось тим, що LRM змогла краще фіксувати поведінку, яка часто спостерігається при використанні пам'яті OldGen програми Java у купі поколінь, яка, як правило, зростає відносно лінійно (незалежно від фактичного темпу та нахилу) між MaGC. Приклад такої поведінки наведено на рисунку 3.5. Там можна помітити, як користування пам’яттю OldGen поступово зростає до виснаження.

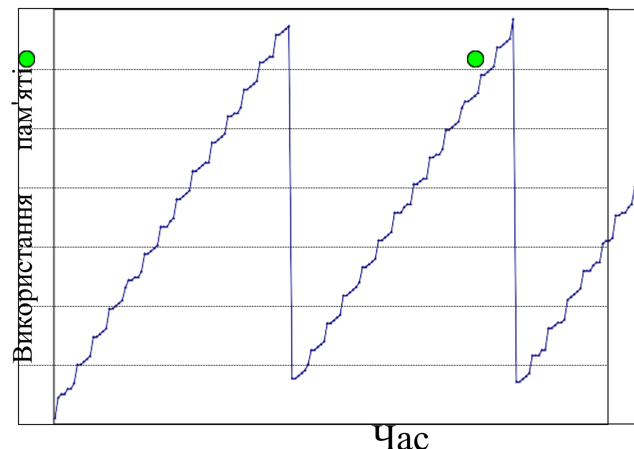


Рисунок 3.5 - Використання пам'яті OldGen у часі.

Період очікування вибірки. Нарешті, процес очікує кількість мілісекунд, налаштовану в інтервалі вибірки, перед початком наступного раунду ітераційних кроків алгоритму.

3.3 Алгоритми балансування навантаження GC-Aware

Дооцінюючи приріст продуктивності, який можна досягти шляхом адаптації балансування навантаження на основі інформації про прогноз MaGC, змінено чотири добре відомі алгоритми балансування навантаження. Серед діапазону доступних алгоритмів обрано чотири: круговий (RR), випадковий (RAN), зважений круговий (WRR) та зважений випадковий (WRAN). Як покажуть експериментальні результати, досягнуті покращення продуктивності очевидні для чотирьох алгоритмів, і тому очікується, що M2_JAVA може дати подібні результати, застосовуючи до інших алгоритмів балансування навантаження.

Основна відмінність моїх алгоритмів (у порівнянні з їх початковими аналогами) полягає в тому, що вони проводять додаткову перевірку при виборі наступного вузла. Тобто, якщо попередньо вибраний вузол (відповідно до їх оригінальних критеріїв відбору) збирається зазнати MaGC в межах заданого порогу (час, коли вузол перестає вважатися можливим кандидатом, оскільки наступний MaGC знаходиться занадто близько), це вузол пропускається, а наступний вузол оцінюється. Після закінчення MaGC уражений вузол знову доступний для вибору. Наприклад, якщо поріг часу становить 5 секунд, а поточний час - 17:00:00 PM, будь-які вузли, для яких передбачено, що MaGC відбудеться між 17:00:00 і 17:00:05, будуть пропущені в ітерації балансування навантаження, оскільки їх прогнози потрапляють до заданого порогу. Додатковою зміною алгоритмів, що усвідомлюють GC, було включення умови виходу, щоб запобігти нескінченному циклу у випадку, коли всі вузли мали зазнати MaGC в межах визначеного порогу. Якщо це трапляється, алгоритми, що знають про GC, поводитимуться як свої оригінали.

Приклад запропонованих алгоритмів представлений в алгоритмі, який показує зважений круговий робот (GC-WRR), який відповідає GC. При порівнянні щодо оригінального WRR можна помітити дві застосовані зміни (рядки 14-21). Додаткова перевірка для розгляду близькості MaGC у виділенні вузла та умова виходу (змінна fTries), яка веде підрахунок

оцінених вузли для запобігання раніше обговорюваному нескінченному циклу.

Інтегруючи GC-обізнаність у чотири обрані алгоритми, виявлено певну схожість між виконаними змінами. Це дозволило мені абстрагувати зміни до загальної версії алгоритму збалансування навантаження, який відповідає GC. Там можна помітити, як після того, як відбувається вихідний вибір балансування навантаження (представлений функцією original-Selection) , алгоритм виконує додаткові кроки для вибору наступного вузла, який буде використовуватися.

Ця нова логіка вкладена в функції IsMaGcClose, getFore- *акторський склад*, *markAsEval*, AreNodesToEval і resetNodesToEval. Функція IsMaGcClose (показана в алгоритмі 4) відповідає за перевірку, чи не є наступний MaGC "занадто близько" для попередньо вибраного вузла. Якщо це так, потрібно вибрати інший вузол. Внутрішньо ця функція використовує getForecast (який є обгорткою алгоритму MaGA, обговореного в розділі 4.3), і markAsEval (який відповідає за маркування, можливо, через структуру даних, як хеш-таблиця [5] або вектор [6], вузлів після їх оцінки для поточного рішення про балансування). Тим часом відповідальність AreNodesTo-Eval полягає в тому, щоб перевірити, чи всі вузли були оцінені для поточного рішення про балансування (щоб уникнути потенційного нескінченного циклу). Нарешті, функція re-setNodesToEval відповідає за очищення позначок після прийняття рішення.

Серед альтернативних стратегій розробки політики для M2_JAVA я спочатку зосередився на автоматизації вибору FWS. Це пояснюється тим, що проведена оцінка точності показала, що точність алгоритму MaGA особливо чутлива до цієї конфігурації. Ця чутливість виникає через те, що FWS обмежує ступінь знань (з точки зору даних історичної пам'яті), що використовується для прогнозування MaGC. Крім того, в цих експериментах немає одиночна FWS досягла найнижчої похибки прогнозу у всіх випадках, показуючи, що не існує FWS, що найкраще підходить для всіх.

Крім того, ці експериментальні результати показали, що алгоритм MaGA, як правило, отримує користь від наявності більше історичних даних. Однак це зростання зазвичай не є монотонним. Навпаки, оптимальний FWS

може відчувати корита. Цю поведінку фіксує метрика MittCCV [113] (яка вимірює коефіцієнт варіації з точки зору кількості MiGC, які виникають між MaGC). Цей підхід робить метрику MittCCV відповідним критерієм для класифікації різних програм поведінки в сім'ях. Наприклад, коли є великі розбіжності у кількості МіГК, що трапляються між МаГК (що відображається у великому значенні MittCCV), використання більше історичних даних не є корисним, оскільки ця історія неправильно фіксує драматичне (кілька порядків величини tude) зміни у поведінці пам’яті.

Висновки до розділу 3

На основі спостережуваної поведінки експериментально було визначено три сімейства програм MittCCV: Низький (MittCCV $\leq$0,1), середній (0,1 <MittCCV <1,0) і високий (MittCCV $\geq$1,0). Для кожної родини функція тренду FWS було похідних, зосереджуючись на тих MaGC, які отримують вигоду від використання приростів в історії MiGC (залишаючи при цьому поза межі тенденції). Валідність отриманих моделей відображалася в їх обчислених значеннях коефіцієнта детермінації [26], які перевищували 0,9 (поріг, загальноприйнятий у статистиці як мінімальне значення для врахування трендової функції, що представляє модельовані дані). Потім ці засновані на функціях політики дозволяли автоматизувати вибір відповідного FWS для кожного випадку окремо. Отримані експериментальні результати продемонстрували, що ці функції змогли точно передбачити хороший відсоток подій MaGC. Нарешті, отримані результати точності також показали, що кількість вибіжників, як правило, зменшується у більших розмірах (наприклад, розміри купи. Така поведінка підтвердила рішення про ігнорування відхилень від похідних функцій.

Розділ 4
Експериментальне оцінювання балансування навантаження

4.1 Постановка експериментів

На поведінку стратегії балансування навантаження сильно впливає точність прийнятих рішень щодо збалансування та обсяг використовуваних ресурсів [35]. Глибоке розуміння цих факторів є ключовим для розуміння доцільності будь-якої стратегії балансування навантаження. Тому було проведено вичерпну оцінку M2_JAVA з точки зору точності, загальності, масштабованості та надійності. У цьому розділі представлені чотири експерименти, проведені для оцінки переваг та витрат від використання M2_JAVA. По-перше, оцінено точність алгоритму MaGa у наборі програм Java та розмірах купи. По-друге, оцінено загальну поведінку M2_JAVA у наборі різних алгоритмів балансування навантаження. По-третє, оцінено масштабованість поведінки M2_JAVA в різних кластерах. Нарешті, Я оцінив надійність поведінки M2_JAVA протягом тривалих періодів часу. Розділ завершується дискусією для практиків, де узагальнено ключові висновки та спостереження.

Тут метою було визначити, які характеристики GC програми можуть працювати краще для визначення початкового набору сімейств програм. Щоб досягти цього, я оцінив точність алгоритму MaGa щодо широкого набору програм Java та експериментальних конфігурацій з метою аналізу умов, в яких алгоритм працює краще.

У наступних параграфах я представляю розроблений прототип, тестове середовище та параметри, що визначали оцінені експериментальні конфігурації. Вибраний діапазон FWS, тести Java та стратегії GC.

Прототип. На основі аналізу інших робіт [46], спрямованими на мінімізацію потенційних впливів на моніторингове середовище, логіка прогнозу була впроваджена зовні (не нав'язливо) для JVM. Для цього я використав розширення Java Management Extension (JMX) [29] для

взаємодії з моніторинговим JVM. JMX був обраний, оскільки це стандартна технологія Java, яка може отримати всю інформацію, необхідну для прогнозування подій MaGC (наприклад, використання пам'яті або знімки GC). Прототип також використовує бібліотеку OpenForecast [36] для всіх статистичних розрахунків.

Java Тести. Орієнтир DaCapo 9.12 [40] та орієнтир SPEC-JVM 2008 [38] були обрані, оскільки вони пропонують широкий спектр різних програмних поведінок для тестування. Більше того, це два найпоширеніші тести Java у літературі (як це обговорювалось у розділі 2.1.10). Оскільки бенчмарк DaCapo пропонує різні тестові навантаження на програму [8], найбільше навантаження для кожної програми було використано, щоб максимально наголосити на GC. Також їх кількість ітерацій було встановлено таким чином, що MaGC запускалися для всіх перевірених розмірів купи. Ця інформація узагальнена в таблиці 4.1. Подібним чином програми SPECJVM були налаштовані для запуску MaGC. Це передбачало встановлення часу ітерації кожної програми на 60 хвилин. Оскільки програма сонячного світла присутня в обох тестах, вона запускалась лише один раз. Крім того, виявилося, що проміжок часу розминки 5 секунд є достатньо великим, щоб дозволити всім програмам закінчити завантаження своїх класів до того, як було сформовано перший прогноз. Нарешті, було перевірено 5 різних розмірів купи на програму (100, 200, 400, 800 та 1600 МБ) з метою диверсифікації більш оціненої поведінки GC (оскільки розмір купи є основним фактором, що впливає на поведінку GC [57]).

Стратегії GC. Були використані три стратегії, обговорені у Розділі 2.1: послідовний GC (sGC), паралельний GC (pGC) та одночасний GC (cGC). Це рішення було прийнято з метою диверсифікації більш оціненої поведінки ГК (оскільки вибір стратегії ГК також є основним фактором, що впливає на поведінку ГК [57]).

Алгоритм прогнозування MaGC. Випробовано широкий діапазон значень FWS: [10..3000] з кроком 10. Крім того, значення інтервалу вибірки було обрано 100 мс, припускаючи, що протягом цього періоду часу не відбуватиметься більше одного MiGC (отже, не будь-який зразок MiGC). Нарешті, було використано вікно прогріву розміром 50 (результат ділення визначеного часу розминки на інтервал вибірки).

Середовище. Всі тести проводились на віртуальній машині (ВМ) з 4 віртуальними процесорами, 3 ГБ оперативної пам'яті та 50 ГБ HD; під управлінням Linux Ubuntu 12.04L та OpenJDK JVM 7u25-2.3.10. Крім того, JVM було налаштовано на ініціалізацію своєї кучі до максимального розміру, щоб підтримувати її постійною під час експериментів, і виклики програмного запиту MaGC були відключені. Віртуальна машина розташована на обладнаному сервері Dell PowerEdge T420 [12] з 2 процесорами Intel Xeon на частоті 2,20 ГГц (12 ядер / 24 потоки), під управлінням Linux Ubuntu 12.04L 64-біт, 96 ГБ оперативної пам'яті, 2 ТБ HD та використанням KVM [32] для віртуалізації.

Таблиця 4.1. Конфігурації DaCapo.

Розмір робочого навантаження	**Програма**	**# Ітерації**
величезний	h2	10
	котик	960
	торгова квасоля	10
	торгове мило	10
великий	аврора	4000
	батік	120
	затемнення	10
	jython	10400
	pmd	200
	сонячний промінь	640
	ксалан	10400
за замовчуванням	фоп	10400
	лю	10400
	індекс	
	lusearch	10400

Критерії оцінки. Були розраховані наступні три показники. Помилка прогнозу (FE) [15]. Ця метрика є відношенням абсолютної похибки прогнозування (різниця між часом прогнозу та часом реальної події MaGC) як частки часу, що минув з часу попереднього MaGC. Зазвичай це виражається у відсотках, щоб бути порівнянним між різними

програмами, де нижчі значення кращі. В якості альтернативи FE може бути виражена як точність прогнозу (FA), що є різницею між максимально можливою точністю (100%) та FE. З точки зору FA, вищі значення кращі. Середня кількість MiGC, що відбулися між двома подіями MaGC (MittCAV tt) [15]. Ця метрика фіксує взаємозв'язок між розміром купи та розподілом пам'яті, необхідним додатку (основні фактори, що впливають на GC, як доведено відповідно в [16] та [17]). Чим менше MittCAV tt, тим більше спрацьовує MaGC, і в цьому випадку програма частіше вичерпує свої пам'ять старого покоління. Якщо значення наближається до нуля (наприклад, 5 або нижче), програма наближається до винятку, який не вистачає пам’яті. Навпаки, значення, далеке від нуля (наприклад, 1000 або вище), вказує на те, що старе покоління рідко вичерпується. Коефіцієнт варіації (MittCCV) [17]. Ця метрика є стандартним відхиленням MittCAV tt, вираженим у відсотках від середнього, і дозволяє порівнювати різні програми з точки зору їх мінливості у використанні пам'яті.

На першому етапі аналіз був зосереджений на оцінці точності алгоритму прогнозування MaGa. Незважаючи на те, що результати різнились між різними стратегіями ГХ, можна було досягти високої точності (маючи на увазі ФЕ нижче 10%) для всіх випробуваних експериментальних конфігурацій. На рисунку 4.8 представлені результати, отримані для кожної зі стратегій GC. Там можна спостерігати досягнуту середню похибку прогнозу, яка становила від 3% до 5%; тоді як стандартне відхилення становило від 2% до 4%. Оскільки жоден FWS не досяг найнижчого FE для всіх програм, наступний аналіз зосередився на визначенні оптимального FWS (FWS, який досяг FE, найближчий до нуля) для поєднання розміру програми та купи. Цей аналіз показав цікаву тенденцію: загалом алгоритм прогнозування, як правило, отримує вигоду від наявності більше історичних даних. Це призводить до того, що оптимальна FWS має тенденцію до зростання з часом. Однак це зростання не було стабільним у більшості випадків. Навпаки, оптимальний FWS зазнав корита під час виконання

більшості програм (маючи на увазі, що в тих випадках менше історії було краще для досягнення низького FE).

На основі такої поведінки аналіз зосереджувався на розумінні причин цих западин. Щоб оцінити, чи корита були спричинені мінливістю поведінки програми (з точки зору використання пам'яті), я проаналізував MittCCV програм. Цей аналіз показав, що існує взаємозв'язок між коритами FWS та змінами у кількості MiGC, що відбуваються між MaGC (що є ключовим входом, що використовується алгоритмом прогнозу MaGC). Всякий раз, коли ці зміни є «надто різкими» (що відображається у великій вартості MittCCV), використання більше історичних даних не є корисним, оскільки ця історія не фіксує належним чином різких (на кілька порядків) змін у поведінці пам'яті. Навпаки, якщо в цьому сценарії використовується лише найновіша історія (що неявно означає використання меншої FWS), точність прогнозу різко покращується.

Вище зазначений результат привів мене до думки, що функції можуть бути виведені із спостережуваної поведінки (де частота мінімальних показників в оптимальному тренді FWS пов'язана з рівнем MittCCV) і використовуватись як перший набір політик у M2_JAVA. Потім ці засновані на функціях політики дозволять автоматичний вибір відповідної FWS на основі оптимальної тенденції FWS. Визначено три сімейства програм на основі їх значень MittCCV: низький (MittCCV <= 0,1), середній (0,1 <MittCCV <1,0) та високий (MittCCV> = 1,0). Потім, трендові функції були отримані з оптимальних результатів FWS. Цей початковий підхід не працював добре для програм у високій сім'ї та кількох програм у середній сім'ї через їх відносно часті корита. Отримані функції не були репрезентативними для змодельованих даних, оскільки вони давали значення коефіцієнта детермінації (R2) [17] нижче 0,9 (що є порогом, загальноприйнятим у статистиці як мінімальне значення R2 для врахування тенденційної функції, репрезентативної для змодельованих даних). Ці результати змусили мене скорегувати обсяг: Замість того, щоб зосередитись на досягненні високої точності прогнозу для всіх подій MaGC, я зосередився лише на тих MaGC, які дотримуються подібної тенденції зростання, як FWS (отже, виграш від використання приростів в історії MiGC), залишаючи поза

межами політики. Гіпотеза полягала в тому, що, хоча ці недосконалі тенденційні функції FWS можуть пропустити вибір відповідної FWS для точного прогнозування MaGC, представлених видаленими випадами, функції все одно можуть бути корисними для точного прогнозування справедливого відсотка MaGC; інформація, яка, отже, дозволить M2_JAVA покращити роботу кластерної системи. Після видалення викидів, репрезентативні трендові функції FWS були успішно виведені для всіх сімейств програм.

Остаточним спостереженням цього експерименту було те, що, поки загальне навантаження, оброблене в експериментальній конфігурації, не змінюється (умова, яка була виконана для всіх конфігурацій, що використовують одну і ту ж програму), мінливість (з точки зору MittCCV) зменшується розмір купи збільшується. Це пояснюється тим, що MaGCs є більш однорідними (за кількістю MiGC) у таких більших розмірах купи. Така поведінка сприяє обраній стратегії пропуску викидів, оскільки їх кількість зменшується, коли мінливість зменшується. На відміну від цього, MittCAV tt збільшується, коли збільшується розмір купи. Це пов’язано з тим, що перед запуском MaGC залишається більше пам'яті.

Цей експеримент довів, що алгоритм MaGa може точно прогнозувати події MaGC (досягнення FE <10% для всіх перевірених програм та розмірів купи) при правильній настройці. Вибір FWS особливо важливий, оскільки алгоритм чутливий до цього параметра конфігурації. Крім того, експериментальні результати показали, що метрика MittCCV є відповідним критерієм для класифікації різних програмних форм поведінки в сім'ях. Потім ці сімейства сприяли виведенню політик на основі функцій для автоматизації вибору FWS без необхідності ручного налаштування.

Метою наступного експерименту було оцінити загальні переваги та витрати на використання M2_JAVA. Для цього було порівняно поведінку M2_JAVA, застосовану до чотирьох часто використовуваних алгоритмів балансування навантаження. У наступних розділах описується цей експеримент та його результати.

Вибрані алгоритми балансування навантаження, тести Java та стратегії GC. Також описано критерії оцінки, використані в цьому експерименті. Прототип. Він був побудований на верблюді Apache Camel [1], який є

популярним балансиром навантаження. Це рішення було обрано, оскільки воно є відкритим кодом і розроблено на Java, характеристики, що полегшили його інтеграцію з логікою прогнозування MaGC. Крім того, архітектура цього балансира навантаження пропонує чітко визначені точки розширення. Ця характеристика полегшила реалізацію алгоритмів балансування навантаження, які відповідають GC. Середовище. Всі експерименти проводились в ізольованому тестовому середовищі, так, щоб було контрольовано все навантаження. Це середовище складалося з п'ятдесяти двох віртуальних машин: кластер з п'ятдесяти вузлів програми з одним балансувачем навантаження та одним вузлом тестування навантаження. Усі віртуальні машини мали такі характеристики: 4 віртуальні процесори на частоті 2,20 ГГц, 3 ГБ оперативної пам'яті та 50 ГБ HD; під управлінням Linux Ubuntu 12.04L та OpenJDK JVM 7u25-2.3.10 з купою 1,6 Гб. Кожен JVM був налаштований на ініціалізацію своєї купи до максимального розміру та виклики на програмне запит MaGC були відключені. Вузол тестувача навантаження також використовував Apache JMeter 2.9 [2], а на вузлах застосунків працював Apache Tomcat 6.0.35 [3].

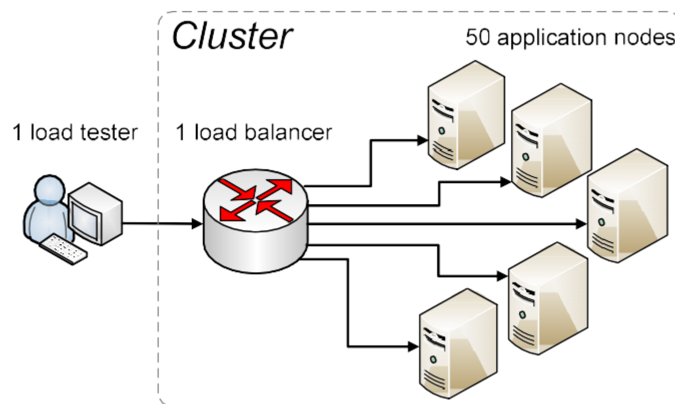


Рисунок 4.2 - M2_JAVA - Тестове середовище.

Віртуальні машини розташовувались на двох серверах Dell PowerEdge M620 [11] усередині шасі Dell PowerEdge VRTX [13]. Кожен сервер був оснащений 2 процесорами Intel Xeon E5-2660 v2 на частоті 2,20 ГГц (10 ядер / 20 потоків), 192 ГБ оперативної пам'яті, мережевою картою 10 Гбіт / с та VMware ESXi 5.5.0 як гіпервізором. Крім того, шасі забезпечило 12,3 ТБ пам'яті SAS (складається з задньої плати жорсткого диска з 14 жорсткими

дисками) через адаптер PERC 6 Гбіт / с.

4.2 Алгоритми балансування навантаження

Були протестовані чотири алгоритми: круговий (RR), випадковий (RAN), зважений круговий (WRR) та зважений випадковий (WRAN); а також їх розроблені аналоги з GC (GC-RR, GC-RAN, GC-WRR, GC-WRAN). Було проведено два типи прогонів: перший тип використовував оригінальну версію кожного алгоритму і вважався базовим при аналізі результатів. Другий тип запуску використовував GC-версію кожного алгоритму. Щодо алгоритму прогнозування MaGC (який внутрішньо використовується алгоритмами, що знають про GC, як пояснено в розділі 4.4), FWS був автоматично обраний функціональними політиками, описаними в розділі 4.5. Крім того, для інтервалу вибірки було обрано значення 100 мс, припускаючи, що протягом цього періоду часу не відбуватиметься більше одного MiGC (отже, не бракує жодного MiGC).

Вибрано два найбільш широко використовувані в літературі тести Java (DaCapo 9.12 [40] та SPECJVM 2008 [38]), оскільки вони пропонують широкий спектр з 23 різних програм для тестування. На відміну від інших еталонів (які генеруються синтетично), це програми з реального життя з різних сфер бізнесу, які широко використовуються в галузі. Розділ 2.1.10 представляє короткий виклад цих орієнтирів та їх програм.

Для того, щоб мати можливість викликати програму з тестового сценарію JMeter HTTP (щоб можна було викликати декілька одночасних викликів на вузол програми), було розроблено обгортковий JSP, який встановлено в екземплярі Tomcat кожного вузла програми. Для кожної програми було створено тестовий сценарій JMeter, який додав деяке контрольоване розмаїття до робочого навантаження. Для програм DaCapo він передбачав зміну розміру навантаження між викликами програм (використовуючи наявні заздалегідь визначені розміри навантаження DaCapo [8]). У випадку програм SPECJVM контрольоване розмаїття передбачало різний час виконання (в діапазоні від 30 до 90 секунд). Кожен тестовий запуск JMeter тривав 60 хвилин і використовував 500 одночасних користувачів. Нарешті, кожен окремий виклик програми вважався

транзакцією.

Стратегії GC. Були використані три стратегії, обговорені у Розділі 2.1: послідовний GC (sGC), паралельний GC (pGC) та одночасний GC (cGC). Це рішення було прийнято з метою диверсифікації більш оціненої поведінки ГК (оскільки вибір стратегії ГК є основним фактором, що впливає на поведінку ГК [57]).

Критерії оцінки. Що стосується продуктивності, основними показниками були пропускна здатність в секунду (tps) і час відгуку (мс). Що стосується часу відгуку, то нижчі значення кращі; тоді як для пропускної здатності вищі значення кращі. Ці показники були зібрані за допомогою JMeter. Що стосується накладних витрат, основними показниками були використання CPU (%) та пам'яті (MB). В обох випадках нижчі значення кращі. Ці показники були зібрані за допомогою команди top [33]. Що стосується точності прогнозу, також були розраховані три метрики, обговорені в Розділі 4.6.1.1: Помилка прогнозу (FE), середня кількість МіГК, що відбулися між двома подіями MaGC (MittCAV tt), та коефіцієнт варіації (MittCCV).

Покращення продуктивності.ДоЩоб зрозуміти поведінку M2_JAVA в оцінюваних експериментальних конфігураціях, я спочатку зосередив свій аналіз на оцінці покращення продуктивності, якої досяг M2_JAVA. У цьому контексті покращення продуктивності для певної метрики (наприклад, часу відгуку) - це різниця між експериментальною конфігурацією, що використовує алгоритм балансування навантаження, відомий GC (наприклад, GC-WRR), та його аналогом із використанням відповідного оригіналу. алгоритм (наприклад, WRR). Що стосується пропускної здатності, покращення продуктивності передбачає позитивну різницю (оскільки вища пропускна здатність краща) і має значення більше ніж 0%. Що стосується часу відгуку, поліпшення продуктивності передбачає негативну різницю (оскільки менший час відгуку кращий) і має значення в діапазоні від 0% до 100%.Загальні результати показали, що M2_JAVA працював добре, оскільки всі експериментальні конфігурації, знайомі з GC, досягли поліпшення продуктивності. Що ще важливіше, поведінка чотирьох перевірених GC алгоритмів балансування навантаження була подібною, оскільки вони досягли

порівнянних поліпшень продуктивності. На рисунку 4.11 наведені результати середнього часу відгуку (RTAV tt). Там можна спостерігати досягнуті середні покращення показників, які становили від 28% до 31%. Слід зазначити, що ці результати узагальнюються за повним набором тестових застосунків, які мають широкий діапазон поведінки пам'яті, так що спостережувані стандартні відхилення становили від 21% до 24%. Рисунки 4.12 та 4.13 зображують отримані результати з точки зору максимального часу відгуку (RTMAX) та середньої пропускної здатності (TAV tt). Наступний раунд аналізу був зосереджений на оцінці чутливості M2_JAVA щодо різних використовуваних стратегій GC. Ці результати представлені на рисунках 4.14 (RTAV tt), 4.15 (RTMAX) та 4.16 (TAV tt). Там видно, що середні покращення продуктивності, досягнуті M2_JAVA, були відносно близькими за трьома стратегіями GC, що означає, що M2_JAVA працював добре незалежно від стратегії. Найбільший виграш відбувся з sGC, оскільки ця стратегія GC переживала найбільш трудомісткі події MaGC (отже, маючи найбільший потенційний виграш для використання).

Попередні два аналізи були корисними для отримання високоякісного уявлення про досягнуті покращення продуктивності. Однак ці аналізи не виявити відмінності у поведінці пам'яті у тестованих програмах (що відображається у відносно високих стандартних відхиленнях, отриманих після консолідації результатів). Тому було потрібно додаткове дослідження з точки зору більшої пам'яті / ГХ.

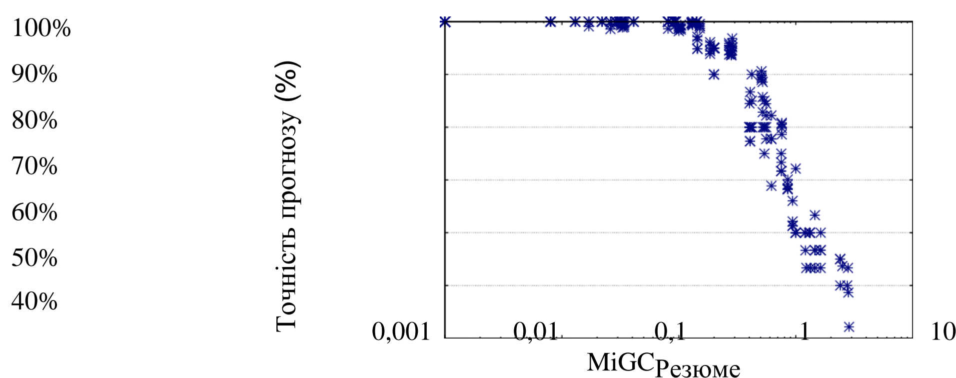


Рисунок 4.3 - Точність прогнозу проти MittCCV.

На наступному кроці проаналізуємо причини досягнення результатів. Для цього проаналізуємо результати з точки зору поведінки MittCCV, як

показано на рисунку 4.3. Там можна спостерігати чіткий взаємозв'язок між точністю прогнозу, досягнутою M2_JAVA, та MittCCV для різних видів поведінки застосунків. Загалом, чим менша мінливість, тим точніше M2_JAVA. Що ще важливіше, точність прогнозу досягає практично 100%, коли мінливість нижче 0,1. Така поведінка зберігалася незалежно від обраного алгоритму балансування навантаження або стратегії GC. Ці результати показують, як MittCCV є відповідною метрикою для характеристики програмної поведінки в сім'ях.

У своїх експериментах визначено, що покращення продуктивності M2_JAVA в основному зумовлене двома факторами. Загальний час, витрачений на MaGC у всіх вузлах програми (MattCD), оскільки він фіксує величину потенційного посилення, яке можна отримати. Точність прогнозу (FA) M2_JAVA, яка є фактичним фактором, що дозволяє перетворити потенційні посилення у фактичні прибутки (шляхом перенаправлення робочого навантаження з будь-якого вузла, який страждає від MaGC).

Загалом, покращення продуктивності, як правило, стає більшим, коли MattCD довгий (оскільки є більше потенційного вигоди для використання). Як-коли-небудь фактичні вигоди залежать від кількості MattCD, яка фактично розглядається (A-MattCD). Така поведінка зображена на рисунку 4.18, де показано досягнуті покращення продуктивності (з точки зору RTAV tt) щодо A-MattCD та FA. A-MattCD виражається у відсотках від загального часу виконання. ФА групується у три рівні: низький (30% ≤FA≤50%), середній (50% <FA≤80%) та високий (FA> 80%). На рисунку 4.18 видно, як покращення для певного рівня коефіцієнта корисної дії (наприклад, високого), як правило, стає більшим, коли A-MattCD довший. Також можна помітити, як A-MattCD сильно впливає на досягнення покращених показників. Наприклад, найбільших поліпшень досягли ті конфігурації, які мали найдовший A-MattCD, хоча вони досягли лише середнього рівня FA.

У цьому контексті MaGC вважався адресованим, якщо його прогнозували досить точно, щоб можна було запобігти надсиланню транзакцій на постраждалий вузол під час MaGC. За цих умов єдиними транзакціями, на які вплинула подія MaGC, були транзакції, які були оброблені вузлом, який страждав від MaGC.

Ця поведінка додатково пояснюється рисунком 4.19, який показує, як FA перекладається у A-MattCD. Загалом, чим вищий коефіцієнт корисної речовини, тим більша кількість A-MattCD. Однак відносини не зовсім лінійні. Це пов'язано з тим, що кількість A-MattCD залежить не тільки від кількості MaGC, які не були розглянуті (як вимірюється FA), але і від тривалості цих MaGC. Наприклад, це не те саме вплив на продуктивність для неточного прогнозування MaGC, який триває дві хвилини, ніж MaGC, який триває дві секунди (хоча обидві події MaGC однаково фіксуються метрикою FA).

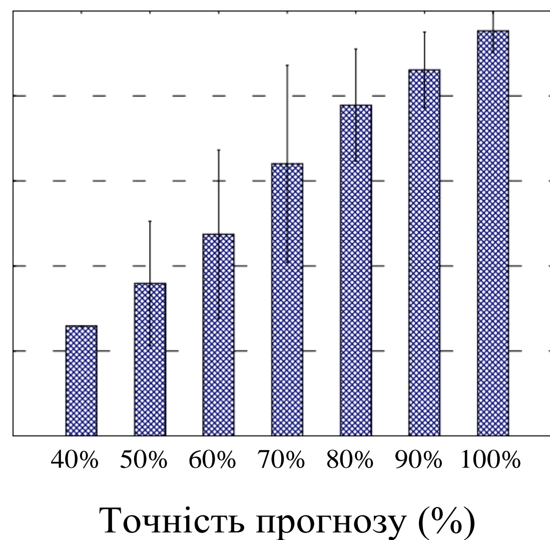


Рисунок 4.4 - A-MattCD для рівня точності прогнозу.

Накладні витрати. Досліджено витрати на використання M2_JAVA. Для цього аналізу я класифікував можливі накладні витрати на два типи: накладні витрати, введені у вузлах програми, та накладні витрати у вузлі балансування навантаження.

У вузлах застосунків M2_JAVA виявився незначним з точки зору процесора та пам'яті для всіх алгоритмів балансування навантаження. Приріст середнього використання центрального процесора (ΔCPUAV tt) у всіх тестованих програмах становив 1,46% при стандартному відхиленні 0,43%; тоді як приріст середнього використання пам'яті (ΔMEMAV tt) становив 0,55%, із стандартним відхиленням 0,35%. Ці збільшення були спричинені процесом збору даних, який збирає інформацію з різних вузлів програми (виконується через JMX, як пояснено в Розділі 4.6.1.1). Ці

результати представлені на рисунках 4.20 та 4.21, на яких показано ΔCPUAV tt та ΔMEMAV tt, відповідно.

У вузлі балансувача навантаження введені накладні витрати були вищими (порівняно з вузлами застосунків), але все ще в межах розумного рівня для кластера з 50 вузлами. ΔCPUAV tt становив 27,88%, зі стандартним відхиленням 1,30%; тоді як ΔMEMAV tt становив 6,34%, із стандартним відхиленням 0,71%. Крім того, чотири алгоритми балансування навантаження виконувались однаково, припускаючи, що рівень введених накладних витрат не залежав від алгоритму. Ці результати представлені на рисунках 4.22 (ΔCPUAV tt) і 4,23 (ΔMEMAV tt). ΔCPUAV tt в основному був спричинений алгоритмом прогнозування, оскільки він постійно генерує оновлений прогноз MaGC для кожного вузла програми. Що стосується споживання пам'яті, приблизно 4% ofMEMAV tt було спричинено ініціалізацією M2_JAVA. Збереження зросло завдяки історичним даним, які зберігались для прогнозування.

Класифікація програм DaCapo / SPEC.Результати цього експерименту також дозволили класифікувати 23 тестовані програми відповідно до їх характеристик ГХ (MattCD та MittCCV). Ця класифікація програм наведена в таблиці 4.2.

Таблиця 4.2. Класифікація програм DaCapo / SPEC за поведінкою GC.

MiGCCV	MaGCD		
	Короткий	Середній	Довго
Низький	омпілятор, jython ——	avrora, компрес, fop, luindex, lusearch, mpegaudio, tomcat, запуск, соняшник, ксалан	
Середній		батік, крипто, затемнення, pmd, торгова квасоля	h2, scimark, tradesoap, xml
Високий		____	дербі, серіал

Цей експеримент продемонстрував приріст продуктивності, який M2_JAVA може принести кластеру. Уникаючи впливу більшості подій MaGC на окремі вузли, продуктивність кластерних застосунків була

значно покращена. Що ще важливіше, поліпшення були досягнуті незалежно від використовуваного алгоритму балансування навантаження або стратегії GC, що підтверджує загальність M2_JAVA. Що стосується накладних витрат, то збільшення обсягу процесора та пам'яті у вузлах застосунків було мінімальним, отже, не впливаючи на їх нормальну роботу. Незважаючи на те, що рівень допустимих накладних витрат у вузлі балансувача навантаження залежав би від конкретного сценарію використання, отримані прирости вважалися прийнятними, оскільки вузол балансира навантаження був далеко не вичерпуючи свої ресурси (особливо враховуючи відносні скромні характеристики балансира навантаження вузол,

Тут метою було оцінити масштабованість M2_JAVA шляхом оцінки його поведінки в різних розмірах кластерів. У наступних розділах описується цей експеримент та його результати.

Налаштування було подібним до того, що було використано в експерименті №2, з наступними відмінностями: Розмір кластера був змінним, охоплюючи діапазон [5..50] вузлів програми з кроком в 5. Кількість одночасних користувачів було збільшено пропорційно розміру кластера (наприклад, 5-вузловий кластер використовував 50 користувачів, 10-вузловий кластер використовував 100 користувачів тощо), так що навантаження відповідно збільшувалось. Оскільки експеримент №2 довів, що M2_JAVA добре працює незалежно від алгоритму балансування навантаження, я зосередився на WRR, оскільки в даний час це найбільш широко використовуваний алгоритм балансування навантаження [59]. Так само я зосередився на стратегії Serial GC, оскільки вона, як правило, переживає найдовші паузи [127], отже, отримуючи більше користі від моєї роботи. Нарешті, я зупинився на 5 програмах, які були репрезентативними для класифікації програм, представленої в розділі 4.6.2.2. Ця конфігурація дозволила протестувати різноманітний набір поведінки GC з меншим набором експериментальних конфігурацій. Ця класифікація (показана в таблиці 4. 2) згрупували програми тестів DaCapo та SPECJVM відповідно до їх характеристик GC (MattCD та MittCCV). У таблиці 4.2 підкреслені програми, які використовуються в цьому експерименті: jython, sunflow,

eclipse, scimark та derby.

У цьому експерименті аналізовано два основних аспекти: оцінка покращення продуктивності, яку дає M2_JAVA в різних розмірах кластерів; та оцінка поведінки накладних витрат у таких кластерах.

Покращення продуктивності. Гіпотеза полягала в тому, що покращення продуктивності не повинно погіршуватись, коли збільшується розмір кластера, оскільки кожен процес прогнозування не залежить один від одного (отже, не залежить від кількості відстежуваних вузлів програми). Це підтвердили результати експерименту. Незважаючи на те, що були незначні відмінності у відсотках покращення продуктивності, яких досягла M2_JAVA, ці вдосконалення були близькими, за різними розмірами кластера, на одну перевірену програму. Рисунок 4.5 показує отримані покращення продуктивності RTAV tt. Там можна помітити, як вдосконалення були відносно постійними для кожної програми через різні розміри кластера. Крім того, відмінності в удосконаленнях між тестованими програмами були зумовлені їх різноманітністю в пам'яті / поведінці GC. Наприклад, програма scimark досягла найбільших покращень, оскільки вона випробовувала найдовший MattCD, а також досягла високої точності прогнозу (понад 90%). Навпаки, програма jython отримала найменші вдосконалення, оскільки зазнала найкоротшого MattCD (тобто мала найменші потенційні вигоди). Подібні тенденції спостерігалися щодо RTMAX та TAV tt.

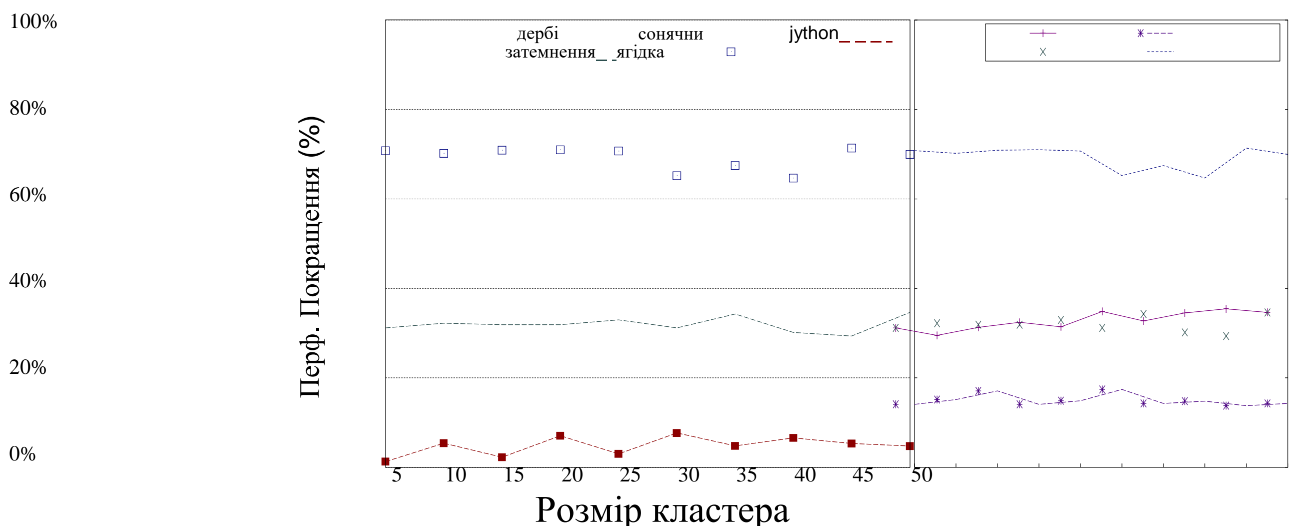


Рисунок 4.5 - RTAV tt - Поліпшення за розмір кластера.

Накладні витрати. Два основні висновки були визначені з точки зору накладних витрат. По-перше, вартість використання вузлів M2_JAVA у вузлах застосунків була мінімальною, відносно постійною і не залежала від розміру кластера. Як пояснювалося раніше, процес прогнозування для кожного вузла програми не залежить один від одного. Таким чином, той самий принцип застосовується до збору даних, який відбувається у вузлах. Це можна помітити в результатах аналізу ∆CPUAV tt і ∆MEMAV tt у вузлах програми за розміром кластера (показано на рисунках 4.25 та 4.26 відповідно). Там можна спостерігати, як прирости використання обох ресурсів були дуже низькими. Крім того, вони представили відносно рівномірний розподіл за всіма тестованими розмірами кластера.

По-друге, накладні витрати у вузлі балансувача навантаження залежали від розміру кластера, дотримуючись відносно плавної тенденції зростання. У випадку з ∆CPUAV tt,ці прирости були в основному спричинені збільшенням кількості одночасних процесів прогнозування (оскільки на кожен моніторинговий вузол програми був один процес прогнозування). Це пояснює відносно лінійний характер зростання. У випадку з ∆MEMAV tt,спостережувані прирости в основному були спричинені обсягом даних, зібраних з вузлів програми для цілей прогнозування. За цих обставин, якщо застосунок ініціює значно велику кількість MaGC та / або MiGC, обсяг пам'яті, необхідний для збереження цих історичних даних, може стати значним. Там можна помітити, як програма дербі мала відносно більший нахил (порівняно з іншими програмами). Це пов’язано з тим, що дербі дало не лише найбільшу кількість історичних даних GC, але й значно більше (на кілька порядків), ніж інші програми. Варто згадати, що, незважаючи на відносно великий нахил, обсяг пам'яті, необхідний M2_JAVA для підтримки дербі, все ще був менше 10% загальної доступної пам'яті (на вузлі балансувача навантаження), навіть з 50 вузлами програми. Цей рівень використання залишає значну кількість простоїв ресурсів для підтримки набагато більше вузлів застосунків.

На закінчення результати цього експерименту показали, як M2_JAVA може витончено масштабуватися для більших скупчень. Досягнуті покращення продуктивності не погіршувались при збільшенні розміру

кластера, тоді як обчислювальні ресурси, що використовуються M2_JAVA, не суттєво зросли.

Тут метою було оцінити надійність M2_JAVA шляхом оцінки його поведінки в більш тривалих (24-годинних) експериментальних пробігах. У наступних розділах описується цей експеримент та його результати.

Налаштування було подібним до того, що було використано в експерименті №3 (представленому в розділі 4.6.3.1), з двома відмінностями: По-перше, оцінюваний кластер був складений з 50 вузлів програми (такого ж розміру, як експеримент №2, описаний у Розділ 4.6.2.1). По-друге, тривалість тестових прогонів була збільшена з 1 до 24 годин для оцінки M2_JAVA на більш тривалій, більш реалістичній тривалості.

Покращення продуктивності.ДоЩоб зрозуміти покращення продуктивності, досягнутої M2_JAVA за допомогою експерименту, я проводив розбивку поведінки кожної експериментальної конфігурації щогодини. Результати цього аналізу не показали серйозного погіршення отриманих поліпшень під час 24-годинних тестових пробігів, що доводить, що поведінка M2_JAVA (з точки зору поліпшення продуктивності) залишається стабільною протягом часу (що відображається в низькому стандартному відхиленні). Серед перевірених програм найбільше середньоквадратичне відхилення було у програмі scimark. Ця поведінка була компенсована досягнутими покращеннями продуктивності (наприклад, середнє значення 69% у перерахунку на RTAV tt), які були найвищими серед тестованих програм. Рисунок 4.29 показує результати в перерахунку на RTAV tt. Подібні результати були отримані щодо RTMAX і TAV tt. Результати аналізу показали, що M2_JAVA не погіршує поведінку вузлів програми з часом. Це пов’язано з тим, що M2_JAVA викликає у них лише мінімальні (і відносно постійні) накладні витрати. ΔCPUAV tt у всіх тестованих застосунках становив 1,02% при стандартному відхиленні 0,38%; тоді як ΔMEMAV tt у всіх тестованих застосунках становив 0,42%, із стандартним відхиленням 0,15%. У вузлі балансувача навантаження результати мого аналізу показали, що $\Delta CPUAV_{tt}$ спричинене M2_JAVA залишалося досить

стійким протягом усіх експериментальних проб (25,57% при стандартному відхиленні 2,15%). Це пов’язано з тим, що основним внеском у це збільшення є кількість прогнозованих процесів, на які впливає не час, а розмір кластера. З точки зору пам'яті, ΔMEMAV tt у всіх тестованих застосунках становив 6,08%, зі стандартним відхиленням 0,86%. Це збільшення залишалося в межах чітко визначеного діапазону протягом 24-годинних пробних прогонів. Ця стабільність у розмірі пам'яті M2_JAVA є результатом ефективного управління історичними даними (наприклад, подіями MiGC), які тимчасово зберігаються M2_JAVA. Ці дані ретельно контролюються і контролюються, так що, коли вони стають старшими за необхідний FWS (що обмежує історію, яка використовується для прогнозування), дані автоматично очищаються.

Висновки до розділу 4

Результати цього експерименту продемонстрували надійність M2_JAVA з часом, оскільки M2_JAVA був здатний поліпшити роботу кластерної системи, не зазнаючи погіршення її поведінки. Що стосується накладних витрат, M2_JAVA відчував відносно рівномірний ΔCPUAV tt протягом усіх тестових запусків. Подібна поведінка спостерігалася і з точки зору ΔMEMAV tt у вузлах програми. Нарешті, M2_JAVA зазнав мінімального збільшення в перерахунку на ΔMEMAV tt у вузлі балансувача навантаження. Це було пов’язано з даними, які M2_JAVA тимчасово зберігались для цілей прогнозування.

Загальні висновки

Кластерні обчислення набули популярності як потужне та економічно ефективне рішення для паралельної та розподіленої обробки. Наприклад, корпоративні програми зазвичай розгортаються в кластерних екземплярах для досягнення вищої продуктивності системи в порівнянні з рішеннями для однієї машини. Однак також добре відомо, що використання кластерів суттєво збільшило складність систем, ще більше ускладнюючи всі дії, пов'язані з оптимізацією продуктивності кластерних систем.

Довирішуючи вищезазначену проблему, ця робота сприяє оптимізації продуктивності кластерних систем на Java (що є однією з найбільш переважаючих технологій на рівні підприємства), особливо націлених на широкомасштабне середовище (оскільки ці середовища, швидше за все, страждають проблемами продуктивності). Зокрема, пропонуються дві методики для вирішення проблем ефективного виявлення проблем, пов'язаних з робочим навантаженням, та ефективного уникнення наслідків великого збору данних на продуктивність - двох важливих проблем, які типова кластерна система Java, ймовірно, зазнає у великих середовищах.

Виявлення проблем продуктивності в кластерних середовищах є складним і трудомістким. Незважаючи на те, що дослідники розробляють інструменти діагностики для спрощення цього завдання, у цих інструментів існують різні обмеження, які перешкоджають їх ефективному використанню в тестуванні продуктивності. Для вирішення цих обмежень представлено нову адаптивну структуру, M1_JAVA, для автоматизації використання інструменту діагностики ефективності в кластерному середовищі. Внутрішньо M1_JAVA використовує набір адаптивних політик для управління різними наборами процесів, які зазвичай беруть участь у конфігурації та використанні інструменту діагностики. Прототип був розроблений навколо набору добре відомих інструментів діагностики для експериментальної оцінки структури. Отримані експериментальні результати продемонстрували, що відповідну економію часу можна отримати, застосувавши запропоновану структуру: значно зменшились не лише зусилля та досвід, необхідні для використання різних інструментів діагностики (від 95% до 99,9%), але й загальна

тривалість тестування продуктивності була значно скорочена (між 66% і 98%). Результати також продемонстрували, що такий адаптивний механізм, що підтримує політику, спрощує конфігурацію інструменту діагностики. Це було досягнуто шляхом автоматичного вирішення компромісів, визначених у кожному інструменті, без необхідності втручання тестера вручну. Таким чином, система продемонструвала спрощення використання інструменту діагностики та скорочення часу, необхідного для аналізу проблем з продуктивністю,

Інша найважливіша проблема кластерних обчислень полягає в тому, як ефективно розподілити навантаження між вузлами кластера. Щоб вирішити цю проблему, представлено M2_JAVA, нову адаптивну стратегію балансування навантаження з урахуванням ГХ. Це покращує продуктивність кластерних систем Java, уникаючи впливу на продуктивність Основного збору данних (що є загальною причиною погіршення продуктивності в системах Java). Внутрішньо, M2_JAVA використовує дані JVM та набір визначених сімейств програм для адаптації до характеристик GC базового застосунка. Впроваджено прототип, і M2_JAVA було всебічно оцінено з точки зору загальності, масштабованості та надійності, щоб запропонувати цінну інформацію щодо поведінки M2_JAVA за таких обставин. Експериментальні результати продемонстрували, що M2_JAVA може значно покращити час відгуку та пропускну здатність кластера (наприклад, середнє поліпшення продуктивності в RTAV tt коливалося між 28% та 31%). Поліпшення були досягнуті незалежно від використовуваного алгоритму балансування навантаження, що підтверджує загальність M2_JAVA. Результати також показали, що M2_JAVA є масштабованим для різних розмірів кластера та надійним у часі, оскільки отримані поліпшення продуктивності не помітно погіршуються, коли збільшується або розмір кластера, або тривалість тестового циклу. Що стосується накладних витрат, M2_JAVA ввів мінімальні накладні витрати на вузли програми кластера (наприклад, приріст середнього використання пам'яті коливався в межах від 0,52% до 0,58%). Крім того, додаткові витрати в балансаторі навантаження були низькими (наприклад, приріст середнього використання пам'яті коливався між 6,19% і 6,49%), особливо

враховуючи скромні характеристики використовуваного вузла балансувача навантаження. З наведених вище результатів можна зробити висновок, що стратегія балансування навантаження, що знає GC, така як M2_JAVA.

Основні наукові та практичні результати доповідалися на наукових семінарах кафедри комп'ютерних наук та інформаційних технологій Хмельницького національного університету (2020 р.), на Всеукраїнській конференції молодих вчених «Актуальні проблеми комп'ютерних наук» (Хмельницький, 9-10 листопада 2020 року, Хмельницький національний університет).

Положення дипломної роботи магістра автором висвітлено в 1 науковій публікації [83],.

Перелік посилань

1. A. Haroon Malik, Bram Adams. Pinpointing the subsys responsible for the performance deviations in a load test. Software Reliability Engineering, 2010.

2. K. S. Ho and H. V. Leong. Improving the scalability of the corba event service with a multi-agent load balancing algorithm. Software: Practice and Experience, 32(5):417–441, 2002.

3. V. Hoorn, M. Rohr, W. Hasselbring, J. Waller, J. Ehlers, and D. Kieselhorst. Continuous Monitoring of Software Services: Design and Application of the Kieker Framework. 2009.

4. J. Howells. Software as a service (saas). Wiley Encyclopedia of Management, 2014.

5. C. C. Hui and S. T. Chanson. Flexible and extensible load balancing. Software: Practice and Experience, 27(11):1283–1306, 1997.

6. A. IIBA. guide to the business analysis body of knowledge (babok guide). International Institute of Business Analysis (IIBA), 2009. [81] International Data Corporation (IDC). Java : Two and a Half Years After the Acquisition. Technical Report August, 2012.

7. D. J. Kephart. The vision of autonomic computing. Computer, Jan. 2003.

8. J. C. Jeremy Singer, Gavin Brown, Ian Watson. Intelligent Selection of Application-Specific Garbage Collectors. In 6th international symposium on Memory management, 2007.

9. Z. M. Jiang. Automated analysis of load testing results. International Symposium on Software Testing and Analysis, page 143, 2010.

10. Z. M. Jiang, A. E. Hassan, G. Hamann, and P. Flora. Automated performance analysis of load tests. In International Conference on Software Maintenance, pages 125–134, 2009.

11. T. Kalibera. Replicating real-time Garbage Collection for Java. International Workshop on Java Technologies for Real-Time and Embedded Systems, 2009.

12. V. Kalogeraki, P. Melliar-Smith, L. E. Moser, and Y. Drougas. Resource management using multiple feedback loops in soft real-time distributed object systems. Journal of Systems and Software, 81(7):1144– 1162, 2008.

13. J. O. Kephart and D. M. Chess. The vision of autonomic computing. Computer, 36(1):41–50, 2003. 14. M. L. C. Briand, Y. Labiche. Using genetic algorithms for early schedulability analysis and stress testing in real-time systems. Genetic Programming and Evolvable Machines, 2006.

15. E. Laskowski, M. Tudruj, and R. Olejnik. Dynamic load balancing based on applications global states monitoring. International Symposium on Parallel and Distributed Computing, pages 11–18, 2013.

16. W. Y. Lee, S. J. Hong, J. Kim, and S. Lee. Dynamic load balancing for switch-based networks. Journal of Parallel and Distributed Computing, 63(3): 286–298, 2003.

17. P. Lengauer and H. M¨ossenb¨ock. The taming of the shrew: increasing performance by automatic parameter tuning for java garbage collectors. International Conference on Performance Engineering, pages 111–122, 2014.

18. E. Li, Mengchen. Dynamically Validating Static Memory Leak Warnings. International Symposium on Software Testing and Analysis, 2013.

19. Y. Liu, L. Wang, and S. Li. Research on self-adaptive load balancing in EJB clustering system. International Conference on Intelligent Systems and Knowledge Engineering, 2008.

20. J. M. S. Bayan. Automatic stress and load testing for embedded systems. Computer Software and Applications Conference, 2006.

21. L. M. Salehie. Self-adaptive software: Landscape and research challenges. ACM Transactions on Autonomous and Adaptive Systems, 2009.

22. S. G. Makridakis, S. C. Wheelwright, and R. J. Hyndman. Forecasting: Methods and Applications 3rd Edition. Wiley, 1997.

23. W. Manning. Scjp sun certified programmer for java 6 exam. Emereo Publishing, 2009.

24. F. Mao, E. Z. Zhang, and X. Shen. Influence of program inputs on the selection of garbage collectors. ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, page 91, 2009.

25. M. L. Massie, B. N. Chun, and D. E. Culler. The ganglia distributed monitoring system: design, implementation, and experience. Parallel Computing, 30(7):817–840, 2004.

26. C. Mateos, A. Zunino, and M. Campo. m-jgrim: a novel middleware for gridifying java applications into mobile grid services. Software: Practice and Experience, 40(4):331–362, 2010.

27. N. Mohamed and J. Al-Jaroodi. Midcloud: an agent-based middleware for effective utilization of replicated cloud services. Software: Practice and Experience, 45(3):343–363, 2015.

28. H. P. Robertson, R. Laddaga. Introduction: the First International Workshop on Self-Adaptive Software. International Workshop on SelfAdaptive Software, 2000.

29. K. Park and V. S. Pai. Comon: a mostly-scalable monitoring system for planetlab. ACM SIGOPS Operating Systems Review, 40(1):65–74, 2006.

30. T. Parsons and J. Murphy. Detecting Performance Antipatterns in Component Based Enterprise Systems. In International Middleware Doctoral Symposium, 2008.

31. H. Pham. Springer handbook of engineering statistics. Springer Science & Business Media, 2006.

32. G. Phipps. Comparing Observed Bug and Productivity Rates for Java and C ++. Software Practice and Experience, Apr. 1999.

33. F. Pizlo, E. Petrank, and B. Steensgaard. A study of concurrent realtime Garbage Collection. ACM SIGPLAN conference on Programming Language Design and Implementation, 2008.

34. P. Pongpaibool, A. Niruntasukrat, C. Issariyapat, K. Meesublak, C. Mokarat, and P. Aiumsupucgul. Netham-nano: A robust and scalable service-oriented platform for distributed monitoring. In Proceedings of the AINTEC 2014 on Asian Internet Engineering Conference, page 51, 2014.

35. P.-L. Poon, T. Tse, S.-F. Tang, and F.-C. Kuo. Contributions of tester experience and a checklist guideline to the identification of categories and choices for software testing. Software Quality Journal, 19(1):141– 163, 2011.

36. A. O. Portillo-Dominguez, M. Wang, D. Magoni, P. Perry, and J. Murphy. Load balancing of java applications by forecasting garbage collections. In International Symposium on Parallel and Distributed Computing, pages 127–134, 2014.

37. M. Puviani, G. Cabri, and F. Zambonelli. A taxonomy of architectural patterns for self-adaptive systems. In International Conference on Computer Science and Software Engineering, pages 77–85, 2013.

38. M. Rolia, Jerry, Artur Andrzejak. Automating enterprise application placement in resource utilities. Self-Managing Distributed Systems, 2003.

39. F. Ruggeri, R. Kenett, and F. W. Faltin. Encyclopedia of statistics in quality and reliability. John Wiley, 2007.

40. L. Rupprecht, A. Reiser, and A. Kemper. Dynamic load balancing in data grids by global load estimation. International Symposium on Parallel and Distributed Computing, pages 243–250, 2012.

41. M. S. Hangal. Tracking down software bugs using automatic anomaly detection. International Conference on Software Engineering, 2002.

42. D. Sanghi, P. Jalote, P. Agarwal, N. Jain, and S. Bose. A testbed for performance evaluation of load-balancing strategies for web server systems. Software: Practice and Experience, 34(4):339–353, 2004.

43. F. Siebert. Limits of parallel garbage collection. ACM SIGPLAN International Symposium on Memory Management, 2008.

44. J. Singer, R. E. Jones, G. Brown, and M. Luj´an. The economics of garbage collection. In ACM Sigplan Notices, volume 45, pages 103– 112. ACM, 2010.

45. R. G. Snatzke. Performance survey. Codecentric AG (2009), 2009.

46. S. Soman, C. Krintz, and D. F. Bacon. Dynamic selection of application-specific garbage collectors. In 4th international symposium on Memory management, New York, New York, USA, 2004. ACM Press.

47. W. Spear, S. Shende, A. Malony, R. Portillo, P. J. Teller, D. Cronk, S. Moore, and D. Terpstra. Making Performance Analysis and Tuning Part of the Software Development Cycle. DoD High Performance Computing Modernization Program Users Group Conference, 2009.

48. D. Spoonhower, J. Auerbach, D. F. Bacon, P. Cheng, and D. Grove. Eventrons: A Safe Programming Construct for High-Frequency Hard Real-Time Applications. ACM SIGPLAN Notices, 41(6):283–294, 2006.

49. Sun Microsystems. Memory Management in the Java HotSpot Virtual Machine. 2006.

50. Y. V. Garousi, L. C. Briand. Traffic-aware stress testing of distributed systems based on uml models. International Conference on Software Engineering, 2006.

51. T. Van Do and C. Rotter. Comparison of scheduling schemes for ondemand iaas requests. Journal of Systems and Software, 85(6):1400– 1408, 2012.

52. R. Van Renesse, K. P. Birman, and W. Vogels. Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining. ACM transactions on computer systems (TOCS), 21(2):164–206, 2003.

53. M. T. Vechev, E. Yahav, and D. F. Bacon. Correctness-preserving derivation of concurrent garbage collection algorithms. ACM SIGPLAN conference on Programming Language Design and Implementation, 2006.

54. M. Wegiel and C. Krintz. Dynamic prediction of collection yield for managed runtimes. ACM SIGPLAN Notices, Feb. 2009.

55. J. Weyns, Danny, M. Usman Iftikhar. Do external feedback loops improve the design of self-adaptive systems? a controlled experiment. International Symposium on Software Engineering for Adaptive and Self-Managing Systems, 2013.

56. M. H. Willebeek-LeMair and A. P. Reeves. Strategies for dynamic load balancing on highly parallel computers. IEEE Transactions on Parallel and Distributed Systems, 4(9):979–993, 1993.

57. P. R. Wilson. Uniprocessor Garbage Collection Techniques. In International Workshop of Memory Management, pages 1–42, 1992.

58. M. Woodside, G. Franks, and D. C. Petriu. The Future of Software Performance Engineering. Future of Software Engineering, 2007.

59. H. Wu, A. N. Tantawi, and T. Yu. A self-optimizing workload management solution for cloud applications. In International Conference on Web Services, pages 483–490, 2013.

60. F. Xian, W. Srisa-an, C. Jia, and H. Jiang. AS-GC: An Efficient Generational Garbage Collector for Java Application Servers. In European Conference on Object-Oriented Programming, 2007.

61. F. Xian, W. Srisa-an, and H. Jiang. Fortune Teller: Improving Garbage Collection Performance in Server Environment using Live Objects Prediction. ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications, 2005.

62. F. Xian, W. Srisa-an, H. Jiang, and A. Hall. Garbage Collection : Java Application Servers Achilles Heel. Science of Computer Programming, 70(2):89–110, 2008.

63. E. Xiao, Xusheng. Context-Sensitive Delta Inference for Identifying Workload-Dependent Performance Bottlenecks. International Symposium on Software Testing and Analysis, 2013.

64. W. Xingen Wang, Bo Zhou. Model-based load testing of web applications. Journal of the Chinese Institute of Engineers, 2013.

65. W. E. Yancey. Evaluating string comparator performance for record linkage. Statistical Research Division Research Report, 2005.

66. J. Yang, D. Evans, D. Bhardwaj, T. Bhat, and M. Das. Perracotta: mining temporal api rules from imperfect traces. International Conference on Software Engineering, 2008.

67. E. Yu, Kai. Practical isolation of failure-inducing changes for debugging regression faults. IEEE/ACM International Conference on Automated Software Engineering, 2012.

68. G. Yu, Tingting, Witawas Srisa-an. SimRacer: an automated framework to support testing for process-level races. International Symposium on Software Testing and Analysis, 2013.

69. J. Zhou and B. Demsky. Memory management for many-core processors with software configurable locality policies. In ACM SIGPLAN Notices, volume 47, pages 3–14. ACM, 2012.

70. Apache Camel. http://camel.apache.org/. Last accessed: 2016-03- 03.

71. Apache JMeter. http://jmeter.apache.org/. Last accessed: 2016- 03-03.

72. Apache Tomcat. http://tomcat.apache.org/. Last accessed: 2016- 03-03.

73. Bugzilla. https://www.bugzilla.org/. Last accessed: 2016-03-03.

74. Class Hashtable. http://docs.oracle.com/javase/7/docs/api/ java/util/Hashtable.html. Last accessed: 2016-03-03.

75. Class Vector. http://docs.oracle.com/javase/7/docs/api/java/ util/Vector.html. Last accessed: 2016-03-03.

76. Command Pattern. http://www.oodesign.com/command-pattern. html. Last accessed: 2016-03-03.

77. DaCapo Benchmark Suite - Programs and Sample Sizes. http://www. dacapobench.org/benchmarks.html. Last accessed: 2016-03-03.

78. Deep Dive on Java Virtual Machine (JVM) Javacores and Javadumps. http://www-01.ibm.com/support/docview.wss?uid= swg27017906&aid=1. Last accessed: 2016-03-03.

79. Defect Severity. http://softwaretestingfundamentals.com/ defect-severity/. Last accessed: 2016-03-03.

80. Dell PowerEdge M620 Blade Server. http://www.dell.com/ie/ business/p/poweredge-m620/pd. Last accessed: 2016-03-03.

81. Dell PowerEdge T420 Tower Server. http://www.dell.com/ie/ business/p/poweredge-t420/pd. Last accessed: 2016-03-03.

82. Dell PowerEdge VRTX Shared Infrastructure Platform. http:// www.dell.com/ie/business/p/poweredge-vrtx/pd. Last accessed: 2016-03-03.

83. Рибчинський Б. О., Доброловський В. В., Медвечук Н. К. Прогонозування завантаженості ресторану з використанням штучного інтелекту / Б. О. Рибчинський, В. В. Доброловський, Н. К. Медвечук // Збірник наукових праць за матеріалами XII всеукраїнської науково-практичної конференції «Актуальні проблеми комп’ютерних наук АПКН-2020». Хмельницький–2020. – С.247-248.

ДОДАТКИ

Додаток А

Ксерокопії наукових публікацій, виконаних при роботі над дипломною роботою магістра

Перелік наукових публікацій:

1. Рибчинський Б. О., Доброловський В. В., Медвечук Н. К. Прогонозування завантаженості ресторану з використанням штучного інтелекту / Б. О. Рибчинський, В. В. Доброловський, Н. К. Медвечук // Збірник наукових праць за матеріалами XII всеукраїнської науково-практичної конференції «Актуальні проблеми комп’ютерних наук АПКН-2020». Хмельницький–2020. – С.247-248.

УДК 004.4

Рибчинський Б. О., Доброловський В. В., Медведчук В. Ю.

Хмельницький національний університет

ПРОГНОЗУВАННЯ ЗАВАНТАЖЕНОСТІ РЕСТОРАНУ З ВИКОРИСТАННЯМ ШТУЧНОГО ІНТЕЛЕКТУ

Розроблений і реалізований метод отримування інформації на підставі аналізу та застосування методів машинного навчання з метою прогнозування завантаженості ресторану у різні періоди функціонування. Запропоновані методи та підходи можна застосовувати в практиному руслі а також в порівняних сферах із предметною областю.

Developed and implemented a method of obtaining information based on the analysis and application of machine learning methods to predict the load of the restaurant in different periods of operation. The proposed methods and approaches can be applied in practice, as well as in comparable areas with the subject area.

Збільшення попиту на туризм і конкуренція між напрямками були пов'язані з наслідками глобалізації. Напрямки конкурують на дедалі більш конкуруючих ринках, мало що відрізняє їх один від одного. Це змусило напрямки зосередитися на відмінності місця, використовуючи місцеві атрибути при запуску інноваційних, нових продуктів і брендів, що може допомогти створити більш унікальну пропозицію продажу.

Асимптотичні рамки, розглянуті тут, є, мабуть, найпростішою нетривіальною моделлю для матриці ознак. Це дозволяє обчислити, в основному будь-яку кількість відсотків, для будь-якого денозера інтересу. Недаментальні елементи матриці денозуючий білий шум, який є менш складними моделями, присутні ще зрозумілі і кількісні. Для наприклад, AMSE будь-якого показника на основі зниження особливого значення містить компонент через шумове забруднення в однині даних, і цей компонент визначає фундаментальну нижню межу. Кон'юнктуря, як результати обрахунків, розраховані в цій моделі, які не прикріплені до конкретного припущення про ранг (наприклад, константи, які визначають мінімакс) залишаються по суті правильними в більш складних моделях.

Спостереження, мінімум відвідувачів, які можемо спостерігати з цього сюжету, майже досягає нуля. Серед відвідувачів 20 відсотків.

Максимальна кількість відвідувачів становить від 55 до 60. спостерігаємо певні дуже високі значення (outlier) більше 60 і навіть більше, ніж 100 відвідувачів. 25-і процентильне та 75-і процентильні значення 13. Спостережeня, поширення бронювання вище, ніж у резервації HPG.

Існує велика кількість бронювань в HPG з відвідувачами розраховувати від 5 до 10. Є кілька застережень в HPG, де кількість відвідувачів більше 20 або навіть досягнення 40.

Це дослідження намагається зрозуміти взаємозв'язок між важливістю і продуктивністю (задоволенням) досвіду місцевої кухні, а також вивчити вплив досвіду їдальні на задоволення призначення. Що стосується важливих факторів для їдальні, найбільш важливим фактором для мандрівників є продовольчі культурні аспекти. Зокрема, туристи очікують, що місцева їжа буде відображати унікальні культурні аспекти для приготування їжі, презентації та зовнішнього вигляду. Дотримуючись культурних аспектів, соціальних аспектів, які є співробітниками та інші поведінки клієнтів, є другим за важливості фактором для іноземних мандрівників.

Крім важливості, респонденти також оцінювали своє задоволення кожним фактором. Як показує аналіз, заклади харчування демонструють високі показники якості продуктів харчування та продовольчих культурних аспектів. Однак якість продуктів харчування не оцінюється як така важлива, як продовольчі культурні аспекти. Отже, продовольчі культурні аспекти являють собою точку диференціації для місцевих ресторанів. Тобто, як цільові організації, так і заклади харчування повинні зосередитися на культурних аспектах харчування та позиціонувати місцеву кухню, підкреслюючи унікальні аспекти харчування. Інша важлива знахідка цього дослідження стосується соціальних аспектів. Незважаючи на те, що респонденти вважають соціальні аспекти важливими, рівень задоволеності був низьким для цього фактора. Цей результат передбачає, що заклади харчування повинні більше зосередитися на службовій поведінці співробітників. Нарешті, фізичне середовище було встановлено, щоб мають низький пріоритет для обідніх стабілізацій.

Машинне навчання має багато застосувань в ресторанах від допомоги кухарям на кухнях до прогнозування майбутніх продажів, що, в свою чергу, допомагає опрацювати вимоги до управління робочою силу та інвентаризацією.

Перелік посилань

1. Jang, SooCheong Shawn, and Young Namkung. 2009. Perceived quality, emotions, and behavioral intentions.
2. Application of an extended Mehrabian–Russell model to restaurants. Journal of Business Research 62: 451–60.
3. Jenkins, Ian, and Andrew Jones. 2003. A Taste of Wales–Blas Ar Gymru': Institutional malaise in promotingWelsh food tourism products. In Tourism and Gastronomy. London: Routledge, pp. 129–45.

Дипломна робота магістра

Модель, метод та засоби оптимізації продуктивності кластерних java-систем

Виконав:

Доброловський В. В.

Керівник:

Бармак О. В.

Актуальність теми

Кластерні середовища зазвичай використовуються в програмах на рівні підприємства, для досягнення більш швидкого часу відгуку та більшої пропускної здатності, ніж середовища окремих машин. Однак цей перехід від монолітної архітектури до розподіленої посилив складність цих застосувань, значно ускладнивши всі заходи, пов'язані з оптимізацією продуктивності таких кластерних систем. Отже, автоматичні методи необхідні для полегшення цих видів діяльності, пов'язаних з продуктивністю, є дуже схильними до помилок і забирають багато часу. Ця робота сприяє оптимізації продуктивності кластерних систем в Java, особливо спрямованих на широкомасштабне середовище.

Постановка задачі

Метою є покращення продуктивності кластерної системи, уникаючи впливів на роботу кластера через значний збір данних, що відбувається на окремих вузлах. Представлені експериментальні результати застосування цих методів до набору реальних програм, демонструють переваги, які ці методи приносять кластерній системі Java. Для досягнення поставленої мети визначені наступні задачі дослідження:

- Здійснити аналіз кластерних сиситем побудованих на java, засоби їх діагностики;
- Удосконалити метод, що базується на адаптивній системі автоматизації, яки розглядає загальні обмеження використання інструмента діагностики для ефективного використання при тестуванні продуктивності кластерних програм;
- Удосконалити метод адаптивного балансування навантаження для збору даних GC;
- Здійснити подальший розвиток технології з визначеним набором метрик, орієнтованих на GC, що характеризують поведінку програм Java з точки зору колекцій данних;
- Розробити відповідні засоби діагностики та провести експериментальні дослідження.

Положення новизни та інновації

- Удосконалено метод, що базується на адаптивній системі автоматизації (M1_JAVA), яки розглядає загальні обмеження використання інструмента діагностики для ефективного використання при тестуванні продуктивності кластерних програм.
- Удосконалено метод адаптивного балансування навантаження для збору даних GC (M2_JAVA).
- Здійснено подальший розвиток технології з визначеним набором метрик, орієнтованих на GC, що характеризують поведінку програм Java з точки зору колекцій данних. Крім того, на основі їхніх характеристик GC представлена класифікація програм у оцінених тестах Java.

Параметри JVM, пов’язані з пам’яттю

Параметр JVM	Опис
-XX: + DisableExplicitGC	Вимкнути явні запити на виконання GC
-XX: PermSize	Ініціалізуйте розмір PermGen
-XX: MaxPermSize	Визначте максимальний розмір PermGen
-Xms	Ініціалізуйте розмір купи
-Xmx	Визначте максимальний розмір купи

Положення новизни та інновації

- Фреймворк виконується одночасно з тестом продуктивності, повністю захищаючи тестер від складностей правильної конфігурації та використання діагностичного інструменту, так що тестувальнику потрібно лише взаємодіяти з інструментом тестування навантаження. Це дозволяє підвищити продуктивність тестера за рахунок зменшення зусиль та досвіду, необхідних для використання діагностичного інструменту. Внутрішньо фреймворк використовує набір політик для автоматичного управління різними ручними процесами, які зазвичай беруть участь у конфігурації та використанні інструменту діагностики продуктивності.
- Цей вдосконалений балансир навантаження обирає вузли програми, для яких не очікується, що в найближчому майбутньому відбуватиметься подія MaGC, як оптимальні вузли для вхідного робочого навантаження. Отже, ця стратегія може допомогти уникнути впливу на роботу кластера через появу подій MaGC в окремих вузлах. Внутрішньо M2_JAVA використовує алгоритм прогнозування MaGC, щоб визначити найкращий спосіб збалансувати навантаження між вузлами програми.

Апробація результатів дипломної роботи магістра та публікації.

Основні наукові та практичні результати *доповідалися* на конференціях:

- доповідь на тему «Прогонозування завантаженості ресторану з використанням штучного інтелекту» на Всеукраїнській конференції молодих вчених «Актуальні проблеми комп'ютерних наук» (Хмельницький, 9-10 листопада 2020 року, Хмельницький національний університет).

За темою кваліфікаційної роботи магістра автором виконано одну *наукову публікацію* [84].

Висновки дослідження

- Кластерні обчислення набули популярності як потужне та економічно ефективне рішення для паралельної та розподіленої обробки. Наприклад, корпоративні програми зазвичай розгортаються в кластерних екземплярах для досягнення вищої продуктивності системи в порівнянні з рішеннями для однієї машини. Однак також добре відомо, що використання кластерів суттєво збільшило складність систем, ще більше ускладнюючи всі дії, пов'язані з оптимізацією продуктивності кластерних систем.
- Довирішуючи вищезазначену проблему, ця робота сприяє оптимізації продуктивності кластерних систем на Java (що є однією з найбільш переважаючих технологій на рівні підприємства), особливо націлених на широкомасштабне середовище (оскільки ці середовища, швидше за все, страждають проблемами продуктивності). Зокрема, пропонуються дві методики для вирішення проблем ефективного виявлення проблем, пов'язаних з робочим навантаженням, та ефективного уникнення наслідків великого збору данних на продуктивність - двох важливих проблем, які типова кластерна система Java, ймовірно, зазнає у великих середовищах.
- Виявлення проблем продуктивності в кластерних середовищах є складним і трудомістким. Незважаючи на те, що дослідники розробляють інструменти діагностики для спрощення цього завдання, у цих інструментів існують різні обмеження, які перешкоджають їх ефективному використанню в тестуванні продуктивності. Для вирішення цих обмежень представлено нову адаптивну структуру, M1_JAVA, для автоматизації використання інструменту діагностики ефективності в кластерному середовищі. Внутрішньо M1_JAVA використовує набір адаптивних політик для управління різними наборами процесів, які зазвичай беруть участь у конфігурації та використанні інструменту діагностики.

- Експериментальні результати продемонстрували, що M2_JAVA може значно покращити час відгуку та пропускну здатність кластера (наприклад, середнє поліпшення продуктивності в RTAV tt коливалося між 28% та 31%). Поліпшення були досягнуті незалежно від використовуваного алгоритму балансування навантаження, що підтверджує загальність M2_JAVA. Результати також показали, що M2_JAVA є масштабованим для різних розмірів кластера та надійним у часі, оскільки отримані поліпшення продуктивності не помітно погіршуються, коли збільшується або розмір кластера, або тривалість тестового циклу. Що стосується накладних витрат, M2_JAVA ввів мінімальні накладні витрати на вузли програми кластера (наприклад, приріст середнього використання пам'яті коливався в межах від 0,52% до 0,58%). Крім того, додаткові витрати в балансаторі навантаження були низькими (наприклад, приріст середнього використання пам'яті коливався між 6,19% і 6,49%), особливо враховуючи скромні характеристики використовуваного вузла балансувача навантаження. З наведених вище результатів можна зробити висновок, що стратегія балансування навантаження, що знає GC, така як M2_JAVA.
- Основні наукові та практичні результати доповідалися на наукових семінарах кафедри комп’ютерних наук та інформаційних технологій Хмельницького національного університету (2020 р.), на Всеукраїнській конференції молодих вчених «Актуальні проблеми комп’ютерних наук» (Хмельницький, 9-10 листопада 2020 року, Хмельницький національний університет).

Щиро дякую за увагу!

Wed Dec 02 12:34:07 EET 2020, Петровський Сергій Степанович, Хмельницький національний університет, ХНУ

Anti-Plagiarism v-15.257

Максимальне співпадіння з одним документом 0.0%

Словники перевірки: en_US, ru_RU, ua_UA. **Помилок в документах: 8%**

ID: 82025 Назва: Модель, метод та засоби оптимізації продуктивності кластерних java-систем Додано в БД: 2020-12-02 Автора: Доброловський Володимир Володимирович Керівники: Бармак О.В. Консультанти: Опоненти:	Документ		Сумарний збіг по Базі Даних	
	Символи	Лексеми	Символи	Лексеми
	156627	1164	711 (0%)	11 (1%)

Джерело плагіату

ID	Опис	Наявність плагіату в документі	
		Символи	Лексеми

РІШЕННЯ КАФЕДРИ КОМП'ЮТЕРНИХ НАУК ТА ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ ПРО ДОПУСК КВАЛІФІКАЦІЙНОЇ РОБОТИ ДО ЗАХИСТУ

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

Назва: **Модель, метод та засоби оптимізації продуктивності кластерних java-систем**
Автор: **Доброловський В.В.**
Спеціальність: **122 Комп'ютерні науки**
Науковий керівник: **д.т.н., проф. Бармак О.В.**

Після аналізу звіту подібності зроблено такий висновок:

№	Висновок	Позначка про відповідність
1	Запозичення, виявлені в роботі, є законними і не є плагіатом (далі – зазначаються підстави віднесення запозичень до правомірних). Робота приймається до захисту.	**відповідає**
2	Виявлені запозичення не є плагіатом, розміщені в розділах, які не описують безпосередньо авторське дослідження, але кількість цитат перевищує обсяг, виправданий поставленою метою роботи (далі – зазначаються детальні та аргументовані підстави віднесення запозичень до правомірних). Робота приймається до захисту, але має бути відкоригована. Відкоригований варіант має бути поданий на кафедру за 2 дні до захисту, разом із заявою щодо самостійності виконання письмової роботи та ідентичності друкованої та електронної версії роботи	-
3	Робота містить навмисні текстові спотворення, передбачувані спроби укриття запозичень або інші прояви академічного плагіату. Робота містить фабрикацію або фальсифікацію даних. Робота не допускається до захисту.	-
4	Інше:	-

Підтвердження: Виявлена системою схожість не є плагіатом т.я. відноситься до власних назв та списку літератури, що не є безпосередньо авторським дослідженням, та складає 3%.

__03.12.2020_____
Дата

Підпис керівника

Підпис завідувача кафедри

Хмельницький національний університет

Кафедра комп'ютерних наук та інформаційних технологій

ВІДГУК ОПОНЕНТА

на дипломну роботу магістра

Магістра *гр. КНм-19-1 Доброловський Володимир Володимирович*

На тему: *Модель, метод та засоби оптимізації продуктивності кластерних java-систем.*

1. Актуальність і значення теми

Кластерні середовища зазвичай використовуються в програмах на рівні підприємства, для досягнення більш швидкого часу відгуку та більшої пропускної здатності, ніж середовища окремих машин. Однак цей перехід від монолітної архітектури до розподіленої посилив складність цих застосувань, значно ускладнивши всі заходи, пов'язані з оптимізацією продуктивності таких кластерних систем. Отже, автоматичні методи необхідні для полегшення цих видів діяльності, пов'язаних з продуктивністю, є дуже схильними до помилок і забирають багато часу. Ця робота сприяє оптимізації продуктивності кластерних систем в Java, особливо спрямованих на широкомасштабне середовище.

2. Оцінка якості та достовірності проведених досліджень.

Отримані результати добре співвідносяться з результатами, наведеними в наукових роботах і довідниках.

3. Оцінка запропонованих заходів та пропозицій, практичної цінності та ефективності.

Проведені дослідження представляють науково-технічну цінність, є ефективним дослідженням в галузі оптимізації продуктивності кластерних java-систем , їх можна використати з метою оптимізації продуктивності кластерних систем в Java, особливо спрямованих на широкомасштабне середовище.

4. Загальний висновок та оцінка

Робота виконана в повному обсязі. Досліджені та проаналізовані дані за допомогою комплексу входять в рамки допустимих відхилень. Пояснювальна записка оформлена в відповідності з нормами. Відмічені недоліки не знижують цінності дипломної роботи. За своєю структурою, практичними цінностями, поставленій меті та вирішеними задачами робота відповідає вимогам вищої школи і вимогам, що пред'являються до освітньо-кваліфікаційного рівня «магістр», а її автор Доброловський В.В. заслуговує присвоєння кваліфікації магістра з комп'ютерних наук та інформаційних технологій.

Робота заслуговує на оцінку «Задовільно».

Опонент Диха О.В., д.т.н., проф., зав.каф. ТАМ ХНУ