ПРАВИТЕЛЬСТВО РОССИЙСКОЙ ФЕДЕРАЦИИ
НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ
«ВЫСШАЯ ШКОЛА ЭКОНОМИКИ»
Факультет компьютерных наук
Образовательная программа «Программная инженерия»

УДК 004.25

СОГЛАСОВАНО	УТВЕРЖДАЮ Академический руководитель образовательной программы «Программная инженерия» старший преподаватель департамента программной инженерии
Директор лаборатории	
Бугаенко Егор Георгиевич	Н.А. Павлочев

Отчет
по исследовательскому курсовому проекту

на тему: Сравнительный анализ скорости работы подсистемы управления памятью в Java и C++

по направлению подготовки бакалавров 09.03.04 «Программная инженерия»

Выполнил
студент группы БПИ234
образовательной программы
09.03.04 «Программная инженерия»

Лопаткин Пётр Сергеевич

29.03.2025

Москва 2025

РЕФЕРАТ

Отчет 27 с., 19 источн. , 1 прил., 1 таб.

СРАВНИТЕЛЬНЫЙ АНАЛИЗ СКОРОСТИ РАБОТЫ ПОДСИСТЕМЫ УПРАВЛЕНИЯ ПАМЯТЬЮ В JAVA И C++

Объектом исследования являются механизмы управления памятью в языках Java и C++.

Цель проекта — узнать какая подсистема памяти быстрее и почему её подход управления памятью быстрее.

Метод исследования — анализ текущих подходов изучения скорости работы подсистем памяти, изучение научной литературы и документации языков Java и C++, написание нагрузочных тестов и их дальнейший анализ.

Результат работы и актуальность — результатом исследования является разработанная коллекция нагрузочных бенчмарков, направленных на тестирование скорости и эффективности работы подсистем управления памятью в Java и C++. Полученные данные проанализированы с учетом особенностей каждой модели управления памятью. В результате даны практические рекомендации по выбору подхода в зависимости от требований к производительности, надежности и контролю над ресурсами. Актуальность работы обусловлена широким применением обеих технологий в промышленной разработке и необходимостью осознанного выбора подходящих инструментов в зависимости от специфики задачи.

Применимость результатов — ценность проведённого исследования заключается в актуальной задаче оптимизации управления ресурсами. В настоящее время разрабатывается язык программирования EOlang, для которого необходимо определить наиболее эффективный подход к управлению памятью. Кроме того, результаты могут быть полезны в промышленной разработке — особенно в случаях, когда заранее известны типы задач, и требуется выбрать язык программирования, обеспечивающий наилучшую производительность и эффективность работы с памятью.

СОДЕРЖАНИЕ

ТЕРМИНЫ И ОПРЕДЕЛЕНИЯ

В настоящем отчете о НИР применяются следующие термины с соответствующими определениями:

Бенчмарк	Тест для измерения производительности программного или аппаратного обеспечения.
EOlang (Elegant Objects Language)	Экспериментальный язык программирования разрабатываемый на принципах чистого объектно—ориентированного подхода компанией Huawei)
Сборщик мусора (Garbage Collector)	Механизм автоматического управления памятью, который освобождает неиспользуемые объекты во время выполнения программы, а также перемещает их между областями памяти.
Область памяти (поколение памяти)	Логическая часть кучи в Java, используемая сборщиком мусора для разделения объектов по времени жизни (например, Young Generation и Old Generation).
Аллокация	Процесс выделения памяти для хранения данных во время выполнения программы.
Деаллокация	Процесс освобождения ранее выделенной памяти.
Поток	Независимая последовательность выполнения операций в программе, которая может работать параллельно с другими потоками, обеспечивая многозадачность
Оператор	Символ или конструкция в языке программирования, которая выполняет определённую операцию над данными
Менеджер памяти	Компонент языка программирования отвечающий за управление памятью программы.
Инлайн (inline)	способ вставки тела функции прямо в место её вызова, чтобы избежать накладных расходов на переход к функции

Синтетическое тестирование	метод тестирования, при котором создаются искусственные данные или сценарии для оценки работы системы в специфических условиях, не всегда отражающих реальную эксплуатацию.
Стек	Структура данных реализующая принцип первый вошел, последний вышел
Ассемблерные инструкции	это низкоуровневые команды, которые управляют процессором на основе его архитектуры, выполняя операции над регистрами, памятью и другими компонентами процессора.
Компиляция	процесс преобразования исходного кода программы, написанного на языке высокого уровня, в машинный код или промежуточный код, который может быть выполнен на компьютере.
Байт—код	промежуточный код, генерируемый компиляторами для языков программирования (например, Java), который затем исполняется виртуальной машиной или интерпретатором, а не напрямую процессором.
Код	Набор инструкций, написанных на языке программирования, которые выполняются компьютером для решения определенной задачи.
JIT—компиляция	техника компиляции, при которой код компилируется в машинный код непосредственно перед его выполнением, что позволяет повысить производительность программы.
Гетерогенная система	система, состоящая из разнородных компонентов (например, разных процессоров, устройств или сред выполнения).

ПЕРЕЧЕНЬ СОКРАЩЕНИЙ И ОБОЗНАЧЕНИЙ

EOlang	Elegant Objects Language(Экспериментальный язык программирования разрабатываемый на принципах чистого объектно-ориентированного подхода компанией Huawei)
GC	Garbage Collector(сборщик мусора)
ЯП	Язык программирования
JIT	(Just—In—Time) аббревиатура, обозначающая метод компиляции, при котором исходный код компилируется в машинный код в процессе выполнения программы, а не заранее.
МС(мс)	миллисекунды

ВВЕДЕНИЕ

В разработке высокопроизводительных приложений есть один важный аспект — управление памятью. В C++ разработчики используют функции malloc/free (или оператор new/delete) для ручного выделения и удаления памяти, в то время как в Java для утилизации памяти используется автоматический сборщик мусора. Хотя оба языка используют различные стили управления памятью, общим для них является необходимость эффективного выделения и удаления кучи и быстрого доступа к памяти. Возрастающая сложность современных приложений и постоянно растущие требования к производительности спровоцировали активное изучение методов управления памятью. Таким образом, понимание характеристик производительности систем с ручным и автоматическим управлением памятью необходимо для выбора подходящей технологии для конкретного приложения.

Предметом исследования являются эффективность и производительность использования различных методов управления памятью, включая кучи, сборщики мусора (Java) и ручное управление (C++). Исследование направлено на встроенные реализации менеджеров памяти. Основной целью данного исследования является ответить на два исследовательских вопроса: какой менеджер памяти работает быстрее и почему он работает быстрее.

Для достижения данной цели необходимо решить блок задач. Сначала необходимо изучить особенности работы каждого из способов работы с памятью. Затем необходимо проанализировать уже написанные исследования на схожие темы и выяснить какие методы сравнения и оценивания в них использовались. Далее необходимо создать таблицу со всеми найденными бенчмарками и на основе неё создать собственный набор тестов, которые помогут точно оценить работу сравниваемых подсистем памяти. После написания бенчмарков необходимо убедиться в их условной идентичности, так как иначе мы будем сравнивать скорость работы абсолютно разных вещей. В конце необходимо составить подробный отчёт на основе данных полученных из бенчмарков.

Оригинальность данного исследования заключается в том, что на данный момент никто явно не проводил сравнения двух данных подсистем памяти. Существуют исследования которые сравнивают скорости работы самописных менеджеров памяти в рамках одного языка, либо которые сравнивают общие параметры языков. Данное исследование поможет точно ответить на вопрос о скорости того или иного языка в области работы с памятью.

Помимо этого на данный момент идёт разработка языка EOlang и разработчикам

необходимо убедиться, что из двух наиболее популярных типов управления памятью они выбирают оптимальный для своего языка исходя из его идей и концепций.

Таким образом, данное исследование решает актуальные научные задачи, связанные со скоростью работы двух популярных видов управления памятью.

ОСНОВНАЯ ЧАСТЬ

1. Обзор исследований скорости подсистем памяти

Очень важно изучить уже схожие исследования, чтобы понять как именно проводится анализ подсистем памяти и какие аспекты при этом выделяют в обоих языках. Если этого не сделать то крайне сложно будет выделить аспекты для сравнения в обоих языках. Однако исследований на данную тему очень мало, т.к большинство данных исследований проводились внутри компаний для повышения эффективности их приложений. В свободном доступе можно найти только их готовые решения и сравнения энтузиастов.

1.1 Обзор исследований подсистемы памяти C++

Исследование Daniel Haggander и Per Lidkn (2001) [1] фокусируется на автоматизации управления динамической памятью в C++. Внутри этого исследования они описали методы, измерения и сравнения их подходов работы с памятью со встроенным решением. Для сравнения они использовали рекурсивное древовидное выделение памяти. На каждом шаге рекурсивно выделялись объекты в дереве на разной глубине. Это позволяет эффективно оценить работу систем управления памятью в условиях с высокой временной локальностью, где объекты многократно создаются и уничтожаются, что характерно для реальных приложений. Такой подход позволяет выявить слабые места в производительности и масштабируемости различных методов управления памятью, особенно в случае с динамическими структурами данных.

1.2 Обзор исследований подсистемы памяти Java

В небольшом исследовании Ionut Balosin [2] очень подробно описываются сценарии тестирования самописных сборщиков мусора для Java, которые очень хорошо помогают понять основные аспекты на которые надо обратить внимание при сравнении подсистем памяти Java и C++. В исследовании анализируются такие аспекты, как выделение и освобождение памяти, а также влияние размера объектов на производительность.

Также в исследовании Carlos Daniel Oliveira Goncalves [3] было уже упоминание готового открытого решения для тестирования памяти в Java. Это позволило далее найти подобные решения с открытым исходным кодом.

2. Поиск готовых бенчмарков с открытым исходным кодом

Как ни странно, исследования очень важны для нас, но их слишком мало чтобы сделать полноценную коллекцию бенчмарков для сравнения именно встроенных подсистем памяти. Именно поэтому далее было принято решение искать бенчмарки, которые позволят увидеть конкретные примеры кода и на основе их и исследований попробовать составить свои бенчмарки.

После тщательного изучения источников с открытым кодом было составлено описание найденных бенчмарков с открытым исходным кодом. В расчёт брались годы в которые данный бенчмарк поддерживался и обновлялся, количество звёзд на платформе GitHub (для оценки его популярности и относительной полезности), количество строк кода(без учета комментариев, переносов строк и прочих неактивных частей), в каких исследованиях он использовался и наконец версия которую мы рассматривали на момент написания нашего бенчмарка. Итоговый набор бенчмарков с зафиксированной версией выложен в открытый доступ и сохранен на GitHub (lopata29435/Comparison-of-Java-vs.-C-Memory-Management-Performance: Repository for research: "Comparison of Java vs. C++ Memory Management Performance").

2.1 Бенчмарки для C++

Для C++ бенчмарки больше ориентированы на низкоуровневую производительность, а также они все синтетические в отличии от Java. Данные бенчмарки предоставляют более детализированный контроль над памятью, что делает их удобными для анализа ручного управления памятью.

2.1.1 Mimalloc

Один из наиболее популярных наборов бенчмарков для анализа malloc—реализаций [4], актуальный с 2019 года по наше время. Этот инструмент тестирует производительность выделения памяти в разных сценариях. Довольно большое количество звёзд на GitHub(382) также показывает, что бенчмарк крайне полезен для оценки работы менеджера памяти. Из этого бенчмарка удалось почерпнуть основные сценарии тестирования которые будут описаны ниже.

2.1.2 STREAM

Бенчмарк [5], направленный на измерение пропускной способности памяти, включая операции чтения и записи. Его длительная история использования (1995–2022)

делает его эталоном для тестирования производительности подсистем памяти. Он также имеет хороший уровень доверия благодаря 348 звёздам на Github. Из него не удалось получить нужных нам сценариев тестирования, так как он направлен немного на другую область анализа менеджера памяти C++. Однако он может быть очень полезен в продолжении исследования.

2.1.3 Adobe Performance Benchmarks

Эти тесты [6] были разработаны для оценки производительности алгоритмов в контексте использования памяти. Хотя они больше не поддерживаются, некоторые их идеи используются в современных исследованиях, например в Mimalloc.

2.2 Бенчмарки для Java

Для Java существует ряд исследований и бенчмарков, сосредоточенных на анализе производительности сборщиков мусора (Garbage Collectors) и управления памятью в различных сценариях. Как правило они все не синтетические, то есть создают нагрузку основываясь на реальных задачах.

2.2.1 DaCapo Benchmarks

Набор бенчмарков [7], разработанный для тестирования производительности Java—приложений с интенсивной нагрузкой на память. Исследование Islam и др. [4] использовало эти бенчмарки для анализа эффективности современных GC в среде JDK 20. Этот набор бенчмарков поддерживается с 2006 года по наше время и имеет 162 звезды. Однако нам он не подходит, так как он тестирует память на основе уже существующих приложений, в то время как мы хотим провести синтетическое тестирование.

2.2.2 Renaissance Benchmarks

Это более современные бенчмарки [8], также как и DaCapo фокусируются на реальных приложениях и нагрузках. Исследования, такие как работа Carlos Daniel Oliveira Goncalves [3], показывают, как различные сборщики мусора справляются с высокоуровневыми задачами, включая вычисления в больших данных. Данный бенчмарк был представлен в 2019 году и продолжает быть актуальным, имеет 312 звёзд и является хорошим примером, но нам он также не подходит из—за несинтетической природы тестов.

2.2.3 GarbageCollectorBenchmark

Синтетический бенчмарк [9], созданный для проверки производительности GC. Этот инструмент позволяет протестировать такие аспекты, как время паузы и производительность при интенсивном выделении и освобождении памяти. Этот бенчмарк был создан энтузиастом в 2015 году и он крайне мал и не имеет доверия из—за отсутствия звёзд, но не отметить его тоже нельзя было, так как он один из немногих бенчмарков с открытым исходным кодом.

2.2.4 JVM Performance Benchmarks

Данный набор [10] включает тесты на ложное разделение, скалярное замещение и другие аспекты работы JVM. Этот бенчмарк напрямую нам не подходит, но может помочь в продолжении исследования в будущем.

2.3 Итоговый обзор исследований и открытого кода

Изучив существующие наборы бенчмарков и исследований, можно сделать ряд ключевых выводов. Для Java почти нет синтетических тестов менеджера памяти с открытым исходным кодом, в то время как C++ имеет значительно больше таких тестов. Бенчмарки Java в большей степени ориентированы на реальное применение или очень специфические случаи использования, в то время как бенчмарки C++ стремятся к комплексным синтетическим тестам, проверяющим отдельные области управления памятью. Также бенчмаркинг в Java в значительной степени опирается на фреймворк JMH, в то время как бенчмарки C++ функционируют без соответствующего фреймворка.

Несмотря на различия в реализации, базовые механизмы тестирования — выделение кучи и операции чтения/записи в обоих языках в значительной степени идентичны. Хотя рассматриваемые бенчмарки удобны для сравнения производительности управления памятью, они были разработаны в первую очередь для сравнения конкретных реализаций, а не для прямого сравнения языков. Это создает проблему для сравнения производительности C++ и Java. Чтобы обеспечить равноценное и содержательное сравнение, мы нормализуем все тестовые примеры и организуем их таким образом, чтобы сравнить производительность управления памятью в эквивалентных условиях. Таким образом, мы можем напрямую сравнить эффективность каждого подхода.

3. Анализ специфики работы подсистем памяти

В этом разделе мы рассмотрим основные особенности операций с памятью в языках Java и C++. Прежде всего, нам необходимо понять ключевые аспекты систем управления памятью в каждом языке. Они обеспечивают поддержку трех примитивных типов памяти — стека, кучи и статической области, мы рассмотрим только стек и кучу, так как статическая память почти не меняется во время выполнения программы. Также мы учтем особенности многопоточной работы.

3.1 Стек

3.1.1 Особенности работы со стеком в C++

- Каждый вызов функции добавляет новый стековый фрейм, который содержит все её локальные переменные [11].
- Стековые фреймы автоматически удаляются при возврате из функции (механизм LIFO) [11].
- Размер стека ограничен, и может произойти переполнение из—за глубокой рекурсии или чрезмерного использования локальных переменных [12].
- Каждый поток имеет свой собственный стек, что обеспечивает безопасность локальных данных [13].

3.1.2 Особенности работы со стеком в Java

- Похож на C++, но управляется JVM, с возможностью настройки размера стека [14].
- Локальные примитивные переменные и ссылки на объекты хранятся в стеке, однако сами объекты располагаются в куче [15].
- Каждый поток имеет отдельный стек, что обеспечивает безопасность потоков [16].

3.2 Куча

3.2.1 Особенности работы с кучей в C++

- Используется для динамического выделения памяти (new, malloc) [17].
- Выделенные объекты сохраняются до явного освобождения (delete, free) [17].

- Могут возникать гонки данных, когда несколько потоков обращаются к общей памяти без синхронизации [17].

3.2.2 Особенности работы с кучей в Java

- Объекты выделяются в куче и автоматически управляются сборщиком мусора (GC) [18].
- Куча делится на Молодое поколение и Старое поколение(Young Generation и Old Generation) для оптимизации производительности GC
- Java гарантирует, что освобожденные объекты становятся недоступными, предотвращая ошибки использования после освобождения памяти.
- GC может вызывать паузы, что влияет на производительность в реальном времени [19].

3.3 Непрерывность памяти и фрагментация

- В C++ разработчики контролируют компоновку памяти(есть возможность делать самописные аллокаторы), то есть компоновка объектов в памяти зависит от заранее известного алгоритма, который всегда срабатывает одинаково. Фрагментация в C++ происходит при частом выделении и удалении блоков памяти. Частично с этим помогают справляться пользовательские аллокаторы, но саму проблему решить невозможно, можно только оптимизировать.
- В Java объекты располагаются в куче, место для которых определяется сборщиком мусора. Массивы примитивных типов хранятся непрерывно, но массивы объектов содержат рассеянные ссылки (Oracle GC Tuning Guide). В Java фрагментация минимизируется благодаря сборщику мусора, который сжимает память, однако в старых версиях она могла приводить к серьезным задержкам работы GC.

3.4 Отдельные аспекты многопоточности

- В C++ аллокация и деаллокация могут блокировать потоки при частом вызове, так как куча для всех потоков общая и она не синхронизируется.
- В Java GC блокирует выделение памяти только в крайних случаях, но доступ к общим объектам требует синхронизации, но всю эту работу на себя берёт GC(shipilev.net).

4. Описание созданных бенчмарков и их специфики

4.1 Fixed Size Allocation

- Цель: Оценить производительность выделения памяти фиксированного размера и способность выделять однотипные объекты без фрагментации.
- Реализация: Каждая итерация выполняет выделение памяти фиксированного размера.
- Метрики: Время выделения памяти.

4.2 Complex Object Allocation

- Цель: Оценить затраты на выделение памяти для сложных и короткоживущих объектов.
- Реализация: Выделяется память для объектов, содержащих вектор (C++) или ArrayList (Java) с указанным размером. При этом очищение происходит раз в 10 000 раз.
- Метрики: Общее время выполнения.

4.3 Variable Size Allocation

- Цель: Проверить производительность выделения памяти при различных размерах блоков.
- Реализация: Размеры блоков генерируются заранее с помощью Python—скрипта. Далее бенчмарки на обоих языках выделяют одинаковую случайную последовательность блоков памяти.
- Метрики: Время выделения памяти и очищения.

4.4 Concurrent Multithreaded Allocation Performance

- Цель: Смоделировать реальные условия многозадачности для выделения памяти.
- Реализация: Основано на тестах бенчмарка Variable Size Allocation, но вместо однопоточного выполнения несколько потоков выполняют выделение памяти одновременно с различными размерами блоков.
- Метрики: Время выделения памяти и очищения.

4.5 Fragmentation Test Random Allocation Release

- Цель: Смоделировать и проанализировать эффекты фрагментации памяти.

- Реализация: Каждая итерация выполняет выделение памяти фиксированного размера, при этом шанс очистки памяти составляет 50 к 10 000 000.
- Метрики: Общее время выполнения.

4.6 Recursive Memory Allocation

- Цель: Проверить производительность рекурсивного выделения памяти.
- Реализация: Каждая итерация выполняет рекурсивное выделение памяти с заданной глубиной и размером на каждом уровне.
- Метрики: Время выделения памяти и очищения.

4.7 Общие замечания по бенчмаркам

Чтобы убедиться, что наши бенчмарки точно измеряют накладные расходы на выделение и освобождение памяти, мы приняли меры для предотвращения агрессивной оптимизации компилятора, которая может устранить или изменить наши тестовые случаи. Современные компиляторы могут распознавать неиспользуемые или избыточные операции с памятью и оптимизировать их. Чтобы этого избежать, мы использовали несколько техник:

- Использование квалификаторов volatile и барьеров памяти, чтобы заставить компилятор сохранить операции с памятью.
- Введение искусственных зависимостей от выделенной памяти (например, выполнение ненужных вычислений или запись результатов в структуры, видимые извне).

Также надо сказать, что все бенчмарки стараются быть максимально похожи друг на друга в обоих языках, но не всегда это возможно. Мы старались анализировать каждый выбранный контейнер и инструкцию, но идеальной идентичности достичь невозможно. Так например мы никак не можем явно вызвать очищение памяти, в лучшем случае мы можем использовать инструкцию System.gc() которая не гарантировано вызовет сборщик мусора. Также важным аспектом является то, что языки имеют свои способы оптимизации кода и после оптимизации нет возможности узнать как именно был интерпретирован код. Если на C++ у нас есть возможность посмотреть ассемблерные инструкции после компиляции, то в случае Java мы можем увидеть только байт код для виртуальной машины, который частично может быть преобразован в ассемблерный код благодаря JIT компиляции. Поэтому тут тоже для условного равенства языков мы использовали максимально возможные оптимизации. C++ компилировался с флагом -o3, а в Java мы не отключали JIT компиляцию и какие либо оптимизации. О других ограничениях будет

рассказано в разделе далее. Ниже приведены название бенчмарков на английском языке так как именно такими названиями представлены эти бенчмарки в коде. Сам код вы можете найти и на платформе GitHub, также там есть всё чтобы развернуть у себя локально эти бенчмарки.(lopata29435/Comparison-of-Java-vs.-C-Memory-Management-Performance-Benchmarks)

5. Ограничения

- Синтетическая среда: Бенчмарки основаны на контролируемых и зачастую идеализированных условиях и могут не охватывать всей полноты поведения реальных приложений. В прикладной среде могут наблюдаться смешанные типы нагрузки и неконтролируемое поведение, например, непредсказуемые шаблоны использования памяти и динамически меняющиеся требования к ресурсам.
- Оптимизации компилятора и JVM: Результаты производительности могут существенно варьироваться в зависимости от оптимизаций компилятора C++ и различных конфигураций JVM в Java. Например, уровни оптимизации, режимы работы сборщика мусора и параметры выполнения могут по—разному влиять на производительность. Эти параметры обычно настраиваются под конкретное приложение и не переносятся напрямую на другие системы.
- Покрытие тестов: Несмотря на то, что набор тестов охватывает широкий спектр сценариев выделения памяти, он не включает все возможные случаи, встречающиеся в продуктивных системах. В реальных системах могут происходить сложные взаимодействия между выделением, освобождением памяти и сборкой мусора, которые не полностью моделируются при синтетическом тестировании.
- Учет утечек памяти: На практике могут возникать проблемы, связанные с утечками памяти или неэффективным управлением ресурсами, которые, как правило, не моделируются в искусственных бенчмарках. Эти проблемы могут повлиять не только на производительность, но и на стабильность, однако в тестах предполагается идеальное управление памятью без утечек.
- Изменчивость объема памяти: Объем выделяемой памяти в тестах фиксирован или задается заранее с помощью скриптов. В реальных приложениях объем потребляемой памяти может быть как значительно меньше, так и существенно больше. Динамический характер использования памяти может влиять на скорость и эффективность выделения иначе, чем в условиях контролируемого тестирования.

- Аппаратные и внешние факторы: Бенчмарки, как правило, запускаются на проверенном оборудовании и с заданными конфигурациями операционной системы. Однако в реальных условиях окружения могут быть гетерогенными с различными уровнями производительности оборудования, разной степенью загрузки системы и другими внешними факторами, влияющими на производительность.
- Настройки размера кучи и стека JVM: В тестах на Java использовались параметры по умолчанию для размера кучи и стека, заданные через опции JVM, чтобы избежать переполнения памяти и стека. Мы указали увеличенные максимальные размеры кучи и стека, чтобы JVM имела достаточно ресурсов для работы с большими объемами памяти без потерь производительности. Однако в обычных или менее ресурсоемких приложениях такие настройки используются редко.
- Мы не можем часто вызывать сборщик мусора: Сборщик мусора при вызове останавливает все потоки для безопасного освобождения памяти. Поэтому слишком частые вызовы System.gc() приводят к огромным затратам ресурсов и времени. По этой причине мы ввели ограничения на частоту вызовов сборщика мусора.
- Фиксированные типы данных: В каждом бенчмарке надо было обеспечить одинаковые размеры выделяемой памяти. Для этих целей в C++ были использованы фиксированные типы данных int8_t, а в Java byte, которые эквивалентны 8 битам или 1 байту.

6. Результаты

Все бенчмарки запускались один раз, если не указано иное. Java—бенчмарки, выполненные с использованием JMH, включали 3 итерации разогрева и 5 итераций измерения. Запуски проводились в автоматическом режиме на отдельном сервере черз GitHub actions, а также на локальном устройстве. Аппаратно мы никак не ограничивали бенчмарки, но замеры проводили в абсолютно идентичных условиях. Эти значения были определены экспериментально, и их изменение не дало заметной разницы в результатах. Ниже приведено описание того, как проводились тесты:

- Fixed Size Allocation (Выделение памяти фиксированного размера): Выполнялось 100 000 000 итераций, в каждой из которых выделялся массив длиной 100 байт.
- Complex Object Allocation (Выделение памяти под сложные объекты): Выполнялось 50 000 000 итераций, каждый объект содержал 1 000 элементов. Удаление объектов происходило один раз каждые 10 000 итераций.

- Variable Size Allocation (Выделение памяти переменного размера): Выполнялось один раз. Размеры аллокаций в байтах загружались из внешнего файла, содержащего 1 000 000 записей, каждая из которых указывает длину массива байтов для выделения.
- Concurrent Multithreaded Allocation Performance (Параллельное многопоточное выделение памяти): Выполнялось один раз. Размеры аллокаций в байтах загружались из внешнего файла (200 000 записей), каждая запись — длина массива байтов. Аллокации выполнялись одновременно в 40 потоках.
- Fragmentation Test (Тест фрагментации — случайное выделение и освобождение памяти): Выполнялся один раз, использовались 10 000 000 аллокаций и соответствующие шаблоны освобождения памяти, загруженные из внешних файлов. Вероятность освобождения памяти, указанная в конфигурационном файле, варьировалась от 50 до 10 000 000.
- Recursive Memory Allocation (Рекурсивное выделение памяти): Выполнялось один раз с глубиной рекурсии 100 000 и размером аллокации 32 768 байт на каждый вызов.

По итогам этих тестов были получены следующие результаты, которые можно увидеть в Таблице 1. Отрицательные значения в данном случае означают, что Java отработало на указанное количество процентов медленней.

Бенчмарк	C++ (мс)	Java (мс)	Процентное преимущество Java над C++
Fixed Size Allocation	4502.77	5536.21	-22.95%
Complex Object Allocation	8162.08	18685.64	-128.93%
Variable Size Allocation	822.81	145.69	82.29%
Concurrent Multithreaded Allocation Performance	7088.08	5099.10	28.06%
Fragmentation Test Random Allocation Release	5738.97	18361.38	-219.94%

Recursive Memory Allocation	13808.80	526.73	96.19%

Таблица 1. Результаты выполнения бенчмарков

7. Выводы

Наконец основываясь на полученных результатах можно сделать значительные выводы. Для удобства мы рассмотрим каждый полученный бенчмарк отдельно.

7.1 Fixed Size Allocation (Выделение памяти фиксированного размера)

Для этого теста разница во времени между C++ и Java незначительна, но всё же присутствует. Скорее всего, в какой—то момент в Java срабатывает сборщик мусора, который начинает перемещать объекты в старое поколение, что и вызывает замедление. В остальном сама операция выделения памяти остаётся примерно одинаковой, особенно при небольших объёмах, где разница практически незаметна.

7.2 Complex Object Allocation (Выделение памяти под сложные объекты)

В этом тесте наблюдается огромная разница по времени. Нужно понимать, что удаление объектов происходит раз в 10 000 итераций. Это связано с тем, что В Java мы явно вызываем сборщик мусора, и если бы мы попытались вызвать сборщик мусора на каждой итерации, то бенчмарк попросту бы не запустился. Это было определено экспериментальным путём. Такое поеведение обусловлено тем, что несмотря на то что сборщик мусора может проигнорировать нашу инструкцию, он всё равно будет срабатывать чаще, а значит будет блокировать потоки и перемещать объекты между поколениями памяти, что катастрофически замедляет работу программы.

7.3 Variable Size Allocation (Выделение памяти переменного размера)

В этом случае разница во времени не такая заметная, значения в целом очень малы. Мы не освобождаем память на каждой итерации, а делаем это в самом конце, поэтому в Java GC не вызывается постоянно, потоки не блокируются и объекты не перемещаются в старое поколение. Это ускоряет выполнение, так как освобождение в молодом поколении (New Gen) происходит практически мгновенно, в отличие от более сложного процесса аллокации в C++. Скорее всего, именно это и создаёт разницу во времени.

7.4 Concurrent Multithreaded Allocation Performance (Многопоточное выделение памяти)

В многопоточном варианте этого теста ситуация становится интереснее, здесь разница уже около 2 секунд. Это связано с тем, что в Java сборщик мусора сам решает

куда и как распределять память даже в многопотоке и он старается оптимизировать этот процесс, в то время как в C++ все потоки работают с общей кучей. Это может привести к избыточной фрагментации или блокировкам потоков, что замедляет управление памятью и влияет на производительность более заметно.

7.5 Fragmentation Test (Фрагментация памяти — случайное выделение и освобождение)

Тесты с имитацией фрагментации памяти при разных размерах аллокаций и случайном освобождении также показали существенную разницу. Как и в случае со сложными объектами, частые вызовы сборщика мусора в Java сильно замедляют программу: потоки приостанавливаются, объекты перемещаются — это всё занимает очень много времени. В C++ таких проблем нет: память просто освобождается, и всё. Мы получаем фрагментацию, но не теряем время на реорганизацию памяти и блокировку потоков. Возможно, это менее эффективно по использованию памяти, но мы ориентируемся только на скорость.

7.6 Recursive Memory Allocation (Рекурсивное выделение памяти)

Здесь проявляется ограниченность стека в C++. Стек в C++ обычно меньше, и компилятор не оптимизирует рекурсию. В Java же JIT—компилятор умеет инлайнить рекурсию и оптимизировать её выполнение, что ускоряет как выделение памяти, так и общее время работы. Вероятно, Java в этом случае смогла эффективно оптимизировать рекурсивные вызовы и управление памятью, а C++ с этим не справился, в результате чего и возникла огромная разница.

7.7 Общий вывод

Эти результаты показывают, что Java выигрывает в удобстве управления памятью и производительности при многопоточности и в случае когда нам нужны короткоживущие объекты, в то время как C++ остаётся более подходящим для задач, где критична минимизация накладных расходов на управление памятью и необходим прямой контроль. Однако при этом не бралась в расчет эффективность выделения памяти.

8. Потенциальные направления для дальнейших исследований

- Расширение набора бенчмарков с включением реальных приложений.
- Анализ влияния различных оптимизаций компилятора и параметров настройки JVM.
- Проектирование бенчмарка, специально ориентированного на тестирование выделения памяти для сложных объектов с запутанными связями. Это

подразумевает объекты с большим количеством полей, вложенными структурами и ссылками на другие объекты, что приближает тест к реальным сценариям, например, графам объектов в корпоративном ПО или крупным структурам данных в научных вычислениях.

- Добавление в замеры максимальное количество потребляемой памяти в моменте, так как это поможет оценить не только скорость но и эффективность.

ЗАКЛЮЧЕНИЕ

В данной работе был проведён всесторонний сравнительный анализ только скорости управления памятью в C++ и Java с использованием набора синтетических бенчмарков. Тесты охватывали различные сценарии выделения памяти: выделение фиксированного размера, размещение сложных объектов, распределение блоков переменного размера, многопоточное распределение памяти, фрагментация памяти, рекурсивное выделение и случайный доступ к памяти.

В целом, скорость работы с памятью в Java в первую очередь зависит от грамотного вызова сборщика мусора. Однако в реальных приложениях невозможно точно управлять моментом его вызова, поэтому производительность может как превосходить C++, так и уступать ему. В то же время, ручное управление памятью в C++ позволяет гарантировать эффективность, тогда как в Java это невозможно.

Существуют сценарии, в которых Java гарантированно работает быстрее — например, при работе с короткоживущими объектами и в многопоточности, где она имеет специальные оптимизации. В C++ поведение менеджера памяти не меняется вне зависимости от типа объекта и его применения, следовательно чтобы добиться схожей производительности необходимо прибегать к явной синхронизации потоков, что может сильно замедлять процессы, а короткоживущие объекты будут аллоцироваться также как и долгоживущие, в то время как Java будет их выделять в молодое поколение памяти где это происходит сильно быстрее, чем аллокация объекта в C++.

В остальных случаях Java медленнее по той причине, что её менеджер памяти реализует сложную логику: перед удалением объектов необходимо остановить потоки, просканировать объекты, переместить их или удалить. В то же время C++ просто выделяет и освобождает память, изредка оптимизируя процессы во избежание фрагментации — это требует гораздо меньше ресурсов.

СПИСОК ИСПОЛЬЗОВАННОЙ ЛИТЕРАТУРЫ

1. Haggander D., Lidkn P., Lundberg L.: A Method for Automatic Optimization of Dynamic Memory Management in C++ // Blekinge Institute of Technology. 2001. № 3. C. 491–494.
2. Balosin I.: JVM Garbage Collectors Benchmarks Report 19.12 // ionutbalosin. 2019.
3. Goncalves C. D. O.: A performance comparison of modern garbage collectors for big data environments // Examination Committee. 2021. T. 5, № 3. C. 51–76.
4. daanx. mimalloc-bench. // 2023. URL: https://github.com/daanx/mimalloc-bench (дата обращения: 22.11.2024).
5. University of Virginia (Charlottesville, Virginia). STREAM // URL: https://www.cs.virginia.edu/stream/ref.html (дата обращения: 22.11.2024).
6. Adobe Systems Incorporated. STLab Performance: A Collection of C++ Performance Benchmarks // URL: https://stlab.adobe.com/performance/ (дата обращения: 22.11.2024).
7. DaCapo Benchmarks Developers. DaCapo Benchmarks // URL: https://github.com/dacapobench/dacapobench (дата обращения: 22.11.2024).
8. Renaissance Benchmarks Developers. Renaissance Benchmarks // URL: https://github.com/renaissance-benchmarks/renaissance/ (дата обращения: 22.11.2024).
9. K. Lamkiewicz. GarbageCollectorBenchmark: // 2015. – URL: https://github.com/KLamkiewicz/GarbageCollectorBenchmark/ (дата обращения: 22.11.2024).
10. I. Balosin. JVM Performance Benchmarks // 2019. – URL: https://github.com/ionutbalosin/jvm-performance-benchmarks (дата обращения: 22.11.2024).
11. Cppreference. C++ Functions // URL: https://en.cppreference.com/w/cpp/language/function (дата обращения: 15.12.2024).
12. Cppreference. C++ атрибут [[noreturn]] // URL: https://en.cppreference.com/w/cpp/language/attributes/noreturn (дата обращения: 15.12.2024).
13. Cppreference. C++ библиотека thread // URL: https://en.cppreference.com/w/cpp/thread (дата обращения: 15.12.2024).
14. Oracle. The Java® Virtual Machine Specification, Java SE 17 Edition. // URL: https://docs.oracle.com/javase/specs/jvms/se17/html/jvms-2.html (дата обращения: 15.12.2024).

15. Oracle. The Java™ Tutorials: Data Types. // URL: https://docs.oracle.com/javase/tutorial/java/nutsandbolts/datatypes.html (дата обращения: 15.12.2024).
16. Oracle. Thread (Java SE 17). // URL: https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/lang/Thread.html (дата обращения: 15.12.2024).
17. cppreference. new (C++). // URL: https://en.cppreference.com/w/cpp/memory/new (дата обращения: 15.12.2024).
18. Oracle. The Java® Virtual Machine Specification, Java SE 17. // URL: https://docs.oracle.com/javase/specs/jvms/se17/html/jvms-2.html (дата обращения: 15.12.2024).
19. Shipilev, Aleksey. JVM Anatomy Quarks. // URL: https://shipilev.net/jvm/anatomy-quarks/ (дата обращения: 15.12.2024).

ПРИЛОЖЕНИЕ 1

Этап проекта	Дата завершения
Поиск и анализ уже проведенных исследований	10.11.2024
Поиск существующих бенчмарков	22.11.2024
Определение сценариев тестирования	10.12.2024
Разработка бенчмарков	20.02.2025
Сбор и анализ данных, корректировка бенчмарков	23.03.2025
Итоговый анализ и отчет	10.04.2025