МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РФ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ
УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«НОВОСИБИРСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ
ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»
(НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ, НГУ)

Факультет информационных технологий
Кафедра Систем информатики

Направление подготовки 09.04.01 Информатика и вычислительная техника
Направленность (профиль): Технология разработки программных систем

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА МАГИСТРА

Пантелеева Евгения Владимировича

Тема работы:

ОПТИМИЗАЦИЯ ВРЕМЕНИ ДОСТУПА К ПОЛЯМ К JAVA ОБЪЕКТА

«К защите допущена»	**Руководитель ВКР**
Заведующий кафедрой,	к.ф.-м.н., доцент
д.ф-м.н, профессор	Кафедра СИ НГУ
Лаврентьев М.М. /……..	Быстров А.В. /……...
(ФИО) / (подпись)	(ФИО) / (подпись)
«……»……20…г.	«……»……20…г.

Новосибирск, 2023

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РФ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«НОВОСИБИРСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»
(НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ, НГУ)

Факультет информационных технологий

Кафедра Систем информатики

(название кафедры)

Направление подготовки: 09.04.01 ИНФОРМАТИКА И ВЫЧИСЛИТЕЛЬНАЯ ТЕХНИКА
Направленность (профиль): Технология разработки программных систем

УТВЕРЖДАЮ

Зав. кафедрой Лаврентьев М.М.
(фамилия, И., О.)

…….
(подпись)
«……»……20…г.

ЗАДАНИЕ
НА ВЫПУСКНУЮ КВАЛИФИКАЦИОННУЮ РАБОТУ МАГИСТРА

Студенту *Пантелееву Евгению Владимировичу*, группы 21224
(фамилия, имя, отчество, номер группы)

Тема: Оптимизация времени доступа к полям Java объекта
(полное название темы выпускной квалификационной работы магистра)

утверждена распоряжением проректора по учебной работе от 1 марта 2022г № 0032
Срок сдачи студентом готовой работы: 20 мая 2023 г.
Исходные данные (или цель работы) оптимизация времени доступа к Java объекту
Структурные части работы:

1. Анализ похожих работ
2. Разработка данной структуры данных
3. Тестирование производительности на различных случаях

Консультанты по разделам ВКР (при необходимости, с указанием разделов)
Зяблицкий Антон Сергеевич
(раздел, ФИО)

Руководитель ВКР
Кафедра систем информатики ФИТ НГУ,
к.ф.-м.н, доцент
Быстров А.В./……..
(ФИ О) / (подпись)

«…»……20…г.

Задание принял к исполнению
Пантелеев Е.В./……..
(ФИО студента) / (подпись)

«…»……20…г.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Современные объектно ориентированные языки программирования, такие как Java, предоставляют удобные механизмы для написания программ. Java приложения часто работают с большим количеством различных объектов. Однако, это может приводить к негативному влиянию на производительность. Огромное число мелких объектов приводит к фрагментации памяти, плохой локальности процессорного кеша, а также к увеличению числа чтений указателей из памяти. Эти факторы неизбежно приводят к ухудшению производительности Java приложения.

В квалификационной работе предлагается способ уменьшения негативного влияния вышеперечисленных недостатков введением в язык Java новой структуры данных, названной flattened array или FlatArray. Эта структура представляет собой последовательность объектов, расположенных в одном линейном участке памяти. Для Java программиста flattened массив имеет интерфейс обычного Java класса.

Flattened array может быть применен для реализации многих других структур данных и алгоритмов, зависящих от скорости доступа к памяти. В частности, в данной работе была реализована альтернативная версия хеш-таблицы, которая в некоторых случаях показала лучшую производительность операции поиска.

На рисунке 1 изображен граф объектов, который может быть частью объектно-ориентированного приложения. Класс Point содержит в себе два поля x, y типа int. Как описывалось ранее, Java работает с такого рода объектами как со ссылками. Потому массив ”pts” содержит в себе указатели. В таком случае при доступе к полю ”x” элемента i требуется дополнительное разыменования указателя, и только потом поступаемся по поля x.

```
Point value = pts[i].x;
```

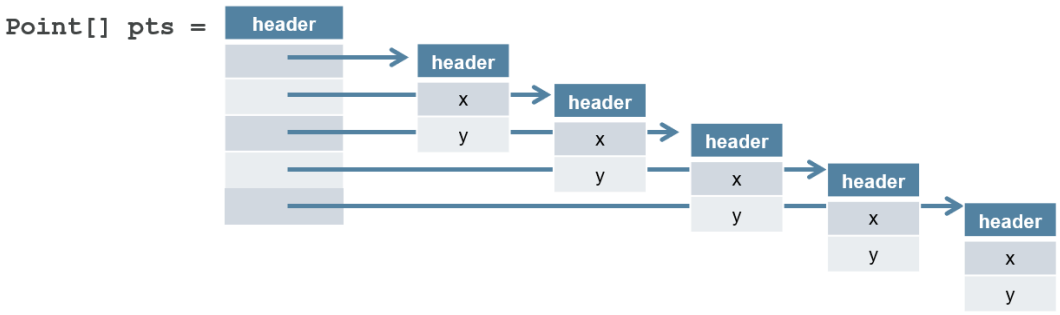

Рис. 1 – Массив объектов класса Point

Однако в реальности мы имеем дополнительные издержки, связанные с исполнением load барьера и проверкой прочитанного значения на null:

```
Point* tmp = &pts[i];
Point actual_value = load-barrier(tmp);
if (actual_value == null) {
  throw new NullPointerException();
}
```

В итоге это приводит к довольно дорогостоящему доступу к элементу массива. Было бы полезно избавиться от дополнительных накладных расходов на доступ к полю. Одним из возможных способов является расположение всего массива в виде линейного участка памяти, как указано на рисунке 2.

По сравнению с предыдущей раскладкой объектов, у последней есть ряд преимуществ:

– Улучшается локальность процессорного кеша. Поскольку объекты находятся рядом, а не разбросаны по всей куче, при чтении одного элемента в линейку кеша могут быть записаны соседние элементы. Это потенциально увеличивает производительность последовательного доступа к элементам массива.
– Удаляются load барьер и проверка на null. Адрес конкретного поля i-того элемента могут быть вычислен с помощью простой арифметики. Во многих случаях это обходится дешевле чтения ссылки из памяти. И

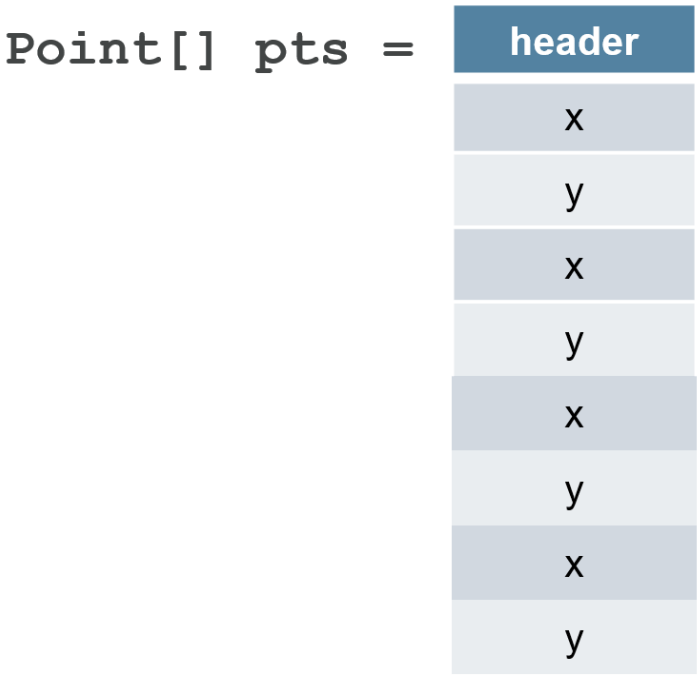

Рис. 2 – Массив значений

поскольку виртуальная машина подразумевает, что в таком массиве не бывает указателей, равных null, проверки на null исчезают.

– Последовательный доступ к элементам может быть автоматически векторизован компилятором

К сожалению, на данный момент JVM и язык Java не поддерживают возможность создания такого рода структуры данных. Разработка такой структуры и поддержка ее в компиляторе и виртуальной машине Java является целью квалификационной работы. Для этой цели необходимо решить следующие задачи:

а) Изучить аналогичные работы

б) Реализовать данную структуру данных, что подразумевает разработку интерфейса для Java программиста и поддержку на уровне среды исполнения.

в) Измерить производительность работы со структорой и проанализировать результаты

г) Изучить применимость новой структуры данных для реализации других структур. Провести тестирование производительности для них.

Данная работа проводилась на базе виртуальной машины Zing. Главными отличиями этой виртуальной машины от OpenJDK является замена C2 компилятором Falcon, базирующемся на LLVM, и применением конкурентного сборщика мусора C4[1]. Ее более подробное описание можно найти в следующей главе.

ОБЗОР ПРЕДМЕТНОЙ ОБЛАСТИ

2.1 Язык Java

Язык программирования Java разработан как язык общего назначения. Его синтаксис похож на C++, однако большинство небезопасных операций, вроде адресной арифметики, недопустимы. Вместо этого язык предлагает механизм сборки мусора и синхронизации потоков.

Большим преимуществом Java является мультиплатформенность, а потому она используется в написании программ для различных операционных систем и процессорных архитектур. Маленькие встраиваемые системы с ограниченным количеством ресурсов, такие как мобильные телефоны, могут быть запрограммированы с помощью Java Platform Micro Edition. Для рабочих станций существует Java Platform Standard Edition, предоставляющее модули для работы с UI, обработки XML и так далее. Разработка корпоративных приложений может быть упрощена с помощью Java Platform Enterprise Edition.

Поскольку Java работает на большом количестве платформ, то приложения не распространяются как платформо-зависимый машинный код. Вместо этого используется концепция виртуальной машины. Исходный код Java программ компилируется в Java байткод, который вместе с таблицей символов и другой метаинформацией сохраняется в class файл. Виртуальная машина Java определяется независимо от самого языка спецификацией[3]. Высокая производительность достигается путем just-in-time компиляции.

2.2 Profile-guided optimization

По сравнению со статически собранными бинарными файлами, исполнение кода с помощью виртуальной машины дает некоторые преимущества, такие как переносимость, безопасность, автоматическое управление памятью,

динамическая загрузка кода. Однако это требует применение других алгоритмов оптимизации для получения высокой производительности.

Оптимизация во время исполнения обычно базируется на профиле, собираемого в течение выполнения кода. Эта техника позволяет найти наиболее часто исполняемые участки программы, называемыми горячими точками “hot spots”. Горячий код может быть оптимизирован jit компилятором более агрессивно, чем статическим, поскольку виртуальная машина владеет большим контекстом об исполняемой программе. Оптимизации, основанные на сборе профиля, называются profile-guided.

2.3 JVM

Разработанная компанией Sun Microsystems и поддерживаемая в настоящее время компанией Oracle виртуальная машина HotSpot доступна для большого числа платформ и операционных систем. Ею поддерживаются архитектуры x64, arm и другие. Эта JVM может работать в операционных системах на базе ядра linux и bsd, а также windows. Виртуальная машина является частью JDK (Java Developmenbt Kit) и обычно поставляется вместе с ним. Существует основанная на открытом исходном коде реализация JDK — OpenJDK, которая доступна всем. Тем не менее, множество компаний предлагают свои реализации JDK, в большей или меньшей степени отличающихся от OpenJDK. Сюда относятся Microsoft OpenJDK, Liberica JDK, Amazon Corretto, Azul Platform Prime и другие.

Azul Platform Prime (или, другими словами, Zing) в первую очередь предназначена для использования в больших корпоративных приложениях, такие как высоконагруженные сервера, в которых важны короткие паузы сборки мусора и высокая производительность. Более того, в ее состав входит множество специфичных инструментов для анализа работы приложений, которые будут полезны Java разработчику, и виртуальная машина Zing, о которой пойдет

речь дальше. В отличии от HotSpot, Zing имеет альтернативу стандартному C2 компилятору под названием Falcon, базирующуюся на фреймворке LLVM (описан в следущем разделе) и собственный сборщик мусора C4. Это конкурентный, поколенный сборщик мусора[12], предназначенный для крупных систем, чувствительным к паузам виртуальной машины.

2.4 LLVM

LLVM – это акроним к ”Low Level Virtual Machine”. Это набор инструментов и библиотек для создания различных компиляторов. С его помощью были разработаны компиляторы для языков C, C++ и многих других. Кроме того, LLVM подходит для написания JIT-компиляторов языков с управляемой средой исполнения, поскольку имеет расширения для поддержки деоптимизаций и сборщика мусора. Важнейшим плюсом этой инфраструктуры является хорошо специфицированный промежуточный язык LLVM IR[7] (Intermediate representation), который фактически представляет собой высокоуровневый ассемблер. На основе этого представления работают различные анализаторы кода и оптимизаторы. LLVM IR базируется на форме статического однократного присваивания (с англ. static single assignment). SSA – это граф потока управления, в котором каждая переменная имеет только одно присваивание[8].

Как упоминалось ранее, компилятор Falcon базируется на инструментарии LLVM. Это позволяет виртуальной машине Zing делать более сложный анализ Java кода и производить более сильные оптимизации, чем стандартный C2 компилятор[11].

Важно отметить одну особенность, как C2 и Falcon работают с интринсиками. Интринсики – это специальные функции, реализующиеся самим компилятором. В JVM к их числу относятся широкий перечень математических функции, алгоритмы шифрования, в частности AES, Unsafe операции и так далее. JVM применяют специально подготовленные разработчиками виртуальной

машины функции, называемыми стабами (на англ. StubRoutines). Они обычно пишутся на ассемблере под конкретную архитектуру и набор инструкций. Другой подход – это генерация кода интринсика на лету для последующей открытой подстановки в тело вызывающей функции, что используется C2 компилятором для реализации Unsafe функций. Поскольку Falcon базируется на LLVM, он использует интринсики, написанные на LLVM IR. Это дает ряд преимуществ. Во-первых, независимость от архитектуры процессора, во-вторых, упрощает разработку интринсиков. Важно отметить, что работа с LLVM IR в Zing невозможна без понимания принципов взаимодействия сборщика мусора и компилятора.

2.4.1 Взаимодействие сборщика мусора и компилятора

Для упрощения взаимодействия между сборщиком и скомпилированном кодом, многие сборщики организованы в терминах 3 абстракций: load и store barrier, safepoint[10]:

– Load barrier, или барьер чтения – это кусок кода, исполняемый после операции загрузки указателя из памяти, но перед использованием прочитанного значения. Следует отметить, что некоторые сборщики могут не иметь барьера чтения.
– Store barrier, или барьер записи – по аналогии с load barrier это фрагмент кода, который работает перед (или после) выполнением машинной инструкции сохранения указателя, но после получения сохраняемого значения. Write барьеры широко применяются для реализации поколенных сборщиков мусора
– Safepoint, или безопасная точка – это позиция в коде, где поток исполнения может безопасно остановиться для дальнейшего анализа его контекста. После этой точки видимые компилируемому коду указатели могут быть изменены во время сборки мусора[9]. Обычно безопасные точки

расставляются в прологах функций и в циклах. Однако бывают случаи, когда поток-мутатор не может достигнуть безопасной точки. Такого рода ошибки называются TTSP (Time To Safepoint issue). Они возникают из-за ошибок внутри виртуальной машины. В этой работе одной из подзадач было устранение TTSP.

Сборщик C4 имеет оба вида барьеров. Их наличие приводит к ухудшению общей производительности, поскольку приходится исполнять дополнительный код при чтении или записи указателя. Потому при генерации кода разработчики стараются использовать как можно более простые барьеры, либо вовсе исключать их. Данная работа помогает частично решить проблему удаления load-барьеров.

2.4.2 Представление Java объекта в памяти

JVM использует единую модель памяти для всех видов объектов, включая массивы и внутренние структуры данных. Ссылки на объекты представляют собой прямые указатели. Это обеспечивает быстрый доступ к полям объекта, однако требуется уметь распознавать тип ссылки во время исполнения. Каждый объект в Java имеет специальный заголовок, называемый markWord. Он необходим для отображения состояния объекта внутри JVM и имеет сравнительно небольшой размер. markWord это внутренняя структура данных, невидимая для Java программиста. Его устройство может быть различным в зависимости от версии JVM и от ее марки. В частности, в Zing размер markWord составляет 8 байт, что в 2 раза меньше, чем в Oracle Hotspot.

2.5 Система типов Java

Все типы в языке Java можно разделить на две больших категории: примитивные и ссылочные типы [3]. На сегодняшний момент существует 8 при-

митивных типов: boolean, int, long, float, double, byte, char, short. К ссылочным типам относятся объекты классов, интерфейсов, массивов и специальный тип null. Можем выделить некоторые отличия:

Примитивные типы:

а) Передача по значению.

б) Операция ”==” сравнивает значения

Ссылочные типы:

а) Передача по ссылке.

б) Имеют единый родительский класс java.lang.Object

в) Операция ”==” сравнивает указатели на объекты, а не их значения

У ссылочных типов можно выделить подмножество, называемое boxing объекты. Это обертка для примитивных типов, которая позволяет представить их как ссылочные. Это необходимо в ряде случаев, например, когда требуется передать примитивный тип в параметр шаблона. В некоторых случаях Java может неявно преобразовывать примитивные типы в ссылочные. Следует отметить, что, как и Java, JVM оперирует с примитивными и ссылочными типами[4]. Более того, для работы с примитивами существуют отдельные инструкции iadd, ladd, fadd, dadd и так далее.

В связи с вышеперечисленным, на данный момент в Java нет возможности создавать собственные типы-значения.

ОСНОВНАЯ ЧАСТЬ

3.1 Обзор похожих работ

В этом разделе будут описываться похожие решения и идеи.

3.1.1 Automatic Object Inlining

В первую очередь стоит отметить работу Christian Wimmer ”Automatic Object Inlining in a Java Virtual Machine”[5]. Автор этой работы разработал алгоритм, который может перемещать элементы к самому массиву. Для этих целей автор во время загрузки классов находит методы, которые меняют значения ссылочных полей. Помимо стандартного профилирования Java, описанного в разделе 2.2, разработчик применят особые барьеры записи, которые подсчитывают количество чтений данного поля. Когда этот счетчик достигает некоторого порога, то JVM производит object colocation. Она позволяет разместить группу объектов в виде единого линейного куска памяти. В дальнейшем компилятор может сгенерировать более оптимизированный код методов, используя информацию о расположении объектов. В частности, компилятор может избежать дополнительное чтение и разыменование указателя. Однако во время компиляции для операций записи ссылки в горячее поле вставляется специальная проверка (guards for field stores), которая распознает попытку изменить значение поля. Это приводит к деоптимизации, поскольку подобная подстановка объекта поля оказывается невозможной. Следует отметить, что техника object inlining применяется не только для элементов массивов, но и для полей объектов.

Предложенная оптимизация дала прирост производительности на тестах SPECjvm98[15]. Но сожалению, данная методика имеет ряд ограничений,

связанных с использованием рефлексии и Unsafe операций. Более того, она приводит к увеличению числа компиляций методов.

3.1.2 Проект ”Valhalla”

Valhalla – это проект по изменению системы типов в языке Java[16]. Он расширяет объектную модель Java определяемыми программистами объектами-значениями (value-objects) и примитивными типами, сочетая абстракции ООП с производительностью примитивов. Благодаря этому изменению Java сможет создавать массивы объектов[17].

На момент написания работы проект ”Valhalla” находится в состоянии разработки. Однако уже доступны ранние сборки, которые все программисты могут опробовать.

В отличии от проекта ”Valhalla”, предлагаемая структура FlatArray не требует переработки системы типов языка.

3.1.3 Contained Objects

org.ObjectLayout – это пакет, предоставляющий набор классов, разработанных с учетом оптимизации схемы памяти[18]. Библиотека предоставляет следующие примитивы:

а) StructuredArray – интерфейс для создания массивов объектов как линейный участок памяти.

б) Contained object – позволяет создавать объекты со ”встроенными” в них полями

К сожалению, на текущий момент пакет не имеет поддержки ни в одной существующей JVM, а потому не дает прироста производительности.

3.2 Высокоуровневое представление FlatArray

Для Java программиста FlatArray выглядит как обычный Java класс со следующими методами:

```
static <T, S extends FlatArray<T>> S newArray(final Class<S>
   faClass, final int length)
```

Создает новый FA класса S длиной length. Неявно вызывает конструктор с сигнатурой для всех элементов типа T. Проверяет ограничения на S и T перед созданием. Если проверки не были пройдены, кидает соответствующее исключение. Более подробно описание ограничений изложено в разделе 3.4.

```
T get(final int index)
```

Возвращает соответствующий индексу элемент. Бросает ArrayOutOfBoundException, если индекс выходит за границы массива.

```
void set(T object, final int index).
```

Копирует поля object в элемент по данному индексу. Если object не null, помечает объект, что то был инициализирован явно. Бросает ArrayOutOfBoundException, если индекс выходит за границы массива.

```
bool sentinel(final int index).
```

Проверяет, что элемент по данному индексу явно инициализирован.

```
int length()
```

Возвращает длину массива.

Элемент flattened array реализует интерфейс FAElement, который имеет один метод void set(FAElement object). Он копирует поля object в this. Пример использования

```
// Реализацияклассаэлемента
class Point implement FAElement {
  int x, y;
  Point() {
    this.x = 0;
    this.y = 0;
  }
```

```
  void set(FAElement p) {
    this.x = ((Point)p).x;
    this.y = ((Point)p).y;
  }
}

// FlatArray дляданногокласса
class PointArray extends FlatArray<Point> {
  private PointArray(final int length) {
    super(length);
  }
  public Point get(final int index) {
    return super.get(index);
  }
  public void set(Point value, final Point index) {
    super.set(value, index);
  }
  public static PointArray newArray(final int length) {
    return FlatArray.newArray(PointArray.class, length);
  }
}
```

3.3 Представление FlatArray внутри виртуальной машины

Несмотря на то, что идея flattened array достаточно проста, ее реализация потребовала переработки многих компонент виртуальной машины: загрузчика классов, сборщика мусора, внутренний CI (compiler interface), runtime представление классов и Falcon компилятора.

Стоит отметить, что все Java методы класса FlatArray реализованы внутри виртуальной машины, однако подходы сильно различаются. Так, для режима интерпретации, а также для исполнения C1 скомпилированных методов, используется native вызов кода внутри виртуальной машины, написанного

на C++. Этот подход достаточно прост в исполнении, поскольку не требует глубокой переработки виртуальной машины, однако приводит к дорогостоящему runtime вызову, который потенциально нивелирует все преимущество FlatArray. Более того, при таком подходе невозможны различные оптимизации компилятора, вроде открытой подстановки (inlining). В итоге это не позволяет использовать все возможные плюсы flattened array. Потому, когда falcon компилирует вызывающий метод, он использует интринсики. Как упоминалось ранее, это конструкции, которые выглядят для программиста как обычные функции, но они реализуются компилятором. В нашем случае это подготовленные куски кода на LLVM IR, которые могут быть встроены в вызывающего и оптимизированы под конкретный тип flattened array и его элемента.

Одной из самых острых проблем со стороны компилятора, которая привела к изменению в среде исполнения, оказалась свертка размера элемента в константу при выполнении адресной арифметики. Это происходило из-за того, что виртуальная машина не могла связать идентификатор конкретного FlatArray класса (kid - “klass” identifier) с ekid (element “klass” identifier). С помощью kid компилятор может вычислить размер объекта. Кроме того, falcon считал, что вычисленный с помощью адресной арифметики указатель может быть равен null, хотя это не так.

Работа с FlatArray начинается с момента инициализации виртуальной машины. Он загружается вместе с другими ”well-known” классами и для него создается специализированное внутреннее представление. Когда JVM начинает загружать наследников класса FlatArray, она так же создает для них специализированное представление. Это необходимо из-за следующих причин:

а) Альтернативный подход к трассировке объекта сборщиком мусора

б) Необходимость связывать идентификатор класса FlatArray или его наследника с идентификатором элемента массива. В частности, компилятору необходима такая информация для проведения оптимизаций.

На данный момент FlatArray интегрирован в JDK. Он представляется как расширение JDK, поставляемой вместе с виртуальной машиной.

3.4 Ограничения

FlatArray может быть успешно использован далеко не во всех случаях, поскольку имеет ряд ограничений. На данный момент их существует несколько:

а) Элемент не должен иметь тип интерфейса, абстрактного класса, массива или flattened array. Другими словами, размер типа элемента массива должен быть вычислим во время компиляции и быть константой.

б) Класс-элемент реализовывает интерфейс FAElement. Это требование связано с необходимостью распознавать возможные элементы массива во время загрузки класса.

в) Наследник Flat Array не должен иметь поля. Были проведены измерения, что дополнительные поля внутри FlatArray приводят к ухудшению производительности. Более того, виртуальная машина ожидает конкретную раскладку массива, которая описана в разделе 4.3.

Кроме того есть и другого рода ограничения, связанные с особенностью реализации FlatArray внутри JVM:

а) Нельзя применять Unsafe операции и рефлексию.

б) Максимальный размер FlatArray равен 16 GB. Это ограничение появилось из-за требований сборщика мусора

в) Если хотя бы один элемент внутри массива живой с точки зрения сборщика мусора, то весь массив будет считаться живым. Это связано с ограничениями C4. Если FlatArray достаточно велик, то он будет выделен в отдельном регионе памяти ”big space”, из которого сборщик мусора не переносит объекты в другие регионы. Игнорирование этого ограничения может привести к тому, что в итоге мелкие объекты окажутся в ”big space” регионе.

3.5 Примеры использования

В этом разделе будет приведено сравнение интерфейсов для работы с Java массивом объектов и FlatArray. Создание объекта выглядит так:

```
// Java objects array
T[] array = new T[length];
// FlatArray
FA array = FlatArray.newArray(FA.class, length);
```

Проверка на ”null” и доступ к элементу:

```
// Java objects array
if (array[idx] != null) {
  field = array[idx].field;
}
// FlatArray
if (array.sentinel(idx)) {
  field = array.get(idx).field;
}
```

Инициализация элемента:

```
// Java objects array
array[idx] = new T(arg1, arg2);
// FlatArray
array.get(idx).arg1 = arg1;
array.get(idx).arg2 = arg2;
array.mark(idx, true);
```

ОПИСАНИЕ РЕШЕНИЯ

4.1 Первая версия FlatArray

Раскладка полей внутри FlatArray изображена на рисунке 3. Данная структура состоит из нескольких частей:

а) header; заголовок объекта, который включает в себя markWord, поля для хранения длины массива и KID элемента.

б) последовательность ссылок на объекты

в) сами объекты

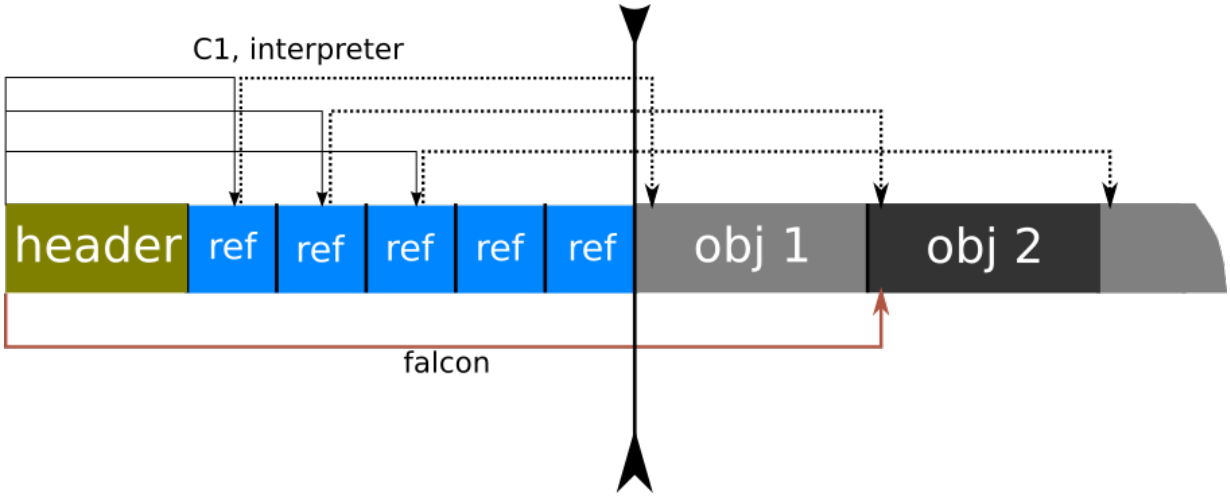

Рис. 3 – Первая версия FlatArray

В реальности этот FlatArray представляет собой обычный Java массив класса Type[].class, у которого позади расположены элементы. Доступ к определенному элементу может быть осуществлен двумя способами: стандартным, через чтение и разыменование указателя, или путем вычисления адреса. JVM использует оба подхода. При интерпретации или исполнении C1 скомпилированного кода применяется первый подход, falcon - второй. Как упоминалось ранее, режим C1 и интерпретатор используют JNI методы для работы с FA. Однако при falcon компиляции JNI реализации подменяются специально подготовленными интринсиками. Свойство nullability достигается путем добавления поля

’isSentinel’ во внутрь элемента. В итоге такая раскладка FlatArray имеет ряд недостатков:

а) Проверка nullability достаточно медленная. Особенно в случае not-existing, поскольку мы имеем большее число промахов кеша по сравнению с классическим Java массивом.

б) Необходимо вычислять размер элемента массива для адресной арифметики, что дает значительные накладные расходы. Либо необходимо хранить размер элемента где-то внутри структуры данных, что и было применено в последствии.

в) Нет поддержки сборки мусора, поскольку такой FlatArray неотличим от обычного Java массива объектов. Необходимо корректно предоставлять метаинформацию о этом объекте.

4.2 Представление FlatArray с sentinel array

В этом разделе представлена альтернативная версия раскладки полей FlatArray. Она изображена на рисунке 4. В данной версии FlatArray учтены недостатки предыдущего подхода. Этот массив состоит из следующих полей:

Рис. 4 – FlatArray с sentinel array

- header[8 byte]; markWord этого объекта
- length [4 byte]; длина массива
- ekid [4 byte]; KID элемента массива
- sentinel array; массив элементов типа bool, который показывает, какие элементы внутри были явно инициализированы
- pad; поле для выравнивания
- последовательность элементов вместе с метаинформацией

Поля ’isSentinel’ вынесены в одну компактную область памяти, что в итоге уменьшает число промахов кеша в режиме ’not-existing’. Результаты тестирования производительности можно увидеть в таблицах 1, 2 и 3. Описание тестов можно найти в разделе 4.4.

Таблица 1 – Производительность HashMap only-existing.

Benchmark	*Score, op/s*
faMapLoop	*5523 -2%*
mapLoop	*5646*

Таблица 2 – Производительность HashMap fifty-fifty.

Benchmark	*Score, op/s*
faMapLoop	*6122 +5%*
mapLoop	*5833*

Таблица 3 – Производительность HashMap not-existing.

Benchmark	*Score, op/s*
faMapLoop	*34216 +12%*
mapLoop	*30371*

Из результатов в таблицах можно сделать вывод, что в случае not-existing и fifty-fifty хеш-таблица, основанная на FlatArray, имеет лучшую производительность, чем стандартная HashMap. Однако в случае only-existing все наоборот. Нам требуется ускорить операцию поиска в хеш-таблице в тестах not-existing и fifty-fifty. Это связано с тем, что в реальных приложениях эти два случая встречаются чаще, чем not-existing. Потому эти тесты более приоритетны для оптимизации, чем not-existing.

4.3 Текущее представление FlatArray

Новое внутреннее представление FlatArray изображено на рисунке 5. Этот массив имеет несколько полей:

Рис. 5 – Текущее представление FlatArray

- markWord [8 byte];
- length [4 byte]; длина массива
- ekid [4 byte]; KID элемента массива
- Последовательность элементов вместе с метаинформацией:
  - offset[4 byte]; Смещение от заголовка текущего элемента до заголовка FlatArray.
  - bool[1 byte]; показывает, что данный элемент был инициализирован.
  - pad; выравнивание.
  - object; элемент массива.

Результаты тестирования производительности можно увидеть в таблицах 4, 5 и 6. Описание тестов можно найти в разделе 4.4.

Таблица 4 – Производительность HashMap only-existing.

Benchmark	*Score, op/s*
faMapLoop	*6032 +7%*
mapLoop	*5654*

Таблица 5 – Производительность HashMap fifty-fifty.

Benchmark	*Score, op/s*
faMapLoop	*5393 -7%*
mapLoop	*5824*

Таблица 6 – Производительность HashMap not-existing.

Benchmark	*Score, op/s*
faMapLoop	*19401 -25%*
mapLoop	*25757*

Как видно из таблиц, поиск существующего ключа в хеш-таблице, базирующейся на FlatArray, менее эффективно, чем в стандартной HashMap, однако в других случаях мы имеем хорошую производительность.

В отличие от FlatArray с sentinel array, эта раскладка показывает плохую производительность в случаях fifty-fifty и not-existing. Это связано с возрастанием числа промахов кеша в этих тестах. Однако данная раскладка имеет преимущество в размере. Она меньше FlatArray с sentinel array на ”length” байт, поскольку sentinel массив удаляется.

Таблица 7 – Производительность FlatArray only-existing.

Benchmark	*Score, op/s*
flatArrayLoop	*58843 +77%*
basicArrayLoop	*33184*
flatArrayLoopSentinel	*41450 +35%*
basicArrayLoopSentinel	*30810*
flatArray	*653577811 +13%*
basicArray	*579119776*
flatArraySentinel	*387515918 +4%*
basicArraySentinel	*373422833*

Результаты тестирования производительности элементарных операций над FlatArray и Java массивом объектов приведены в таблице 7. Как мы видим, FlatArray имеет лучшие результаты, чем обычный массив.

После анализа было выяснено, что является главной причиной, почему производительность FlatArray в HashMap не дает похожего улучшения. Это связано с тем, что в HashMap чтение памяти не является узким местом, то есть производительность поиска по ключу ограничена пропускной способностью процессора, а не памяти. Связи с этим, FlatArray будет полезен в тех местах, где бутылочным горлышком будет доступ к памяти.

4.4 Тестирование

4.4.1 Особенности тестирования производительности кода в JVM языках

Поскольку целью данной работы является оптимизация производительности, то важной задачей становится подбор подходящих метрик для ее из-

мерения. Как говорилось ранее, JVM это управляемая среда исполнения с Just In Time компиляцией и функцией автоматической сборки мусора. Это сильно сказывается на методике измерения, поскольку на одном и том же тесте JVM может выдавать нестабильную производительность. В первую очередь это связано с тех, что JVM достигает пиковой производительности далеко не сразу после старта приложения. Среде исполнения необходимо время, чтобы загрузить требуемые классы, создать профиль исполнения приложения, скомпилировать горячие методы и так далее. Процесс разгона JVM до пиковой производительности называют ”warm-up”, или ”разогревом”.

Для тестирования производительности кода, написанного на JVM языках, существуют специализированные инструменты. В данной работе применялся Java Microbenchmark Harness[6]. Это набор библиотек для измерения производительности небольших кусков кода. Этот инструмент позволяет разработчику настраивать время warm-up, количество итераций измерений, а также выбирать формат представления результатов. В работе используется throughput, которая выражается в количестве операций в секунду, то есть op/s.

4.4.2 Методика тестирования производительности FlatArray

Поскольку FlatArray — это новая структура данных для Java мира и готовых бенчмарков для ее тестирования не существовало пришлось тесты для нее создавать. Была разработана коллекция тестов, которые измеряют производительность FlatArray по сравнению с обычным Java массивом с учетом различных аспектов. В частности, тесты измеряют скорость доступа к элементу массиву в зависимости от его длины. Учитываются также такие факторы, как nullability элемента. Измерение скорости доступа проводится в трех режимах:

а) not-existing; структура данных полностью пуста

б) fifty-fifty; структура данных на 50% пуста

в) only-existing; полная инициализация всех элементов

Помимо сравнения массивов, были созданы тесты, измеряющие производительность HashMap из JDK8 и хеш-таблицы, основанной на FlatArray. Эти бенчмарки позволяют оценить потенциальную применимость данной структуры данных внутри JDK. Они измеряют производительность операции поиска по ключу в трех режимах, описанных в этом разделе. Все измерения проводились на компьютере с процессором Intel(R) Xeon(R) E-2134 CPU @ 3.50GHz с объемом оперативной памяти 64Gb.

В следующих разделах описаны конкретные тесты производительности, пояснено, что они измеряют и зачем. В дальнейшем в работе будут приведены результаты бенчмарков для различных версий FlatArray.

4.4.3 Тесты flatArrayLoop и basicArrayLoop

Эти бенчмарки тестируют производительность обхода массива в цикле. Код теста выглядит примерно так:

```
measure() {
  for (int i = 0; i < length; i++) {
    long value = array[indexes[i]].field;
    acc = doSimpleMath(acc, value)
  }
  return acc;
}
```

В случае когда ’array’ это FlatArray, бенчмарк носит название flatArrayLoop и basicArrayLoop, если в качестве ’array’ применяется стандартный Java массив. Тест обходит ’array’ по заранее подготовленным индексам, содержащихся в ’indexes’. Из массива извлекается объект, из которого читается поле типа long. В дальнейшем оно используется для вычисления простой арифметики в функции ’doSimpleMath’. Массив ’index’ предварительно заполняется индексами. При этом существует два режима обхода ’array’:

a) random; каждый последующий индекс – это псевдослучайное число.

б) forward; индексы идут по возрастающей от 0 до ’length’

4.4.4 Тесты flatArray и basicArray

Эти тесты используются для измерения скорости одиночного доступа к полю элемента массива. Псевдокод приведен в листинге:

```
measure() {
  long value = array[index].field;
  return value;
}
```

В отличие от тестов из предыдущего раздела, эти тесты хорошо показывают стоимость адресной арифметики. Компилятор способен выносить некоторые вычисления за пределы цикла, что в итоге сказывается на полученных результатах. В случае FlatArray важно понимать реальные накладные расходы вычисления адреса конкретного поля в зависимости от длины массива.

4.4.5 Тесты flatArraySentinel и basicArraySentinel

Эти тесты измеряют производительность доступа к элементу с явной проверкой на nullability. Псевдокод показан ниже:

```
measure() { // FlatArray
  if (array.sentinel(index)) {
    acc = array.get(index);
  }
  return acc;
}
measure() { // Java object array.
  Element value = array[index];
  if (value != null) {
    acc = value.field;
  }
  return acc;
```

```
}
```

4.4.6 Тесты flatArraySentinelLoop и basicArraySentinelLoop

Этот набор тестов похож на предыдущий, но тесты в нем работают в цикле. Ниже приведен псевдокод для версии с FlatArray и с обычным Java массивом.

```
measure() {
  for (int i = 0; i < array.length(); i++) {
    if (array.sentinel(indexes[i])) {
      long value = array.get(indexes[i]).field;
      acc = doSimpleMath(acc, value)
    }
  }
  return acc;
}

measure() {
  for (int i = 0; i < array.length(); i++) {
    Element value = array[indexes[i]];
    if (value != null) {
      acc = doSimpleMath(acc, value.field)
    }
  }
  return acc;
}
```

4.4.7 Тесты faMapLoop и mapLoop

Помимо тестов непосредственно на массивы, измерялась производительность двух реализаций java.util.HashMap. Обе базируются на хеш-таблице из JDK8, но одна из них использует FlatArray.

Эта пара бенчмарков тестирует скорость поиска элемента по ключу в цикле. Их псевдокод приведен ниже.

```
measure() {
  for (int i = 0; i < length; i++) {
    Element value = hashmap.get(keys[i]);
    if (value != null) {
      acc = doSimpleMath(acc, value.field)
    }
  }
  return acc;
}
```

4.4.8 Тесты faMap и map

Помимо тестирования производительности поиска ключей в цикле, полезно измерить скорость одиночного доступа к хеш-таблице.

```
measure() {
  Element value = hashmap.get(key);
  return value.field;
}
```

Эти бенчмарки отличаются от faMapLoop и mapLoop. Было замечено, что компилятор по разному оптимизирует эти два типа тестов, а потому необходимы результаты измерений обоих.

ЗАКЛЮЧЕНИЕ

В квалификационной работе предложена и реализована оптимизация времени доступа к полям Java объекта на примере поддержки в Java новой структуры данных FlatArray. С точки зрения программиста эта структура выглядит как обычный Java класс, но реализованный внутри Java машины. В работе были решены следующие задачи:

а) Создано и протестировано несколько версий FlatArray с различной раскладкой полей

б) Реализована загрузка классов наследников FlatArray

в) Устранены проблемы с плохой оптимизацией nullcheck и typecheck при доступе к элементу FlatArray

г) Реализована поддержка сборки мусора для FlatArray

д) Разработаны и измерены тесты производительности

е) FlatArray интегрирован в JDK.

Результаты проведенного тестирования показали, что хотя предложенное решение и не является универсальными, оно дает некоторый прирост производительности в определенных сценариях. Например, FlatArray может стать заменой обычному Java массиву объектов в тех случаях, когда бутылочным горлышком горячего кода является доступ к памяти.

В качестве направления дальнейших исследований по теме квалификационной работы можно назвать ослабление ограничений на использование структуры FlatArray, а также изучение возможностей применения ее для построения других сложных структур данных помимо HashMap.

Выпускная квалификационная работа выполнена мной самостоятельно и с соблюдением правил профессиональной этики. Все использованные в работе материалы и заимствованные принципиальные положения (концепции) из опубликованной научной литературы и других источников имеют ссылки на них. Я несу ответственность за приведенные данные и сделанные выводы.

Я ознакомлен с программой государственной итоговой аттестации, согласно которой обнаружение плагиата, фальсификации данных и ложного цитирования является основанием для не допуска к защите выпускной квалификационной работы и выставления оценки «неудовлетворительно».

_____	_____
ФИО студента	Подпись студента

_____

(заполняется от руки)

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ И ЛИТЕРАТУРЫ

1. Gil Tene, Balaji Iyengar, Michael Wolf. C4: The Continuously Concurrent Compacting Collector // ACM SIGPLAN Notices. — 2011. — Vol. 46, No. 11. — P. 79-88.

2. Types and Programming Languages / Benjamin C. Pierce // The MIT Press, — 2002.

3. Gosling, J., Joy, B., Steele, G., Bracha, G., Buckley, A. The Java Language Specification. Java SE 8 Edition. — 2015. — Oracle America, Inc.

4. Lindholm, T., Yellin, F., Bracha, G., Buckley, A. The Java Virtual Machine Specification. Java SE 8 Edition. — 2015. — Oracle America, Inc.

5. Christian Wimmer. Automatic Object Inlining in a Java Virtual Machine. — 2008. — Institute for System Software Johannes Kepler University Linz.

6. Java Microbenchmark Harness. [Электронный ресурс]. — 2022. — URL: https://openjdk.org/projects/code-tools/jmh/ (дата обращения: 20.05.2023)

7. LLVM Language Reference Manual. [Электронный ресурс]. — 2022. — URL: https://llvm.org/docs/LangRef.html (дата обращения: 20.05.2023)

8. Static Single Assignment Book. [Электронный ресурс]. — 2022. — URL: https://pfalcon.github.io/ssabook/latest/book-full.pdf (дата обращения: 20.05.2023)

9. The Garbage Collection Handbook: The Art of Automatic Memory Management / Chapman & Hall, 2012 // CRC Press. — 2012.

10. Garbage Collection with LLVM. [Электронный ресурс]. — 2023. — URL: https://llvm.org/docs/GarbageCollection.html (дата обращения: 20.05.2023)

11. Gil Tene. Zing hits the trifecta. [Электронный ресурс]. — 2017. — URL: https://stuff-gil-says.blogspot.com/2017/ (дата обращения: 20.05.2023)

12. Azul C4 Garbage Collector. [Электронный ресурс]. — 2023. — URL: https://www.azul.com/products/components/pgc/ (дата обращения: 20.05.2023)

13. Chris Lattner. The Architecture of Open Source Applications (Volume 1). [Электронный ресурс]. — 2023. — URL: LLVM https://aosabook.org/en/v1/llvm.html (дата обращения: 20.05.2023)

14. Gil Tene on Zing, Low Latency GC, Responsiveness. [Электронный ресурс]. — 2017. — URL: https://www.infoq.com/interviews/tene-low-latency/ (дата обращения: 20.05.2023)

15. The SPECjvm98 Benchmarks. [Электронный ресурс]. — 2008 . — URL: http://www.spec.org/jvm98 (дата обращения: 20.05.2023)

16. Project Valhalla. [Электронный ресурс]. — 2022 . — URL: https://openjdk.org/projects/valhalla/ (дата обращения: 21.05.2023)

17. JEP draft: Value Objects (Preview). [Электронный ресурс]. — 2023. — URL: https://openjdk.org/jeps/8277163 (дата обращения: 21.05.2023)

18. org.ObjectLayout. [Электронный ресурс]. — 2023. — URL: http://objectlayout.github.io/ObjectLayout/ (дата обращения: 21.05.2023)