Санкт-Петербургский государственный университет

Кафедра системного программирования

Группа 24М.71-мм

Трефилов Степан Захарович

Реализация универсального компиляторного интерфейса на основе JVMCI

Отчёт по учебной практике
в форме «Решение»

Научный руководитель:
Доцент кафедры системного программирования, к.ф.-м.н., Д. В. Луцив

Санкт-Петербург
2025

Оглавление

1. Введение

Java является широко используемым [18] индустриальным языком программирования. Java программы распространяются в виде class [13] файлов, содержащих байткод для виртуальной машины Java. Для достижения максимальной производительности исполнения Java байткода применяется техника JIT компиляции.

Популярной реализацией спецификации языка Java SE [19] является OpenJDK [15]. Он также является главной реализацией языка Java, и большое количество крупных компаний (например, Amazon, Red Hat, Azul) имеют свои дистрибутивы OpenJDK. Исходный код OpenJDK был открыт в 2007 году.

В OpenJDK применяется 2 компилятора: C1, также известный как клиентский, и C2, также известный как серверный. Несмотря на то, что C2 широко используется и работает хорошо, существуют ряд компаний, предлагающих альтернативные реализации Java с компиляторами, заменяющими C2. Данные альтернативные реализации, как правило, имеют компиляторы, генерирующие более производительный код, чем C2. Одной из таких альтернативных реализаций является GraalVM, разработчики которой добавили в OpenJDK интерфейс JVMCI.

Начиная с 9 версии языка Java в OpenJDK появился новый интерфейс JVMCI [8] (Java VM compiler interface) для встраивания в JDK стороннего компилятора, написанного на Java. Он позволяет относительно простым образом разрабатывать сторонние компиляторы никак не модифицируя OpenJDK, а также получать необходимую для компиляции информацию из HotSpot VM с помощью набора определенных классов и методов из модуля jdk.internal.vm.ci [10].

Но что если мы захотим использовать использовать не Java, а, например, C++, для реализации стороннего компилятора? Можно либо изменить исходный код OpenJDK разных версий, добавив в него новый компилятор, либо использовать JVMCI и одну версию компилятора, но тогда придется реализовывать компилятор на Java.

В рамках данной работы было решено разработать новый компиляторный интерфейс на основе JVMCI, который будет предоставлять аналогичные возможности, но работать на основе технологий сериализации, а не Java.

2. Постановка задачи

Целью работы является разработка универсального компиляторного интерфейса на основе JVMCI. Для реализации данной цели были поставлены следующие задачи.

1. Провести обзор JVMCI и существующих Java JIT–компиляторов.
2. Спроектировать и разработать новый компиляторный интерфейс.
3. Провести апробацию интерфейса: написать автоматические тесты.

3. Обзор предметной области

3.1. Компиляторы C1 и C2

C1 и C2 – стандартные компиляторы в OpenJDK. Они написаны на C++ и являются частью кодовой базы OpenJDK, в отличие от компиляторов на основе JVMCI (как, например, Graal Compiler), которые отделены от OpenJDK.

C1 (клиентский компилятор[3] HotSpot JVM) – JIT компилятор, нацеленный в первую очередь на быструю компиляцию и уменьшенное использование памяти (по сравнению с серверным компилятором HotSpot JVM). До версии Java 8, пользователь должен был выбирать, какой из двух компиляторов он хочет использовать, однако в современных версиях JDK оба компилятора, по умолчанию, работают совместно.

Многоуровневая компиляция[5] (multi–tiered compilation) – технология, направленная одновременно на ускорение работы HotSpot JVM и на улучшение качества производимого во время компиляции кода. Суть ее заключается в том, что сначала HotSpot JVM запускает клиентский компилятор, а далее, при необходимости, использует серверный. Подробнее, когда HotSpot JVM замечает, что метод часто используется, она отправляет его на компиляцию в C1. Это позволяет ускорить выполнение метода, а также собрать дополнительную метаинформацию, которая может помочь серверному компилятору произвести более качественный машинный код. Далее, если метод остается "горячим", он отправляется в C2 и компилируется для максимальной производительности.

C2 (серверный компилятор[11] HotSpot JVM, top-tier компилятор) – JIT компилятор, нацеленный в первую очередь на скорость работы производимого кода. Он затрачивает больше ресурсов, чем клиентский компилятор HotSpot JVM, однако используя метаинформацию времени исполнения, которую накапливает HotSpot JVM во время работы каждого конкретного метода, удается добиться относительно хорошей производительности производимого кода.

Несмотря на то, что C2 способен компилировать сравнительно хороший машинный код, существуют различные проекты, целью которых является замена C2 на другой, лучший компилятор. В частности, компилятор Graal, работающий в OpenJDK с помощью JVMCI интерфейса, является одной из таких замен.

3.2. Компилятор Graal

Graal[7] - JIT компилятор, написанный на Java для замены C2 в HotSpotVM. Во многом Graal похож на C2, так как тоже использует в качестве промежуточного представления Sea of nodes[17], однако является более продвинутым компилятором и имеет, в отличие от C2, проприетарную версию.

В контексте данной работы Graal нам интересен тем, что для его создания в

OpenJDK был реализован JVMCI. Это важно, так как других компиляторов, использующих JVMCI, на текущий момент нет. Из-за этого JVMCI сильно заточен именно под Graal, и поддерживается в наиболее свежем состоянии лишь для версий JDK, в которых развивается Graal.

3.3. JVMCI-интерфейс

Как уже было упомянуто ранее, JVMCI — интерфейс для встраивания в Java стороннего компилятора, написанного на Java. С точки зрения данного интерфейса, компилятор это наследник абстрактного класса JVMCICompiler[1], в котором определен метод compileMethod. Каждый раз, когда OpenJDK требуется скомпилировать очередной метод, виртуальная машина вызывает компилятор с помощью compileMethod, передавая единственным аргументом объект типа CompilationRequest[2], содержащий информацию о том, какой метод необходимо скомпилировать. Также для случаев, когда компилятор поддерживает OSR[3] компиляции, CompilationRequest содержит номер байткода в методе, от которого нужно сделать компиляцию метода.

Для получения информации из виртуальной машины, компилятор может использовать классы, определенные в модуле jdk.internal.vm.ci [10]. Структурно, основными частями данного модуля:

- Директории meta и hotspot - содержат интерфейсы для работы с сущностями виртуальной машины, а также реализации этих интерфейсов для виртуальной машины HotSpot. Например:
  - HotSpotConstantPool - сущность для доступа к constant pool [14];
  - HotSpotResolvedJavaMethod - сущность для доступа к информации о конкретном методе;
  - HotSpotResolvedObjectType - сущность для доступа к информации о конкретном классе.
- Директория code - содержит классы, необходимые для установки скомпилированных методов в Code cache. Например:
  - BytecodeFrame - сущность, позволяющая описать, в каких регистрах или стековых слотах находятся конкретные локальные переменные, стековые переменные и мониторы. Данная информация необходима для деоптимизаций и сборки мусора;

[1] JVMCICompiler.java
[2] CompilationRequest.java
[3] On Stack Replacement

  - CodeCacheProvider - сущность, с помощью которой можно взаимодействовать с Code Cache. Данная абстракция необходима для добавления скомпилированного кода для Java методов.

- Директории по названиям архитектур (в частности, amd64 и aarch64) - содержат классы для предоставления архитектурно специфичной информации. Например, с помощью методов класса AMD64.java[4] можно получить информацию о расширениях, поддержанных процессором, а также номера регистров архитектуры X86 с точки зрения HotSpot VM.

Большой плюс данного интерфейса заключается в том, что он позволяет подключить Java компилятор к самым свежим сборкам OpenJDK с помощью набора флагов, без необходимости перекомпиляции или изменения кодовой базы OpenJDK. В частности, для Java 21 нужно сделать следующее:

1. Собрать свой Java компилятор в виде Jar архива.
2. Добавить его в *classpath* целевого приложения.
3. Использовать флаги $-XX : +UseJVMCICompiler$ для включения JVMCI и $-Djvmci.Compiler$ для указания имени компилятора.

Существенным же минусом данного интерфейса является то, что не для самых свежих версий OpenJDK данный интерфейс является заброшенным. В частности, на текущий момент актуальными версиями OpenJDK для Graal являются 17, 21 и 24. Тем не менее, даже в версиях 17 и 21 внутренности JVMCI интерфейса существенно отличаются. В версиях ниже JVMCI отличается еще более существенно, и алгоритм подключения компиляторного агента может существенно отличатся.

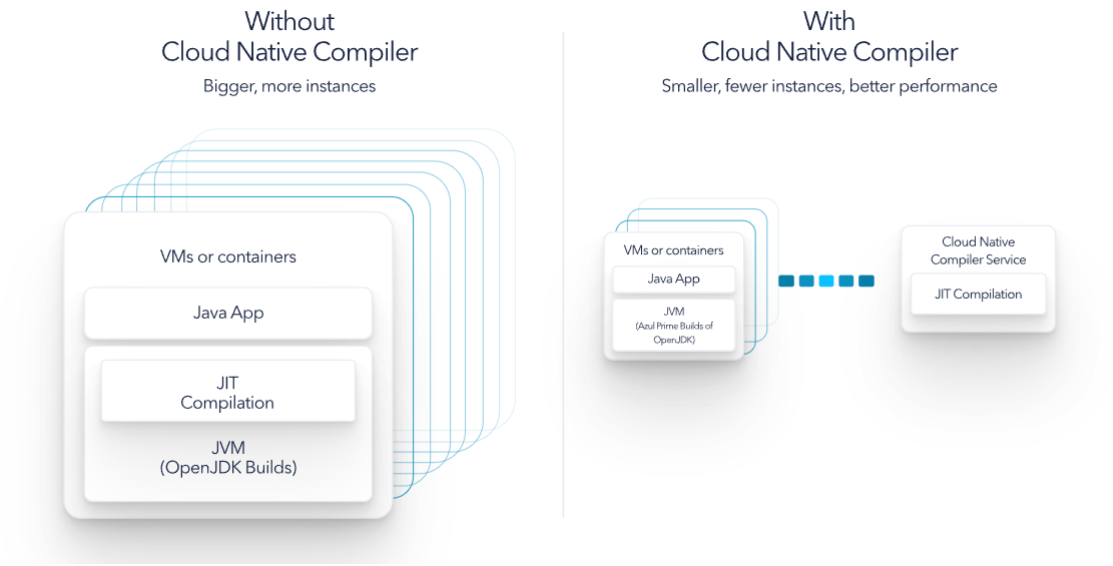
3.4. Cloud Native Compiler

Кроме Graal, работающего с HotSpot VM через JVMCI, существуют и другие альтернативные JIT компиляторы для Java. В частности, существует Falcon JIT - компилятор для Java, использующийся вместо C2 в Zing VM (также известной как Azul Platform Prime[1]). Данный компилятор для нас особенно интересен тем, что он может быть запущен отдельно от Zing VM на клиентской машине. Таким образом, можно создать некоторый сервер компиляции, пользователи которого будут подключаться через некоторый сетевой интерфейс.

Реализация подобного сетевого компиляторного интерфейса не является целью данной работы, однако разработанный интерфейс также может быть использован

[4] AMD64.java

Рис. 1: Демонстрация Cloud Native Compiler



для сетевого JIT компилятора, так как работает с сериализованными сообщениями, в отличие от JVMCI, работающего с Java объектами.

Можно также упомянуть, что кроме Zing VM подобную технологию отделения JIT компилятора в отдельный сервер поддерживает OpenJ9[4] - еще одна альтернативная реализация Java VM Specification.

3.5. Языки описания данных

Для реализации нового универсального интерфейса необходимо выбрать некоторый формат данных, с помощью которого компилятор и виртуальная машина будут общаться. Так как JIT-компилятор должен работать параллельно с приложением и максимально быстро, вариант описания данных с помощью JSON или любого другого текстового представления был сразу отброшен. Было решено использовать некоторый независимый от языка бинарный протокол.

Первым в качестве такого формата рассматривался Protobuf[16]. Protocol Buffers – это не зависящий от языка расширяемый механизм сериализации структурированных данных.

Были выделены следующие плюсы использования Protobuf:

1. Удобный формат описания интерфейса в виде отдельного файла со структурами данных.

2. Хорошая поддержка во множестве языков, что, в частности, позволяет удобно общаться с виртуальной машиной из C++ компилятора.

Хотя Protobuf хорошо подходит для выбранной задачи, вместо него было решено использовать Cap'n Proto[2]. Cap'n Proto это очень похожая на Protobuf технология

от того же автора, но является более легковесной. Это достигается за счет применения Zero-Copy подхода: Cap'n Proto использует одно и то же представление данных при записи, чтении и передаче, что позволяет сэкономить время на сериализации и десериализации данных.

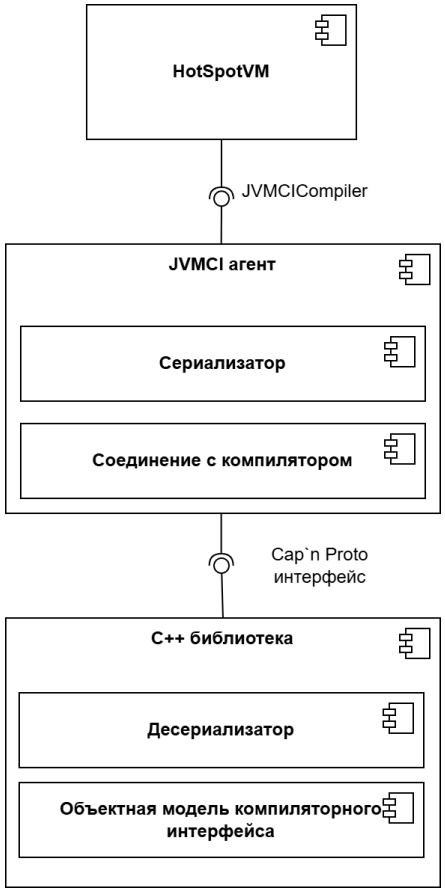
Также в качестве языка для описания данных рассматривался Flatbuffers[6]. Аналогично Cap'n Proto, Flatbuffers позволяет сэкономить за счет применения Zero-Copy подхода, а также имеет поддержку в большом количестве языков.

4. Проектирование и реализация интерфейса

4.1. Общая схема решения

Новый универсальный компиляторный интерфейс было решено реализовывать в виде набора связанных компонентов:

Рис. 2: Общая диаграмма компонентов решения



- JVMCI агент – входная точка для нового компиляторного интерфейса. В качестве входной точки было решено выбрать именно JVMCI так как это существенно упрощает интеграцию, и, как следствие, можно быстрее перейти к реализации остальных компонентов интерфейса. В будущем планируется отказаться от JVMCI в целом, но эта задача за рамками данной работы. Внутри JVMCI агента также были реализованы:
  - Сериализатор сообщений – компонента, переводящая JVMCI объекты в сериализованные Cap‘n Proto сообщения.
  - Компонента отвечающая за соединение с компилятором. Внутри нее происходит подключение компилятора в виде динамической библиотеки, а также отправка и принятие сообщений.

- C++ библиотека. Внутри данной библиотеки были реализованы:
  - Десериализатор сообщений – компонента, переводящая сериализованные Cap'n Proto сообщения в C++ объекты.
  - Объектная модель компиляторного интерфейса.

Общую диаграмму решения можно увидеть на Рис. 2.

4.2. Описание интерфейса

Используя язык описания Cap'n Proto, было создано описание нового компиляторного интерфейса для OpenJDK. В частности, было необходимо описать запросы для следующих сущностей виртуальной машины:

- Классы, методы и поля.
- Сущности из constant pool.
- Запросы на создания различных зависимостей для скомпилированного кода. Например, unique concrete method зависимость, которая позволяет с помощью CHA девиртуализовать виртуальный метод.
- Некоторые другие запросы, как, например, константное значение поля, если поле является константой.

Во многом описанный интерфейса дублирует JVMCI, однако есть и некоторые существенные различия:

- Используется меньшее количество различных сущностей. JVMCI активно использует наследование, что затрудняет использование интерфейса. Судя по всему, изначально это было сделано для гибкости, так как подразумевалось, что JVMCI может быть реализован не только для виртуальной машины HotSpot. Так как мы хотим работать только с OpenJDK, было решено избавится от большого количества промежуточных сущностей в новом интерфейсе.
- Так как новый интерфейс основан на сериализации, а не Java объектах, каждой сущности представляющий объект, класс, метод и поле было добавлено специальное *id* поле, позволяющее однозначно идентифицировать некоторую сущность. Во время общения с виртуальной машиной компилятор должен запоминать *Id* сущностей, так как некоторые ответы виртуальной машины могут ссылаться друг на друга по *id*.

В результате был разработан файл в формате Cap'n Proto, содержащий полное описание интерфейса. Пример описания одного из сообщений можно видеть на Рис. 3.

Рис. 3: Описание сообщения типа ResolvedJavaType на языке Cap‘n Proto

```
struct ResolvedJavaType {
  id @0: ProtoID;
  javaType @1: JavaType;
  superClassId @2: ProtoID; # ResolvedJavaType id
  interfaceIds @3: List(ProtoID); # ResolvedJavaType ids
  layoutHelper @4: Int32;
  superCheckOffset @5: Int32;
  union {
    instanceType @6: ResolvedInstanceType;
    arrayType @7: ResolvedArrayType;
  }
  accessFlags @8: AccessFlags;
}
```

4.3. Реализация компиляторного агента

Для подключения компилятора к OpenJDK было решено реализовать компиляторный JVMCI агент. Данный агент выполняет 3 функции:

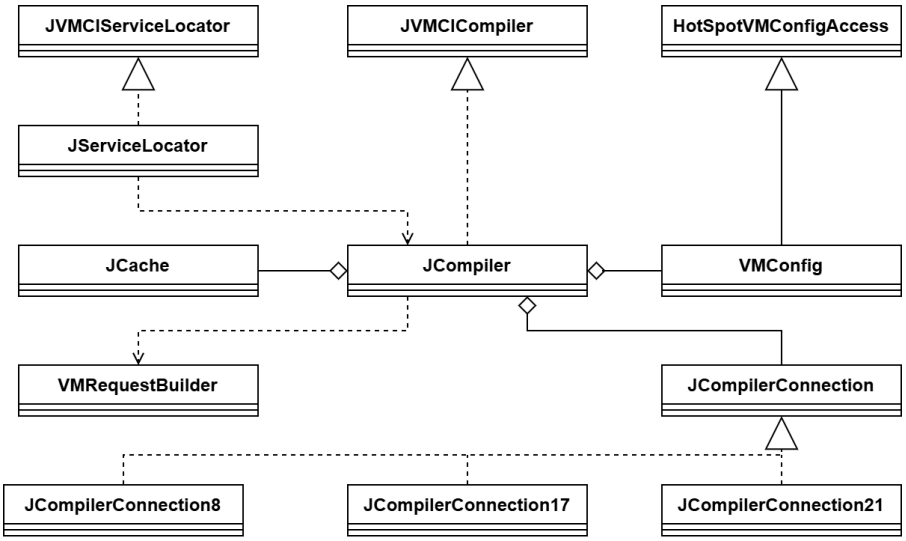
1. Подключается к виртуальной машине OpenJDK с помощью JVMCI интерфейса.
2. Подключает компилятор к приложению. На текущий момент был реализован агент, который подключает компилятор в виде динамической библиотеки, однако ничего не мешает модифицировать агент так, чтобы он мог, например, подключаться к компиляторному серверу.
3. Сериализует запросы и ответы между компилятором и виртуальной машиной, а также перенаправляет запросы на компиляцию.

Рис. 4: Взаимосвязь HotSpotVM и компилятора через новый интерфейс

В качестве целевых версий OpenJDK для реализации агента были выбраны наиболее свежие LTS версии Java 17 и 21, а также последняя более менее активно поддерживаемая Java 8.

Большую часть кода агенты для 3 версий Java разделяют между собой. В частности, все, что связано с обработкой запросов и построением ответов, является общим для 3 агентов. Основные различия заключаются в специфичных для каждой версии Java инструментах сериализации и взаимодействия с native кодом (наследники класса JCompilerConnection на Рис. 5).

Рис. 5: Диаграмма классов компиляторного агента

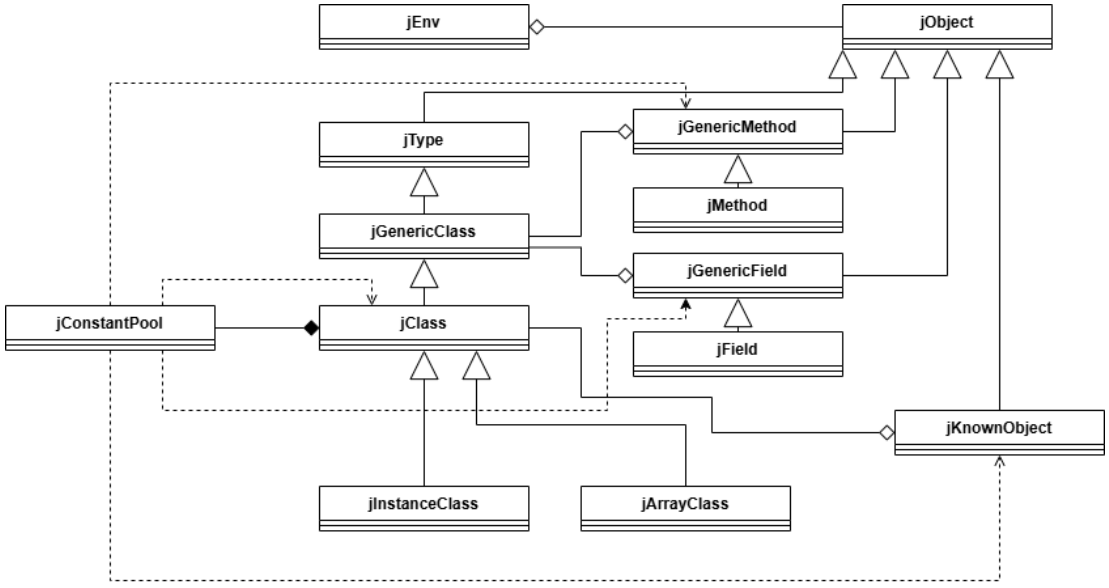


Для Java 17 и 21 для общения с компилятором применяется Foreign Function and Memory API[9]. Используя данный API, можно выделять память за пределеами Java кучи и вызывать методы динамической библиотеки прямо на уровне Java кода.

Для аналогичных целей в Java 8 пришлось применить Unsafe[5] в комбинации с JNI[12].

4.4. Реализация интерфейса на стороне компилятора

Рис. 6: Диаграмма классов объектной модели компилятора



Для работы с предложенным интерфейсом на C++ был реализован десериализатор приходящих Cap'n Proto сообщений в объектную модель, подходящую для непо-

[5] Unsafe.java

средственно компиляции (диагрмма на Рис. 6).

Данная объектная модель почти полностью является калькой с аналогичной, применяемой в OpenJDK C1 и C2 компиляторами. Кроме того, был реализован также сериализатор для уходящих от компилятора запросов в виртуальную машину.

5. Апробация интерфейса

Для аппробации интерфейса был реализован компилятор-заглушка, способный получать и запрашивать сериализованные сообщения. На базе данной заглушки было написано порядка 300 различных тестов, проверяющих различные сценарии возможного взаимодействия между виртуальной машиной и компилятором.

Кроме того, для упрощения тестирования, был реализован специальный класс, позволяющий производить некоторые подготовительные действия с HotSpot VM перед инициализацией компиляции. Например, данная обертка для запуска тестов позволяет:

- Запустить тестовый метод в интерпретаторе некоторое количество раз.
- Скомпилировать тестовый метод в C1 компиляторе для сбора профиля.
- Вызвать компиляцию тестового метода как On Stack Replacement

Для верификации того, что сериализатор и десериализатор отработали как ожидается была реализована возможность распечатать все отправленные сообщения, а также содержимое $jEnv$ – объекта содержащего все принятые от виртуальной машины сообщения в виде объектной модели компилятора. Проверка того, что вывод компилятора соответствует ожидаемому, осуществляется с помощью FileCheck[6].

Пример распечатанного $jEnv$ можно увидеть на Рис. 7.

Рис. 7: Пример содержимого $jEnv$ в распечатанном виде

```
{
    "181": {
        "_name": "ClassWithReferenceField",
         ...
    },
    "183": {
        "_holder": {
            "181": "ClassWithReferenceField"
        },
        "_name": "myClone",
        "_signature": "()java.lang.Object",
        "flags": "public (x1)",
        "force_inline": false,
    },
...
}
```

[6] LLVM FileCheck

6. Заключение

В ходе работы на текущий момент были получены следующие результаты.

1. Проведен обзор предметной области. Были рассмотрены существующие JIT–компиляторы для Java (C1, C2, Graal), JVMCI интерфейс, а также языки описания данных Protobuf и Cap'n Proto.

2. Реализован новый компиляторный интерфейс:

   - Реализован JVMCI агент, сериализующий сущности виртуальной машины в Cap'n Proto сообщения и отправляющий их компилятору.
   - Реализована C++ библиотека, позволяющая десериализовать Cap'n Proto сообщения в C++ объекты.

3. Написаны автоматические тесты, позволяющие проверить работоспособность JVMCI агента в сочетании с C++ библиотекой.

Планы на развитие работы:

1. Избавиться от JVMCI в качестве промежуточного звена между компилятором и виртуальной машиной.

Список литературы

[1] Azul Platform Prime. — URL: https://www.azul.com/products/prime/ (online; accessed: 2024-4-24).

[2] Cap'n Proto введение. — URL: https://capnproto.org/ (online; accessed: 2024-12-19).

[3] Design of the Java HotSpot Client Compiler for Java 6. — 2008. — URL: https://dl.acm.org/doi/pdf/10.1145/1369396.1370017 (online; accessed: 2024-12-19).

[4] Eclipse OpenJ9: A Java Virtual Machine for OpenJDK that's optimized for small footprint, fast start-up, and high throughput. — URL: https://github.com/eclipse-openj9/openj9 (online; accessed: 2025-3-16).

[5] Efficient Code Management for Dynamic Multi-Tiered Compilation Systems. — 2014. — URL: https://dl.acm.org/doi/10.1145/2647508.2647513 (online; accessed: 2024-12-19).

[6] Flatbuffers документация. — URL: https://flatbuffers.dev/ (online; accessed: 2024-12-19).

[7] Graal Compiler. — URL: https://www.graalvm.org/latest/reference-manual/java/compiler/ (online; accessed: 2025-3-22).

[8] JEP 243: Java-Level JVM Compiler Interface. — 2022. — URL: https://openjdk.java.net/jeps/243 (online; accessed: 2024-12-19).

[9] JEP 442: Foreign Function Memory API. — URL: https://openjdk.org/jeps/442 (online; accessed: 2025-3-16).

[10] JVMCI модуль OpenJDK. — URL: https://github.com/openjdk/jdk17u-dev/tree/master/src/jdk.internal.vm.ci/share/classes (online; accessed: 2024-12-19).

[11] The Java HotSpot Server Compiler. — 2001. — URL: https://www.usenix.org/legacy/event/jvm01/full_papers/paleczny/paleczny.pdf (online; accessed: 2024-12-19).

[12] Java Native Interface Specification. — URL: https://docs.oracle.com/en/java/javase/23/docs/specs/jni/index.html (online; accessed: 2025-3-16).

[13] Java SE спецификация. The class File Format. — URL: https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-4.html (online; accessed: 2024-5-02).

[14] Java VM Specification глава 4. — URL: https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-4.html#jvms-4.4 (online; accessed: 2024-12-19).

[15] OpenJDK project. — URL: https://github.com/openjdk/ (online; accessed: 2024-12-19).

[16] Protocol Bruffers документация. — URL: https://protobuf.dev/ (online; accessed: 2024-12-19).

[17] Semantic reasoning about the sea of nodes. — URL: https://www.graalvm.org/latest/reference-manual/java/compiler/ (online; accessed: 2025-3-22).

[18] TIOBE Index for December 2024. — URL: https://www.tiobe.com/tiobe-index/ (online; accessed: 2024-12-19).

[19] Спецификации Java SE. — URL: https://docs.oracle.com/javase/specs/jls/se20/html/index.html (online; accessed: 2024-12-19).