Санкт-Петербургский государственный университет

Кафедра системного программирования

Группа 20Б.11-мм

Трефилов Степан Захарович

Использование JVMCI для интеграции стороннего компилятора в Java 17

Отчёт по учебной практике
в форме «Решение»

Научный руководитель:
ассистент, А. П. Козлов

Санкт-Петербург
2022

Оглавление

1. Введение **3**

2. Постановка задачи **5**

3. Обзор **6**

3.1. C1 Java Client Compiler 6

3.2. C2 Java Server Compiler 6

3.3. Graal JIT Compiler 7

3.4. Falcon JIT Compiler 8

3.5. JIT специфичные задачи 8

3.6. Интеграция JVMCI компилятора в OpenJDK 10

3.7. Взаимодействие HotSpot JVM и JVMCI компилятора . . 11

4. Протокол взаимодействия Falcon JIT и JVM **13**

4.1. Получение метаинформации о компилируемом методе . . 13

4.2. Получение метаинформации о произвольном классе 14

4.3. Реализуемость требований Falcon JIT 15

5. Интеграция произвольного компилятора через JVMCI **16**

5.1. Пример компиляции простого метода с помощью JVMCI компилятора 16

6. Заключение **19**

Список литературы **20**

1. Введение

Java – широко используемый промышленный язык программирования[11]. Важной оптимизацией популярных реализаций JDK[1], которая, в том числе, делает этот язык программирования популярным, является возможность JIT–компиляции программ. До недавнего времени не существовало возможности встроить в Java произвольный компилятор без изменения исходного кода конкретного JDK, однако в Java 9 появился JVMCI.

JVMCI (Java VM compiler interface) – интерфейс для встраивания в Java стороннего компилятора, написанного на Java[8]. Он позволяет относительно простым образом встраивать сторонние компиляторы в произвольный дистрибутив JDK, а также получать необходимую для компиляции информацию из HotSpot VM[7] с помощью набора определенных методов.

GraalVM [4] – высокопроизводительный дистрибутив JDK с открытым исходным кодом. Главными особенностями GraalVM по отношению к другим дистрибутивам JDK являются новые виды оптимизаций программ, реализованные на базе GraalVM, а также возможность интеграции Java с другими языками, такими как Javascript, R или Python, с помощью Truffle Language Framework. Для нас также важно, что компилятор внутри GraalVM написан на Java и интегрируется с помощью JVMCI, благодаря чему его можно воспринимать как пример использования JVMCI для JIT компиляции Java программ.

Не смотря на то, что JVMCI существует уже не первый год, это все же довольно молодой интерфейс, к тому же разработанный, в первую очередь, для интеграции Graal JIT компилятора в OpenJDK. Как следствие, материалов, разъясняющих как встроить свой собственный компилятор в Java используя JVMCI, не существует.

Для решения данной проблемы было решено изучить исходный код OpenJDK[10] и GraalVM[6] связанный с JVMCI интерфейсом, а также

[1] https://docs.oracle.com/javase/8/docs/technotes/guides/index.html#:~:text=Java%20SE%20Development%20Kit%20(JDK)

используя JVMCI научиться взаимодействовать с Falcon JIT[3].

2. Постановка задачи

Целью работы является выделение и систематизация информации, необходимой для интеграции стороннего компилятора через JVMCI. Для её выполнения были поставлены задачи:

1. сделать обзор существующих JIT компляторов в Hotspot JVM и их отличия от классических компляторов;
2. сделать обзор протокола взаимодействия Falcon JIT и JVM;
3. разработать прототип модуля, если возможно, связывающий Falcon JIT с HotSpot JVM с помощью JVMCI.

3. Обзор

3.1. C1 Java Client Compiler

Исторически в HotSpot JVM существовало 2 типа JIT компиляторов: клиентский и серверный.

C1 (клиентский компилятор[1] 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[2] (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.

3.2. C2 Java Server Compiler

C2 (серверный компилятор[9] HotSpot JVM) – 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.

Также, для улучшения производительности кода путем применения различных оптимизаций, в C2 используется графовое SSA[2] промежуточное представление для Java байткода – Sea of nodes.

Несмотря на то, что C2 способен компилировать быстрый машинный код, существет мнение, что он уже достаточно сильно устарел[5]. Большая кодовая база с длинной историей, написанная на C++, является довольно сложной для освоения и модификации новыми инженерами. В связи с этим, в разных компаниях начали разрабатываться замены для C2.

3.3. Graal JIT Compiler

Graal – JIT компилятор, разработанный, в первую очередь, с целью замены C2 (серверного компилятора HotSpot JVM) на новый компилятор, который будет легче писать, расширять и поддерживать. Исходный код данного компилятора написан на Java, в связи с чем в современных версиях JDK появился JVMCI.

JVMCI (Java VM Compiler Interface) – интерфейс для использования внутри HotSpot JVM стороннего компилятора, написанного на Java. JVMCI позволяет, используя возможности Java Service Provider Interface, предоставить реализацию интерфейса JVMCIServiceLocator из OpenJDK, благодаря чему появляется возможность замены C2 на внешний по отношению к HotSpot JVM компилятор.

Примечательно, что в Graal используется свой Graal IR – графовое SSA промежуточное представление для Java байткода, которое, по сути, является аналогом для Sea of nodes из C2. Благодаря ему Graal может производить большое количество различных оптимизаций, которые позволяют, в конечном итоге, генерировать машинный код более производительный, чем генерирует C2.

[2] https://en.wikipedia.org/wiki/Static_single-assignment_form

3.4. Falcon JIT Compiler

Falcon JIT – JIT компилятор, разработанный внутри компании Azul Systems с целью замены компилятора C2.

Компилятор построен на базе LLVM[3], в связи с чем имеет большое количество оптимизаций, предоставляемых этой инфраструктурой. На многих тестах Falcon JIT генерирует более быстрый код, чем C2, и разница между ними может достигать до четырех раз в пользу Falcon JIT.

3.5. JIT специфичные задачи

В отличии от классической Ahead of Time компиляции, JIT компиляция производится во время работы приложения, благодаря чему JIT компилятор может использовать данные о текущем состоянии программы для проведения оптимизаций. В частности, такой информацией может быть информация о загруженных классах или профиль приложения. Таким образом, при интеграции стороннего компилятора через JVMCI нам необходимо думать об этих оптимизациях и о задачах, которые необходимо решать для реализации данных оптимизаций, без которых не может обойтись конкурентноспособный JIT компилятор.

3.5.1. Сборка мусора и деоптимизации

Сборщик мусора – инструмент для автоматического управления динамически выделенной памятью. Для реализации данной оптимизации HotSpot JVM необходимо иметь информацию об указателях на использованную память.

Деоптимизации – инструмент, который позволяет JIT компилятору, в частности, делать агресивные оптимизации. Наиболее простой пример, где может пригодится деоптимизация - условная конструкция. Компилятор может предположить, что одна из ветвей условного оператора никогда не достигается, и не скомпилировать ее. Если это предположение однажды нарушиться, компилятор деоптимизирует метод

[3] https://llvm.org

и вернется исполнять его в интерпретатор. Чтобы произвести такой возврат, компилятору необходимо сохранить дополнительную информацию – состояние интерпретатора после деоптимизации.

Таким образом, что сборщик мусора, что деоптимизации требуют для себя точки внутри программы, в которых программа будет находится в некотором консистентном состоянии с дополнительной информацией. Такие точки называются safepoint.

3.5.2. Синхронизация

Синхронизация необходима для многопоточного программирования. Для ее организации в HotSpot JVM предусмотрены специальные механизмы, однако нас особенно интересует их часть, связанная с взаимодействием между JVM и JIT компилятором.

Понятно, что и в режиме интерпретации, и в режиме исполнения скомпилированного кода JVM может понадобиться взять замок для синхронизации. Проблема возникает тогда, когда после взятия замка приходится произвести деоптимизацию. Для обеспечения возврата в интерпретатор в таком случае, на местах потенциальных деоптимизаций, которые являются safepoint–ами, также сохраняется информация о всех замках, взятых методом, и об их местоположении в памяти.

3.5.3. Инлайниг методов и анализ иерархии классов

Инлайнинг методов – оптимизация, направленная на уменьшение накладных расходов, необходимых для вызова метода. В частности, основным из таких накладных расходов является диспетчеризация, так как все методы в Java являются виртуальными. Для того чтобы выбрать, какой конкретный метод необходимо подставить вместо некоторого виртуального вызова применяется анализ иерархии классов.

Суть анализа иерархии классов заключается в том, чтобы посмотреть все загруженные классы и попытаться определить, может ли на месте некоторого виртуального вызова быть вызвано более одного конкретного метода. Если нет, то можно вставляем код этого конкретного

метода вместо виртуального вызова.

Такая оптимизация может проводится лишь в предположении, что позже не будет загружено других классов, которые будут содержать другую реализацию для метода, который был подставлен вместо виртуального вызова. Если это предположение в некоторый момент выполнения программы нарушается, то нам необходимо деоптимизироваться и скомпилировать код заново.

3.5.4. Исключения

В Java поддерживается конструкция try – catch – finally, позволяющая пользователям работать с исключениями. При создании Java байткода, все блоки try – catch – finally реализуются с помощью таблицы исключений, которая определяет, каким байткодам могут соответствовать какие обработчики и для каких исключений.

С точки зрения JIT компиляции нам интересно, что вместо того, чтобы компилировать сложную логику работы исключений, можно использовать деоптимизацию, и отправиться исполнять обработку исключения в интерпретатор.

3.6. Интеграция JVMCI компилятора в OpenJDK

Для интеграции стороннего компилятора в JVMCI используется Java Service Provider Interface.

С точки зрения практического применения, существуют 2 способа (примеры[12] можно найти здесь) интеграции стороннего компилятора через JVMCI.

Первый способ заключается в том, чтобы предоставить реализацию для интерфейса `jdk.vm.ci.services.JVMCIServiceLocator` используя `META-INF`. Для этого необходимо в META-INF создать папку services. В папке services необходимо создать файл с названием нужного нам сервиса `jdk.vm.ci.services.JVMCIServiceLocator`, в котором будет содержаться единственная строка – полное имя класса, содержащего реализацию данного интерфейса. Далее необходимо упаковать нашу реализацию в jar файл

и добавить ее в classpath программы, в которой нужно использовать наш компилятор.

Второй способ является более современным аналогом первого и заключается в том, чтобы создать Java 9 модуль и предоставить в нем реализацию для интерфейса `jdk.vm.ci.services.JVMCIServiceLocator`. Далее, аналогично первому способу, нужно упаковать наш модуль в jar файл и модифицировать module-path программы, в которой есть необходимость использовать наш компилятор.

Также важно отметить, что в любом случае для активации JVMCI компилятора нам необходимо использовать следующие опции java:

1. `-XX:+UnlockExperimentalVMOptions` – позволяет использовать экспериментальные опции java;

2. `-XX:+EnableJVMCI` и `-XX:+UseJVMCICompiler` – позволяют использовать JVMCI компилятор для запуска программы;

3. `-Djvmci.Compiler=<name>` – опция, в которой необходимо указать имя нашего JVMCI компилятора.

3.7. Взаимодействие HotSpot JVM и JVMCI компилятора

Когда HotSpot JVM решает, что какой–либо метод необходимо скомпилировать, он отправляет запрос на компиляцию в JVMCI компилятор. Данный запрос содержит в себе внутреннее JVMCI представление метода, и позволяет получить полезную для компиляции метода информацию. Например, это может быть:

- класс, в котором объявлен метод;
- модификаторы доступа метода;
- может ли метод быть статически связан;
- какие обработчики исключений есть в методе, каким они соответсвуют байткодам;

- какой у метода байткод;
- какая у метода сигнатура.

Кроме уже упомянутой информации о методе, который HotSpot JVM хочет скомпилировать, JVMCI имеет и другие методы для взаимодействия с HotSpot JVM. В частности, можно получить:

- `MetaAccessProvider` – класс, позволяющий получить метаинформацию о произвольных классах, полях и методах;
- `CodeCacheProvider` – класс, позволяющий взаимодействовать с кешом кода в Java. В частности, позволяет добавлять скомпилированный код для некоторого метода;
- `HotSpotStackIntrospection` – класс, позволяющий исследовать стек вызовов HotSpot JVM.

После того как произойдет компиляция метода через JVMCI, нужно оповестить HotSpot JVM об этом. В случае, если компиляцию произвести не удалось, то JVMCI компилятор должен оповестить HotSpot JVM о том, что произошла ошибка.

4. Протокол взаимодействия Falcon JIT и JVM

Falcon JIT не является непосредственной частью HotSpot JVM. По этой причине, если появляется необходимость воспользоваться Falcon JIT с помощью JVMCI, то придется передавать ему не только байткод метода, который мы хотим скомпилировать, но и некоторое количество дополнительной информации о состоянии виртуальной машины.

В частности, Falcon JIT может запросить дополнительную информацию:

1. о модификаторах доступа для метода;
2. сигнатуру конкретного метода;
3. о состоянии constant pool для компиляции конкретного метода;
4. модификаторы доступа, суперкласс, интерфейсы для некоторого класса, необходимого для компиляции конкретного метода.

Также Falcon JIT может понадобиться информация, необходимая для выполнения JIT специфичных задач, описанных ранее, но в первую очередь было решено найти, как извлечь перечисленную выше информацию через JVMCI.

4.1. Получение метаинформации о компилируемом методе

Как уже было описано выше, для компиляции конкретного метода в JVMCI приходит некоторый запрос на компиляцию. Данный запрос представляется как объект класса `HotSpotCompilationRequest`, и содержит в себе метод `getMethod()`. С помощью данного метода можно получить объект класса `HotSpotResolvedJavaMethod` – внутреннее JVMCI представления для метода, отправленного HotSpot JVM на компиляцию. Уже с помощью данного объекта можно удовлетворить часть требований, необ-

ходимых Falcon JIT для проведения компиляции. В частности, можно узнать:

- байткод метода;
- модификаторы доступа метода;
- сигнатуру метода.

4.2. Получение метаинформации о произвольном классе

Внутри себя любой Java метод может создавать новые объекты произвольных классов и вызывать их методы. В связи с этим, появляется необходимость получать метаинформацию о классах, задеиствованых внутри метода.

Для получения метаинформации о классе, в котором компилируемый метод объявлен, внутри `HotSpotResolvedJavaMethod` присутствует метод `getDeclaringClass()` для получения объекта `HotSpotResolvedObjectType` – внутреннее JVMCI представление для некоторого класса. С помощью данного объекта мы можем удовлетворить такие требования Falcon JIT, как:

- получить состояние constant pool для компиляции метода;
- получить модификаторы доступа, суперкласс, интерфейсы для класса, в отором объявлен компилируемый метод.

Кроме этого, Falcon JIT может понадобиться информации о классах, задействованых внутри метода и не являющихся классом, в котором метод объявлен. Эта информация может быть получена из аналогиных объектов класса `HotSpotResolvedObjectType`, которые могут быть взяты из внутреннего JVMCI представления для constant pool компилируемого метода.

4.3. Реализуемость требований Falcon JIT

После изучения JVMCI был сделан вывод, что требования Falcon JIT через данный интерфейс теоретически можно удовлетворить. Однако на данный момент интеграция Falcon JIT через JVMCI в HotSpot JVM еще не произошла, и, как следствие, требования, которые JVMCI не сможет удовлетворить, еще могут возникнуть.

В качестве демонстрации того, что JVMCI теоретически может удовлетворить требования Falcon JIT, был разработан прототип модуля, связывающий Falcon JIT с HotSpot JVM, однако так как Falcon JIT является проприетарным програмным обеспечением, в данной работе этот модуль обсуждаться не будет.

5. Интеграция произвольного компилятора через JVMCI

Перед тем как создать прототип модуля, связывающий Falcon JIT с HotSpot JVM, понадобилось решить набор задач, которые могут быть полезны для интеграции произвольного компилятора через JVMCI. Такими задачами стали:

- интегрировать в OpenJDK сторонний "пустой" компилятор через JVMCI
- научиться добавлять скомпилированный для метода код в code cache

Как интегрировать в OpenJDK сторонний JVMCI компилятор уже было освещено ранее.

5.1. Пример компиляции простого метода с помощью JVMCI компилятора

Для демонстрации полного цикла взаимодействия Java и стороннего компилятора через JVMCI можно реализовать JVMCI компилятор, который компилирует ровно один метод `add` (полный код для данного примера доступен здесь[12]).

Листинг 1: Метод add

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

Такую компиляцию можно разбить на несколько шагов:

1. HotSpot JVM определяет метод как "горячий" и хочет скомпилировать. Как уже упоминалось выше, на данном этапе HotSpot JVM шлет запрос на JIT компиляцию метода, который представляется в виде объекта класса `HotSpotCompilationRequest`.

2. Получен запрос на компиляцию, и теперь нужно сгенерировать машинный код для метода `add`. Для простоты, в нашем случае компилятор уже знает машинный код для метода `add`, он зашит в компилятор. В общем случае, можно использовать любой сторонний компилятор, к которому можно получить доступ из Java. В частности, это может быть Falcon JIT.

3. Когда есть скомпилированный код в виде массива байтов,

Листинг 2: Ассемблерный код для метода `add`

```
private static final byte[] asmCodeForAddMethod = {
    (byte) 0x8d, (byte) 0x04, (byte) 0x16, (byte) 0xc3
};
```

необходимо создать экземпляр класса `HotSpotCompiledNmethod`. Для конкретного метода `add` это может выглядеть так:

Листинг 3: JVMCI представления для машинного кода

```
CompiledCode compiledCode =
    new HotSpotCompiledNmethod(method.getName(),
        asmCodeForAddMethod, asmCodeForAddMethod.length,
        new Site[]{}, new Assumptions.Assumption[]{},
        new ResolvedJavaMethod[]{method}, new
            HotSpotCompiledCode.Comment[]{},
        new byte[]{}, 16, new DataPatch[]{}, false, 32,
            StackSlot.get(ValueKind.Illegal, 0, false),
        method, request.getEntryBCI(),
        method.allocateCompileId(request.getEntryBCI()),
        ((HotSpotCompilationRequest) request).getJvmciEnv(), false);
```

Также, при генерации `HotSpotCompiledNmethod` из произвольного Java метода может понадобиться указать такие дополнительные данные, как:

- Массив `Site`-ов – массив, содержащий некоторые дополнительные данные, ассоциированные с некоторыми местами в коде метода. В частности, такими дополнительными данны-

ми может быть информация для деоптимизации метода.

- Массив `Assumptions.Assumption`-ов – массив, содержащий предположения о коде метода, которые могут быть нарушены в ходе выполнения. Например, это может быть предположение о том, что вызов некоторого виртуального метода можно заменить на конкретный метод.

4. После генерации `HotSpotCompiledNmethod`–а, можно просто вставить скомпилированный код в кеш кода Java. Для этого используем специальный экземпляр класса `HotSpotCodeCacheProvider`:

Листинг 4: Установка кода в code cache

```
CodeCacheProvider hotSpotCodeCacheProvider =
   HotSpotJVMCIRuntime.runtime().getHostJVMCIBackend().getCodeCache();
hotSpotCodeCacheProvider.setDefaultCode(method, compiledCode);
```

Далее лишь необходимо вернуть HotSpot JVM сообщение об успешной компиляции, и с этого момента вызов метода `add` будет использовать скомпилированный машинный код.

6. Заключение

В ходе работы были выполнены следующие задачи:

- сделан обзор существующих JIT компиляторов в HotSpot JVM;
- сделан обзор требований, выдвигаемых Falcon JIT к JVM;
- показана реализуемость части требований Falcon JIT.

Планы на продолжение работы:

- используя JVMCI, интегрировать в OpenJDK Falcon JIT компилятор.

Список литературы

[1] Design of the Java HotSpot Client Compiler for Java 6. — 2008. — URL: https://dl.acm.org/doi/pdf/10.1145/1369396.1370017 (online; accessed: 2022-5-9).

[2] Efficient Code Management for Dynamic Multi-Tiered Compilation Systems. — 2014. — URL: https://dl.acm.org/doi/10.1145/2647508.2647513 (online; accessed: 2022-5-9).

[3] Falcon JIT Compiler. — URL: https://www.azul.com/products/components/falcon-jit-compiler/ (online; accessed: 2022-5-5).

[4] Get started with GraalVM. — URL: https://www.graalvm.org/docs/getting-started/ (online; accessed: 2022-4-26).

[5] Getting to Know Graal, the New Java JIT Compiler. — 2022. — URL:

[6] Graal project. — URL: https://github.com/oracle/graal (online; accessed: 2022-5-5).

[7] HotSpot (virtual machine). — URL: https://en.wikipedia.org/wiki/HotSpot_(virtual_machine) (online; accessed: 2022-4-26).

[8] JEP 243: Java-Level JVM Compiler Interface. — 2022. — URL: https://openjdk.java.net/jeps/243 (online; accessed: 2022-4-26).

[9] The Java HotSpot Server Compiler. — 2001. — URL: https://www.usenix.org/legacy/event/jvm01/full_papers/paleczny/paleczny.pdf (online; accessed: 2022-5-9).

[10] OpenJDK project. — URL: https://github.com/openjdk/ (online; accessed: 2022-4-26).

[11] TIOBE Index for December 2021. — URL: https://www.tiobe.com/tiobe-index/ (online; accessed: 2022-4-26).

[12] Пример интеграции пустого компилятора через JVMCI. — 2022. — URL: https://github.com/IAmFunkyFrog/JVMCICompilerInsertExamples (online; accessed: 2022-5-12).