Научная статья
Original article
УДК 004.4`423

РАЗРАБОТКА ШАБЛОННОГО ИНТЕРПРЕТАТОРА OPENJDK С ПРИМЕНЕНИЕМ LLVM

DEVELOPING AN OPENJDK TEMPLATE INTERPRETER USING LLVM

Лукьянов Антон Юрьевич, студент магистратуры 2 курс, факультет «Электромеханический», Сибирский государственный университет водного транспорта, Россия, г. Новосибирск

Lukyanov Anton Yurievich, 2nd year master's student, faculty "Electromechanical", Siberian State University of Water Transport, Russia, Novosibirsk

Аннотация: Для поддержки виртуальной машины Java (JVM) «HotSpot» на разных видах архитектур требуется создание портированной версии. Портирование обычно ведётся путем переписывания макроассемблера с использованием языка ассемблер. Поэтому актуальна задача создания более универсального и упрощенного способа портирования JVM на разные виды архитектур. В статье представлен способ для универсальной кодогенерации шаблонов интерпретатора JVM с помощью проекта по реализации программной инфраструктуры для создания компиляторов и их утилит - LLVM. Использование именно LLVM в качестве базовой системы программирования обусловлено большими возможностями адаптации дополнительных

компонентов системы программирования, например, таких как дизассемблер, компоновщик и отладчик для использования с новыми архитектурами.

Summary: To support the Java virtual machine (JVM) "HotSpot" on different types of architectures, a ported version is required. Porting is usually done using assembly language. Therefore, the task of creating a more universal and simplified way of porting the JVM to different types of architectures is relevant. The article presents a method for universal code generation of JVM interpreter templates using the LLVM project for the implementation of a software infrastructure for creating compilers and their utilities. The use of LLVM as the basic programming system is due to the large possibilities for adapting additional components of the programming system, for example, such as a disassembler, linker, and debugger for use with new architectures.

Ключевые слова: трансляция программ, интерпретатор, шаблонный интерпретатор, виртуальная машина Java, JVM, OpenJDK, LLVM.

Keywords: program translation, interpreter, template interpreter, Java virtual machine, JVM, OpenJDK, LLVM.

Введение

Язык программирования Java на сегодняшний день является одним из самых востребованных и распространённых языков программирования на рынке. По состоянию на май 2022 года занимает 3 место по популярности среди всех языков программирования по индексу TIOBE [1]. Наиболее известной особенностью языка Java является его кроссплатформенность – код можно писать и запускать практически на всех платформах и устройствах, **для которых существует виртуальная машина Java.** Благодаря этой возможности, Java по праву считается самым универсальным и гибким языком программирования. Например, написанную программу для Windows можно беспрепятственно запустить на устройстве, работающем под управлением Mac OS или Linux [2]. Но что если попробовать запустить программу на устройстве с другим процессором? Для удачного запуска на устройстве должна быть установлена портированная версия JVM под целевую архитектуру. На данный момент для

большинства самых популярных архитектур уже есть портированные JVM [3]. Но для создания новых портов или обновления старых необходимо переписать значительную часть JVM в части кодогенерации. В частности это актуально для Российских процессоров “Эльбрус”.

Данное исследование проводилось над шаблонным интерпретатором JVM “HotSpot”. Для кодогенерации шаблонный интерпретатор использует макроассемблер. На этапе инициализации VM создаются кодобуферы, которые заполняются платформозависимым машинным кодом различных стабов и байт-кодов VM. Отладить такое непросто, во-первых сгенерированный код не имеет прямого сопоставления с исходным кодом, во-вторых для генерации используется ассемблер.

При создании портированной версии шаблонного интерпретатора обычно переписывается макроассемблер, в результате используется ассемблер с платформозависимым набором команд процессора. Для этого необходимо знать не только устройство JVM, но и набор команд процессора, для которого создается порт. Так как наборы команд различных архитектур сильно отличаются, то использование ранее написанных кодогенераторов невозможно.

Таким образом, можно сказать что кроссплатформенность, с точки зрения запуска одной и той же программы на разных архитектурах процессоров, в JVM достигается с помощью портирования виртуальной машины.

1. Шаблонный интерпретатор JVM

В данной работе исследование касается интерпретатора JVM. Интерпретатор первая ступень для генерации и исполнения машинного кода в виртуальной машине Java. Так как программа на данном этапе представлена в виде последовательности байт-кодов, самый быстрый способ начать ее выполнение это поочередное исполнение каждого байт-кода, чем и занимается интерпретатор.

В JVM существует два вида интерпретатора, CppInterpreter(C++) и TemplateInterpreter(шаблонный). Первый, это упрощенная версия интерпретатора, написанная в основном на C++. Основная цель этого

интерпретатора - экспериментировать и быстро создавать прототипы новых вариантов JVM.

Настоящий интерпретатор, он же шаблонный, написан на макроассемблере, зависящем от платформы, Этот интерпретатор генерируется динамически при каждом запуске JVM.

Как и большинство программ, интерпретатор нуждается в инициализации. Эта процедура выполняет генерацию машинного кода для каждого байт-кода отдельно, в виде шаблонов, этот код хранится в специальной таблице - DispatchTable. Шаблон представляет собой отрывок машинного кода, выполняющий конкретную задачу, это может быть какой-либо стаб JVM или байт-код. Во время исполнения байткодов интерпретатор прыгает по уже сгенерированным шаблонам, таким образом, исполняя программу целиком, генерируя код для байт-кодов единожды.

2. LLVM

Избежать создание портированных версий JVM для каждой архитектуры процессоров можно, если это позволяет кодогенерация, способная генерировать нужный код для целевой архитектуры. Создать такой кодогенератор позволяет проект программной инфраструктуры для создания компиляторов и сопутствующих им утилит – LLVM. LLVM используется в широко распространённом компиляторе Clang. На его примере рассмотрим принцип работы LLVM, изображенный на рисунке 1.

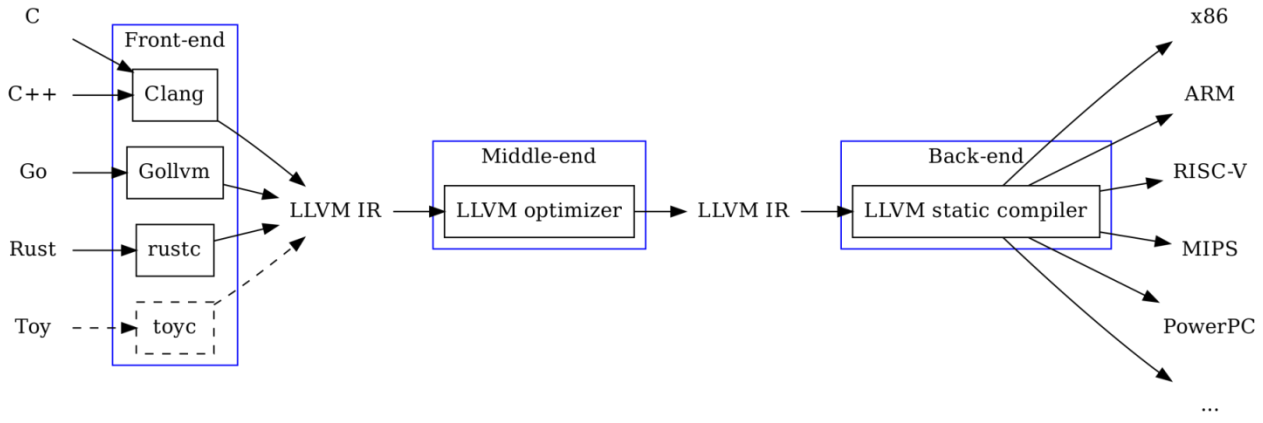

Рис 1. Описание работы LLVM на примере Clang

Работу Clang можно разделить на 3 части:

1. Front-End – В этой части происходит разбор исходного кода запускаемой программы и создание LLVM IR.
2. Optimizer – На этом этапе ведется обработка LLVM IR, применение оптимизаций.
3. Back-End – Генерация машинного кода под заданную архитектуру. На данный момент LLVM поддерживает более 26 архитектур.

Где LLVM IR или просто IR(intermediate representation) – внутренне представление LLVM, над которым производятся все основные оптимизации, по сути представлен в виде байткода, который LLVM понимает. Таким образом, создав из исходного кода, написанного на любом языке программирования IR, мы можем использовать LLVM для компиляции нашей программы.

На данный момент LLVM в качестве Front End поддерживает следующие языки: C/C++, Go, Haskell, Fortan, Rust, Javascript, Ruby, CIL(C#).

Как видно языка Java нет в перечислении, думаю это связанно с тем, что множество функций самого языка Java выполняет её виртуальная машина. Таким образом, если мы хотим построить IR для Java и создавать оптимизированный с помощью LLVM код для множества архитектур, используя один и тот же интерпретатор, то нам следует объединить виртуальную машину Java с технологиями LLVM.

Успешное применение LLVM в JVM есть и в других исследованиях. Например, в Azul Platform Prime используется JIT-компилятор Falcon [4], использующий LLVM. Так же есть статья об одном из первых исследовании о применении LLVM в JVM – VmKit [5].

Так как JVM является стековой виртуальной машиной, то для примера можно рассмотреть статью, рассказывающую о создании компилятора на базе LLVM для стековой процессорной архитектуры TF16 [6].

На данный момент аналога интерпретатора JVM с LLVM не существует.

3. Применение LLVM

Разработкой шаблонного интерпретатора с использованием LLVM занимаются в рамках проекта “Alhambra” в Новосибирском центре

информационных технологий “УНИПРО”. Данный проект нацелен на создание универсальной JVM, способной генерировать код для множества архитектур, без портирования, используя LLVM. В проекте разрабатывается, как и интерпретатор, так и компилятор.

Для реализации универсального кодогенератора в шаблонном интерпретаторе JVM необходимо создать новый порт, как это делалось и для других портов, с заменой макроассемблера, но на этот раз не на новый ассемблер, а на LLVM C-API. Наш порт создается на базе JVM “HotSpot” для x86_64. Помимо замены макроассемблера, так же необходимо создать новые утилиты. На рисунке 2 показаны классы, в которые необходимо внести изменения и классы, которые необходимо создать.

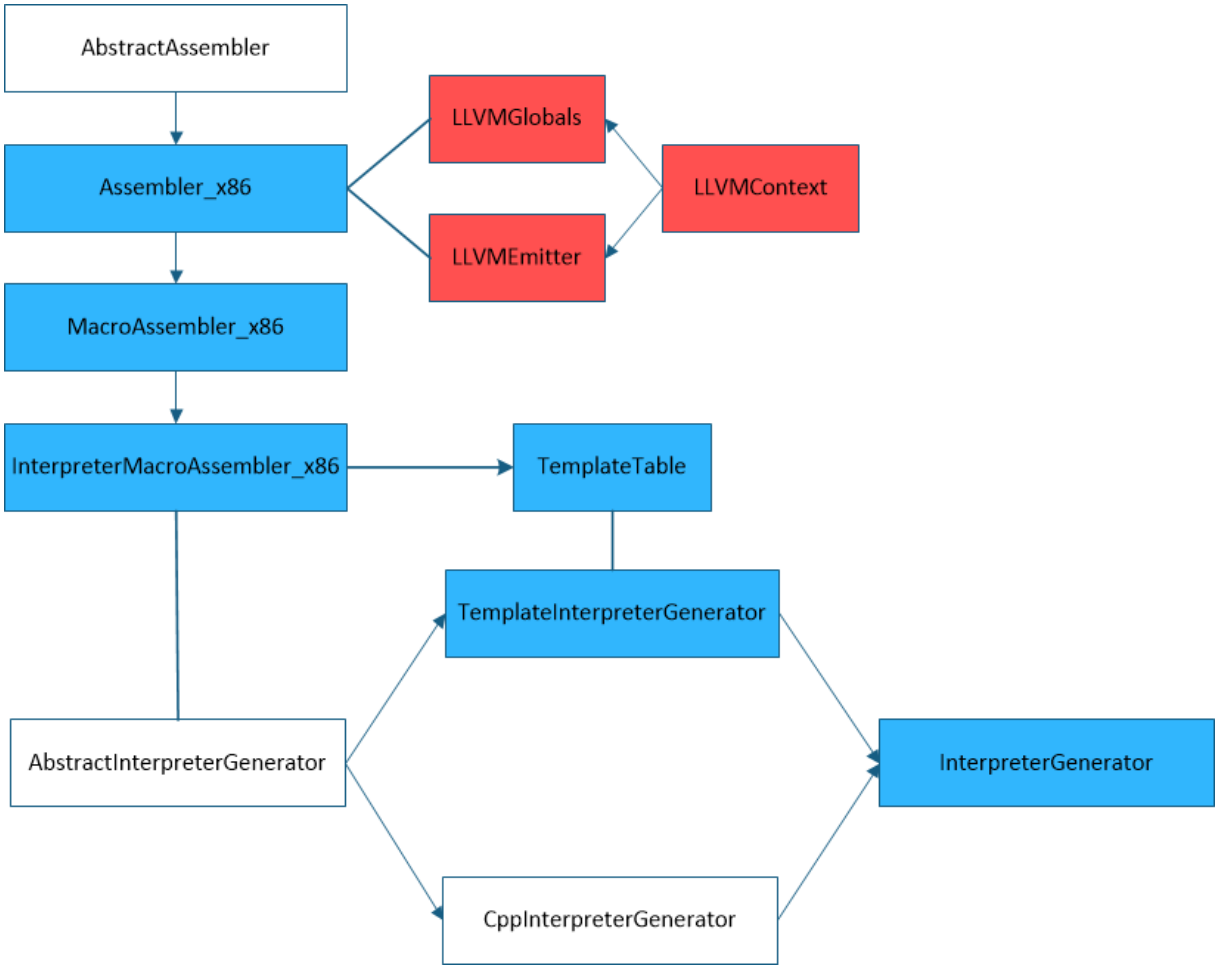

Рис 2. Изменения классов в порте JVM LLVM

Красные – это классы, которые были созданы.

- LLVMContext – класс хранения глобального контекста LLVM

- LLVMGlobals – класс реализующий глобальные переменные, используемые вместо конкретных регистров
- LLVMEmitter – класс для записи сгенерированного кода в кодобуфер JVM.

Синие – это классы, в которых был заменен макроассемблер на LLVM C-API.

Помимо универсальной кодогенерации от LLVM так же наследуется AOT-компиляция, чего нет в стандартных JVM.

Заключение

В ходе исследования была выявлена актуальная проблема кроссплатформенности JVM для непопулярных архитектур и для тех архитектур, которые обновляют свои наборы команд процессоров. Представлено современное решение в виде LLVM, позволяющее реализовать универсальную кодогенерацию как в интерпретаторе, так и в компиляторе JVM.

Литература

1. TIOBE Index. [Электронный ресурс]. URL:https://www.tiobe.com/tiobe-index/ (дата обращения 10.05.2022)
2. Is Java cross platform? [Электронный ресурс]. URL:https://stackoverflow.com/questions/5978005/is-java-cross-platform/5978036#5978036 (дата обращения 10.05.2022)
3. Портированные версии JVM [Электронный ресурс]. URL:https://openjdk.java.net/census#porters (дата обращения 10.05.2022)
4. Azul Platform Prime’s Falcon JIT Compiler [Электронный ресурс]. URL:https://www.azul.com/products/components/falcon-jit-compiler/ (дата обращения 10.05.2022)
5. Nicolas Geoffray. Fostering Systems Research with Managed Runtimes. [Электронный ресурс]. URL:https://vmkit.llvm.org/publications/thesis.html (дата обращения 10.05.2022)
6. Скворцов Л.В. Баев Р.В. Долгорукова К.Ю. Шарыгин Е.Ю. Разработка компилятора для стековой процессорной архитектуры TF16 на основе

LLVM [Электронный ресурс]. URL:https://elibrary.ru/item.asp?id=47410920 (дата обращения 10.05.2022)

Literature

1. TIOBE Index. [Electronic resource]. URL: https://www.tiobe.com/tiobe-index/ (Accessed 05/10/2022)
2. Is Java cross platform? [Electronic resource]. URL: https://stackoverflow.com/questions/5978005/is-java-cross-platform/5978036#5978036 (Accessed 05/10/2022)
3. Ported versions of JVM [Electronic resource]. URL: https://openjdk.java.net/census#porters (Accessed 05/10/2022)
4. Azul Platform Prime’s Falcon JIT Compiler [Electronic resource]. URL: https://www.azul.com/products/components/falcon-jit-compiler/ (Accessed 05/10/2022)
5. Nicolas Geoffray. Fostering Systems Research with Managed Runtimes. [Electronic resource]. URL: https://vmkit.llvm.org/publications/thesis.html (Accessed 05/10/2022)
6. Skvortsov L.V. Baev R.V. Dolgorukova K.Yu. Sharygin E.Yu. Development of a compiler for the stack processor architecture TF16 based on LLVM [Electronic resource]. URL: https://elibrary.ru/item.asp?id=47410920 (accessed 05/10/2022)

© Лукьянов А.Ю., 2022 Научно-образовательный журнал для студентов и преподавателей «StudNet» №5/2022.

Для цитирования: Лукьянов А.Ю., РАЗРАБОТКА ШАБЛОННОГО ИНТЕРПРЕТАТОРА OPENJDK С ПРИМЕНЕНИЕМ LLVM// Научно-образовательный журнал для студентов и преподавателей «StudNet» №5/2022.