Министерство науки и высшего образования Российской Федерации Федеральное государственное автономное образовательное учреждение высшего образования «Московский физико-технический институт (национальный исследовательский университет)»

Физтех-школа Радиотехники и Компьютерных Технологий
Кафедра микропроцессорных технологий в интеллектуальных системах управления

Выпускная квалификационная работа бакалавра

Разработка модуля автогенерации платформо-зависимых констант для целей кросс-компиляции

Автор:
Студент Б01-909а группы
Кофанов Даниил Сергеевич

Научный руководитель:
научная степень
Ишин Павел Андреевич

Москва 2023

Аннотация

Разработка модуля автогенерации платформо-зависимых констант для целей кросс-компиляции

Кофанов Даниил Сергеевич

В Главе 1 данной работе определена практическая ценность кросс-компиляции в контексте виртуальных машин. В Главе 2 на примере работы абстрактной виртуальной машины дано определение платформо-зависимых констант и сформулирована связанная с ними проблема, возникающая при кросс-компиляции. В Главе 3 предложена идея гибкого и масштабируемого решения этой проблемы, основанного на создании некоторого интерфейса и использовании системного (т. е. не относящегося к виртуальным машинам) кросс-компилятора. Произведено сравнение с альтернативным решением. В Главе 4 дан обзор имплементации найденного решения в виртуальной машине с открытым исходным кодом ArkCompiler, основанного на модуле ExternalProject утилиты cmake. В Главе 5 отмечены преимущества, включающие высокую степень автоматизированности, удобства в использовании разработанного интерфейса. Также выделены недостатки текущей имплементации, предложены пути их устранения.

Оглавление

Глава 1

Введение

На момент написания этой работы сложно переоценить влияние мобильных устройств на повседневную жизнь. Смартфоны предоставляют огромный спектр возможностей, начиная от коммуникации и игр, заканчивая рабочими потребностями, навигацией и отслеживанием физического состояния человека. Количество использующихся телефонов значительно преобладает над количеством ПК и ноутбуков. Однако такой массовый характер был бы невозможен в отсутствии многообразия устройств на рынке. Действительно, развитие технологий безусловно исходит из потребности в новых возможностях, однако финансирование этого развития поступает из уже существующих и отлаженных технологий. Кроме того, только что изобретённой технологии требуется некоторое время, чтобы спрос на неё появился и вырос. Исходя из этого, а также готовности потребителя пойти на компромисс между ценой за устройство и количеством и качеством предоставляемых им функций, представленные на рынке устройства коренным образом отличаются между собой.

С другой стороны, имея такое разнообразие, разрабатывать приложения и отлаживать под каждое устройство было бы слишком дорогой задачей. Чтобы решить эту проблему, вводится понятие виртуальной машины, то есть некоторого абстрактного вычислителя, способного исполнять инструкции из определённого набора, тем самым обеспечивая разработчиков приложений некоторым количеством гарантий. Кажется очевидным, что этот подход аналогичен договорённости между программистами и разработчиками микропроцессоров в виде архитектуры системы команд и почти в той же степени необходим. Главное отличие заключается в более высоком уровне абстракции, что позволяет переложить ответственность за управление ресурсами приложения, например выделение и освобождение памяти, на саму виртуальную машину, тем самым упрощая код и снижая требовательность к опыту разработчика. В случае мобильных устройств, виртуальная машина рассматривается практически как неотъемлимая часть операционной системы, которая, помимо обеспечения гарантий разработчикам приложений, служит барьером безопасности для пользователей, устанавливающих и использующих самые разные приложения.

Высокоуровневость виртуальных машин позволяет выражать программы более компактно в сравнении с выражением в нативном коде. В то же время, поочерёдное исполнение инструкций виртуальной машины (интерпретация байткода) существенно замедляет алгоритм. Выходом из этой ситуации стало использование just-in-time компиляции, то есть генерация оптимизированного нативного кода для часто вызываемых функций, происходящая параллельно выполнению основного алгоритма. Использование как классических оптимизаций, например межпроцедурной оптимизации, раскрутки циклов, так и специфичных JIT-оптимизаций, например профилирование типов в случае динамических языков, позволяет достичь производительно-

сти, нивелирующей накладные расходы вносимые этим уровнем абстракции. В виртуальных машинах так же используется ahead-of-time компиляция, производящаяся до запуска приложения. Это позволяет избежать излишних затрат энергии при повторных запусках приложений связанных с JIT-компиляцией, скомпилировать большее количество методов, и, возможно, применить большее количество оптимизаций, тем самым экономя заряд батареи мобильных устройств, ускоряя время запуска и улучшая производительность приложения за счёт хранения результатов компиляции (AOT-файлов) на устройстве.

Очевиден тот факт, что множество приложений в своих подзадачах часто используют идентичные алгоритмы, такие как сортировка, работа со строками и т. д., и что множественная имплементация того или иного алгоритма ведёт к практической сложности поддержки такого кода. В виду этого, многие алгоритмы, оптимизированные с целью быть приемлимо эффективными для самых различных нагрузок, образуют стандартную библиотеку языка и поставляются вместе с виртуальной машиной. Будучи часто переиспользуемой, стандартная библиотека AOT-компилируется и хранится на устройстве, что позволяет любому приложению гарантировано использовать оптимизированный нативный код, не совершая излишнюю работу по JIT-компиляции.

Выпуская огромное количество устройств, запускать AOT-компиляцию стандартной библиотеки на каждом из них было бы нерационально для производителя и сопровождалось бы ощутимыми издержками производства. Их можно избежать путём единовременной компиляции и последующей загрузки полученного образа на устройства, например при прошивке. Для подготовки образа виртуальной машины и операционной системы используется специальный набор утилит, включающий в себя кросс-компилятор и опции компиляции, который работает на отличной от целевого устройства платформе и генерирующий исполняемые файлы в виде нативного кода целевого устройства. Для такого набора утилит принят термин таргет-тулчейн, в то время как аналогичный набор утилит, предназначенный для получения исполняемых файлов, нативных для той же платформы на которой они и были сгенерированы, обозначается как хост-тулчейн.

Однако, таргет-тулчейн не подходит для непосредственной подготовки AOT-файла стандартной библиотеки. Это связано с тем, что стандартная библиотека расчитана на исполнение в среде виртуальной машины, отличающейся от окружения создаваемого операционной системой и может проявляться в том, что тулчейн-компилятор и AOT-компилятор виртуальной машины могут использовать разное соглашение о вызовах. В виду этого появляется вопрос о кросс-компиляторе в терминах виртуальной машины, что на самом деле означает возможность генерации AOT-файлов виртуальной машины для определённого устройства на другой платформе, например на сервере. Эта задача имеет множество тонкостей, одной из которых посвящена данная работа - проблема зависимости некоторых численных значений от платформы (платформо-зависимых констант). Наиболее ярким примером является размер указателя. Однако в этом случае зависимость от платформы достаточно легко "предсказать": на AMD64-совместимой платформе это скорее всего 64 бита, а на ARM32 - 32. Другим, более трудноразрешимым и содержательным примером является смещение до полей в структурах данных. Будучи собранным на одной и той же платформе одним и тем же компилятором, в заисимости от опций компиляции результат может быть различным. В последующих главах излагается подробное описание этой проблемы, на примере абстрактной виртуальной машины демонстрируется один из источников таких значений, а также предлагается решение, основанное на использовании таргет-тулчейна, нашедшее применение в виртуальной

машине ArkCompiler. Как будет показано в дальнейшем, эта имплементация отличается высоким уровнем автоматизации и масштабируемости, делающей её малозаметной и лёгкой в использовании. Кроме того, помимо основной функции, она вносит дополнительную степень верификации AOT-файлов.

Глава 2

Постановка задачи

Примем следующие определения.

- *Тулчейн* - набор утилит, позволяющий транслировать C++-программы в некоторый бинарный код.
- *Платформа* - предполагаемое тулчейном окружение, в котором будет исполняться сгенерированный им код. Подразумевается, что каждая платформа в достаточной степени однозначно определяется некоторым тулчейном. В качестве синонима, в данной работе будет также использоваться термин *архитектура*.
- *Платформо-зависимая константа* - имеющее определённое имя, определённое в терминах C++ константное выражение, численное значение которой каким-либо образом зависит от платформы.

Итак,
Целью данной работы является разработка модуля, способного каким-либо способом вычислять заранее обговорённый набор платформо-зависимых констант для интересующих разработчика платформ.

Глава 3

Обзор существующих решений

Здесь надо рассмотреть все существующие решения поставленной задачи, но не просто пересказать, в чем там дело, а оценить степень их соответствия тем ограничениям, которые были сформулированы в постановке задачи.

Глава 4

Исследование и построение решения задачи

Здесь надо декомпозировать большую задачу из постановки на подзадачи и продолжать этот процесс, пока подзадачи не станут достаточно простыми, чтобы их можно было бы решить напрямую (например, поставив какой-то эксперимент или доказав теорему) или найти готовое решение.

Глава 5

Описание практической части

Если в рамках работы писался какой-то код, здесь должно быть его описание: выбранный язык и библиотеки и мотивы выбора, архитектура, схема функционирования, теоретическая сложность алгоритма, характеристики функционирования (скорость/память).

Глава 6

Заключение

Здесь надо перечислить все результаты, полученные в ходе работы. Из текста должно быть понятно, в какой мере решена поставленная задача.

Литература

[1] *Mott-Smith, H.* The theory of collectors in gaseous discharges / H. Mott-Smith, I. Langmuir // *Phys. Rev.* — 1926. — Vol. 28.

[2] *Морз, Р.* Бесстолкновительный PIC-метод / Р. Морз // Вычислительные методы в физике плазмы / Ed. by Б. Олдера, С. Фернбаха, М. Ротенберга. — М.: Мир, 1974.

[3] *Киселёв, А. А.* Численное моделирование захвата ионов бесстолкновительной плазмы электрическим полем поглощающей сферы / А. А. Киселёв, Долгоносов М. С., Красовский В. Л. // Девятая ежегодная конференция «Физика плазмы в Солнечной системе». — 2014.