

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 данной работе определена практическая ценность кросс-компиляции в контексте виртуальных машин. На примере работы абстрактной виртуальной машины дано определение платформо-зависимых констант и сформулирована связанная с ними проблема, возникающая при кросс-компиляции. 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 подходом. Дан обзор имплементации найденного решения в виртуальной машине с открытым исходным кодом ArkCompiler, основанного на модуле ExternalProject утилиты cmake. Отмечены преимущества, включающие высокую степень автоматизированности, удобства в использовании разработанного интерфейса. Также выделены недостатки текущей имплементации, предложены пути их устранения.

Оглавление

1	Введение	2
2	Постановка задачи	5
2.1	Формулировка цели	5
2.2	Обоснование цели	5
3	Сравнение подходов к проблеме платформо-зависимых констант	8
3.1	Фиксирование констант	8
3.2	Вычисление констант	9
3.3	Итог сравнения	10
4	Исследование и построение решения задачи	11
5	Описание практической части	12
6	Заключение	13

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 используемых 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 неотъемлемая 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 производительности

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 (АОТ-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 приемлемо эффективными для самых различных нагрузок, образуют стандартную библиотеку языка и поставляются вместе с виртуальной машиной. Будучи часто переиспользуемой, стандартная библиотека АОТ-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 АОТ-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 АОТ-файла стандартной библиотеки. Это связано с тем, что стандартная библиотека рассчитана на исполнение в среде виртуальной машины, отличающейся от окружения создаваемого операционной системой и может проявляться в том, что тулчейн-компилятор и АОТ-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 АОТ-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 зависимости 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

2.1 Формулировка цели

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*.
- *Платформозависимая константа* – именованное, 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 платформозависимых констант для интересующих разработчика платформ, а так же имеющего интерфейс, предоставляющий доступ к значению любой из констант на каждой из архитектур.

Для обоснования этой задачи рассмотрим более подробно один из источников платформозависимых констант.

2.2 Обоснование цели

Рассмотрим более подробно язык программирования, который поддерживает какая-либо интересующая нас виртуальная машина, реализованная, для определённости, на языке C++. Положим также, что в нём существуют некоторые встроенные типы, это могут быть числа, строки массивы, словари и т.д., а операции с этими сущностями, например их создание, модификация, получение состояния, выражаются в специальных инструкциях виртуальной машины. Тогда, при исполнении программы, созданию массива будет соответствовать выполнение некой инструкции *newarray*, а получению длины этого массива – выполнение *arraylength* (рис. 2.1).

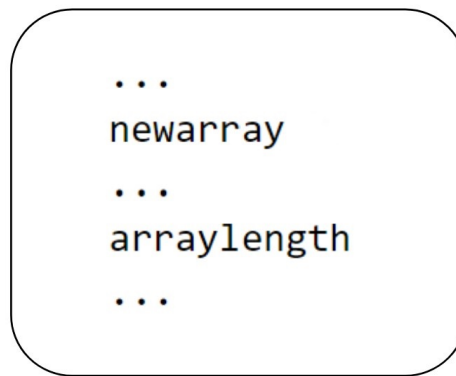


Рис. 2.1: Пример инструкций в программе

С другой стороны, каждый встроенный тип находит отражение в некотором C++-классе. При этом, при исполнении инструкции *newarray* будет происходить аллокация экземпляра этого класса, а при исполнении *arraylength* будет произведено чтение поля *size_* одного из экземпляров (рис. 2.2).

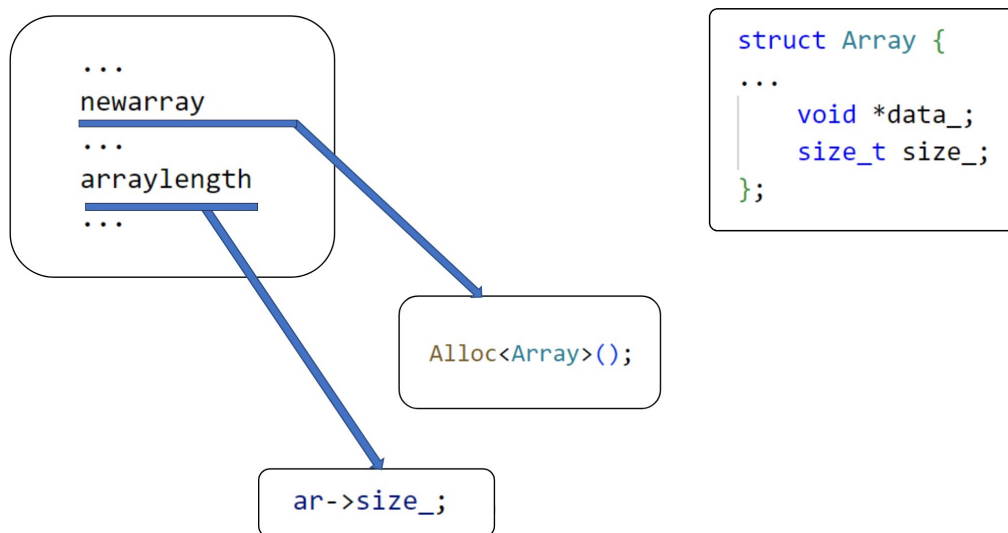


Рис. 2.2: Реализация семантики языка

В нативном формате, обращение к этому полю будет выражено в виде машинной инструкции загрузки по указателю с константным смещением. Возвращаясь к кросс-компилятору виртуальной машины, эта инструкция будет создана на финальном этапе, кодогенерации, из соответствующей инструкции промежуточного представления. Фактически, эта инструкция промежуточного представления абстрагирует бинарный формат машинной инструкции загрузки по смещению конкретной архитектуры, отделяя зависимость от платформы. Однако, такой абстракции для отделения оказывается недостаточно. Дело в том, что смещение до поля класса (эквивалентное в рассматриваемой ситуации смещению в инструкции загрузки), в общем случае, зависит от платформы. Это может происходить по разным причинам, например – из-за зависимости размера какого-либо элемента структуры данных от платформы, выравнивания по разным адресам. Так, рассматриваемое ранее смещение до поля *size_* структуры *Array* на 64-битной архитектуре AMD64 будет равно 8 байтам, хотя на архитектуре ARM32 – 4. Если упустить из рассмотрения этот

факт, то кросс-компилятор виртуальной машины, запущенный на AMD64-сервере, при генерации ARM32-инструкций будет использовать соответствующее текущей платформе значение смещения. Тогда, при запуске такого кода на ARM32-телефоне, исполнение такой инструкции приведёт к некорректному поведению и является программной уязвимостью. Вопрос корректности и безопасности генерируемого кода является первостепенным в процессе компиляции, что подчёркивает значимость этой проблемы. Можно выделить два подхода, позволяющие разрешить этот конфликт: первый – сделать смещения для всех платформ одинаковыми, второй – каким-либо образом сообщать компилятору значение константы на платформе, для которой происходит компиляция. В следующей главе будет произведено сравнение этих подходов из которого становится ясным, почему предпочтение было отдано последнему из них.

Глава 3

Сравнение подходов к проблеме платформено-зависимых констант

В этой главе описываются два подхода к решению проблемы платформено-зависимых величин – фиксирование величин между всеми платформами и, контрастное ему, вычисление величин для каждой платформы, а также производится сравнение этих подходов. Хочется отметить, что данная работа посвящена последнему из них, а так же что её возникновение обязано успешной попытке устранить недостатки, связанные с первым подходом.

3.1 Фиксирование констант

Этот подход заключается в поддержании кода в таком состоянии, при котором величины будут иметь одно и то же значение на каждой из платформ. Из достоинств данного подхода можно отметить простоту идеи, однако на практике, следование такому подходу сопровождается нетривиальными сложностями.

Например, к этому может привести использование атрибутов, не относящихся к стандарту языка. При использовании разных тулчейнов, хотя и относящихся к одной и той же платформе (например GCC и LLVM), может получаться отличающийся результат.

Далее, остаётся открытым вопрос самого фиксирования этих величин: ранее был рассмотрен лишь один из примеров – смещения полей в структуре данных, однако фиксирование величин другой природы может быть связано со значительными сложностями, если будет возможно в принципе.

Кроме этого, для верификации, каждой константе должна соответствовать статическая проверка на то, что текущее значение величины действительно совпадает с некоторым жёстко закодированным числом. Из-за этого, исходный код проекта приобретает избыточную и малоинформативную логику, необходимую лишь для проверки корректности самого подхода, причём на фоне большого числа таких констант сложно гарантировать существование такой проверки для каждой из них. Более того, при активной разработке и модификации виртуальной машины, фактические значения констант могут меняться (что можно легко проследить на примере изменения смещений при добавлении нового поля в структуру), инициируя необходимость модификации уже существующих проверок. Разработчик неизбежно будет сталкиваться с последствиями такого подхода, поддержка проекта будет становиться всё сложнее.

3.2 Вычисление констант

Альтернативой является более консервативный подход, описываемый как вычисление величин для каждой платформы, который имплементирован в ходе данной работы. Под консервативностью следует понимать отсутствие каких-либо требований и ограничений на значения констант, описанных ранее, и отсутствие вмешательств в исходный код, связанных с этими ограничениями. Хотя на первый взгляд способ вычисления "гостевых" значений кажется неочевидным, предположив что он существует, множество описанных ранее проблем устраняются сами собой или вовсе лишены смысла.

Так, не появляются затруднения при использовании платформо-зависимого функционала какого-либо тулчейна. Также, не приходится изобретать способ фиксирования для констант разного происхождения, что делает такой подход более однородным и гибким. Не возникает и вопроса сверки констант на разных платформах, так как допускается их различие. Это в свою очередь допускает решения, являющимися локально оптимальными для каждой из платформ. Например, рассмотрим структуру данных с двумя 4-байтными полями. Выравнивание на 8 байт на 32-битной архитектуре будет связано с накладными расходами на размер используемой памяти, в то время как выравнивание на 4 байта может повлечь снижение производительности на 64-битной платформе. Данный подход позволяет использовать 4-байтное выравнивание на 32-битной платформе и 8-байтное выравнивание на 64-битной платформе для одной и той же структуры данных.

Таким образом, этот подход, а именно отказ от "модификации" констант, устраняет большинство, если не все, недостатки, описанные в предыдущей секции. С другой стороны, формально он допускает ранее описанный подход, то есть в определённом смысле не противоречит ему: значения каждой из величин на всех платформах могут (искусственно или естественно) совпасть. С практической точки зрения, это означает, что миграция с использования первого подхода на использование второго может происходить поэтапно, то есть сначала можно наладить инфраструктурную часть, а затем, по мере необходимости, ослаблять введённые искусственно ограничения.

Важно сделать следующее замечание. Необходимо, чтобы полученный кросс-компилятором, рассчитывающим на какую-то конкретную платформу, код действительно запускался в контексте виртуальной машины, собранной для той же платформы. В описываемой ниже имплементации эта проблема устраняется путём вычисления контрольной суммы CRC32 из значений констант на целевой платформе и последующей её записи как в каждый АОТ-файл, генерируемый кросс-компилятором виртуальной машины, так и в сам целевой образ виртуальной машины. Во время инсталляции АОТ-файла происходит сверка этих значений, что позволяет обнаружить потенциальные ошибки, связанные с несовместимостью между кросс-скомпилированным кодом и самой виртуальной машиной. Однако в таком виде, эта особенность не является недостатком. Действительно, значение какой-либо константы может изменяться в зависимости от версии виртуальной машины, а совпадение версии кросс-компилятора, находящегося на сервере, и самой виртуальной машины, находящейся на устройстве, при ошибках инфраструктуры может нарушиться. Такая контрольная сумма вносит дополнительную степень верификации, косвенно проверяя совместимость версий и несёт дополнительную полезную нагрузку, никак не связанную с выбранным подходом.

3.3 Итог сравнения

Исходя из вышесказанного, второй подход, связанный с вычислением констант для каждой платформы, является более предпочтительным в виду своей универсальности. Характерная ему однородность, определённая выше, позволяет в высокой степени автоматизировать весь процесс, связанный с вычислением значений и поддержанием их в консистентном состоянии (как в случае чистой, так и инкрементальной сборки), фактически отделяясь в независимый инкапсулирующий модуль. Разработчик сталкивается с этим модулем лишь через интерфейс и у него нет необходимости вручную вносить изменения в этот модуль: достаточно лишь определения платформо-зависимой константы, после чего её можно использовать в основной части проекта.

Дальнейшее повествование раскрывает детали реализации выбранного подхода.

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, Долгонос М. С., Красовский В. Л.* // Девятая ежегодная конференция «Физика плазмы в Солнечной системе». — 2014.