Оптимизация динамической загрузки библиотек на архитектуре ARM

Е.А. Кудряшов <eugene.a.kudryashov@gmail.com>
Д.М. Мельник <dm@ispras.ru>
А.В. Монаков <amonakov@ispras.ru>
Институт системного программирования РАН,
109004, Россия, г. Москва, ул. А. Солженицына, д. 25

Аннотация. В статье рассматривается подход к оптимизации вызовов внешних функций в позиционно-независимом коде, основанный на выдаче вызовов непосредственно через глобальную таблицу смещений (GOT), минуя таблицу компоновки процедур (PLT). Стандартные механизмы кодогенерации на операционной системе Linux предполагают создание PLT не только для основного модуля (который является позиционно-зависимым и полагается на механизм PLT для вызовов внешних процедур), но и для динамических библиотек, где PLT используется также для организации ленивого связывания; однако, использование PLT требует дополнительной инструкции перехода, может иметь низкую локальность по кешу и на некоторых архитектурах накладывает дополнительные ограничения на работу компилятора в месте вызова. Реализация вызовов внешних функций в виде косвенных переходов на адреса, загруженные непосредственно из GOT в месте вызова, позволяет избежать недостатков вызовов через PLT, ценой отказа от возможности ленивого связывания и, возможно, увеличением размера кода. Была исследована реализация этой оптимизации для архитектур x86 и ARM в компиляторе GCC. Было обнаружено, что на архитектуре ARM отсутствуют типы релокаций, которые позволили бы генерировать оптимальный код для загрузок из GOT. Для решения этой проблемы в GCC и Binutils (в ассемблере и компоновщике) были реализованы недостающие типы релокаций, позволяющие построить адрес позиции в GOT относительно счетчика команд, используя инструкции movt, movw. Проведенное тестирование свидетельствует, что предложенная оптимизация позволяет получить увеличение производительности, несмотря на увеличение размеров динамических библиотек.

Ключевые слова: оптимизация программ; динамический загрузчик; глобальная таблица смещений; таблица компоновки процедур; релокация; архитектура ARM

DOI: 10.15514/ISPRAS-2016-28(1)-4

Для цитирования: Кудряшов Е.А., Мельник Д.М., Монаков А.В. Оптимизация динамической загрузки библиотек на архитектуре ARM. Труды ИСП РАН, том 28, вып. 1, 2016 г., стр. 63-80. DOI: 10.15514/ISPRAS-2016-28(1)-4

1. Введение

Операционные системы обычно поддерживают два типа библиотек: статические, которые становятся частью загружаемого образа программы при ее сборке, и динамические, которые загружаются в память отдельно (если их там нет) при запуске программы.

Выбор между статическими или динамическими библиотеками требует компромиссов. Использование динамических библиотек позволяет сократить расходование оперативной памяти, когда запущены одновременно несколько различных программ, использующих один набор библиотек, но это достигается ценой как правило более сложного позиционно-независимого кода (PIC) в библиотеках [1], и сложным процессом динамического связывания библиотек.

Загрузка и связывание динамических библиотек выполняется динамическим загрузчиком, который получает управление после того, как ядро операционной системы разместило в памяти программу. Перед началом выполнения программы он выполняет поиск и загрузку всех файлов с динамическими библиотеками, которые использует программа. Загрузка выполняется на заранее неизвестные адреса, и поэтому часть данных в загруженных образах библиотек подвергается процедуре перемещения: динамический загрузчик должен переписать часть загруженных данных в зависимости от адреса, на который загружена библиотека, в соответствии со специальными аннотациями – релокациями. Кроме того, библиотека может использовать функции из других загруженных модулей, и для них нужно выполнить динамическое связывание, то есть подставить конкретные адреса в соответствие символическим именам. Существует два подхода к решению данной задачи: разрешение ссылок во время запуска или во время исполнения. Оба имеют негативные эффекты: разрешение во время запуска увеличивает время запуска программы, разрешение во время исполнения создаёт накладные расходы на каждый вызов функции.

Цель данной работы – оптимизировать взаимодействие программ с динамическими библиотеками в режиме PIC, с помощью поиска компромисса между временем запуска программы и накладными расходами на вызов функций. В дополнении к этому планируется добиться генерации более эффективного кода посредством ввода новых типов релокаций.

Дальнейшее изложение будет построено следующим образом. Сначала будет описано использование позиционно-независимого кода в динамических библиотеках. После этого следует описание оптимизации и её особенности реализации на архитектуре ARM. В заключении будут приведены результаты оптимизации, протестированные на встраиваемой реляционной базе SQLite.

2. Особенности позицинно-независимого кода

Режим позиционно-независимого кода подразумевает возможность загрузки модуля по произвольному базовому адресу без изменений загруженного

программного кода (именно это позволяет разделять одну копию кода библиотеки между всеми использующими ее процессами). Это означает, что все инструкции, обращающиеся к динамическим адресам, не содержат их как операнд, а загружают их косвенно из области данных, и все динамические релокации применяются к секции с данными, а не с кодом.

2.1 Смещение относительно text и data секций

Одним из ключевых аспектов PIC является то, что код полагается на разницу в адресном пространстве между text и data секциями программы, которые известны компоновщику. Когда он соединяет несколько объектных файлов вместе, он собирает все их секции (т.е. все text секции различных файлов будут собраны в одну большую). Следовательно, компоновщик имеет информацию как о размерах секций, так и об их относительном положении.

Например, data секция может идти непосредственно сразу после text секции, тогда смещение любой инструкции в text секции до начала data секции будет составлять длину text секции за вычетом смещения данной инструкции от начала text секции. Обе составляющие (длина text секции и смещение инструкции относительно начала text секции) известны компоновщику.

На рис. 1 text секция была загружена по определенному адресу (неизвестный во время компоновки) 0xXXXX0000 (где XXXX – некий адрес, который в данном контексте не имеет значения), сразу после неё была загружена data секция со смещением 0xXXXXA000. Если некой инструкции со смещением 0x100 в text секции понадобится что-то в data секции, компоновщик знает относительный сдвиг (в данном случае 0x9F00) и может использовать это в инструкции.

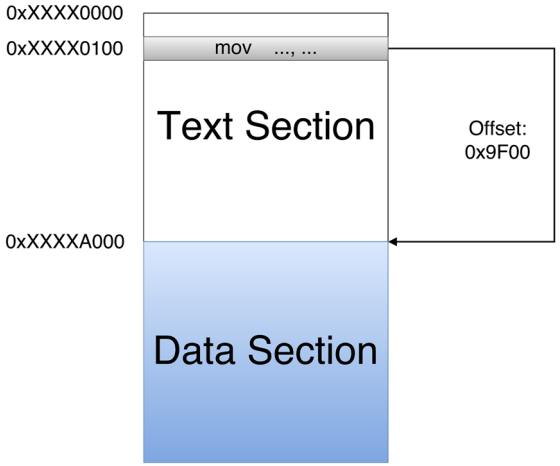

Рис. 1. Смещения относительно text и data секций

Fig. 1. Offsets between text and data sections

Стоит заметить, что компоновщик сможет высчитать верное смещение и в случаях, когда имеются дополнительные секции между text и data, и в случаях, когда data предшествует text секции: достаточно лишь того, что секции не перемещаемы друг относительно друга после завершения компоновки.

2.2 Глобальная таблица смещений

Глобальная таблица смещений (Global Offset Table, GOT) это таблица абсолютных адресов [1]. Когда инструкция кода обращается к глобальной переменной, то вместо обращения по абсолютному адресу (которое требует релокации в text секции), она обращается к GOT по относительному. Таким образом, смещение в коде относительное, и оно известно компоновщику. На рис. 2 представлена схема GOT с адресами на глобальные переменные.

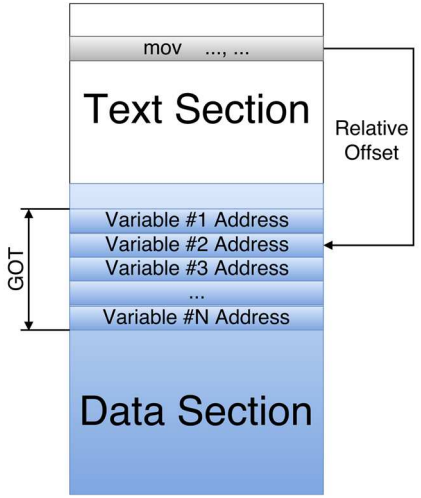

Рис. 2. Обращение к глобальной таблице смещений

Fig. 2. Reference to the the Global Offset Table

Данный метод позволяет вынести релокации из секции кода в секцию data путём перенаправления ссылок на переменные через GOT.

2.3 Вызов функций в режиме PIC

До этого речь шла о взаимодействии с переменными в режиме PIC. Теоретически, точно такой же подход можно использовать и для работы с вызовом функций: глобальная таблица смещений будет содержать адреса функций, которые будут заполняться во время загрузки программы. Но в действительности вызов функций работает иначе и немного сложнее.

2.3.1 Ленивое связывание

Когда программа ссылается на некоторую внешнюю функцию, то настоящий адрес данной функции неизвестен во время загрузки. Определение и присваивание данного адреса называется связыванием, и этим занимается динамический загрузчик, когда динамическая библиотека загружается в память. Процесс связывания требует времени, т.к. загрузчик должен пройтись по специальным таблицам в поисках необходимой функции. Время для поиска одной функции не велико, но библиотеки обычно содержат большое количество функций. Кроме того, большинство из этих поисков будет выполняться напрасно, потому что программы редко используют полный функционал библиотек. Также не стоит забывать о различных функциях обработки ошибок, которые, как правило, не вызываются вовсе.

Для ускорения данного процесса был разработан подход ленивого связывания. Прилагательное «ленивое» является обобщающим названием оптимизаций, в которых выполнение операций откладывается на последний момент, когда её выполнение действительно необходимо. Данный подход позволяет избежать выполнения ненужных операций, которые никогда не понадобятся во время исполнения программы, однако это достигается ценой усложнения динамического загрузчика и опасностью аварийного завершения программы, если при ленивом связывании нужный символ не может быть найден. Ленивое связывание реализуется с помощью таблицы компоновки процедур.

2.3.2 Таблица компоновки процедур

Таблица компоновки процедур (Procedure Linkage Table, PLT) – часть секции text [1]. Она состоит из множества небольших одинаковых фрагментов исполняемого кода, по одному для каждой внешней функции. Вместо вызова функции напрямую происходит перенаправление на код из таблицы процедур, который совершает вызов необходимой функции. Данный участок кода иногда называется «трамплином». В случае, если используется ленивое связывание, трамплин в случае первого вызова функции разрешает и вызывает её, иначе сразу вызывает её, т.к. разрешение уже было произведено и абсолютный адрес известен. Использование трамплина для каждой функции обуславливается необходимостью загрузки адреса функции и подготовкой аргументов для её разрешения.

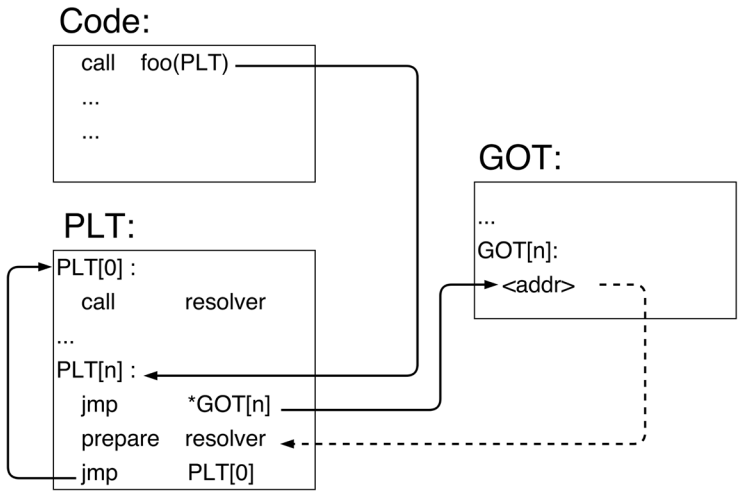

Рис. 3. Первый вызов функции из динамической библиотеки

Fig. 3. First invocation of a function from a dynamic library

Процесс первого вызова функции изображен на рис. 3, рассмотрим его детальнее:

- когда происходит вызов внешней функции *foo*, то компилятор заменяет данный вызов на вызов кода в PLT (в данном примере *foo(PLT)*);
- таблица компоновки процедур состоит из специального нулевого элемента (о нём будет сказано позже) и из последовательности структурно одинаковых фрагментов кода; каждый фрагмент кода отвечает за динамическую загрузку определенной функции;
- каждый элемент PLT, кроме нулевого, состоит из следующих частей:
  - переход по соответствующему адресу глобальной таблицы смещений;
  - подготовка аргументов для процедуры *resolver;*
  - переход на нулевой элемент PLT;
- нулевой элемент PLT – это вызов процедуры *resolver*, которая расположена в самом динамическом загрузчике; данная процедура занимается получением абсолютного адреса функций;

- до того, как адрес функции будет разрешен, *n*-ый элемент глобальной таблицы смещений имеет адрес следующей инструкции в PLT; именно поэтому на рис. 3 стрелка обозначена пунктиром.

При вызове *foo* в первый раз происходит следующее:

- вызывается PLT[*n*], и происходит переход на адрес, расположенный в GOT[*n*];
- адрес указывает на следующую инструкцию в PLT[*n*], для подготовки аргументов процедуры *resolver*;
- вызывается процедура *resolver*;
- *resolver* получает абсолютный адрес *foo*, и помещает его в GOT[*n*] и вызывает *foo*.

При последующих вызовах *foo*, в силу ленивого связывания, картина изменится (см. рис. 4):

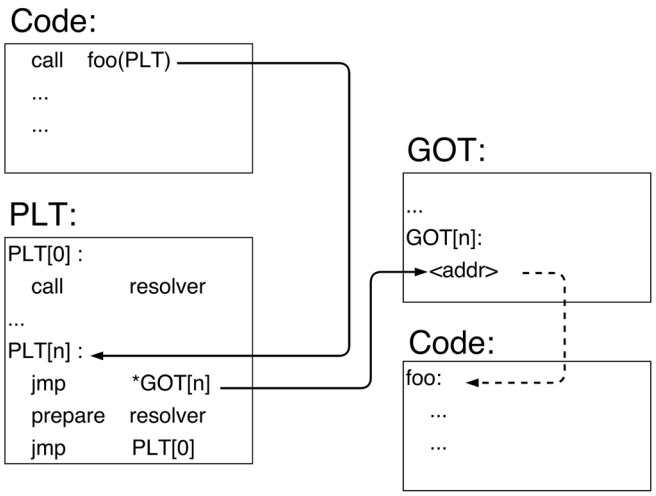

Рис. 4. Последующие вызовы функций после связывания

Fig. 4. Subsequent invocations of bound dynamic functions

- вызывается PLT[*n*], и происходит переход на адрес, расположенный в GOT[*n*];
- GOT[*n*] указывает на *foo*, соответственно происходит вызов *foo*.

Другими словами, все последующие вызовы *foo* происходят в обход процедуры *resolver* и стоят одного дополнительного перехода.

Описанный подход позволяет использовать ленивое связывание функций. Также он делает text секцию позиционно независимой, т.к. единственное место, где используется абсолютный адрес – это в глобальной таблице смещений, которая расположена в data секции.

3. Отказ от использования PLT

Хотя использование PLT необходимо в программных модулях, скомпонованных из позиционно-зависимого кода (в них все места вызова используют абсолютный адрес, на место которого компоновщик вынужден подставить адрес PLT-трамплина), в коде динамических библиотек использовать PLT не обязательно: как отмечалось раньше, позиционно-независимый код мог бы загружать адрес вызываемых функций непосредственно из GOT-таблицы. Выгода от использования PLT заключается исключительно в возможности ленивого связывания функций. Но для программ, где производительность является критически важной, в этом нет особой необходимости. Исключение составляет случай, когда программа используется очень часто, её время исполнения крайне невелико, и большая часть функций не используется: тогда полное динамическое связывание при каждом запуске программы будет вносить заметный вклад в общее время работы; однако в таком случае статическая компоновка позволит существенно ускорить загрузку.

С другой стороны, вызовы через PLT требуют выполнения дополнительных инструкций, и переход на трамплин может не обладать локальностью по кэшу инструкций. Таким образом, генерация кода, который выполняет вызов внешних функций непосредственно через GOT, минуя PLT, может быть оптимальнее.

Оптимизация заключается в следующем: когда компилятор встречает вызов внешней функции, то он не формирует обычный вызов, который будет преобразован в PLT вызов (см. рис. 3, *foo(PLT)*), а записывает в регистр адрес функции из глобальной таблицы смещений и делает вызов функции по регистру. Для этого необходимо, чтобы GOT была уже проинициализирована, т.е. она заполняется адресами функций во время запуска. Как следствие, необходимость в ленивом связывании пропадает.

Данная оптимизация была реализована нами на архитектурах x86, x86-64, позднее она появилась и на архитектуре AArch64 (64-битный ARM) [2]. Однако, реализация данной оптимизации на архитектуре ARM имеет существенные отличия.

4. Устранение PLT-переходов на x86-архитектурах

Изначально вариант генерации кода, избегающий использования PLT, был реализован в GCC для процессорных архитектур IA-32 и AMD64. Было отмечено, что в компиляторе присутствует функциональность, называемая «function CSE», решающая похожую задачу: обеспечить возможность оптимизации нескольких вызовов одной и той же функции путем загрузки адреса этой функции на регистр; это выделяет взятие адреса функции как отдельную инструкцию во внутреннем представлении компилятора и позволяет автоматически удалить дублирующиеся загрузки в ходе дальнейших оптимизационных проходов. Для большинства архитектур эта функциональность не активна, вероятно из-за исключительной редкости случаев, когда за счет нее возможно существенное улучшение кода.

Соответственно, при выдаче RTL-кода для вызовов функций, при активных флагах -fno-plt и -fPIC, вызов выдается как последовательность двух RTL-инструкций: первая вычисляет абсолютный адрес вызываемой функции в псевдорегистр, а вторая выполняет косвенный вызов по этому регистру. Поскольку генерируется PIC-код, первая инструкция соответствует загрузке из таблицы GOT (в позиционно-зависимом коде это было бы записью абсолютного адреса непосредственно в регистр).

Для 32-битных x86 архитектур помимо исключения перехода на PLT-трамплин есть и второй фактор улучшения кода: согласно соглашению о вызовах, при переходе на PLT-трамплин, регистр ebx должен указывать на GOT-таблицу. Поскольку на x86 нет относительной адресации относительно текущей инструкции (регистра eip), PLT-трамплины используют ebx как базу для загрузки целевого адреса из GOT. Однако ebx является call-saved регистром: вызываемая функция должна восстанавливать его значение перед возвратом. Таким образом, оптимизация хвостовых вызовов при переходе на PLT-трамплин невозможна. Использование же переходов через GOT позволяет использовать любой регистр общего назначения в качестве базы для позиционно-независимой адресации, так что для хвостовых переходов компилятору достаточно распределить адрес GOT на стираемый при вызовах регистр, например, eax. Кроме того, открываются и другие возможности оптимизации за счет переноса инструкции загрузки из GOT: вынос из циклов, планирование, удаление избыточности.

При разработке и тестировании этой оптимизации были обнаружены и исправлены упущения в компиляторе. В частности, порядок перечисления регистровых классов был неоптимален для 32-битного x86, что иногда не позволяло хорошо выбрать регистр для адресации GOT. При выдаче хвостовых переходов без надобности была запрещена косвенная адресация по регистру eax.

Для оценки улучшения производительности использовался Clang/LLVM, скомпилированные так, что динамически загружается более 100 библиотек с более 24000 динамических символов. Для компиляции тривиального C++

файла, Clang с -fno-plt на 20% медленнее, чем базовый, из-за отсутствия ленивого связывания, но уже на 10% быстрее, когда оно запрещено. При компиляции большого файла наблюдается существенное (10-12%) ускорение [3].

5. Применение на архитектуре ARM

Существует несколько особенностей, которые препятствуют применению данной оптимизации на архитектуре ARM:

- обратная оптимизация на этапе combine;
- отсутствие необходимых типов релокаций для генерации эффективного кода.

5.1 Combine

Особенность на этапе combine заключается в следующем: после того, как была произведена запись адреса функции из глобальной таблицы в регистр (инструкция вызова функции разделилась на несколько инструкций), происходит объединение данных инструкций в одну, которая была изначально.

Combine – это оптимизационный проход в компиляторе GCC, выполняющий объединение нескольких инструкций в одну. Могут объединяться как две инструкции, так и триплеты и даже четвёрки команд. Комбинирование происходит подстановкой значений регистров в более поздних инструкциях, которые ссылаются на данные регистры. Если итогом такого комбинирования является допустимая инструкция (инструкция подошла к какому-либо из шаблонов инструкций), то результат сохраняется, с удалением предшествующих инструкций (из которых подставлялись значения регистров) и обновлением информации о потоке данных [4].

Когда компилятору необходимо сделать вызов по регистру, то данный регистр получает пометку о том, что он эквивалентен ссылке на функцию (данная пометка может использоваться в других оптимизациях). Тогда, combine видит, что есть инструкция по помещению адреса функции в регистр, а затем вызов функции по этому регистру, и он совершенно справедливо пытается объединить их в одну инструкцию вызова функции, помещая значения регистра в аргумент функции вызова. Данная команда корректна, комбинирование завершается успешно, и в итоге снова получается вызов через PLT, которого хотели избежать.

Решение данной проблемы было заимствовано у архитектуры AArch64 [2]: во время проверки допустимости инструкции вызова функции были добавлены условия:

- компилируется PIC;

- не используется PLT для всех или для конкретно данного вызова;
- происходит вызов внешней функции.

Выполнение данных условий означает, что функция должна быть вызвана через регистр, а это значит, что скомбинированная инструкция не подойдёт к шаблонам вызова и combine не сможет их объединить в одну.

5.2 Генерация эффективного кода

Архитектура ARM имеет несколько систем команд. Одна из них это стандартная ARM, в данной системе используются только 32-битные команды. Система команд THUMB состоит из команд, взятых из ARM и преобразованных до 16-разрядных кодов. Данный режим сокращает объем используемой памяти, повышая плотность компилируемого кода. Этот набор команд имеет ограничение: работать напрямую можно только с 8-ю младшими регистрами общего назначения. Компромиссом между режимами ARM и THUMB является система команд THUMB-2, в которой сохранены 16-битные инструкции для повышения плотности кода, а также добавлены 32-битные инструкции, которые позволят приблизиться к производительности полного 32-битного набора команд ARM. Далее будут рассмотрены системы команд ARM и THUMB-2.

5.2.1 Система команд ARM

После исправления обратного комбинирования, получаем необходимый эффект (см. рис. 5).

Рассмотрим данный код немного детальнее:

- в регистр *r3* загружается смещение до глобальной таблицы смещений относительно метки *.LPIC0* (в данной метке это смещение будет сложено с регистром *pc*; таким образом будет получено смещение до глобальной таблицы смещений);
- в регистр *r2* загружается смещение функции *foo* в таблице смещений;
- в регистр *r3* помещается адрес функции *foo*.

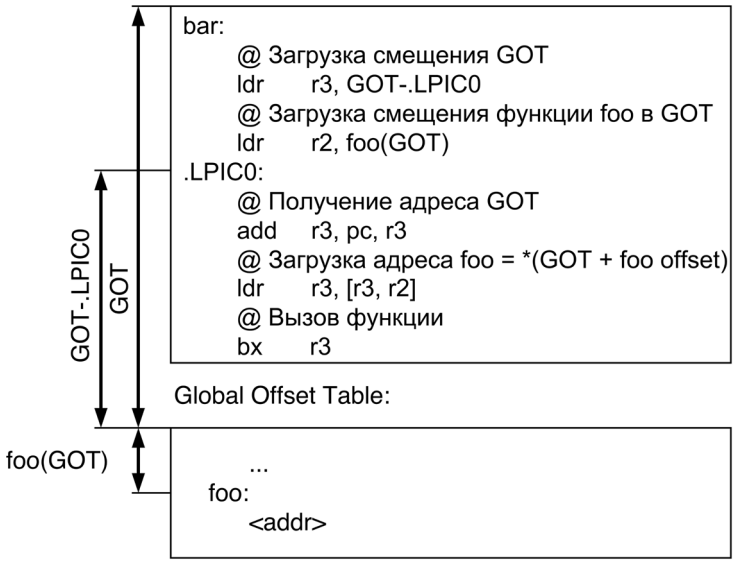

Рис. 5. Загрузка адреса функции и вызов

Fig. 5. Loading the address of the function prior to call

Таким образом мы получаем загрузку функций с помощью GOT без использования PLT. Но полученный код не выглядит оптимальным: на вызов функции необходимо выполнить три инструкции загрузки из памяти, которые являются дорогостоящей операцией.

Как было упомянуто ранее, компоновщик знает размеры секций и их смещения. Но данная информация никак не используется при генерации ассемблерного кода. Ввод релокации с адресом смещения до ячейки функции в глобальной таблице смещений невозможен, т.к. в архитектуре ARM нельзя записать 32-битную константу в регистр за одну инструкцию. Поэтому мы будем использовать две инструкции:

- *movw* (от англ. move wide) – помещает 16 битовую константу в регистр, обнуляя 16 старших битов регистра назначения;
- *movt* (от англ. move top) – помещает 16 битовую константу в 16 старших битов регистра назначения, при этом не изменяя младшие 16 битов.

```
.equ    label, 0x12345678
movw    r0, #:lower16:label
movt    r0, #:upper16:label
```

Рис. 6. Использование метки в инструкциях movw/movt

Fig. 6. Using a label in movw/movt instructions

Поводом для использования данных инструкций является возможность загрузки значения через метку (см. рис. 6). Данный код говорит о наличии функционала, который может быть использован при реализации релокаций. Причем данный функционал присутствует как в компиляторе, так и в компоновщике. Необходимо помнить, что использование инструкций *movw* / *movt* накладывает ограничение на версию архитектуры – они были введены сравнительно недавно и доступны на архитектурах ARMv6T2 и выше.

Заметим, что порядок данных команд имеет значение. Таким образом, потребуется две новые релокации, которые линкер должен будет разрешить: первая необходима для предоставления младших 16 бит смещения до ячейки функции в GOT, вторая – старших 16 бит.

На рис. 7 представлен псевдокод, в стиле ассемблера архитектуры ARM. Рассмотрим его подробнее:

- *#:lower16:got:foo* – младшие 16 битов смещения относительно текущего положения до ячейки функции *foo* в GOT;
- *#:upper16:got:foo* – старшие 16 битов смещения относительно текущего положения до ячейки функции *foo* в GOT;
- символ «.» означает смещение относительно начала программы до текущей исполняемой программы, аналогично метке *.LPIC0,* которая содержит смещение до инструкции, следующей за ней;
- разность (*.LPIC - .)* даёт смещение относительно текущего положения инструкции до инструкции, в которой данный адрес будет использоваться (в нашем случае *ldr r3, [pc, r3]*).

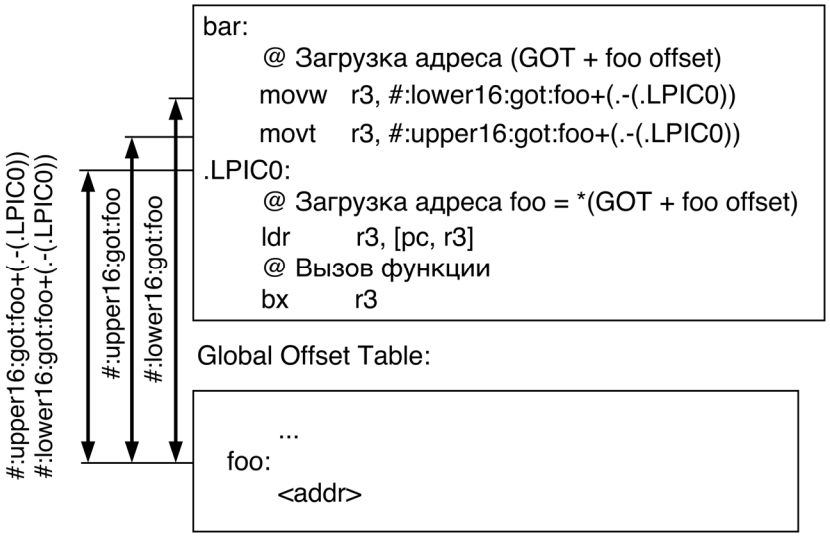

Рис. 7. Оптимизированная загрузка адреса функции

Fig. 7. Optimized function address load sequence

Таким образом, с помощью двух инструкций получилось избавиться от двух загрузок из памяти и одного сложения во время исполнения за счет расчета адресов ячеек в глобальной таблице смещений во время компоновки.

5.2.2 Система команд THUMB-2

Изначальная генерация кода схожа с системой команд ARM, и мало чем отличается от той, что представлена на рис. 5. Инструкции *movw* / *movt* также имеются в наборе команд. Но возникает проблема другого рода. Если при загрузке адреса ранее использовалась команда *ldr r3, [pc, r3]*, то в режиме THUMB-2 она невозможна. Согласно спецификации, при сложении с регистром *pc*, в качестве первого слагаемого, возможен только следующий случай: *ldr r0, [PC, imm12]*, где *imm12* – константное 12-битовое значение. В случае же если мы захотим поменять порядок сложения воспользовавшись коммутативностью, то результат выполнения команды вида *ldr r0, [r0, pc]* является неопределенным и непредсказуемым [5].

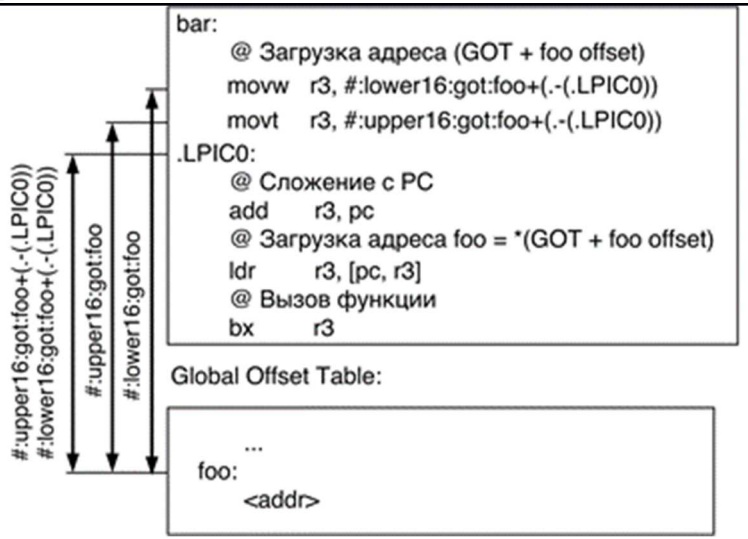

Рис. 8. Оптимизированная загрузка в режиме THUMB-2

Fig. 8. Optimized function address load sequence for THUMB2

Для решения данной проблемы необходимо использовать дополнительное сложение. На рис. 8 представлен псевдокод оптимизированной загрузки.

6. Результаты тестирования

Для тестирования было решено использовать встраиваемую реляционную базу данных SQLite [6], собранную в режиме динамической библиотеки. Во всех тестовых случаях были включены следующие флаги:

-O2 -march=armv7-a -mtune=cortex-a9 -mfpu=neon -mfloat-abi=softfp.

Для тестирования использовался стандартный набор тестов, поставляемый с исходными кодами SQLite и содержащий все основные операции с базой данных: INSERT, SELECT по числовым и строковым значениям, UPDATE числовых и строковых значений, INSERT таблиц друг в друга, сложные SELECT запросы.

В табл. 1 приведено среднегеометрическое время выполнения тестов, а также улучшения по отношению к базовому случаю – PLT. В табл. 2, аналогично табл. 1, приведены размеры динамической библиотеки при тестах.

Табл. 1. Результаты тестирования на SQLite. Время выполнения в секундах (среднегеометрическое по тестам).
В скобках указано улучшение по сравнению с базовым вариантом (с PLT).

Table 1. Evaluation results on SQLite. Times in seconds (geometric mean over multiple individual tests).
In parenthesis: improvement relative to base time (using PLT).

Набор команд	PLT	Без PLT	Без PLT с оптимизацией
ARM	1.28	1.26 (1.6%)	1.19 (7.0%)
THUMB-2	1.30	1.31 (-0.8%)	1.21 (6.9%)

Рассмотрим сначала ARM режим: время исполнения, как и предполагалось, уменьшается при отказе от PLT, причем, оно уменьшается, даже если каждый вызов функции будет требовать три загрузки из памяти вместо одного прыжка по таблице компоновки процедур. Когда включается оптимизация вызова через инструкции *movw* / *movt* происходит ускорение на 7.0%.

Табл. 2. Результаты тестирования на SQLite. Размер динамической библиотеки в килобайтах.
В скобках указано улучшение по сравнению с базовым вариантом.

Table 2. Evaluation results on SQLite. Dynamic shared library size, in kilobytes.
In parenthesis: improvement relative to base size.

Набор команд	PLT	Без PLT	Без PLT с оптимизацией
ARM	548	629 (-14.8%)	621 (-13.3%)
THUMB-2	400	445 (-11.3%)	458 (-14.5%)

С размером кода не так всё однозначно, после отказа от использования PLT размер динамической библиотеки растёт, это связано с тем, что теперь вместо одной инструкции прыжка на PLT таблицу генерируется 5 инструкций, размер кода увеличивается на 14.8%, после применения оптимизации кода на один вызов генерируется 4 инструкции, соответственно, размер кода, относительно PLT, увеличивается на 13.3%.

При использовании команд THUMB-2, отключение PLT приводит к незначительному замедлению, но после оптимизации кода даёт ускорение на 6.9%. Размер кода также увеличивается во всех случаях, но в отличии от ARM, не удалось сократить количество инструкций, получилось лишь их поменять на менее дорогостоящие по времени исполнения.

7. Заключение

В рамках данной работы проведён анализ работы позиционно-независимого кода и динамического загрузчика в операционной системе Linux с использованием компилятора GCC. Была рассмотрена оптимизация по отказу от использования таблицы компоновки процедур для вызовов функций. Также были обнаружены трудности в реализации данной оптимизации на архитектуре ARM.

Проблемы были преодолены с помощью улучшения оптимизации комбинирования инструкций, а также добавления новых релокаций в GCC и GNU Binutils для генерации оптимизированного ассемблерного кода.

Изменения, внесенные в компилятор и динамический загрузчик, повлекли за собой ускорение выполнения тестов на встраиваемой реляционной базе данных SQLite, собранной с динамической библиотекой. Получилось ускорить время выполнения на 7.0% за счет отказа от ленивого связывания во время выполнения исполняемого файла.

Данная оптимизация полезна в программах, которые используют динамические библиотеки и для которых производительность является критически важным параметром.

Список литературы

[1]. J. Levine. Linkers and Loaders. Morgan-Kauffman, p. 256, October 1999.
[2]. J. Greenhalgh. [AArch64] Tighten direct call pattern to repair -fno-plt. https://gcc.gnu.org/ml/gcc-patches/2015-08/msg00152.html.
[3]. A. Monakov. PIC calls without PLT, generic implementation. https://gcc.gnu.org/ml/gcc-patches/2015-05/msg00225.html.
[4]. D. Melnik. Developing Interblock Combine Pass in GCC. GNU Tools Cauldron 2013. https://gcc.gnu.org/wiki/cauldron2013.
[5]. ARM Architecture Reference Manual, ARMv7-A and ARMv7-R edition, Section A2.8.6.60, 04 June 2009.
[6]. Веб-сайт SQLite. http://www.sqlite.org/about.html.

Dynamic loader optimization for ARM

E.A. Kudryashov <eugene.a.kudryashov@gmail.com>
D.M. Melnik <dm@ispras.ru>
A.V. Monakov <amonakov@ispras.ru>
Institute for System Programming of the Russian Academy of Sciences,
25, Alexander Solzhenitsyn st., Moscow, 109004, Russia

Abstract. The paper discusses an optimization approach for external calls in position-independent code that is based on loading the callee address immediately at the call site from the Global Offset Table (GOT), avoiding the use of the Procedure Linkage Table (PLT). Normally the Linux toolchain creates the PLT both in the main executable (which comprises position-dependent code and has to rely on the PLT mechanism to make external calls) and in

shared libraries, where the PLT serves to implement lazy binding of dynamic symbols, but is not required otherwise. However, calls via the PLT have some overhead due to an extra jump instruction and poorer instruction cache locality. On some architectures, binary interface of PLT calls constrains compiler optimization at the call site. It is possible to avoid the overhead of PLT calls by loading the callee address from the GOT at the call site and performing an indirect call, although it prevents lazy symbol resolution and may cause increase in code size. We implement this code generation variant in GCC compiler for x86 and ARM architectures. On ARM, loading the callee address from the GOT at call site normally needs a complex sequence with three load instructions. To improve that, we propose new relocation types that allow to build a PC-relative address of a given GOT slot with a pair of movt, movw instructions, and implement these relocation types in GCC and Binutils (assembler and linker) for both ARM and Thumb-2 modes. Our evaluation results show that proposed optimization yields performance improvements on both x86 (up to 12% improvement with Clang/LLVM built with multiple shared libraries, on big translation units) and ARM (up to 7% improvement with SQLite, average over several tests), even though code size on ARM also grows by 13-15%.

Keywords: program optimizations; dynamic loader; global offset table; procedure linkage table; relocations; ARM

DOI: 10.15514/ISPRAS-2016-28(1)-4

For citation: Kudryashov E.A., Melnik D.M., Monakov A.V. Dynamic loader optimization for ARM. *Trudy ISP RAN/Proc. ISP RAS*, vol. 28, issue 1, 2016, pp. 63-80 (in Russian). DOI: 10.15514/ISPRAS-2016-28(1)-4

References

[1]. J. Levine. Linkers and Loaders. Morgan-Kauffman, p. 256, October 1999.
[2]. J. Greenhalgh. [AArch64] Tighten direct call pattern to repair -fno-plt. https://gcc.gnu.org/ml/gcc-patches/2015-08/msg00152.html.
[3]. A. Monakov. PIC calls without PLT, generic implementation. https://gcc.gnu.org/ml/gcc-patches/2015-05/msg00225.html.
[4]. D. Melnik. Developing Interblock Combine Pass in GCC. GNU Tools Cauldron 2013. https://gcc.gnu.org/wiki/cauldron2013.
[5]. ARM Architecture Reference Manual, ARMv7-A and ARMv7-R edition, Section A2.8.6.60, 04 June 2009.
[6]. Веб-сайт SQLite. http://www.sqlite.org/about.html.