Министерство образования и науки Российской Федерации
Московский физико-технический институт (национальный исследовательский университет)

Факультет радиотехники и компьютерных технологий
Кафедра Микропроцессорные технологии в интеллектуальных системах управления

Выпускная квалификационная работа бакалавра по направлению 010900
«Прикладные математика и физика»

Исследование влияния бинарных оптимизаций на модификацию таблиц переходов в архитектуре ARM

Студент 909 группы
Колобаев Дмитрий Анатольевич

Научный руководитель
Петушков Игорь Вячеславович

Долгопрудный
2023

Аннотация

В работе рассмотрено влияние оптимизаций, связанных с перестановкой горячих блоков кода, на консистентность таблиц переходов. Исследовалась оптимизация приложений для архитектуры AArch64 с помощью открытого инструмента LLVM BOLT (Binary Optimization Layout Tool). Исследование стабильности BOLT обнаружило проблемы при оптимизации приложений, содержащих в себе таблицы переходов – конструкции осуществляющие косвенный переход по адресу, зависящему от контекста исполнения программы.

В работе предложены алгоритмы поиска, анализа и модификации таблиц переходов. Предложенные алгоритмы были добавлены в инфраструктуру LLVM BOLT. Исследование стабильности модифицированного оптимизатора показало, что описанные в работе методы обработки таблиц переходов исправляют все ошибки, обнаруженные в открытой версии оптимизатора.

Содержание

Глава 1

Введение

В современной технике оптимизация работы программ для электронных вычислительных машин имеет очень большое значение. Для этих целей используется множество средств, применяемых на разных уровнях программного и аппаратного обеспечения. Одним из способов улучшения производительности являются бинарные оптимизации - преобразования исполняемого кода приложения на основе профиля его исполнения[17]. Профиль приложения дает дополнительную информацию о логике работы программы недоступную компилятору на этапе оптимизации исходного кода. Учет этой информации при оптимизации приложения позволяет добиться прироста производительности. Описанный подход используется в Binary Layout Optimization Tool (LLVM BOLT) [12]. Этот инструмент использует профиль исполнения приложения для нахождения оптимального расположения часто исполняемых (горячих) блоков кода. Локализация горячих блоков кода позволяет эффективнее использовать ресурсы вычислительной машины, на которой исполняется приложение, за счет уменьшения количества промахов в L1I-cache и iTLB. Авторы оригинальной статьи добились увеличения производительности реальных серверных приложений на 7% [12].

Любые оптимизирующие трансформации должны сохранять логику работы программы. В случае LLVM BOLT сложность представляют машинные инструкции, оперирующие с адресами и смещениями в исполняемом файле. После изменения компоновки кода такие инструкции нуждаются в модификации.

В рамках представленной работы было проведено исследование корректности оптимизаций LLVM BOLT приложений для архитектуры AArch64. После применения бинарных оптимизаций в 4% программ из тестового набора были обнаружены ошибки. Было установлено, что причиной ошибок являются таблицы переходов - конструкции в бинарном коде, реализующие ветвление с произвольным количеством вариантов.

В работе описан алгоритм поиска и модификации таблиц переходов. Данный алгоритм реализован как часть бинарного оптимизатора LLVM BOLT. Для проверки результатов проведено повторное исследование корректности оптимизаций, которое

показало, что предложенный подход исправляет все задетектированные случаи ошибок.

Глава 2

Постановка задачи

Стабильность оптимизатора LLVM BOLT была исследована на наборе из 602 тестов в который вошли тесты из открытого набора тестов LLVM Test Suite и ряд бенчмарков, например, SPEC2006 и SPEC2017 [1]. Тестовый набор содержит большое количество разнообразных приложений, в том числе достаточно сложных, что обеспечивает большое покрытие кода LLVM BOLT и репрезентативность результатов. Целевая платформа имеет архитектуру AArch64 и поддерживает исполнение только не зависящих от адреса загрузки исполняемых файлов (PIE - position independent executable). Поэтому все тестовые приложения были скомпилированы в соответствующем режиме. Исследование выявило ошибки после оптимизации в 24 тестах, что составило 4% от общего числа тестовых приложений.

Дальнейшее исследование показало, что ошибки заключаются в нарушении потока управления программы из-за невалидных записей в таблицах переходов оптимизированных приложений.

Цель работы: разработать алгоритм анализа и корректной модификации таблиц переходов и имплементировать его в инфраструктуре LLVM BOLT.

Задачи:

- определить причину ошибок при оптимизации приложений с таблицами переходов с помощью LLVM BOLT.
- Разработать решение для автоматизированного поиска и анализа таблиц переходов.
- На основе предложенного решения разработать алгоритм обработки таблиц переходов в LLVM BOLT.

Глава 3

Обзор существующих решений

3.1 Бинарная оптимизация

Технология бинарных оптимизаций применяется для увеличения производительности приложений, исходный код которых по тем или иным причинам недоступен [17]. Для оптимизации используется профиль исполнения приложения. Ключевым подходом в инструментах бинарной оптимизации с профилем является оптимизация основанная на обратной связи (FDO, Feedback Driven Optimization). В данном подходе компилятор использует информацию полученную в результате предыдущих исполнений программы, для выполнения более агрессивных преобразований кода [2].

Современные компиляторы такие как GCC или Clang поддерживают FDO [8]. Однако, они не способны работать с приложениями без исходного кода.

Еще одно отличие компиляторов от бинарных оптимизаторов состоит в точности использования профилировочной информации, полученной методом сэмплирования [3]. Компиляторы пытаются соотнести информацию полученную из профиля приложения с промежуточным представлением программы. С одной стороны это позволяет большему количеству оптимизаций использовать информацию из профиля. С другой стороны, чем больше промежуточное представление отличается от того представления, для которого был получен профиль, тем менее точно применяется профильная информация . В случае бинарного оптимизатора профиль используется с наибольшей точностью, но отсутствует информация о высокоуровневой логике программы.

За последние несколько лет наблюдается резкий рост популярности бинарных оптимизаторов: BOLT [12], Propeller [10], Janus [16], HALO [14], Ispike [9]. Эти инструменты используют профильную информацию исполнения для принятия решений по оптимизации.

3.2 LLVM BOLT

LLVM BOLT (Binary Optimization and Layout Tool) [12] - бинарный оптимизатор с открытым исходным кодом, являющийся частью проекта LLVM [7]. Оптимизатор способен обрабатывать бинарные файлы двух архитектур x86 и AArch64. Целевыми приложениями для LLVM BOLT являются программы с большим размером исполняемого кода. В таких приложениях за счет локализации часто исполняемых участков удается добиться существенного прироста производительности [11]. Создатели данного бинарного оптимизатора достигли прироста производительности в 7% на серверных приложениях, скомпилированных с оптимизациями времени связывания и использованием профиля [12].

Основной оптимизацией BOLT является перекомпоновка кода на основе информации об исполнении приложения. Оптимизации применяются на уровне бинарных базовых блоков - линейных участков кода, заканчивающихся инструкцией перехода. С помощью статического анализа бинарного файла и по информации о переходах, встретившихся в процессе исполнения программы, оптимизатор строит граф потока управления и оценивает *температуру* базовых блоков. Чем больше раз исполнялся блок кода, тем он считается горячее. Собранная информация используется для того чтобы перекомпоновать код программы, располагая горячие участки кода рядом друг с другом. Алгоритм схематично изображен на рисунке 1. Подобная перестановка позволяет уменьшить количество промахов в L1I кэш процессора и iTLB, и, как следствие, повышает производительность [13].

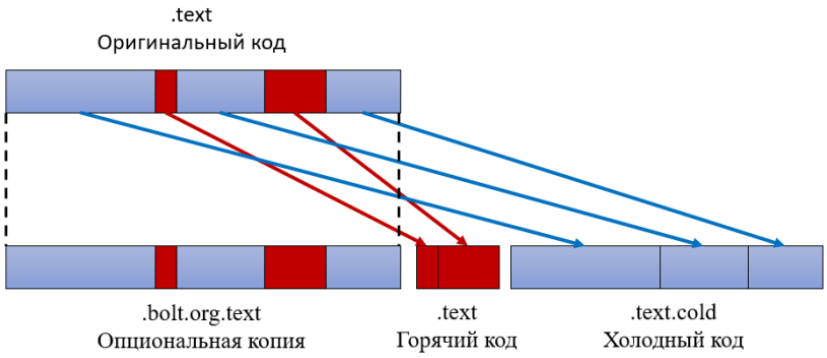

Рис. 1. Схема работы бинарного оптимизатора BOLT [17].

3.3 Релокационная информация

Релокации - механизм, применяемый линкерами для связывания символов в программе с их конечными адресами в бинарном файле [6]. Описание формата релокаций и их смысл являются частью спецификации формата исполняемого файла. Для ELF файлов, используемых в операционной системе Andrioid, статическая релокационная информация помещается в отдельные секции, имена которых помечены префиксами

.rel или .rela. Эти секции содержат записи следующего формата [4]:

```
typedef struct {
    Elf64_Addr      r_offset;
    Elf64_Xword     r_info;
} Elf64_Rel;

typedef struct {
    Elf64_Addr      r_offset;
    Elf64_Xword     r_info;
    Elf64_Sxword    r_addend;
} Elf64_Rela;
```

- Поле r_offset обычно кодирует адрес в памяти, к которому надо применить релокацию.
- Поле r_info кодирует тип релокации и символ с которым релокация связана.
- Поле r_addend есть только у релокаций типа rela и кодирует отступ от символа.

Статическая процедура связывания подразумевает вычисление адреса по релокационной информации и запись полученного значения по адресу, закодированному в поле r_offset. Правила вычисления адреса отличаются в зависимости от типа релокации. Например, для релокаций типа `R_AARCH64_PREL32`, означающих 32-битное смещение относительно PC (Program Counter), правило вычисления следующее:

$$V = S + A - P \mod 2^{32} \tag{1}$$

Здесь V - значение релокации, S - адрес символа, A - значение поля r_addend, P - значение поля r_offset.

3.4 Таблицы переходов в архитектуре AArch64

Замечание: В данной работе рассматриваются приложения скомпилированные для архитектуры AArch64 в режиме PIE. Поэтому все рассматриваемые конструкции в бинарном коде не должны зависеть от абсолютных адресов [5].

Таблица переходов в исполняемом PIE файле состоит из двух частей – шаблона доступа и массива смещений. Ее структура приведена на рисунке 2.

Шаблон доступа - это последовательность инструкций в программе, которая отвечает за формирование адреса и осуществление косвенного перехода по нему. Для формирования адреса используется массив смещений – участок памяти в секции данных, содержащий массив констант одинакового размера. Размер констант может

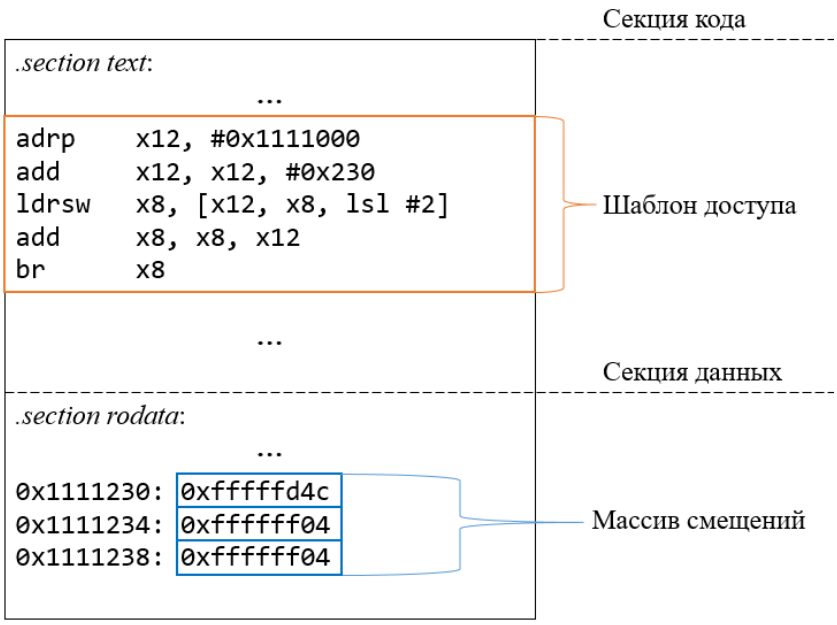

Рис. 2. Структура таблицы переходов в архитектуре ARM.

варьироваться от 1 до 4 байт. Из логики работы инструкций шаблона доступа следует формула для подсчета адреса косвенного перехода:

$$TargetAddress = Base + sign\_extend_{64}(OffsetArray[Index]) \ll Shift \qquad (2)$$

В формуле (2) использованы следующие обозначения:

- $TargetAddress$ - адрес перехода.
- $Base$ - база отсчета адреса.
- $OffsetArray[Index]$ - элемент массива смещений с индексом $Index$.
- $sign\_extend_{64}$ - операция знакового расширения числа до 64 бит.
- $Shift$ - значение опционального битового сдвига влево. Может быть равно 0.

Такой алгоритм подсчета адреса позволяет выполнять в данной точке программы переход на разные адреса в зависимости от значения $Index$. При этом для определения адреса массива смещений используется зависящая от PC инструкция `ADRP`. Сам массив содержит смещение адреса перехода относительно базы отсчета. Таким образом эта формула инвариантна относительно изменения адреса загрузки приложения в память, а значит может использоваться в PIE файлах.

Примером высокоуровневой конструкции, которая компилируется в таблицу переходов, является `switch-case` оператор из языков C и C++.

Компилятор для целевой платформы генерирует всего два типа таблиц переходов. Они приведены в таблице рис. 1. Типы отличаются способом формирования базы отсчета $Base$ и наличием релокаций для элементов массива смещений.

`adrp x1, -7585792`
`add x1, x1, 692`
`ldrh w0, [x1, w0, uxtw 1]`
`adr x1, 12`
`add x0, x1, w0, sxth 2`
`br x0`
	`adrp x12, 21544960`
	`add x12, x12, 1696`
	`ldrsw x8, [x12, x8, lsl 2]`
	`add x8, x8, x12`
	`br x8`

Шаблон доступа первого типа. В качестве *Base* используется конец базового блока. Релокаций нет.	Шаблон доступа второго типа. В качестве *Base* используется начало массива смещений. Есть релокации.

Таблица 1. Примеры шаблонов доступа.

3.5 Таблица перехода с адресацией относительно кода

В случае шаблона доступа первого типа адресация происходит относительно кода. Инструкция `adr` загружает в регистр адрес следующий за инструкцией косвенного перехода. Затем этот адрес используется в качестве базы адресации таблицы переходов. Таким образом формула вычисления адреса косвенного перехода становится:

$$TargetAddress = (BranchAddress+4)+sign\_extend_{64}(OffsetArray[Index]) \ll Shift$$

где $BranchAddress$ - адрес инструкции косвенного перехода.

В таблицах такого типа база отсчета и точки программы, на которые может передать управление таблица переходов, находятся в пределах одной исполняемой секции. Поэтому относительное смещение между ними не может измениться в результате процедуры связывания. По этой причине для элементов массива релокации не нужны.

3.6 Таблица перехода с адресацией относительно данных

В случае шаблона доступа второго типа адресация происходит относительно данных. За базу отсчета принимается начало массива смещений. Формула для расчета адреса перехода принимает следующий вид:

$$TargetAddress = OffsetArray + sign\_extend_{64}(OffsetArray[Index])$$

где $OffsetArray$ - адрес массива смещений.

В таблицах такого типа база отсчета и точки программы, на которые может передать управление шаблон доступа, находятся в разных секциях. Из-за этого компилятор не может вычислить значения элементов в массиве смещений до процедуры связывания. Поэтому для записей в массиве смещений требуется релокационная ин-

формация. Она кодируется с помощью релокаций типа `R_AARCH64_PREL32` рис. 3.

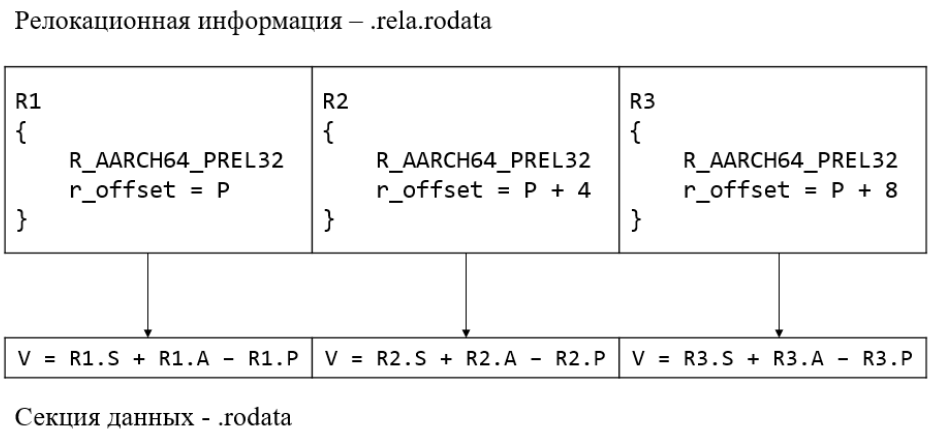

Рис. 3. Релокационная информация для массива смещений.

3.7 Поддержка таблиц переходов в LLVM BOLT для AArch64

Существующий алгоритм обработки таблиц переходов LLVM BOLT детектирует только таблицы переходов с адресацией относительно кода. Оптимизатор запрещает изменение компоновки базовых блоков функций, в которых обнаружена такая таблица. Данное ограничение не позволяет получить прирост за счет локализации горячих блоков функции, но исключает возможность ошибок в таблице переходов, за счет PIE свойств кода.

Второй тип таблиц переходов LLVM BOLT не детектирует. Вместо этого он полагается на анализ релокационной информации. Все релокации типа `R_AARCH64_PREL32` обрабатываются единообразно, в том числе те что не относятся к таблицам переходов. BOLT пересчитывает значение релокаций на основе расположения объектов в памяти оптимизированного приложения:

1. Подсчитывается адрес на который указывает релокация в оригинальном адресном пространстве.
2. Этот адрес отображается в адресное пространство оптимизированного приложения.
3. Вычисляется значение релокации в новом адресном пространстве.
4. Значение релокации записывается по адресу, закодированному в поле `r_offset` релокации.

Как будет показано далее, в случае таблиц переходов такой подход не работает - записи в массиве смещений оказываются невалидными.

Глава 4

Исследование и построение решения задачи

4.1 Влияние оптимизаций на корректность таблиц переходов

Целевыми блоками таблицы переходов будем называть все базовые блоки, на которые может передать управление данная таблица переходов.

В результате применения LLVM BOLT целевые блоки таблицы переходов могут изменить свой адрес. В таком случае оптимизатор должен исправить массив смещений. Пусть адрес целевого блока таблицы переходов $Target$ и база отсчета $Base$ после оптимизации стали равны $Target'$ и $Base'$ соответственно. Тогда согласно формуле (2)

$$Target' = Base' + sign\_extend_{64}(OffsetArray[Index]) \ll Shift \tag{3}$$

Для того чтобы данное равенство выполнялось в новом адресном пространстве, оптимизатору нужно изменить элемент массива смещений с индексом $Index$. Новое значение элемента массива можно выразить из равенства (3):

$$OffsetArray[Index]' = (Target' - Base') \gg Shift \mod 2^{8 \cdot sizeof(OffsetArray[Index])} \tag{4}$$

В последнем равенстве нужно учесть, что адреса $Target'$ и $Base'$ занимают в 64-битной архитектуре 8 байт, а элемент массива может занимать 1, 2 или 4 байта. Поэтому результат вычислений в правой части должен быть обрезан до размера элемента массива.

Формула (4) показывает, что значение элемента массива смещений зависит только от разности $Target'$ и $Base'$, но не от их абсолютных значений. Таким образом такая таблица переходов может быть использована в PIE исполняемых файлах.

4.1.1 Влияние оптимизаций на таблицы перехода с адресацией относительно кода

В случае детекции таблиц переходов с адресацией относительно кода LLVM BOLT запрещает изменение относительного расположения базовых блоков в данной функции - функция может перемещаться только целиком. Таким образом выполняются равенства:

$$Target - Base = Target' - Base'$$
$$\Rightarrow OffsetArray[Index] = OffsetArray[Index]'$$

элементы массива оказываются валидными в новом адресном пространстве без дополнительной модификации со стороны оптимизатора. Данная стратегия не работает, если целевой блок таблицы переходов окажется в другой функции, но такие случаи LLVM BOLT не обрабатываются.

4.1.2 Влияние оптимизаций на таблицы перехода с адресацией относительно данных

Рассмотрим таблицы перехода с адресацией относительно данных. Так как LLVM BOLT их не детектирует, то смещение между адресами целевых блоков и базой отсчета в новом адресном пространстве может измениться.

$$Target - Base \neq Target' - Base'$$
$$\Rightarrow OffsetArray[Index] \neq OffsetArray[Index]'$$

Требуется изменить записи в массиве в соответствии с формулой (4). Для модификации массива оптимизатор полагается на механизм релокаций.

Листинг 1 Псевдокод шаблона доступа с адресацией относительно данных.

```
# x8 = Index
adrp    x12, OffsetArray[31:12]
add     x12, x12, OffsetArray[11:0]
ldrsw   x8, [x12, x8, lsl 2]
add     x8, x8, x12
br      x8
```

На каждый элемент массива смещений в исполняемом файле есть релокация типа `R_AARCH64_PREL32`. Поле `r_offset` в этих релокациях равно адресам элементов массива. Тогда согласно шаблону доступа (лист. 1) значение релокации V в ориги-

нальном адресном пространстве должно удовлетворять выражению:

$$Target = OffsetArray + sign\_extend_{64}(V) \tag{5}$$

$$V = Target - OffsetArray \mod 2^{32} \tag{6}$$

С другой стороны по правилу вычисления значения релокации (1)

$$V = S + A - P \mod 2^{32} \tag{7}$$

В данном случае P равно значению поля `r_offset`, то есть адресу элемента массива. Адрес элемента массива получается прибавлением $Index$ к базе отсчета $OffsetArray$. Тогда выражение (7) принимает вид.

$$V = S + A - OffsetArray - 4 \cdot Index \mod 2^{32}$$

Подставим сюда выражение (6) и выразим закодированный в релокации адрес $S + A$.

$$S + A = Target + 4 \cdot Index \mod 2^{32} \tag{8}$$

Таким образом, из логики работы инструкций в шаблоне доступа и правила вычисления значений релокаций следует, что в исходном исполняемом файле релокации элементов массива смещений не указывают на адрес целевого базового блока. Вместо этого в них закодирован адрес целевого блока смещенный на $4 \cdot Index$. Это порождает неявную зависимость значений всех элементов массива смещений от адреса начала массива.

Для формирования релокационной информации оптимизированного приложения LLVM BOLT отображает адрес закодированный в оригинальных релокациях в новое адресное пространство. Оригинальный адрес вычисляется по правилу общему для всех релокаций `R_AARCH64_PREL32`.

$$S + A = V + P$$

Согласно (8) этот адрес не соответствует адресу целевого базового блока, поэтому адрес, который получает LLVM BOLT в адресном пространстве оптимизированного приложения, в общем случае является невалидным и при формировании новых релокаций в оптимизированном файле порождает невалидные записи в массиве.

Пример ситуации, которая приводит к возникновению ошибок представлен на рис. 4. Значение смещения от адреса базового блока $4 \cdot Index$ может оказаться больше чем размер самого базового блока и попасть в следующий базовый блок. В таком случае если в оптимизированном приложении разность адресов между этими двумя блоками изменится на Δ, то итоговое значение релокации изменится на Δ. Это приведет к тому, что адрес перехода, получаемый из связанного с релокацией элемента

массива, также изменится на Δ и станет не равен адресу целевого блока в оптимизированном адресном пространстве. Таким образом при оптимизации нарушается целостность потока управления.

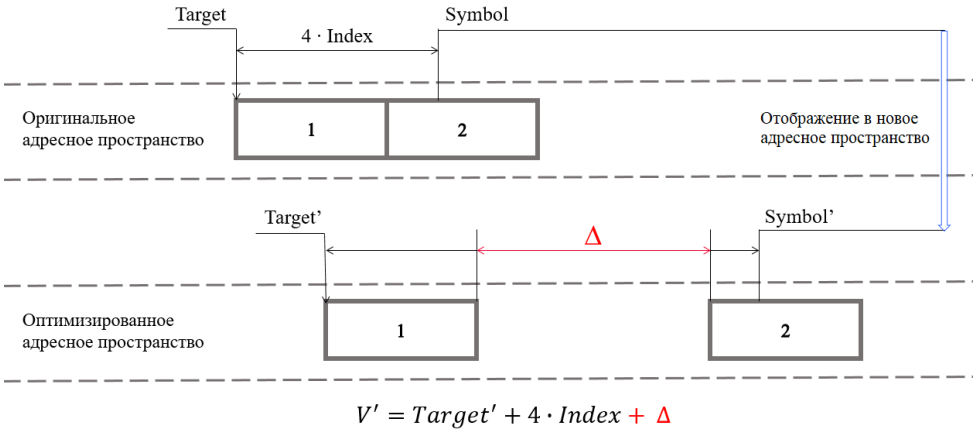

Рис. 4. Пример ошибки при пересчете значения релокации.

Для корректной обработки таблиц переходов при пересчете релокаций следует опираться не на значение релокации, а непосредственно на адрес целевого блока. Получить адрес целевого блока $Target$ можно зная значение релокации и адрес начала массива смещений по формуле (5). Полученный адрес следует отобразить в адресное пространство оптимизированного приложения. Затем сформировать новую релокацию:

$$P' = P$$
$$S' = Target'$$
$$A' = P - OffsetArray$$

Значение такой релокации удовлетворяет выражению (6) в адресном пространстве оптимизированного приложения.

$$V' = S' + A' - P' = Target' - OffsetArray \mod 2^{32}$$

Подобным образом нужно пересчитать только те релокации, которые относятся к массиву смещений данной таблицы переходов. То есть для произвольно взятой релокации секции данных нужно иметь возможность определить, принадлежит ли она массиву смещений. Так как элементы массива должны идти подряд, начиная с адреса начала массива, задача определения принадлежности каждого следующего элемента массиву сводится к определению его размера.

Таким образом для правильной обработки таблицы переходов с адресацией относительно данных требуется вычислить адрес массива смещений этой таблицы, или, что тоже самое, определить базу отсчета, и определить размер массива смещений. В существующем алгоритме LLVM BOLT ошибки связаны с тем, что каждый элемент

массива смещений трактуется как начало массива.

4.2 Статический анализ таблиц переходов с адресацией относительно данных

Задача анализа таблиц переходов в бинарном файле оригинального приложения состоит в определении границ массива смещений. Обладая этой информацией, можно статически определить адреса всех целевых блоков таблицы. Зная адреса целевых блоков, можно корректно сформировать массив смещений для оптимизированного файла.

4.2.1 Определение начала массива смещения

Начало массива смещений определяется по шаблону доступа таблицы переходов. Инструкции `adrp` и `add` в шаблоне отвечают за формирование адреса массива смещений. `adrp` загружает в регистр старшие биты с 12 по 31, инструкция `add` добавляет к адресу младшие 12 бит (с 0 по 11). Таким образом, по аргументам этой пары инструкций можно установить адрес начала массива смещений. Задача сводится к нахождению в массиве инструкций исходного приложения шаблона доступа. В общем случае, инструкции шаблона доступа могут идти не подряд. Поэтому поиск шаблона осуществляется с помощью анализа графа потока данных по следующему алгоритму (обозначения, используемые в алгоритме, отображены в лист. 2):

1. В массиве инструкций исходного приложения найти инструкцию косвенного перехода `br reg1`.
2. Найти инструкцию сложения формата `add reg1, reg2, reg3`, определяющую значение регистра `reg1`.
3. Найти инструкцию загрузки значения из памяти, использующую один из двух регистров `reg2` или `reg3` в качестве базы адресации. Пусть это будет `reg3`.
4. Найти инструкцию сложения `add reg3, reg4, imm1` которая определяет значение базы адресации инструкции загрузки из памяти.
5. Найти инструкцию `adrp`, записывающую значение `imm2` в регистр reg4.
6. Если все инструкции были найдены, то они образуют шаблон доступа таблицы переходов. Адрес начала массива смещений равен сумме `imm1` и `imm2`.

Пусть инструкция i использует множество регистров R как аргументы. Обозначим $Def(i)$ множество инструкций, которые при каком-либо сценарии исполнения программы могут определять значения регистров R в данной точке. Алгоритм 1 определяет адрес массива смещений по шаблону доступа таблицы переходов.

Алгоритм 1 Алгоритм поиска шаблона доступа таблицы переходов. Адреса массивов смещений всех таблиц перехода в программе записываются в JT.

```
I - массив инструкций программы
JT - множество адресов таблиц переходов.
for each i ∈ I do
    if i инструкция косвенного перехода then
        JT.insert(GetOffsetArrayStart(i))
    end if
end for
function GetOffsetArrayStart(i_br)
    for each d ∈ Def(i_br) do
        if d инструкция сложения двух регистров then
            i_load ← LoadLookup(d)
            b_add ← AddLookup(i_load)
            b_adrp ← AdrpLookup(b_add)
            page ← константный аргумент b_adrp
            offset ← константный аргумент b_add
            return page + offset
        end if
    end for
end function
function LoadLookup(i_add)
    for each d ∈ Def(i_add) do
        if d инструкция загрузки из памяти по регистру then
            return d
        end if
    end for
end function
function AddLookup(i_load)
    for each d ∈ Def(i_load) do
        if d инструкция сложения регистра с числом then
            return d
        end if
    end for
end function
function AdrpLookup(b_add)
    for each d ∈ Def(b_add) do
        if d инструкция verb|adrp| then
            return d
        end if
    end for
end function
```

4.3 Уравнения потока данных

Для поиска шаблона доступа используется граф потока данных. Построение такого графа подразумевает решение definition-use задачи для машинных инструкций. В данной работе для этого был использован метод уравнений потока данных для задачи доступных определений (Reaching definitions).

Листинг 2 Пояснения к алгоритму поиска шаблона доступа.

```
# reg - register
# imm - constant
adrp    reg4, imm2
add     reg3, reg4, imm1
ldrsw   reg2, [reg3, reg, lsl 2]
add     reg1, reg3, reg2
br      reg1
```

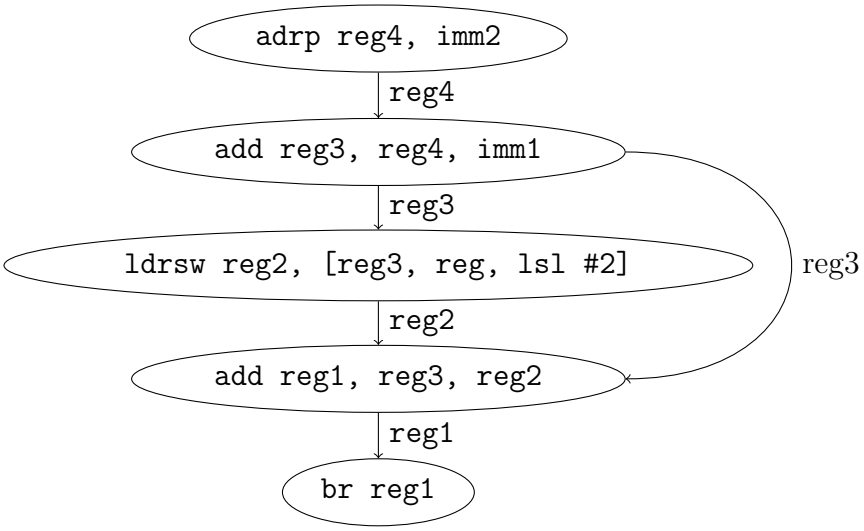

Рис. 5. Граф потока данных для примера лист. 2.

При обработке бинарного файла LLVM BOLT конструирует граф потока управления. Вершинами данного графа являются базовые блоки. Каждый блок содержит последовательность линейно исполняемых инструкций, заканчивающихся инструкцией перехода. Этот граф используется для построения и решения уравнений потока данных.

Определение регистра - выражение, меняющее значение регистра, например, инструкция загрузки значения из памяти в данный регистр. Определение регистра r доступно в точке программы P, если в графе потока управления существует путь от определения r до точки P, на котором значение r не изменяется. Если на пути встречается другое определение r, то говорят что оно "убивает"первое определение. Задача доступных определений состоит в нахождении всех определений регистра r доступных в данной точке программы P.

Требуется вычислить доступные определения в точках до и после каждой инструкции в программе. Для этого в каждой точке формулируются соотношения, которые оперируют четырьмя логическими величинами:

in Определение достигает точки перед исполнением инструкции.

out Определение достигает точки после исполнения инструкции.

gen Инструкция создает новое определение.

kill Инструкция убивает определение.

В каждом случае *gen* и *kill* это величины которые мы можем легко определить для каждой инструкции, основываясь на логике ее работы, а *in* и *out* это величины, которые мы хотим в конечном счете установить. Все эти величины являются логическими функциями от инструкции i. Для величин *gen* и *kill* должно выполняться условие:

$$gen(i) \wedge kill(i) = \text{FALSE}$$

Теперь можно определить несколько типов уравнений с введенными величинами на графе потока управления программы.

Граничные условия Определяют значения логических величин на входе или выходе из программы.

Уравнения распространения Определяют как значения распространяются из одного базового блока в другой.

Уравнения переноса Определяет как значения изменяются одной инструкцией.

Есть два основных типа задач на поток данных: те в которых значения распространяются в направлении исполнения программы (прямые задачи), и те в которых значения распространяются в направлении противоположном исполнению программы (обратные задачи). Величины *in*, *out* и прочие представляются как битовые вектора, в которых каждое измерение соответствует какому-то определению в программе.

Задача доступных определений относится к прямым задачам. Обозначим s точку входа в программу (первая инструкция в графе потока управления), $\text{pred}(i)$ - все инструкции, имеющие ребро в инструкцию i в графе потока управления (поток управления - направленный граф). Тогда уравнения для задачи доступных определений:

$$in(s) = \emptyset$$
$$in(i) = \bigcap_{x \in \text{pred}(i)} out(x) \text{ for } i \neq s$$
$$out(i) = (in(i) - kill(i)) \cup gen(i)$$

Чтобы построить граф потока данных, значения битовых векторов вычисляются по приведенным выше правилам для каждой инструкции в графе потока управления. Из-за наличия циклов в графе, недостаточно обойти весь граф один раз. Поэтому алгоритм применяется рекурсивно. Изначально в очередь обработки попадают все инструкции из графа. Далее они извлекаются по одной. Для извлеченной инструкции вычисляется значение *out* множества. Если значение этого множества изменилось, то в очередь помещаются все инструкции, которые идут в графе потока управления после данной. Таким образом алгоритм продолжается до тех пор, пока в графе потока данных происходят изменения.

В результате применения данного алгоритма для каждой инструкции можно сказать, какая инструкция определяет регистры, используемые данной инструкцией. Эта информация необходима для описанного выше алгоритма поиска шаблона доступа.

4.4 Вычисление размера таблицы переходов

Целевые адреса связаны со значением релокаций следующим равенством:

$$Target = Base + V$$

Это выражение выполняется только для релокаций, которые относятся к массиву смещений с адресом начала *Base*. По этой причине, чтобы правильно определить все множество целевых блоков таблицы переходов, нужно знать размер массива.

4.4.1 Оценка размера массива смещений по релокационной информации

Элементы массива располагаются в памяти последовательно. Для каждого элемента существует релокация с полем `r_offset` равным адресу этого элемента. Основываясь на этих фактах, можно предложить способ оценки размера массива по релокационной информации. Он состоит в подсчете количества релокаций, которые начинаются с адреса *Base* и идут подряд. Псевдокод этого метода приведен в алгоритме 2.

Алгоритм 2 Алгоритм оценки размера массива смещений по релокациям. Вычисляет количество элементов массива в переменной *Len*.

```
Len ← 0
Addr ← Base
while Relocation at Addr is R_AARCH64_PREL32 do
    Len ← Len + 1
    Addr ← Addr + 4
end while
```

Этот алгоритм имеет большой недостаток. Если после массива смещений окажутся релокации типа `R_AARCH64_PREL32`, не относящиеся к таблице переходов, то посчитанный размер окажется больше реального значения.

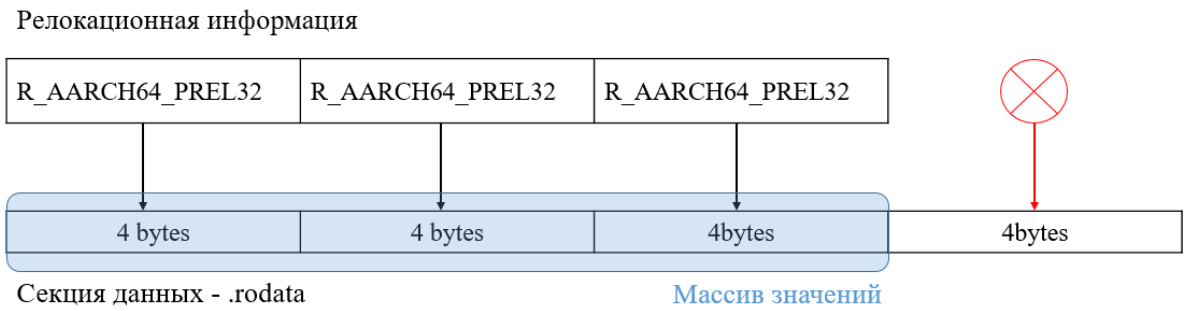

Рис. 6. К алгоритму вычисления размера массива по релокациям.

4.4.2 Оценка размера массива смещений по расширенному шаблону доступа

Второй предлагаемый метод оценки размера массива основывается на анализе инструкций в коде программы, также как и метод поиска шаблона доступа. Рассмотрим расширенный шаблон доступа таблицы переходов. Он приведен в лист. 3. По сравнению с обычным шаблоном, рассмотренным ранее, здесь добавлены еще две инструкции - инструкция сравнения `cmp` и инструкция условного перехода `b.hi`. С точки зрения логики программы эти инструкции отвечают за проверку того, что значение $Index$ лежит в пределах размера массива. Схематично граф потока управления изображен на рисунке рис. 7. Регистр, используемый как индекс в массиве смещений, сравнивается с константой равной размеру массива минус 1. В случае, если индекс оказывается больше константы, то инструкции шаблона доступа не исполняются. Такую логику работы имеет, например, ключевое слово `default` из языков C и C++.

Листинг 3 Пример расширенного шаблона доступа.

```
adrp    x28, 0x10000
add     x28, x28, 0xab8
cmp     w8, 0x2f
b.hi    0x9bc
ldrsw   x8, [x28, x8, lsl 2]
add     x8, x8, x28
br      x8
```

Для определения размера массива смещений по расширенному шаблону доступа достаточно по графу потока данных найти инструкцию сравнения `cmp`. Определение флагов сравнения, порождаемое этой инструкцией, должно быть доступно инструкции условного перехода `b.hi` в смысле определения, данного выше для задачи доступных определений. Константный аргумент инструкции `cmp` позволяет определить размер массива смещений. Данный подход никак не опирается на релокационную информацию и использует только анализ исполняемого кода приложения. Обозначим BB_{load} базовый блок, содержащий обращение к массиву смещений (инструкцию загрузки из памяти), $pred(BB)$ - множество базовых блоков, имеющих ребро ведущее

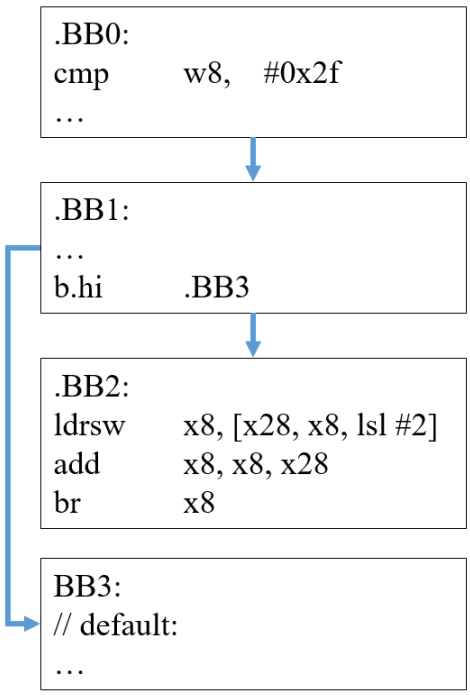

Рис. 7. Граф потока управления для расширенного шаблона доступа таблицы переходов.

к блоку BB в графе потока управления. При обработке графа нужно учитывать, что искомый блок с инструкцией `cmp` может не быть непосредственным предшественником блока BB_{load}. Псевдокод приведен в алгоритме 3.

Алгоритм анализа расширенного шаблона доступа позволяет точно определить размер массива смещений.

4.5 Устойчивость алгоритма анализа к изменению шаблона доступа

Алгоритм анализа таблиц переходов полагается на поиск определенных последовательностей инструкций в коде приложения. В общем случае, искомые инструкции могут располагаться в бинарном коде приложения произвольным образом. Однако алгоритм работает с представлением программы в форме графа потока управления и графа потока данных. Поэтому он не зависит от расположения инструкций в бинарном файле приложения.

В некоторых случаях компилятор может генерировать последовательности инструкций, которые реализуют ту же самую логику, но отличаются набором инструкций от заданного шаблона. Примеры таких последовательностей приведены в лист. 5 и лист. 4.

Количество сохраняющих логику преобразований шаблона доступа бесконечно велико. Например, шаблон можно бесконечно дополнять инструкциями `nop` не выполняющими никакого действия (пример лист. 6). Однако такое преобразование не

Алгоритм 3 Алгоритм оценки размера массива смещений по расширенному шаблону доступа. Количество элементов в массиве смещений записывается в переменную Len.

$Q \leftarrow$ пустая очередь
$Visited \leftarrow$ пустая очередь
$Len \leftarrow 0$
$Q.push(pred(BB_{load}))$
while Q not empty **do**
  $BB \leftarrow Q.pop()$
  if $Visited$ содержит BB **then**
    пропустить итерацию цикла
  end if
  $Visited.push(BB)$
  if BB содержит инструкцию b.hi **then**
    for each определения доступные b.hi **do**
      if определение - инструкция cmp **then**
        Imm - аргумент cmp
        $Len \leftarrow Imm + 1$
        return
      end if
    end for
  else
    for each $p \in pred(BB)$ **do**
      if $Visited$ не содержит p **then**
        $Q.push(p)$
      end if
    end for
  end if
end while

Листинг 4 Пример шаблона доступа с инструкцией mov.

```
adrp    reg1, imm1
add     reg2, reg1, imm2
mov     reg6, reg2
ldrsw   reg3, [reg6, reg4, lsl 2]
add     reg5, reg6, reg3
br      reg5
```

Листинг 5 Пример шаблона доступа с инструкцией nop.

```
adrp    reg1, imm1
add     reg2, reg1, imm2
nop
ldrsw   reg3, [reg2, reg4, lsl 2]
add     reg5, reg2, reg3
br      reg5
```

влияет на граф потока данных, поэтому не влияет и на алгоритм анализа шаблона доступа.

Существуют более сложные случаи преобразований, меняющие граф потока данных, но сохраняющие логику работы шаблона доступа таблицы переходов (пример лист. 4). Назовем две последовательности инструкций в потоке исполнения программы равными с учетом множества регистров R, если их исполнение приводит программу в одну и ту же точку графа потока управления, и в этой точке значения всех регистров множества R равны для обеих последовательностей. Рассмотрим пример шаблона доступа лист. 4 и покажем, что существует бесконечное множество последовательностей равных шаблону доступа с учетом регистров $\{reg3,\ reg4,\ reg5,\ reg6\}$. Действительно, шаблон можно дополнить произвольным количеством пар инструкций `mov` копирования значений из одного регистра в другой (лист. 7). Каждая новая пара инструкций `mov` меняет граф потока данных, но оставляет последовательности равными с учетом регистров $\{reg3,\ reg4,\ reg5,\ reg6\}$.

Листинг 6 Пример шаблона доступа с множеством инструкций nop. Граф потока данных не изменяется. Логика работы не изменяется.

```
adrp    reg1, imm1
add     reg2, reg1, imm2
nop
nop
...
nop
ldrsw   reg3, [reg2, reg4, lsl 2]
add     reg5, reg2, reg3
br      reg5
```

Листинг 7 Пример шаблона доступа с множеством инструкций mov. Граф потока данных изменяется. Логика работы не изменяется.

```
adrp    reg1, imm1
add     reg2, reg1, imm2
mov     reg6, reg2
mov     reg2, reg6
mov     reg6, reg2
mov     reg2, reg6
...
mov     reg6, reg2
ldrsw   reg3, [reg6, reg4, lsl 2]
add     reg5, reg6, reg3
br      reg5
```

Хороший алгоритм поиска шаблона доступа должен быть устойчив к сохраняющим равенство преобразованиям шаблона, то есть уметь находить шаблон доступа в любых последовательностях инструкций, возможно отличающихся от шаблона доступа набором инструкций, но обладающих такой же логикой работы. Обработать все такие последовательности в общем случае невозможно по двум причинам:

1. Как было показано число таких последовательностей бесконечно велико. В общем случае граф потока данных, сохраняющий логику шаблона доступа таблицы переходов, может иметь произвольную сложность.

2. Статический анализ приложения не позволяет обработать последовательности в которых значения регистров сначала сохраняются, а потом загружаются из памяти (лист. 8), или в которых присутствуют инструкции косвенного перехода.

Листинг 8 Пример шаблона доступа с передачей значения через память. Такой случай нельзя обработать статически.

```
adrp    reg1, imm1
add     reg2, reg1, imm2
strw    reg2, [regA]
...
ldr     reg2, [regB]
ldrsw   reg3, [reg2, reg4, lsl 2]
add     reg5, reg2, reg3
br      reg5
```

Код реальных приложений, которые являются целевыми для LLVM BOLT, оптимизируется и генерируется компилятором в процессе создания программы. Поэтому можно предположить, что в абсолютном большинстве случаев бинарный код шаблона доступа будет минимален или близок к минимальному. Тогда обработка минимального шаблона и небольшого числа наиболее частых его модификаций должны обеспечить достаточную надежность в определении таблиц переходов.

Глава 5

Результаты и обсуждение

Описанные в рамках данной работы алгоритмы детекции и анализа таблиц переходов были реализованы в инфраструктуре бинарного оптимизатора LLVM BOLT. Встраивание результатов работы в существующую иерархию классов LLVM BOLT позволяет оптимизатору формировать корректные таблицы переходов в адресном пространстве оптимизированного приложения.

5.1 Сравнение алгоритмов оценки размера массива смещений

В работе предложено два алгоритма для оценки размера массива смещений.

1. Алгоритм оценки размера массива смещений по релокационной информации в главе 4.4.1.

2. Алгоритм оценки размера массива смещений по расширенному шаблону доступа в главе 4.4.2.

Первый алгоритм проще в смысле реализации, однако обладает рядом важных недостатков по сравнению со вторым. Для обоих алгоритмов существуют примеры входных данных, которые приведут к ошибке в оценке размера таблицы переходов. Однако в случае алгоритма анализа релокаций ошибка не детектируется, тогда как в случае алгоритма анализа расширенного шаблона доступа всегда точно известно если оценить размер не удалось. Это свойство делает второй алгоритм более подходящим для промышленных продуктовых решений, каким является бинарный оптимизатор LLVM BOLT.

Второе преимущество анализа по шаблону доступа, состоит в том, что этот подход не использует дополнительную информацию, в частности, релокации. Это свойство полезно при оптимизации приложений для конечных пользователей. В целях уменьшения размера из бинарных файлов таких приложений часто удаляют релокации, например, с помощью инструмента `strip` [15]. Анализ релокаций в таких бинарных

файлах невозможен. Метод анализа по шаблону доступа позволяет обрабатывать бинарные файлы без релокационной информации.

Преимущества метода анализа по расширенному шаблону обосновывают выбор этого алгоритма для использования в LLVM BOLT.

5.2 Оценка устойчивости алгоритма определения размера массива смещений к изменению шаблона доступа

Для проверки гипотезы о надежности определения таблиц переходов по минимальному шаблону, высказанной в главе 4.5, были подсчитаны метрики, характеризующие точность определения таблиц переходов. Для каждого приложения a в тестовом наборе были посчитаны полное количество таблиц переходов N_a и количество таблиц, для которых удалось установить адрес начала и конца массива смещений $N_a^{correct}$. По этим данным были посчитаны количество приложений, в которых все таблицы обработаны полностью верно $N^{correct}$, и суммарное количество приложений, в которых встретились таблицы перехода N. Далее были вычислены следующие метрики:

- Доля приложений без ошибок:

$$A = \frac{N^{correct}}{N}$$

- Средняя доля неверно обработанных таблиц среди приложений с ошибками:

$$E = \frac{\sum_a \frac{N_a - N_a^{correct}}{N_a}}{N - N^{correct}}$$

Полученные значения приведены в таблице 2.

Метрика	Значение %
A	99.4%
E	0.33%

Таблица 2. Метрики надежности алгоритма анализа таблиц переходов.

Полный размер тестового набора составил 621 приложение. Из них 170 содержали таблицы переходов. Ошибка обработки была обнаружена лишь в 1 приложении. В нем для 2 таблиц из 606 не получилось вычислить размер массива смещений.

Близкое к 100% значение A, говорит о том, что абсолютное большинство приложений обрабатывается полностью корректно. Это означает что приложения с проблемными случаями встречаются очень редко. Близкое к 0% значение E говорит о том, что в тех приложениях, где ошибки все-таки встречаются, их очень мало.

5.3 Оценка улучшения стабильности работы оптимизатора

Для оценки результатов работы было проведено повторное исследование стабильности LLVM BOLT на том же наборе тестов, который выявил проблемы при оптимизации таблиц переходов. Ожидалось, что внесенные изменения повысят процент приложений, работающих корректно после оптимизации. Результаты тестирования показали, что все 602 теста из набора работают корректно после оптимизации. Таким образом, подход предложенный в данной работе решает проблему оптимизации приложений с таблицами переходов.

Глава 6

Заключение

В ходе работы была исследована корректность работы приложений после оптимизации с помощью LLVM BOLT для архитектуры AArch64. В открытой версии инструмента были выявлены ошибки формирования таблиц переходов в оптимизированных приложениях. В 24 из 602 (4%) тестовых приложений после оптимизации наблюдались ошибки времени исполнения.

Были предложены алгоритмы поиска, анализа и модификации таблиц переходов для генерации корректных оптимизированных приложений. Алгоритм поиска использует графы потока управления и потока данных приложения для обнаружения последовательностей инструкций, которые исполняют логику таблиц переходов.

Алгоритм анализа определяет начало и размер массива смещений - области данных, содержащей информацию о целевых адресах таблицы переходов. Определение границ массива смещений позволяет статически вычислить все целевые базовые блоки таблицы. Для оценки размера массива смещений предложено два способа - с использованием релокационной информации (глава 4.4.1) и по расширенному шаблону доступа (глава 4.4.2). Приведено сравнение двух способов (глава 5.1), обосновывающее выбор второго алгоритма для использования в LLVM BOLT.

Описанные в работе алгоритмы реализованы как часть бинарного оптимизатора LLVM BOLT.

Дана численная оценка частоты возникновения ошибок при определении размера массива смещений по шаблону доступа таблицы переходов. Полученные значения приведены в таблице 2. Результаты показывают высокую надежность алгоритма.

Исследована корректность работы приложений, оптимизированных с помощью LLVM BOLT с применением описанных в данной работе техник обработки таблиц переходов (глава 5.3). Все 602 приложения из тестового набора работают корректно при использовании модифицированной версии оптимизатора. Таким образом, достижением данной работы является существенное улучшение стабильности работы LLVM BOLT для архитектуры AArch64.

Список литературы

[1] URL: https://www.spec.org/benchmarks.html.

[2] Dehao Chen, David Xinliang Li, Tipp Moseley. "AutoFDO: Automatic Feedback-Directed Optimization for Warehouse-Scale Applications". *CGO 2016 Proceedings of the 2016 International Symposium on Code Generation and Optimization*. New York, NY, USA, 2016, с. 12—23.

[3] Dehao Chen и др. "Taming Hardware Event Samples for FDO Compilation". *Proceedings of International Symposium on Code Generation and Optimization (CGO)*. 2010.

[4] *ELF for the Arm 64-bit Architecture (AArch64)*. URL: https://github.com/ARM-software/abi-aa/blob/main/aaelf64/aaelf64.rst.

[5] Dongsoo Ha, Wenhui Jin, Heekuck Oh. "REPICA: Rewriting Position Independent Code of ARM". *IEEE Access* **6** (2018), с. 50488—50509. DOI: 10.1109/ACCESS.2018.2868411.

[6] Stephen Kell, Dominic P. Mulligan, Peter Sewell. "The Missing Link: Explaining ELF Static Linking, Semantically". *SIGPLAN Not.* **51** 10 (сент. 2016), с. 607—623. ISSN: 0362-1340. DOI: 10.1145/3022671.2983996. URL: https://doi.org/10.1145/3022671.2983996.

[7] C. Lattner, V. Adve. "LLVM: a compilation framework for lifelong program analysis and transformation". *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* 2004, с. 75—86. DOI: 10.1109/CGO.2004.1281665.

[8] *LLVM Clang Compiler User's Manual*. URL: https://clang.llvm.org/docs/UsersManual.html.

[9] C.K. Luk и др. "Ispike: a post-link optimizer for the Intel/spl reg/ Itanium/spl reg/ architecture". *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* 2004, с. 15—26. DOI: 10.1109/CGO.2004.1281660.

[10] Angélica Aparecida Moreira, Guilherme Ottoni, Fernando Magno Quintão Pereira. "VESPA: Static Profiling for Binary Optimization". *Proc. ACM Program. Lang.* **5** OOPSLA (окт. 2021). DOI: 10.1145/3485521. URL: https://doi.org/10.1145/3485521.

[11] Guilherme Ottoni, Bertrand Maher. “Optimizing function placement for large-scale data-center applications”. *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2017, с. 233—244. DOI: 10.1109/CGO.2017.7863743.

[12] Maksim Panchenko et al. “BOLT: A Practical Binary Optimizer for Data Centers and Beyond” (2018). arXiv: 1807.06735 [cs.PL].

[13] Karl Pettis, Robert C. Hansen. “Profile Guided Code Positioning”. *SIGPLAN Not.* **25** 6 (июнь 1990), с. 16—27. ISSN: 0362-1340. DOI: 10.1145/93548.93550. URL: https://doi.org/10.1145/93548.93550.

[14] Joe Savage, Timothy M. Jones. “HALO: Post-Link Heap-Layout Optimisation”. *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*. CGO 2020. San Diego, CA, USA: Association for Computing Machinery, 2020, с. 94—106. ISBN: 9781450370479. DOI: 10.1145/3368826.3377914. URL: https://doi.org/10.1145/3368826.3377914.

[15] *strip(1) - Linux manual page*. URL: https://man7.org/linux/man-pages/man1/strip.1.html.

[16] Ruoyu Zhou, Timothy M. Jones. “Janus: Statically-Driven and Profile-Guided Automatic Dynamic Binary Parallelisation”. *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2019, с. 15—25. DOI: 10.1109/CGO.2019.8661196.

[17] Лисицын С.А. “Исследование и оптимизация применения трасс исполнения приложения для статической бинарной трансляции под RISC архитектуры”. Дис. ... док. ФГАОУ ВО Московский физико-технический институт (национальный исследовательский университет), 2022.