Институт системного программирования им. В.П. Иванникова РАН

На правах рукописи

Батузов Кирилл Андреевич

Исследование и разработка методов оптимизации программ для систем динамической двоичной трансляции

05.13.11 – Математическое и программное обеспечение вычислительных машин, комплексов и компьютерных сетей

ДИССЕРТАЦИЯ

на соискание ученой степени

кандидата физико-математических наук

Научный руководитель

к. ф.-м. н.

Белеванцев А. А.

Москва – 2018

Оглавление

Введение

Актуальность темы исследования. Программные эмуляторы позволяют выполнять машинный код, созданный для одной процессорной архитектуры, на другой процессорной архитектуре. Без этого не обойтись при использовании ПО с закрытым исходным кодом на несовместимых аппаратных платформах и при разработке ПО для встраиваемых систем, когда разработка и итоговое выполнение программ происходят на различных архитектурах. Программные эмуляторы позволяют разрабатывать и отлаживать приложения, даже не имея в распоряжении аппаратуры, например, в случае, когда выпуск соответствующей аппаратуры еще не налажен.

Кроме того, эмуляторы позволяют сохранять, модифицировать и воссоздавать отдельные состояния эмулируемой системы и ход ее работы. Это дает дополнительные возможности по изучению и отладке ПО, такие как фаззинг-тестирование и обратная отладка, которые невозможно либо очень трудно реализовать на реальной аппаратуре. Возможности эмуляторов по контролю и анализу состояния всей вычислительной системы делают их важным инструментом во вопросах информационной безопасности. Они применяются в таких задачах, как обратная инженерия вредоносного ПО и раннее обнаружение атак.

Эффективность эмулятора очень важна во всех вышеперечисленных задачах. Эмулятор должен обеспечивать не только минимальную производительность, необходимую для корректной работы программ, взаимодействующих с внешним миром через, например, сетевые интерфейсы, но и приемлемую скорость работы, чтобы быть использованным при отладке приложений, которая предполагает многократный запуск исследуемого приложения.

Основным методом построения эффективных эмуляторов является динамическая двоичная трансляция. В этом методе во время выполнения программы для ее кода строится и выполняется эквивалентный код для процессора, на котором запущен эмулятор. Это позволяет достичь большей производительно-

сти, чем при интерпретации. С другой стороны, данный подход не обладает такими ограничениями аппаратной виртуализации, как необходимость выполнять виртуализованный код на процессоре того же семейства, что и эмулируемый.

В контексте производительности динамической двоичной трансляции актуальные следующие ее аспекты:

- эффективность сгенерированного в ходе трансляции кода;
- эффективность самого процесса трансляции, в том числе кэширования оттранслированного кода и обработки событий.

Из перечисленного лучше изучен вопрос построения эффективного процесса трансляции и выполнения оттранслированного кода. Так, например, в эмуляторе с открытым исходным кодом QEMU реализованы хранение, быстрый поиск и повторное использование сгенерированных участков кода, сцепление участков кода и другие оптимизации.

Методы оптимизации программ в статических компиляторах хорошо изучены, однако они не могут быть использованы напрямую в динамической двоичной трансляции. Основной сложностью является необходимость применять эти методы во время выполнения программы, то есть выгода от их использования должна превышать нужные затраты времени и памяти. Особенностью, связанной непосредственно с двоичной трансляцией, является специфика оптимизируемого внутреннего представления, полученного из бинарного кода, как правило, уже оптимизированного компилятором. Кроме того, регионы программы, подвергаемые оптимизации, небольшие, что делает применение многих оптимизаций невозможным или неэффективным, и, более того, отсутствует высокоуровневая информация об этих регионах (типы данных, конструкции потока управления и т.п.).

Вопросом оптимизации динамической двоичной трансляции занимаются как во многих научных центрах, таких как университет Колумбии, университет Висконсина, Национальный университет Тайваня, ИСП РАН, МЦСТ, так и

во многих коммерческих организациях, таких как IBM, WindRiver, Linaro, Red Hat. Многие результаты, полученные в коммерческих организациях, остаются закрытыми, и их детали недоступны. Открытые результаты не всегда имеют реализацию в распространенных эмуляторах с открытым исходным кодом. Кроме того, они не покрывают весь спектр возможных улучшений.

Целью диссертационной работы является разработка методов оптимизации кода программ во время динамической двоичной трансляции и их реализация в системе динамической двоичной трансляции с открытым исходным кодом QEMU.

Для достижения поставленных целей были сформулированы и решены следующие **задачи**.

1. Разработка методов машинно-независимых оптимизаций, учитывающих особенности динамической двоичной трансляции.

2. Разработка алгоритмов распределения регистров, учитывающих особенности динамической двоичной трансляции.

3. Разработка методов трансляции инструкций одной процессорной архитектуры в инструкции другой процессорной архитектуры, позволяющих задействовать расширенные вычислительные возможности целевого процессора.

Научная новизна. В работе были получены следующие результаты, обладающие научной новизной.

1. Разработан метод решения задачи анализа потока данных для ациклических графов потока управления для применения во время динамической двоичной трансляции.

2. Разработан однопроходный алгоритм локального распределения регистров для применения во время динамической двоичной трансляции.

Данный алгоритм учитывает имеющуюся информацию о времени жизни переменных для более эффективного выбора регистров для сброса их содержимого в память.

3. Разработан однопроходный комбинированный (глобальный и локальный) алгоритм распределения регистров для применения во время динамической двоичной трансляции. Данный алгоритм позволяет выполнять глобальное распределение регистров, совмещенное с генерацией кода, без введения дополнительного внутреннего представления.

4. Разработан метод динамической двоичной трансляции векторных инструкций одной процессорной архитектуры в векторные инструкции другой процессорной архитектуры.

Теоретическая и практическая значимость. Разработанный в диссертации однопроходный алгоритм анализа потока данных может быть использован для построения других машинно-независимых оптимизаций для двоичной трансляции, которые сводятся к задаче анализа потока данных. Разработанные алгоритмы распределения регистров могут использоваться в исследованиях по расширению регионов, подвергаемых динамической двоичной трансляции. Предложенный метод выражения векторных инструкций одной процессорной архитектуры через векторные инструкции другой процессорной архитектуры является общим и может быть применен к любой паре процессорных архитектур. Разработанные в диссертации методы организации динамической двоичной трансляции могут быть использованы при преподавании курсов в МГУ, МФТИ и ВШЭ. Все разработанные методы были реализованы в эмуляторе с открытым исходным кодом QEMU. Модифицированный эмулятор используется как часть систем динамического анализа бинарного кода, разрабатываемых в ИСП РАН. Разработанные машинно-независимые оптимизации были включены в QEMU.

Положения, выносимые на защиту.

1. Алгоритм решения задачи анализа потока данных для ациклических графов потока управления для применения во время динамической двоичной трансляции.

2. Однопроходный алгоритм локального распределения регистров для применения во время динамической двоичной трансляции. Данный алгоритм учитывает имеющуюся информацию о времени жизни переменных для более эффективного выбора регистров для сброса их содержимого в память.

3. Однопроходный комбинированный (глобальный и локальный) алгоритм распределения регистров для применения во время динамической двоичной трансляции. Данный алгоритм позволяет выполнять глобальное распределение регистров, совмещенное с генерацией кода, без введения дополнительного внутреннего представления.

4. Метод динамической двоичной трансляции векторных инструкций одной процессорной архитектуры в векторные инструкции другой процессорной архитектуры.

Апробация результатов. Основные результаты диссертации обсуждались на следующих конференциях:

- Конференция «РусКрипто» 2015,
- Открытая конференция ИСП РАН 2016,
- Открытая конференция ИСП РАН 2017,
- Конференция по виртуализации Linux «KVM Forum» 2017 (Прага),
- Научно-исследовательский семинар Института системного программирования им. В.П. Иванникова РАН.

Публикации. Материалы диссертации опубликованы в 4 печатных работах [1—4]. Работа [4] опубликована в журнале, индексируемом системами Scopus и Web of Science. Работы [1—3] опубликованы в журнале, входящем в список ВАК. Вклад автора в работе [1] заключается в разработке машинно-независимых оптимизаций в эмуляторе QEMU.

Личный вклад автора. Все представленные в диссертации результаты получены лично автором.

Структура и объем диссертации. Диссертация состоит из введения, 5 глав, заключения и библиографии. Общий объем диссертации 105 страниц, включая 3 рисунка. Библиография включает 60 наименований.

Глава 1

Обзор методов оптимизации программ при динамической двоичной трансляции

Программные эмуляторы позволяют выполнять машинный код ПО на несовместимой аппаратной платформе. Они могут применяться, например, при отладке приложения для встраиваемых систем. Такие системы обычно имеют очень ограниченные ресурсы, что делает запуск и использование на них среды разработки невозможным. Кроме того, возможности эмуляторов по анализу, сохранению и изменению состояния всей гостевой системы позволяют с их помощью реализовывать такие возможности по отладке и тестированию программ, как фаззинг-тестирование и обратная отладка. Также эти свойства эмуляторов используются в задачах информационной безопасности, например, для анализа вредоносного ПО. Производительность эмулятора является важной его характеристикой, так как все вышеперечисленные задачи предполагают многократный запуск ПО, которое, в свою очередь, может работать продолжительное время. Основным методом построения эффективных программных эмуляторов является *динамическая двоичная трансляция*.

Двоичная трансляция — это процесс получения по программе P программы Q, удовлетворяющей заданным требованиям, если обе программы записаны в виде машинных кодов. Различают два подхода к двоичной трансляции: статическая двоичная трансляция и динамическая двоичная трансляция [5].

Статическая трансляция производится один раз для исполнимого файла, а полученный код выполняется после ее окончания. В данном подходе не обрабатывается самомодифицирующийся код. В архитектуре фон Неймана программа и данные хранятся одинаковым образом в памяти. В статье [6] говорится, что отделение кода программы от данных во время статической трансляции является алгоритмически неразрешимой задачей. В статье [7] утверждается, что

проблема останова может быть сведена к этой задаче.

Динамическая двоичная трансляция производится во время выполнения программы. При этом можно транслировать только участки кода, которые выполняются, что автоматически решает проблему как отделения инструкций от данных, так и самомодифицирующегося кода [8]. В этом случае на вход транслятору попадает текущее состояние самомодифицирующегося кода непосредственно перед тем, как он должен быть выполнен.

В данной работе будет рассматриваться только динамическая двоичная трансляция. С точки зрения производительности эмулятора при использовании динамической двоичной трансляции можно улучшать саму процедуру трансляции, либо повышать эффективность генерируемого при трансляции кода. В данной работе будет рассматриваться только последнее.

В данной главе описывается процесс динамической двоичной трансляции. Более детально сам процесс описан в разделе 1.2. Динамическая двоичная трансляция производится во время работы программы небольшими фрагментами, называемыми блоками трансляции. Она выполняется в несколько этапов. Сначала происходит дизассемблирование входного машинного кода во внутреннее представление. Затем над внутренним представлением выполняются оптимизации. Обзор существующих алгоритмов оптимизации приведен в разделе 1.3. После оптимизаций происходит генерация результирующего кода. На этом этапе происходит распределение регистров, описанное более подробно в разделе 1.6. Увеличивать производительность кода, полученной в результате динамической двоичной трансляции можно также с помощью параллельного выполнения и использования векторных инструкций. Эти вопросы рассматриваются в разделах 1.4 и 1.5 соответственно. В разделе 1.7 рассматриваются примеры программ, использующих динамическую двоичную трансляцию. В разделе 1.8 описывается структура данной работы и обозначаются вопросы, которые должны быть рассмотрены.

1.1. Основные понятия и определения

Программа (операционная система), которая подвергается динамической двоичной трансляции называется *гостевой программой* (*гостевой операционной системой*), а ее код — *гостевым кодом*. Код, который генерируется в результате динамической двоичной трансляции, называется *целевым кодом*. Если гостевой код и целевой предназначены для выполнения на различных архитектурах процессоров, то эти архитектуры называют *гостевой* и *целевой* соответственно.

При использовании динамической двоичной трансляции в эмуляторах (например, QEMU) происходит наложение терминов. С точки зрения эмулятора, целевая архитектура — это та архитектура, которая эмулируется, что соответствует гостевой архитектуре в терминах динамической двоичной трансляции. Чтобы избежать путаницы, архитектуру (операционную систему, процессор), на которой выполняется эмулятор, будем называть *основной*. В данной работе этот термин будет использоваться в контексте динамической двоичной трансляции вместо термина «целевой».

Трансляция гостевого кода происходит небольшими участками, называемыми *блоками трансляции*. Для повышения производительности оттранслированные блоки сохраняются в *кэше трансляций* и могут быть использованы повторно. Каждому блоку трансляции соответствует

- участок гостевого кода, из которого он был получен,
- фрагмент целевого кода, сгенерированный из него в процессе динамической двоичной трансляции,
- некоторое внутреннее представление, используемое во время трансляции.

Базовым блоком будем называть максимальную по включению последовательность инструкций, которая может начать выполняться только с первой,

а закончить — только на последней инструкции из последовательности. Рассмотрим произвольный блок трансляции. Обозначим множество всех его базовых блоков B. Введем также два фиктивных базовых блока $Entry$ и $Exit$, соответствующие входу и выходу из блока трансляции. Тогда граф $\langle V, E \rangle$, где $V = B \cup \{Entry, Exit\}$, а $E \subseteq V \times V$ — множество возможных передач управления от одного базового блока другому, назовем *графом потока управления* для данного блока трансляции.

В данной работе будем считать, что блок трансляции удовлетворяет следующим ограничениям:

- его граф потока управления ациклический,
- из вершины $Entry$ исходит ровно одна дуга.

Заметим, что фрагмент гостевого кода, из которого был получен данный блок трансляции, может иметь несколько входов. Тогда каждому такому входу будет соответствовать свой блок трансляции. Таким образом, одному и тому же участку гостевого кода могут соответствовать несколько блоков трансляции. Кроме того, будем считать, что граф потока управления блока трансляции отсортирован топологически. В случае, если данное условие не выполняется, топологическая сортировка графа потока управления может быть выполнена за $O(|E|)$, то есть время, сопоставимое со временем, необходимым на построение этого графа.

1.2. Процесс динамической двоичной трансляции

Процесс динамической двоичной трансляции можно условно разделить на четыре повторяющихся циклически этапа:

- определение следующего блока трансляции и поиск его в кэше трансляций,

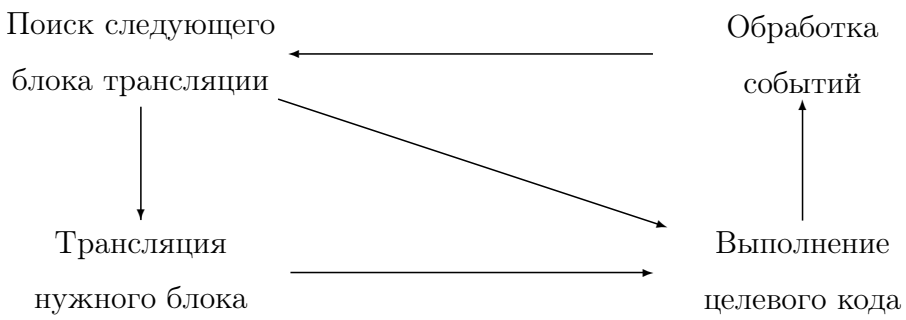


Рис. 1.1. Основной цикл динамической двоичной трансляции

- трансляция блока, если он не был найден,
- выполнение полученного целевого кода,
- обработка внешних событий.

Данные этапы образуют основной цикл динамической двоичной трансляции, который схематично изображен на рис. 1.1.

Процесс начинается с определения нужного блока трансляции. Он вычисляется, исходя из значения счетчика инструкций и других параметров, которые могут варьироваться от системы к системе. После того, как блок трансляции был определен, происходит поиск его в имеющемся кэше трансляций. Если он был найден, он отправляется на выполнение. В противном случае запускается процедура трансляции, которая строит нужный блок из гостевого кода, после чего полученный блок отправляется на выполнение. После того, как блок или несколько блоков были выполнены, происходит обработка внешних событий. Конкретные типы событий сильно зависят от эмулируемой системы. Например, это могут быть сигналы или готовность файловых дескрипторов. Затем определяется следующий блок трансляции, который должен быть выполнен, и процесс повторяется.

Теперь рассмотрим более подробно процедуру трансляции, которая схематично изображена на рис. 1.2. Она начинается с того, что выделяется участок гостевого кода и происходит его дизассемблирование во внутреннее представле-

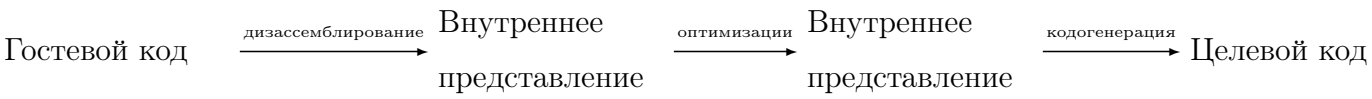


Рис. 1.2. Процедура трансляции

ние. Далее над этим внутренним представлением производятся машинно-независимые оптимизации, после чего из него генерируется целевой код. Во время генерации целевого кода производятся машинно-зависимые оптимизации и преобразования программы, такие как распределение регистров и генерация кода. В данной работе будут рассматриваться два последних этапа, так как на них производятся оптимизации, определяющие в конечном итоге быстродействие полученного кода.

1.3. Алгоритмы оптимизации программ

Задача оптимизации программ в контексте компиляторов изучается уже очень давно и подробно описана в литературе [9; 10]. Однако динамическая двоичная трансляция обладает рядом особенностей, из-за которых многие алгоритмы, эффективно работающие в компиляторах, не могут быть применены. Главными такими особенностями являются:

- оптимизации производятся над ациклическими блоками трансляции, которые, как правило, имеют очень небольшой размер,
- оптимизации производятся во время выполнения, т.е. время, затраченное на оптимизации, также входит в результирующее время работы программы,
- во время оптимизаций отсутствует информация, которую компилятор получает из высокоуровневого представления (например, граф потока управления).

Попытки применить существующие компиляторы для оптимизации во время динамической двоичной трансляции без учета особенностей последней приводят к существенному падению производительности [11]. Одним из способов разрешения данной проблемы является профилирование программы во время выполнения и применение сложных алгоритмов оптимизации только к коду, который выполняется большое количество раз. Данный подход рассмотрен в работе [12]. Другой подход состоит в аккуратном выборе оптимизаций, которые будут производиться над над транслируемым кодом. В работе [13] описывается результат применения возможностей компиляторной инфраструктуры LLVM [14] с учетом этого подхода. В данной работе будут рассматриваться вопрос выбора и реализации оптимизаций в самих эмуляторах, а не использование инфраструктуры компилятора вместо имеющегося динамического двоичного транслятора. Полученные результаты могут быть объединены с профилированием и применением более сложных компиляторных оптимизаций к часто выполняющимся участкам кода, но этот вопрос выходит за рамки данной работы.

1.3.1. Машинно-независимые оптимизации

Машинно-независимые оптимизации производятся над машинно-независимым представлением программы (или фрагмента кода). Они несущественно зависят от особенностей конкретной машины, на которой будет выполняться программа. За основу берется идея уменьшения избыточности: то есть не перевычислять выражение, если его значение было уже посчитано ранее, не выполнять код, не влияющий на результат работы программы, и так далее. Машинно-независимые оптимизации, как правило, сводятся к задаче анализа потоков данных. Задача анализа потока данных подробно описана в литературе, и далее мы будем пользоваться нижеследующей постановкой, основанной на [9].

В анализе потоков данных с каждой точкой программы связывается *значение потока данных* из некоторого множества V. Над элементами множества V определен *оператор сбора* $\wedge$ такой что $(V, \wedge)$ образуют *полурешетку*, т. е.

для любых x, y и z из V выполнены следующие соотношения:

- $x \wedge x = x$,
- $x \wedge y = y \wedge x$,
- $x \wedge (y \wedge z) = (x \wedge y) \wedge z$,
- существует *верхний* элемент $\top$ такой, что $\top \wedge x = x$ для любого x,
- может (но не обязательно) существовать *нижний* элемент $\bot$ такой, что $\bot \wedge x = \bot$ для любого x.

Оператор сбора задает отношение частичного порядка на элементах множества V:

$$x \leqslant y \iff x \wedge y = x.$$

Если значение потока данных перед инструкцией s равно $IN[s]$, а после — $OUT[s]$, то *задача потока данных* состоит в нахождении допустимых значений $IN[s]$ и $OUT[s]$ для всех инструкций s. На значения $IN[s]$ и $OUT[s]$ накладываются два типа ограничений:

- семантические (связанные с семантикой инструкции) и
- ограничения, основанные на потоке управления.

Существует два вида задач анализа потока данных: *прямые* и *обратные*. Они отличаются конкретным видом ограничений, накладываемых на значения $IN[s]$ и $OUT[s]$. Первый тип ограничений описывается уравнением

$$OUT[s] = f_s(IN[s]) \quad \text{(прямая задача)},$$

либо

$$IN[s] = f_s(OUT[s]) \quad \text{(обратная задача)}.$$

Здесь f_s — передаточная функция для инструкции s.

Второй тип ограничений имеет два случая. Для инструкций внутри базового блока данное ограничение говорит, что для двух последовательных инструкций s_{i+1} и s_i выполняется

$$IN[s_{i+1}] = OUT[s_i].$$

А для инструкций, лежащих на границах базовых блоков, выполняется следующее соотношение

$$IN[B] = \bigwedge_{P-\text{предшественник } B} OUT[P] \quad \text{(прямая задача)},$$

либо

$$OUT[B] = \bigwedge_{S-\text{потомок } B} IN[S] \quad \text{(обратная задача)}.$$

Здесь B, P и S обозначают блоки трансляции, а $IN[B]$, $OUT[B]$, $IN[S]$ и $OUT[P]$ соответствуют значениям потока данных перед и после соответствующих блоков.

Наличие соотношения $IN[s_{i+1}] = OUT[s_i]$ позволяет ввести передаточную функцию для базового блока как суперпозицию передаточных функций всех инструкций, в него входящих.

Прямая задача анализа потока данных решается итеративным алгоритмом 1, а обратная — алгоритмом 2. Данные алгоритмы по начальному приближению, используя передаточные функции, строят все более и более точные приближения до тех пор, пока множества $IN[s]$ и $OUT[s]$ не перестанут меняться для всех инструкций s.

Полурешетка называется монотонной, если

$$\forall x, y \in V : f(x \wedge y) \leqslant f(x) \wedge f(y).$$

Высотой полурешетки называется число на единицу меньшее длины максимальной последовательности $x_1, x_2, \ldots, x_n$ такой, что $x_1 < x_2 < \cdots < x_n$. Известно [9], что если полурешетка потока данных монотонна и имеет конечную высоту, то алгоритмы 1 и 2 сходятся. Скорость сходимости алгоритмов зависит

Алгоритм 1 Итеративный алгоритм для прямой задачи потока данных

procedure DATAFLOW-DIRECT

$OUT[Entry] \leftarrow v_{Entry}$

for all B отличный от $Entry$ **do**

$OUT[B] \leftarrow \top$

end for

while Внесены изменения в OUT **do**

for all B отличный от $Entry$ **do**

$IN[B] \leftarrow \bigwedge_{P-\text{предшественник } B} OUT[P]$

$OUT[B] \leftarrow f_B(IN[B])$

end for

end while

end procedure

Алгоритм 2 Итеративный алгоритм для обратной задачи потока данных

procedure DATAFLOW-REVERSE

$IN[Exit] \leftarrow v_{Exit}$

for all B отличный от $Exit$ **do**

$IN[B] \leftarrow \top$

end for

while Внесены изменения в IN **do**

for all B отличный от $Exit$ **do**

$OUT[B] \leftarrow \bigwedge_{S-\text{потомок } B} IN[S]$

$IN[B] \leftarrow f_B(OUT[B])$

end for

end while

end procedure

от конкретных значений потока данных и порядка просмотра базовых блоков в цикле алгоритма. При этом алгоритмы сходится к некоторому консервативному решению, не обязательно совпадающему с идеальным. В терминах отношения частичного порядка это означает, что полученное значение потока данных RES не превосходит идеального решения $IDEAL$:

$$\forall B : RES[B] \leqslant IDEAL[B].$$

1.4. Параллельное выполнение

Важными источником повышения производительности является параллельное выполнение нескольких операций в программе. Можно рассматривать как параллелизм на уровне команд (например, векторные инструкции, производящие вычисления не над отдельными значениями, а над их последовательностями, либо явный параллелизм на уровне команд [15; 16]), так и параллелизм на уровне создания нескольких потоков вычислений, одновременно выполняющихся на нескольких процессорах или ядрах одного процессора.

В случае эмуляторов есть очевидный источник параллелизма, связанный с эмуляцией многоядерной гостевой вычислительной системы на многоядерной основной вычислительной системе. В этом случае эмуляция каждого ядра гостевой системы может быть помещена в отдельный поток, и эти потоки затем за счет основной операционной системы будут распределены на вычислительные ядра основного процессора. Данный подход описан в работе [17]. Приведенные в статье эксперименты показывают, что он обеспечивает прирост производительности, близкий к теоретическому максимуму: параллельное выполнение на 4 ядрах гостевых программ, которые могут полностью использовать ресурсы 4 потоков, дает прирост производительности около 3.8 раз. Сейчас уже ведутся работы по включению многопоточного выполнения кода в эмулятор с открытым исходным кодом QEMU [18; 19]. Этот вопрос можно считать решенным, и

он не будет рассматриваться в данной работе.

Явный параллелизм на уровне команд обеспечивает параллельную работу очень небольших фрагментов кода. Выделение фрагментов кода такого размера в отдельные потоки нецелесообразно, так как накладные расходы превысят выигрыш от параллельного выполнения. Единственным оставшимся источником является использование параллелизма на уровне команд основной процессорной архитектуры для выражения аналогичного параллелизма гостевой процессорной архитектуры. Явный параллелизм на уровне команд встречается крайне редко и уже подробно рассмотрен в работах по системе динамической двоичной трансляции «Эльбрус» [20; 21]. В данной работе будет рассматриваться только параллелизм за счет использования векторных инструкций. Подробнее этот вопрос рассмотрен в разделе 1.5.

1.5. Трансляция векторных инструкций

Векторные инструкции процессора позволяют выполнять операцию не над одним числом, а над последовательностью однотипных чисел. Они оперируют с *векторами* фиксированной битовой длины. Длина векторов может быть различной в зависимости от конкретной архитектуры, но обычно является степенью двойки. В современных архитектурах встречаются вектора длины 64, 128, 256 и 512 бит. Каждый вектор интерпретируется как последовательность чисел. В зависимости от размера этих чисел их количество может варьироваться. Так, например, 128-битный вектор может быть интерпретирован как четыре 32-битных числа или как шестнадцать 8-битных. Интерпретация зависит от того, какие инструкции применяются к вектору.

Для организации вычислений процессорная архитектура предоставляет набор *векторных регистров* и набор *векторных инструкций*. Часто данные наборы бывают оформлены как расширение процессорной архитектуры. Так, например, широко распространенная архитектура `x86` имеет векторные расширения

MMX, SSE и AVX [22], а архитектура ARM имеет векторное расширение NEON (также называемое Advanced SIMD) [23].

Наиболее распространенные высокоуровневые языки программирования, такие, как Java, C, C++, C#, Python,[1] не имеют поддержки векторных типов данных и операций. Векторные инструкции в результирующем коде могут образовываться в результате автоматической векторизации циклов компилятором либо могут быть внесены вручную программистом, используя язык ассемблера или расширенные возможности компилятора.

Динамическая двоичная трансляция широко используется при реализации эмуляторов. Трансляция скалярных инструкций хорошо изучена и проверена на практике. Набор скалярных инструкций характеризуется тем, что есть стандартный набор базовых операций, которые реализованы во всех распространенных архитектурах: базовые арифметические операции, битовые операции, сдвиги, сравнения, условные и безусловные переходы, вызовы подпрограмм. Процессорные архитектуры могут иметь достаточно большое количество специфичных инструкций, но все эти инструкции либо хорошо выражаются через базовый набор, либо для них можно написать вспомогательную функцию, реализующую семантику данной инструкции, и вызывать эту функцию из полученного кода для основной системы.

Ситуация с векторными инструкциями менее однозначна. Сами наборы векторных инструкций предоставляют существенно больше различных операций и сильно варьируются от архитектуры к архитектуре. Выражение одних операций через другие в общем случае затруднительно, так как над всеми элементами вектора должны выполняться одни и те же операции, то есть использование условных переходов возможно только в очень ограниченном количестве ситуаций. Кроме того, реализация функции, выполняющей эквивалентные вычисления, также проблематична. Это связано с тем, что в языках высокого уровня отсутствуют необходимые типы данных, а соглашения о вызовах для

[1] Данные языки занимают первые 5 позиций индекса TIOBE за апрель 2017 года.

векторных значений гораздо сильнее различаются от компилятора к компилятору.

Существуют частные решения задачи выражения одних векторных инструкций через другие. Так, система ExaGear Desktop компании ElTech [24], служащая для запуска программ, написанных для архитектуры x86, на архитектуре ARM, предположительно умеет транслировать векторные инструкции расширения SSE архитектуры x86 в инструкции расширения NEON архитектуры ARM. Данное ПО является закрытым, и информация о его внутреннем устройстве отсутствует в свободном доступе. Однако пользовательская документация однозначно говорит, что для выполнения гостевых программ, содержащих инструкции MMX/SSE, необходима поддержка расширения NEON со стороны основной архитектуры [25]. Следует отметить, что расширение NEON предоставляет более широкий набор инструкций, чем MMX/SSE, и выражать инструкции MMX/SSE с помощью инструкций расширения NEON проще, чем в обратную сторону.

Другим примером динамической двоичной трансляции векторных инструкций в векторные является система Valgrind [26]. Valgrind — среда динамической двоичной трансляции, служащая для динамического анализа бинарного кода с целью выявления ошибок или анализа производительности. Сами анализаторы оформляются в виде машинно-независимых плагинов, которые работают с внутренним представлением. Преобразования из бинарного кода во внутреннее представление и обратно берет на себя среда. Valgrind поддерживает несколько различных архитектур с векторными расширениями, однако всегда работает в ограничении, что гостевая архитектура совпадает с основной. Таким образом, Valgrind всегда транслирует векторные инструкции в векторные инструкции той же самой архитектуры. Тем не менее, для корректной работы инструментов, поддерживаемых данной системой, внутреннее представление должно быть в достаточной мере архитектурно независимым. В частности, на различных архитектурах одни и те же операции внутреннего

представления должны обладать одной и той же семантикой. В Valgrind многие операции внутреннего представления используются для выражения инструкций различных архитектур.

Данные наблюдения позволяют сделать предположение о наличии достаточного количества пересечений между различными наборами векторных инструкции и о возможности выражения части инструкций одного набора через инструкции другого набора.

1.6. Распределение регистров

Задача распределения регистров состоит в назначении элементов данных (псевдорегистров, переменных внутреннего представления) на регистры. Ее можно разбить на две подзадачи:

- выбрать, какие данные должны находиться на регистрах в каждой точке программы, и поместить их на так называемые *псевдорегистры* — виртуальные регистры, число которых не ограничено и каждый из которых содержит ровно одно значение,
- поставить в соответствие псевдорегистрам реальные.

При этом, если распределение регистров происходит в рамках одного базового блока, то такое распределение называется локальным распределением регистров. Если при распределении учитываются несколько базовых блоков, то распределение регистров называется глобальным [9].

1.6.1. Модель

Пусть $\mathcal{R}$ — множество всех аппаратных регистров целевой архитектуры. Обозначим $\mathcal{R}^+ = \mathcal{R} \cup \{\bot\}$, где $\bot$ — специальное значение, означающее отсутствие регистра. Пусть T — множество всех переменных участка кода, для которого производится распределение регистров, а $\mathcal{I}$ — множество всех инструкций.

Тогда распределением регистров называется функция

$$\mathcal{A} : \mathcal{I} \times \{before, after\} \times T \to \mathcal{R}^{+}$$

.

1.6.2. Алгоритмы локального распределения регистров

Задача локального распределения регистров является NP-трудной [27]. Известно несколько алгоритмов ее точного решения: через сведение к задаче линейного программирования [28], либо поиск кратчайшего пути в ациклическом взвешенном ориентированном графе [29]. Однако эти алгоритмы работают достаточно медленно, и вместо них на практике применяются различные эвристики, такие, как:

- Первый дальний (Furthest-First, FF) — когда необходимо поместить псевдорегистр на реальный, выбирается любой свободный. Если свободного нет — то освобождается тот, ближайшее использование содержимого которого находится как можно дальше.

- Первый чистый (Clean-First, CF) — аналогично предыдущему, но для освобождения имеют приоритет регистры, значение которых совпадает со значением соответствующей переменной, записанным в памяти.

Эвристический алгоритм FF сочетает в себе высокую скорость работы и получает достаточно близкое к оптимальному решение [27; 28]. Опишем его более подробно.

1. Рассматривается очередная инструкция (код просматривается в прямом порядке, от начала к концу).

2. Для всех псевдорегистров, которые еще не распределены на регистры, выбираются произвольные свободные регистры.

3. Если свободных регистров недостаточно, то некоторые регистры освобождаются, то есть значение переменной, содержащееся в данных регистрах записывается в память. Такую операцию называют *сбросом регистров* в память. Для освобождения выбираются те регистры, ближайшее использование содержимого которых находится как можно дальше.

4. Алгоритм переходит к следующей инструкции и возвращается на шаг 1.

1.6.3. Алгоритм раскраски графа взаимодействия

В алгоритме глобального распределения регистров, описанном в работах [30; 31], строится граф взаимодействия регистров. Вершинами этого графа становятся переменные внутреннего представления, которые должны быть распределены на регистры. Ребра соединяют две вершины, если соответствующие переменные могут быть живы в один и тот же момент времени. Далее необходимо раскрасить данный граф в k цветов, где k — количество регистров целевой архитектуры.

Определение того, может ли граф быть раскрашен в k цветов, является NP-полной задачей, поэтому в алгоритме применяется эвристический алгоритм. Если в графе взаимодействия есть узел степени, меньшей k, то он удаляется из графа, затем оставшийся граф красится в k цветов рекурсивно, после чего выбирается цвет для удаленной вершины. Так как у удаленной вершины было менее k соседей, то для нее всегда можно выбрать подходящий цвет. Если последовательность таких удалений позволила получить пустой граф, то, возвращаясь из рекурсии, будет построена корректная раскраска графа. Иначе генерируется сброс одной из переменных в память, что разбивает одну их вершин графа на несколько, и алгоритм повторяется [9]. Существуют различные варианты алгоритма в зависимости от способа выбрать переменную для сброса в память [32—36].

1.6.4. Алгоритм линейного сканирования

В работе [37] описывается алгоритм глобального распределения регистров, который называется алгоритмом линейного сканирования. Он начинается с выделения интервалов жизни всех переменных. Под *интервалом времени жизни* подразумевается отрезок $[a, b]$, где a и b — минимальный и максимальный номера инструкций, в которых данная переменная *жива*, т. е. содержит некоторое значение, которое может быть впоследствии использовано в программе. В рассматриваемом алгоритме предполагается, что все базовые блоки программы упорядочены каким-нибудь образом, что позволяет задать сплошную нумерацию всех инструкций. Далее интервалы времени жизни просматриваются в порядке увеличения их начала. При просмотре очередного интервала сначала освобождаются все регистры, с которыми связаны переменные, интервалы жизни которых истекли. Затем для просматриваемого интервала выбирается свободный регистр. Если свободного регистра не нашлось, то выбирается регистр для сброса. В оригинальной статье [37] предлагается брать интервал с самым дальним концом.

Существуют различные модификации данного алгоритма. Так, например, в [38] описывается разновидность данного алгоритма, которая позволяет использовать «дыры» в интервалах жизни переменных, а также повторно распределять на регистры переменные, которые были сброшены в память из-за нехватки регистров ранее. Под «дырой» понимается подинтервал $[c, d]$ интервала $[a, b]$ времени жизни переменной, где переменная не содержит значения, которое будет использовано в дальнейшем. Если интервал времени жизни некоторой другой переменной полностью вписывается в «дыру», то эти две переменных могут быть распределены на один и тот же регистр без создания конфликтов.

1.7. Программы, использующие динамическую двоичную трансляцию

Существует большое множество инструментов, использующих динамическую двоичную трансляцию: Transmeta Code Morphing [39], Pin [40; 41], QEMU [42; 43], Valgrind [26; 44] и другие [45]. Последние два инструмента выделяются на общем фоне тем, что они одновременно активно используются на практике и обладают открытым исходным кодом. Для экспериментов в данной работе был выбран QEMU, поскольку он реализует наиболее общий случай динамической двоичной трансляции, когда про гостевую и основную системы не делается никаких дополнительных предположений. Последнее очень важно, так как не каждая практическая задача может быть модифицирована так, чтобы удовлетворять тем или иным требованиям системы динамической двоичной трансляции.

1.7.1. Valgrind

В работе [26] описывается Valgrind — инфраструктура для написания инструментов для динамического анализа и поиска ошибок в программах. Valgrind полностью реализует систему динамической двоичной трансляции, а инструменты работают только с его внутренним представлением. Valgrind работает на множестве различных архитектур процессоров (x86, x86_64, ARM и другие) и под управлением различных операционных систем (Linux, Darwin). При этом машинно-независимое внутреннее представление полностью скрывает особенности процессора и позволяет переносить инструменты с одной архитектуры на другую без изменений.

В качестве блока трансляции Valgrind использует суперблок [46], т. е. линейную последовательность инструкций с одним входом и несколькими выходами. Гостевой код сначала преобразуется во машинно-независимое внутреннее представление. Оно использует форму с единственным присваиванием [47; 48].

На этом этапе в правой части присваивания может стоять сколь угодно сложное выражение без побочных эффектов. Затем к нему применяются оптимизации с целью уменьшения избыточности (удаление избыточных загрузок-сохранений, продвижение констант и копий и их сворачивание, удаление мертвого кода и удаление общих подвыражений). Также на этом этапе сложные выражения разбиваются на несколько присваиваний так, чтобы в правой части каждого из них стояла только одна операция. Далее внутреннее представление попадает к инструменту, который добавляет к нему код для анализа программы и поиска в ней ошибок, после чего происходит второй проход оптимизаций, состоящий из сворачивания констант и удаления мертвого кода. После этого простые присваивания собираются обратно в сложные выражения, на которых выполняется выбор инструкций. На выходе получается последовательность инструкций. Инструкции являются машинно-зависимыми и соответствуют инструкциям целевой архитектуры, но используют виртуальные регистры. Затем происходит распределение регистров с помощью алгоритма линейного сканирования [49] и генерируется код.

Большое количество внутренних представлений и несколько проходов оптимизаций существенно замедляют процедуру трансляции, но это не являет существенным недостатком для Valgrind, который делает упор на удобство реализации и широкие возможности различных инструментов.

1.7.2. Pin

В работе [40] описывается Pin — система, аналогичная Valgrind. Pin работает на различных архитектурах процессоров (x86, x86_64, IA64, ARM и другие) и под управлением различных операционных систем (Linux, Windows).

Существенными отличиями от Valgrind с точки зрения организации процесса динамической двоичной трансляции является наличие сцепления блоков трансляции (block chaining) [45] и связанных с ним изменениях в алгоритме распределения регистров.

Pin, в отличие от большинства других систем динамической двоичной трансляции, пытается сохранять регистры гостевой архитектуры на регистрах целевой между блоками трансляции. Для этого с каждым блоком связывается начальное распределение регистров гостевой архитектуры на регистры целевой. Изначально это распределение берется равным тому, которое было в блоке, переход из которого вызвал трансляцию. Если впоследствии необходимо совершить переход на данный блок с другим распределением регистров, Pin вставляет компенсирующий код. Анализ времени жизни переменных также расширен на работу между несколькими блоками трансляции. Если на момент трансляции блока информация о времени жизни переменных на части его выходов уже известна (из информации на входах в другие блоки), то эта информация используется во время трансляции.

Сохранение переменных на регистрах между блоками трансляции может иметь как положительный эффект на общую производительность (за счет того, что удается избежать дополнительных записей в память и чтений из памяти на границах блоков трансляции), так и отрицательный (из-за того, что увеличивается количество используемых одновременно регистров и появляется дополнительный компенсационный код). К сожалению, авторы статьи [40] не приводят экспериментальных данных, демонстрирующих влияние данной конкретной оптимизации на общую производительность.

1.7.3. Shade

Еще одна схожая система динамической двоичной трансляции под названием Shade описана в работе [45]. Она работает на процессорах архитектуры SPARC и может эмулировать архитектуры SPARC и MIPS. Основным ее назначением является сбор трасс выполнения программы для их дальнейшего анализа, но гибкие возможности настройки позволяют реализовывать и другие инструменты на базе имеющейся инфраструктуры.

В своей работе Shade следует описанному в разделе 1.2 циклу работы систе-

мы динамической двоичной трансляции. В качестве блока трансляции используется линейная последовательность инструкций без операций перехода внутри. Генерируемый код никак не оптимизируется, поскольку время, затраченное на служебную работу по сбору и анализу трассы, значительно превосходит время трансляции и выполнения кода.

Распределение регистров внутри блока трансляции происходит «по требованию»: при обращении к переменной, находящейся в памяти, для нее подбирается подходящий регистр, при нехватке регистров какая-нибудь переменная записывается обратно в память. Также Shade может статически назначить некоторые переменные на регистры для всех блоков трансляции или только для блоков трансляции из определенного региона программы. В работе [45] авторы не приводят, насколько много переменных распределяется глобально и насколько это влияет на результирующую производительность.

1.7.4. Transmeta Code Morphing

В работе [39] описывается эмулятор архитектуры x86, работающий на процессорах семейства Transmeta Crusoe и использующий в своей работе динамическую двоичную трансляцию. Процессоры Transmeta Crusoe относятся к процессорам с очень длинным словом команд (VLIW). Каждая инструкция может содержать две или четыре операции, каждая из которых выполняется на одном из 5 функциональных устройств процессора. В распоряжении программы имеются 64 целочисленных регистра и 32 регистра с плавающей точкой.

Поскольку гостевая и целевая архитектуры в случае Transmeta Code Morphing фиксированы, их особенности были учтены разработчиками при проектировании эмулятора.

Большое количество целевых регистров сильно упрощает процесс их распределения. Оно позволяет назначить на них все регистры гостевого процессора, и оставшихся регистров хватит на выполнение промежуточных вычислений и работу самого эмулятора.

С другой стороны, для эффективного использования возможностей процессоров с очень длинным словом команд, таких как внутренний параллелизм и спекулятивное выполнение, требуется тщательная оптимизация программ. В результате процесс трансляции в эмуляторе Transmeta Code Morphing очень медленный, и ему подвергаются только горячие участки кода. Остальной же код просто интерпретируется.

1.7.5. Система динамической двоичной трансляции «Эльбрус»

При разработке микропроцессоров «Эльбрус» применялся метод, аналогичный тому, который был использован при разработке процессоров Transmeta Crusoe. «Эльбрус» также относится к процессорам VLIW-архитектуры, и вместе с ним также поставляется система динамической двоичной трансляции, которая позволяет выполнять на данном процессоре код, написанный для архитектуры x86 [20; 50].

Система динамической двоичной трансляции для процессоров «Эльбрус» состоит из интерпретатора и трех трансляторов. Трансляторы отличаются скоростью своей работы и тем, насколько хорошо они оптимизируют код [21].

Самый быстрый из трансляторов, шаблонный транслятор, заменяет каждую инструкцию, используя несколько готовых шаблонов. Следующим по скорости является быстрый региональный компилятор. Он оперирует над линейными участками кода и имеет полноценное промежуточное представление. Быстрый региональный компилятор выполняет ряд оптимизаций: локальное удаление избыточных вычислений (peephole), слияние узлов (if-conversion), удаление избыточных загрузок (redundant load elimination), удаление мертвого кода (dead code elimination), динамический разрыв зависимостей по доступу в память (memory access disambiguation), планирование инструкций (instruction scheduling). В самом медленном трансляторе, оптимизирующем региональном комплияторе, добавляются потимизации циклов, глобальное планирование, явная предварительная загрузка (prefetching) и другие времязатратные оптимизации.

Чем медленнее работает каждый из трансляторов, тем более сложные оптимизации он способен производить и тем более эффективный код получается на его выходе. Более медленные трансляторы применяются только к многократно выполняющимся участкам кода. Большое внимание при разработке транслятора уделялось выбору того, какой из имеющихся трансляторов нужно применять к конкретному участку кода [51].

Архитектура процессоров «Эльбрус» разрабатывалась с учетом необходимости эмулировать на них архитектуру x86. Так, например, в нем предусмотрен «скрытый» канал IDE для загрузки системы динамической двоичной трансляции. Кроме того, в системе динамической двоичной трансляции «Эльбрус» есть оптимизации, которые заточены на эмуляцию архитектуры x86. К таким оптимизациям можно отнести отображение контекста выполнения вычислений с вещественными числами гостевой архитектуры на целевую архитектуру [52].

Важность многих из перечисленных оптимизаций является следствием особенностей процессорной архитектуры «Эльбрус». «Эльбрус» обладает явно выраженным параллелизмом: процессор может обрабатывать по несколько команд каждый такт, эти команды упаковываются в длинное машинное слово. Такой подход позволяет достичь очень высокой производительности при условии, что компилятор сформировал длинные слова оптимальным образом. С другой стороны, VLIW-архитектуры не имеют аппаратного переупорядочивания команд, так что появление зависимостей по данным или управлению может приводить к длительным простоям конвейера существенно снижая производительность. Все это делает планирование инструкций необходимой оптимизацией, также, как и оптимизаций, снижающих количество зависимостей между инструкциями (слияние узлов, удаление избыточных загрузок, разрыв зависимостей по доступу в память) [53]. На архитектурах с аппаратным переупорядочиванием инструкций влияние этих оптимизаций будет существенно ниже.

1.7.6. QEMU

В работе [42] описывается QEMU — эмулятор процессоров и целых вычислительных систем. QEMU поддерживает большое количество различных процессоров и устройств. У него есть два режима работы: эмуляция системы и эмуляция приложения. В первом случае эмулируется вся вычислительная система: процессор и внешние по отношению к нему устройства. Динамической трансляции подвергаются код программы, всех задействованных библиотек и операционной системы. Во втором случае транслируется только код программы и нужных библиотек. Функции операционной системы (в частности, обработку системных вызовов) берет на себя QEMU.

Единицей двоичной трансляции в QEMU является расширенный базовый блок — ациклический участок кода с одним входом.

В QEMU присутствуют только две оптимизации: удаление мёртвого кода и подстановка констант, которая выполняется во время распределения регистров. Процесс распределения регистров в QEMU совмещен с генерацией кода: сразу после того, как для операции внутреннего представления были распределены регистры, для нее генерируется машинный код. Таким образом, изменить выбор регистра впоследствии уже невозможно, и алгоритм распределения регистров в QEMU является однопроходным. Распределение регистров в QEMU локальное, границы базовых блоков переменные пересекают в памяти.

1.8. Применение методов оптимизации в программных эмуляторах

В главе 2 будут рассмотрены оптимизации по продвижению констант и копий, а также сворачивание константных выражений. Удаление мертвого кода уже реализовано в эмуляторе. Примеров возникновения недостижимого кода, равно как и общих подвыражений, обнаружить на существующих примерах

не удалось, поэтому эти оптимизации не реализовывались. Реализованные машинно-независимые оптимизации могут быть сведены к задаче анализа потока данных, которая может быть эффективно решена на ациклическом графе потока управления. Другой подход состоит в реализации глобальной нумерации значений [54; 55], однако данные оптимизации сопряжены с существенно большими накладными расходами, что нежелательно во время динамической двоичной трансляции.

В главах 3 и 4 будут рассмотрены локальное и глобальное распределение регистров. Алгоритмы распределения регистров должны минимизировать накладные расходы, так как они будут выполняться во время работы программы. Поэтому предлагается делать их однопроходными и совмещенными с генерацией кода. Данные условие позволяет избежать дополнительных накладных расходов, связанных с построением еще одного или нескольких вариантов внутреннего представления с выбранными инструкциями и/или распределенными регистрами. Для локального распределения регистров в главе 3 предложены несколько эвристик, которые позволяют снизить количество сбросов регистров в память. В главе 4 строится новый алгоритм глобального распределения регистров, удовлетворяющий изложенным выше требованиям, так как ни один из существующих им не удовлетворяет. Данный алгоритм выбирает, какие переменные должны пересекать границы базовых блоков на регистрах, а затем производит локальное распределение регистров внутри базовых блоков в данных ограничениях.

В главе 5 будет рассмотрено улучшение выбора инструкций. Использование векторных инструкций основной архитектуры для эмуляции векторных инструкций гостевой архитектуры позволяет более эффективно использовать ресурсы основного процессора и тем самым повышает быстродействие генерируемого кода.

Вопросы, связанные с параллельным выполнением гостевого кода рассматриваться не будут, так как они уже описаны в литературе [12; 17] и существует

их реализация в эмуляторе с открытым исходным кодом [18; 19]. В данной диссертационной работе будет рассматриваться только случай, когда в системе динамической двоичной трансляции есть только один уровень оптимизации, применяемый ко всему гостевому коду. В таких ограничениях использование существующих компиляторов для оптимизации показало себя неэффективным [11], поэтому такой подход также рассматриваться не будет.

При описании разрабатываемых методов и алгоритмов будем считать, что мы работаем с некоторым машинно-независимым внутренним представлением системы динамической двоичной трансляции. В этом представлении будут переменные двух типов: глобальные, которые живы во время все работы гостевой программы, локальные, которые живы в пределах одного блока трансляции, и временные, которые живы в пределах одного базового блока. Операции внутреннего представления являются инструкциями некоторого обобщенного языка ассемблера. В своей основе они трехадресные, т. е. оперируют с двумя аргументами и результатов. Операция пересылки будет иметь два аргумента: исходный и целевой. Отдельно будет выделяться операция вызова функции, которая будет иметь переменное число аргументов.

Экспериментальная часть данной работы будет выполняться на базе эмулятора QEMU. Данный эмулятор был выбран потому, что он

- обладает открытым исходным кодом,
- реализует наиболее общий случай динамической двоичной трансляции, когда не делается никаких дополнительных предположений относительно гостевой и основной архитектур.

Глава 2

Машинно-независимые оптимизации

Машинно-независимые оптимизации направлены на упрощение вычислений и устранение избыточности. Данные преобразования не зависят от конкретных особенностей целевой процессорной архитектуры, на которой должен выполняться код. Примерами машинно-независимых оптимизаций являются удаление мертвого и недостижимого кода, продвижение констант и копий, сворачивание константных выражений, удаление общих подвыражений.

Обычно избыточность, устраняемая машинно-независимыми оптимизациями, возникает при трансляции конструкций языка высокого уровня в низкоуровневое представление. В случае динамической двоичной трансляции код гостевой программы, как правило, был хорошо оптимизирован компилятором. Тем не менее, код, получаемый из него при двоичной трансляции, может быть улучшен машинно-независимыми оптимизациями. Причиной этому могут служить несколько фактов.

- Гостевая программа может компилироваться для одной архитектуры, а выполняться после динамической трансляции на другой. Различные архитектуры могут иметь различные ограничения на аргументы и различные побочные эффекты. Таким образом, оптимизации, возможные в одной из архитектур, невозможны в другой.

- Трансляция производится с помощью замены каждой инструкции гостевой программы на последовательность инструкций внутреннего представления системы динамической двоичной трансляции.

- Сама трансляция может выполняться неэффективно и вносить избыточность.

- Часть побочных эффектов инструкции может оказаться неиспользуемой

в дальнейшем.

На стыке инструкций может возникать неэффективный код, который может быть улучшен.

Например, пересылка константы $120034h$ может быть выполнена только в две инструкции на архитектуре ARM:

```
mov r5, #0x34
orr r5, #0x120000
```

Причиной этого является особенность кодирования констант в двоичном представлении инструкций на ARM, накладывающая ограничение на возможные значения констант. На x86 таких ограничений нет, и того же результата можно добиться с помощью одной инструкции:

```
mov $0x120034, %edi
```

2.1. Описание алгоритма

Стандартным подходом к проведению машинно-независимых оптимизаций является анализ потока данных. Для этого формулируются факты потока данных и уравнения потока данных. Далее полученная задача решается итеративным алгоритмом 1 или 2 в зависимости от направления. Данным алгоритмам может потребоваться многократно просмотреть весь код программы.

В случае динамической двоичной трансляции граф потока управления является ациклическим. В этом случае можно выбрать такой порядок просмотра базовых блоков, что алгоритм решения задачи потока данных сойдется за одну итерацию. При этом никаких дополнительных ограничений на структуру задачи потока данных не накладывается. Для прямой задачи потока данных необходимо просматривать вершины графа потока управления в порядке топологической сортировки, а для обратной — в порядке, обратном порядку топологической сортировки.

Теорема 1. *Пусть $G = \langle B, E \rangle$ — ациклический граф потока управления. Пусть все блоки занумерованы в порядке топологической сортировки, т.е. $(b_i, b_j) \in E \implies i < j$. Тогда алгоритм решения прямой задачи анализа потока данных, просматривающий базовые блоки в порядке нумерации, сойдется за одну итерацию.*

Доказательство. Покажем от противного, что вторая и последующая итерации итеративного алгоритма не изменят значение потока данных. Пусть значение $OUT[b_i]$ изменилось для некоторого базового блока b_i на второй или последующей итерациях цикла. Среди всех таких i выберем минимальное i_m.

Рассмотрим уравнение потока данных, которое привело к изменению значения $OUT[b_{i_m}]$.

$$OUT[b_{i_m}] = f(IN[b_{i_m}])$$

Здесь f — передаточная функция, которая зависит от содержимого базового блока. Поскольку код внутри базового блока не менялся, значит, изменилось значение $IN[b_{i_m}]$. Рассмотрим уравнение потока данных для $IN[b_{i_m}]$.

$$IN[b_{i_m}] = \wedge_{b_j:(b_j,b_{i_m})\in E} OUT[b_j]$$

Следовательно, значение минимум одного из $OUT[b_j]$ изменилось с момента предыдущего вычисления $IN[b_{i_m}]$. Из условия упорядоченности базовых блоков следует, что $b_j < b_{i_m}$ и на предыдущей итерации $OUT[b_j]$ были вычислены до $IN[b_{i_m}]$. Значит, значение одного из $OUT[b_j]$ изменилось на текущей итерации алгоритма. Однако это противоречит условию выбора i_m. Полученное противоречие завершает доказательство теоремы. □

Теорема 2. *Пусть $G = \langle B, E \rangle$ — ациклический граф потока управления. Пусть все блоки занумерованы в порядке, обратном порядку топологической сортировки, т.е. $(b_i, b_j) \in E \implies i > j$. Тогда алгоритм решения обратной*

задачи анализа потока данных, просматривающий базовые блоки в порядке нумерации, сойдется за одну итерацию.

Доказательство. Аналогично теореме 1. □

Полученный результат может быть применен для проведения машинно-независимых оптимизаций во время динамической двоичной трансляции. В данной работе для реализации были выбраны продвижение копий и сворачивание константных выражений по следующим причинам.

- Данные оптимизации имеют простую структуру значения потока данных и, как следствие, низкие накладные расходы на их проведение. Это свойство очень важно при динамической двоичной трансляции, так как оптимизации программы производятся во время ее выполнения.
- Существуют известные примеры, где их применение улучшает генерируемый код. Один из таких примеров был рассмотрен в начале главы.

Альтернативой данным оптимизациям является метод нумерации значений, который, кроме продвижения копий и сворачивания констант, позволит удалить общие подвыражения. Однако данный метод обычно делается на базе представления с единичным присваиванием [54; 55]. Существуют способы проводить нумерацию значений в рамках одного базового блока либо в рамках суперблока без использования представления с единичным присваиванием, но они привносят существенные накладные расходы [56]. Поэтому выбор был сделан в пользу более простых, но достаточно легковесных оптимизаций.

Поток данных будет принимать значение из множества $Vars(TB) \times (\mathbb{Z} \cup Vars(TB) \cup \{\texttt{NACC}\})$, где $Vars(TB)$ — множество всех переменных блока трансляции TB, а `NACC` — специальное значение, обозначающее, что данная переменная не является ни копией другой переменной, ни константой. С точки зрения интерпретации это означает, что для каждой переменной внутреннего представ-

ления будем хранить её текущее состояние, которое может принимать одно из трёх значений:

- *является константой*,
- *является копией*,
- *не является ни копией, ни константой*.

Для переменных, являющихся константами, будем хранить значение этой константы; для переменных, являющихся копиями других переменных, будем хранить идентификатор переменной, копией которой является данная переменная.

Данный алгоритм является консервативным, так как анализ потока данных сходится к консервативному решению, то есть замены переменных на их копии или константы он производит только тогда, когда они таковыми являются. При этом некоторые возможные замены могут быть пропущены, так как алгоритм не смог определить, что данная переменная является константой или копией другой переменной. Сложность данного алгоритма — O(количество_инструкций · количество_переменных).

2.2. Реализация алгоритма

В любой заданной точке программы отношение «переменная A является копией переменной B» (обозначим его EQ) является отношением эквивалентности на множестве переменных и задаёт классы эквивалентности в этом множестве. В ходе оптимизаций в каждой точке кода в каждом таком классе выберем представителя и обеспечим, чтобы все остальные переменные данного класса считались копиями этого одного представителя. Введем следующие операции над классами переменных блока трансляции.

- Get-Class(Var) — возвращает представителя класса эквивалентности, к котором относится переменная Var.

- Add-To-Class(Var_1, Var_2) — добавить переменную Var_2 к классу эквивалентности переменной Var_1.

- Remove-From-Class(Var) — удалить переменную Var из её текущего класса эквивалентности.

Опишем сам алгоритм оптимизации (обозначим его алгоритм 3). Пусть у нас есть правильно вычисленные состояния всех переменных на момент начала выполнения операции внутреннего представления. На момент начала выполнения первой инструкции базового блока будем считать, что все переменные находятся в состоянии *не является ни копией, ни константой*, а также что состояние всех переменных на момент начала выполнения инструкции из середины базового блока совпадает с их состоянием на момент конца выполнения предыдущей инструкции. Покажем, как вычислить состояния всех переменных на момент конца выполнения данной инструкции, а также какие изменения должны быть внесены в инструкцию.

Если операция является операцией пересылки переменных «$A = B$», принадлежащих одному классу эквивалентности относительно отношения EQ, данная инструкция является излишней и может быть заменена на пустую операцию (`nop`). Обозначим это преобразование Remove-Instruction. Состояния переменных не изменяются.

Если операция является операцией пересылки переменных «$A = B$» из разных классов, то переменная A переходит из одного класса эквивалентности в другой. При этом если переменная A являлась представителем в своём классе эквивалентности, в этом классе эквивалентности выбирается новый представитель по описанным выше правилам и обновляются пометки для всех переменных из этого класса, чтобы они содержали нового представителя. Также устанавливается пометка, что переменная A является копией представителя класса эквивалентности, содержащего переменную B, а переменная B заменяется на представителя своего класса.

Если все аргументы арифметической операции «$A = B \circ C$» являются константами, то значение A вычисляется, и операция заменяется на операцию пересылки константы. Обновление состояния других переменных происходит аналогично предыдущему пункту. При вычислении значения переменной A необходимо следить за несколькими вещами:

- хотя во внутреннем представлении все константы считаются беззнаковыми, операция может трактовать их как знаковые. В этом случае необходимо приведение типов;
- 32-битные операции на 64-битных системах используют только младшие 32 бита констант. При реализации таких операций, как сдвиги, необходимо обнулять старшие биты входных констант.

Сворачивание константных выражений было реализовано для основных 32-битных арифметических операций и их 64-битных аналогов. В псевдокоде алгоритма обозначим операцию сворачивания константных выражений Fold.

Во всех остальных случаях все входные переменные заменяются на представителей их классов эквивалентности, все выходные переменные операции помечаются как *не является ни копией, ни константой*. Если какие-то из выходных переменных являлись представителями классов эквивалентности, то в этих классах выбираются новые представители, и состояния всех переменных соответствующих классов обновляются, чтобы отразить изменение представителя класса.

Рассмотрим, как производится изменение инструкций во время проведения данной оптимизации на внутреннем представлении QEMU. Заметим, что количество операций не меняется, меняются их типы. Такое изменение внутреннее представление поддерживает. Изменение количества аргументов требует дополнительных действий. Для того, чтобы обеспечить его, будем строить новый массив аргументов. Далее заметим, что в ходе данной оптимизации число аргументов текущей операции либо остаётся неизменным, либо уменьшается (вместе

Алгоритм 3 Алгоритм продвижения констант и копий и сворачивания константных выражений

```
procedure PROPAGATE-AND-FOLD(TB)
    for all I ∈ Instructions(TB) в прямом порядке do
        for all i = 1, ARG-COUNT(I) do
            REPLACE-ARG(I, i, GET-CLASS(Arg(I, i)))
        end for
        if Operation(I) = "=" then
            if GET-CLASS(Arg(I, 1)) = GET-CLASS(Arg(I, 2)) then
                REMOVE-INSTRUCTION(I, TB)
            else
                REMOVE-FROM-CLASS(Result(I))
                ADD-TO-CLASS(Arg(I, 1), Result(I))
            end if
        else if ∀i State[Arg(I, i)] = CONST then
            REMOVE-FROM-CLASS(Result(I))
            FOLD(I)
        else
            REMOVE-FROM-CLASS(Result(I))
        end if
    end for
end procedure
```

Тест	без оптимизаций (время в секундах)	с оптимизациями (время в секундах)	%
164.gzip	754.45	751.19	0.43
175.vpr	728.84	719.84	1.23
176.gcc	459.47	459.95	-0.1
181.mcf	119.4	118.99	0.35
186.crafty	705.12	712.58	-1.06
197.parser	1709.33	1687.56	1.27
252.eon	1496.51	1490.66	0.39
253.perlbmk	1173.34	1164.48	0.76
256.bzip2	665.53	659.77	0.87
300.twolf	1479.2	1467.94	0.76

Таблица 2.1. Результаты работы QEMU на целочисленных тестах из набора SPEC CPU2000.

с изменением самой операции). Так как операции просматриваются в прямом порядке, то новый массив аргументов можно строить в том же самом массиве, что и имеющиеся аргументы, затирая уже прочитанные и, следовательно, ненужные аргументы.

2.3. Результаты

Данная оптимизация была реализована и протестирована в QEMU. QEMU эмулировал архитектуру ARM в режиме приложения и запускался на процессоре Intel Xeon E5520 2.27ГГц. Изучение получающегося кода показывает, что оптимизация работает и успешно продвигает копии и константы, а также сворачивает константные выражения. Удаление мёртвого кода и подстановка констант в инструкции во время распределения регистров заканчивают дело по улучшению промежуточного кода.

Тестирование производилось на целочисленных тестах из набора SPEC CPU2000 [57]. Тесты запускались под QEMU в режиме эмуляции приложения.

Тесты 254.gap и 255.vortex не работали ожидаемым образом в этом режиме. Результаты работы остальных тестов приведены в таблице 2.1. Приведённые числа являются медианой по 5 запускам. Можно видеть небольшое увеличение производительности на всех тестах, кроме 176.gcc и 186.crafty. При этом изменения скорости работы теста 176.gcc находятся в пределах точности измерений. На тестт 186.crafty наблюдается ухудшение производительности.

Глава 3

Локальное распределение регистров

Чтобы избежать дополнительных накладных расходов на создание еще одного внутреннего представления, в котором регистры уже распределены, совместим процесс распределения регистров с генерацией кода. Сразу после того, как для операции внутреннего представления были распределены регистры, для нее генерируется машинный код. Таким образом, изменить выбор регистра впоследствии уже невозможно, а алгоритм распределения регистров получается однопроходным. В данной главе будем рассматривать локальное распределение регистров, границы базовых блоков переменные всегда будут пересекать в памяти.

3.1. Описание алгоритма

В качестве отправной точки возьмем наивный алгоритм локального распределения регистров. Данный алгоритм достаточно близко соответствует тому, который используется в QEMU версии 1.0. Наивный алгоритм распределения регистров может быть описан следующей последовательностью шагов.

1. Рассматривается очередная инструкция (код просматривается в прямом порядке, от начала к концу).

2. Если инструкция являет вызовом функции, то все глобальные переменные сохраняются в память, регистры, которые могут быть испорчены вызываемой функцией, освобождаются, аргументы располагаются на стеке и регистрах в соответствии с соглашениями о вызовах. Алгоритм переходит к шагу 6.

3. Для всех аргументов, которые еще не распределены на регистры, выбирается свободный регистр. Если свободных регистров несколько — то

берется наиболее приоритетный [1].

4. Если на шаге 3 свободных регистров не нашлось, то произвольный не используемый в данной инструкции регистр освобождается, а его содержимое сбрасывается в память. Шаг 3 повторяется.

5. Если данная инструкция является последней в базовом блоке, то все глобальные переменные, находящиеся на регистрах, сохраняются в память.

6. После того, как распределение регистров для данной инструкции завершено, для нее генерируется машинный код и записывается в выходной буфер.

7. Если переменная, содержащаяся в регистре, оказалась мертва после данной инструкции, то этот регистр помечается как свободный.

8. Алгоритм переходит к следующей инструкции и возвращается на шаг 1.

3.1.1. Применение существующих эвристик

Теперь рассмотрим возможности применения во время динамической двоичной трансляции алгоритмов, описанных в разделе 1.6.2 данной работы. Вспомним, что при динамической двоичной трансляции распределение регистров производится во время выполнения программы, поэтому время работы самого алгоритма также важно учитывать.

По соотношению времени работы и качества получаемого кода одним из лучших является алгоритм FF. Для его реализации в рамках наивного алгоритма необходимо внести изменения в четвертый шаг алгоритма. Однако наличие во внутреннем представлении вызовов функций, которые не рассматриваются в алгоритме FF, накладывает два дополнительных ограничения.

1. Вызываемая функция может читать или изменять глобальные переменные, поэтому все они должны быть сохранены из регистров в память,

2. Содержимое некоторых реальных регистров может быть испорчено вызываемой функцией в соответствии с соглашениями о вызовах.

В описанном выше наивном алгоритме эти проблемы частично решены. Корректность генерируемого кода обеспечивается шагом алгоритма. Неравноправие регистров, которое вносит вторая проблема, учитывается в приоритетах регистров при распределении на третьем шаге.

В данной главе предлагается следующее улучшение: сохранять глобальные переменные в память не перед самим вызовом функции, а сразу после их последнего использования до этого вызова. На границах базовых блоков поступать аналогично. Данные глобальные переменные обладают тем свойством, что они живы, занимают регистр, но не могут быть использованы без предварительной их записи в память. Таким образом, предлагается производить такую запись как можно раньше, освобождая регистр для других нужд. В случае, когда регистров не хватает, данная эвристика позволяет сэкономить запись в память, поскольку алгоритм выбора регистра для освобождения может выбрать другой регистр, который еще можно использовать. И в случае, когда еще есть свободные регистры, эта эвристика тоже может экономить сбросы регистров в память. Это связано с неоднородностью реальных регистров — некоторые из них сохраняют свои значения после вызовов, а некоторые нет. Таким образом, освобожденный эвристикой регистр может оказаться более «удобным» для хранения переменной, чем имеющийся свободный регистр.

В описанном выше алгоритме FF сходная ситуация, когда глобальная переменная жива, но не может быть использована без записи в память, может возникать на границах базовых блоков. Однако в той модели

- все регистры равноправны, и любой может быть использован вместо любого другого,

- такие переменные имеют наиболее высокий приоритет при освобождении регистра в случае их нехватки — ни одна переменная не может иметь

использование дальше, чем граница базового блока.

Таким образом, необходимости в подобной эвристике в алгоритме FF не было.

3.1.2. Применение специфичных эвристик

Теперь рассмотрим, какие улучшения алгоритма распределения регистров можно сделать, свойственные именно распределению регистров во время динамической двоичной трансляции.

Рассмотрим шаг 3.1.1 алгоритма распределения регистров. На данном шаге происходит очень много сбросов регистров в память в связи с:

- соблюдением соглашений о вызовах,
- возможным использованием глобальных переменных в вызываемой функции.

В общем виде, в случае вызова произвольной функции с этим ничего поделать нельзя. Однако если про вызываемую функцию имеется дополнительная информация, то количество сбросов можно уменьшить. Так, если известно, к каким глобальным переменным может обращаться функция, то можно сохранять в память только эти переменные, оставляя остальные глобальные переменные на регистрах.

В случае динамической двоичной трансляции большинство вызовов являются вызовами небольших вспомогательных функций самой системы трансляции. Это связано с тем, что гостевые вызовы функций при динамической двоичной трансляции преобразуются в переходы в другой блок трансляции, а непосредственно вызовы могут быть использованы только для обращения к функциям самой системы трансляции. Типичное применение подобных функций — реализация инструкций гостевой системы, которые по каким-то причинам плохо выражаются на внутреннем представлении транслятора. Определить, какая

именно функция вызывается, можно по ее адресу. Если собрать дополнительную информацию про использование этими функциями глобальных переменных, то эту информацию можно использовать в дальнейшем при распределении регистров.

Также необходимо рассмотреть возможность возникновения исключения в вызываемой функции. Оно приведет к досрочному выходу как из функции, так и из всего блока трансляции. Таким образом, перед вызовом функции, которая может генерировать исключения, необходимо сохранять все глобальные переменные. Это ограничение делает данную оптимизацию неэффективной при эмуляции архитектуры процессоров `x86`, поскольку там почти все инструкции, а, следовательно, и эмулирующие их функции, могут приводить к исключениям. Зато для архитектуры `ARM` существует и активно используется большое количество вспомогательных функций, которые вообще не используют глобальные переменные.

3.2. Реализация алгоритма

В ходе данной работы в алгоритме распределения регистров в QEMU были реализованы описанные выше эвристики, а также внесены изменения в анализ времени жизни переменных и алгоритм продвижения констант для поддержки данных изменений.

Анализ времени жизни переменных, который существует в QEMU для удаления мертвого кода, был модифицирован так, чтобы для каждого операнда каждой операции внутреннего представления сохранять, где данный операнд в следующий раз используется. Для этого в нем при проходе от конца кода к началу поддерживается массив, в котором для каждой переменной внутреннего представления записано, в какой последней встреченной операции она использовалась.

При распределении регистров для каждого регистра в каждый момент вре-

мени поддерживается информация, в какой инструкции внутреннего представления будет следующее использование содержимого данного регистра. Данная информация используется в следующих правилах:

- Если в результате нехватки регистров какой-то регистр должен быть освобожден, то выбирается тот регистр, ближайшее использование содержимого которого произойдет как можно позже.

- Если в некоторый момент времени регистр содержит глобальную переменную, и ближайшее использование этой переменной находится в другом базовом блоке или после вызова функции, то переменная сбрасывается в память, а регистр освобождается. Данные переменные все равно будут сброшены в память до их использования, поэтому лучше это сделать как можно раньше, чтобы освободить соответствующий регистр.

- Если регистр может быть испорчен в результате вызова функции, а ближайшее использование его содержимого произойдет после вызова функции, то содержимое данного регистра сбрасывается в память, а регистр освобождается.

Последние два условия обозначим предикатом Spill-Before-Use. Также введем предикаты Is-BB-End, который проверяет, является ли данная инструкция последней в базовом блоке, и Is-Call, который проверяет является ли данная инструкция вызовом функции. С помощью этих предикатов можно записать псевдокод разработанного алгоритма 4. Данный псевдокод также использует операции Spill-Var, которая сохраняет переменную в память, освобождая занятый ей регистр, и Allocate-Reg, которая выбирается регистр для очередного аргумента инструкции.

Кроме того, алгоритму 4 требуется определять, какие глобальные переменные могут быть необходимы функции для ее работы (операция Referenced-Vars). Для этого алгоритм продвижения констант, описанный в главе 2, был

модифицирован, чтобы вычислять адрес вызываемой функции для каждого вызова, если он может быть определен однозначно. Описание вспомогательных функций QEMU было изменено так, чтобы включать информацию про используемые в функции глобальные переменные. Такое описание было сгенерировано для наиболее часто используемых при эмуляции архитектуры `ARM` функций.

Шаг 2 алгоритма распределения регистров был модифицирован так, чтобы брать во внимание нужную информацию. Если удалось однозначно определить вызываемую вспомогательную функцию, и про эту функцию известно, какие глобальные переменные она использует, то только используемые переменные сохраняются в память. Иначе все глобальные переменные сохраняются в память. Последнее правило гарантирует корректность преобразования в случае, когда вспомогательная функция не может быть определена однозначно, а также работоспособность оптимизации в случае неполного описания.

3.3. Результаты

Целью алгоритмов распределения регистров обычно является получение как можно более быстро работающего кода. Однако время работы фрагмента программы зависит от многих факторов, и непосредственно измерять его бывает тяжело. Поэтому эффективность алгоритмов распределения регистров часто оценивают по количеству сгенерированных в результате чтений из памяти и записей в память [29].

Для оценки эффективности произведенных изменений в QEMU было реализовано профилирование генерируемых сбросов в память. Для каждого сброса профилировщик определяет причину: нехватка регистров, затирание регистров вызываемой функцией, сброс глобальных переменных перед вызовом функции, сброс глобальных переменных перед концом базового блока.

В таблицах 3.1 и 3.2 приведены результаты тестирования. Тестирование проводилось на архитектуре x86, в качестве гостевой архитектуры также ис-

Алгоритм 4 Улучшенный алгоритм локального распределения регистров

```
procedure LOCAL-REG-ALLOC(TB)
    for all I ∈ Instructions(TB) в прямом порядке do
        if IS-BB-END(I) then
            for all v ∈ Variables(TB) do
                SPILL-VAR(v)
            end for
        end if
        if IS-CALL(I) then
            for all v ∈ REFERENCED-VARS(I) do
                SPILL-VAR(v)
            end for
        end if
        for all i = 1,ARG-COUNT(I) do
            ALLOCATE-REG(Arg(I, i))
        end for
        for all v ∈ Variables(TB) do
            if SPILL-BEFORE-USE(TB, v, I) then
                SPILL-VAR(v)
            end if
        end for
    end for
end procedure
```

	Нехватка регистров	Конец базового блока	Вызов функции: глобальные переменные	Вызов функции: затираемые регистры	Всего
date	647 (4.5%)	5480 (38.2%)	7640 (53.2%)	582 (4.1%)	14349
bzip2	1469 (7.9%)	6042 (32.5%)	10055 (54.0%)	1041 (5.6%)	18607
linpackc	514 (4.4%)	4455 (37.7%)	6360 (53.8%)	487 (4.1%)	11816
Linux boot	11765 (4.2%)	105556 (37.6%)	149284 (53.1%)	14390 (5.1%)	280995

Таблица 3.1. Результаты профилирования причин сброса регистров существующего алгоритма распределения регистров в QEMU

пользовалась x86. QEMU в режиме динамической двоичной трансляции никак не выделяет случай совпадения гостевой и основной архитектур и всегда выполняет трансляцию общего вида. Таким образом, факт совпадения архитектур никак не влияет на результаты тестирования алгоритма распределения регистров. Первые три теста запускались в режиме эмуляции приложения, последний тест представляет собой загрузку операционной системы GNU/Linux в режиме эмуляции системы.

В первых трех тестах видно существенное снижение сбросов регистров в память из-за нехватки регистров и из-за затирания регистров при вызовах функции (в соответствии с соглашением о вызовах). С другой стороны, количество сбросов глобальных переменных на границах базовых блоков и на вызовах функций увеличивается. Это связано с тем, что в новом алгоритме такие ситуации обнаруживаются как можно раньше, и происходит сброс содержимого регистров в память. В старом алгоритме глобальные переменные сохранятся на регистрах до самого конца базового блока или вызова функции. Однако до это-

	Нехватка регистров	Конец базового блока	Вызов функции: глобальные переменные	Вызов функции: затираемые регистры	Всего
date	102 (0.7%)	5535 (39.0%)	8349 (58.8%)	224 (1.6%)	14210
bzip2	317 (1.7%)	6188 (33.6%)	11588 (62.9%)	324 (1.8%)	18417
linpackc	52 (0.4%)	4491 (38.4%)	6978 (59.6%)	179 (1.5%)	11700
Linux boot	1139 (0.4%)	106543 (39.0%)	160513 (58.8%)	4730 (1.7%)	272925

Таблица 3.2. Результаты профилирования причин сброса регистров измененного алгоритма распределения регистров в QEMU

го момента они могут быть сброшены по причине нехватки регистров. Тем не менее, общее количество сбросов регистров в трех первых тестах уменьшилось на 1%.

Четвертый тест содержит достаточно много недетерминизма, так как в нем даже не был формально определен момент, когда считать загрузку завершенной. Сравнивать работу двух алгоритмов по числу сгенерированных сбросов в память на нем некорректно. Он приведен для того, чтобы продемонстрировать, что распределение причин сброса регистров для режима эмуляции системы остается таким же, равно как и влияние изменений в алгоритме распределения регистров.

Из полученных результатов видно, что почти 60% общего числа сгенерированных сбросов составляют сбросы глобальных переменных перед вызовами функций, более 33% со сбросом глобальных переменных на границах базовых блоков. На долю остальные причины приходится не более 3.5%. Таким образом, дальнейшие улучшения алгоритма, направленные на более удачный выбор ре-

гистров при распределении, не принесут заметных результатов. Для улучшения алгоритма распределения регистров необходимо либо ослаблять ограничения, связанные со сбросом глобальных переменных перед вызовами функций, либо переходить к алгоритму глобального распределения регистров. При этом первое может дать гораздо более сильный эффект, чем второе, но только в том случае, если сбросы этих глобальных переменных в память были несущественными, т. е. вызываемая функция на самом деле не читала их значение. Однако эксперименты со вспомогательными функциями для архитектуры `ARM` показывают, что это не так, и большое количество сбросов являются существенными. Поэтому в качестве следующего шага улучшения алгоритмы был сделан переход к глобальному распределению регистров.

Глава 4

Глобальное распределение регистров

В данной главе рассматривается задача глобального распределения регистров. В отличие от локального, глобальное распределение регистров производится над участком кода, содержащим несколько базовых блоков. Для компиляторов таким участком кода будет являться вся процедура целиком, а для динамической двоичной трансляции — блок трансляции.

4.1. Описание комбинированного алгоритма

Теперь рассмотрим возможности применения существующих алгоритмов во время динамической двоичной трансляции, и на основе проведенного анализа построим новый алгоритм.

Начнем с алгоритма раскраски графа. Этот алгоритм отличается высоким качеством получаемого распределения регистров, но и высокими накладными расходами. Ему требуется много времени на работу и большое количество дополнительных структур данных, что делает его применение нежелательным в рамках динамической двоичной трансляции.

Алгоритм линейного сканирования, напротив, имеет низкие накладные расходы и решает (неоптимально) задачу глобального распределения регистров за линейное время. Однако оригинальный алгоритм предполагает очень жесткое ограничение на распределение регистров: любая переменная за время своей жизни может находиться только на одном регистре, причем только в течение непрерывного интервала времени. Его модифицированная версия — алгоритм упаковки со вторым шансом [38] — не имеет такого ограничения, однако она требует второго прохода с возможностью переписывать выбор регистров для того, чтобы разрешать противоречия в точках объединения нескольких путей в графе потока управления. Действительно, в случае, если у базового блока

есть несколько предшественников в графе потока управления, необходимо обеспечить, чтобы при входе в этот блок расположение переменных на регистрах было одинаковым независимо от того, из какого предшественника в него попало управление. Второй проход требует наличия промежуточного представления с уже распределенными регистрами. В случае его отсутствия добавление такого представления также повлечет за собой дополнительные накладные расходы.

Прежде чем строить новый алгоритм, отметим одну особенность промежуточного представления, получаемого при динамической двоичной трансляции. При трансляции гостевого кода во внутреннее представление каждая инструкция переводится в последовательность команд внутреннего представления. Все входные аргументы данная последовательность получает на глобальных переменных (соответствующих регистрам гостевой архитектуры) либо в памяти. Выходные данные располагаются там же. Все же остальные переменные живы только внутри таких последовательностей и используются для хранения результатов промежуточных вычислений. Таким образом, количество переменных, интервалы жизни которых пересекают границы базовых блоков, существенно меньше общего числа переменных.

Алгоритм, полученный в главе 3, может быстро и эффективно распределять регистры в рамках одного базового блока. Не составляет труда немного модифицировать его так, чтобы он принимал во внимание условия на границах базового блока: какие глобальные переменные должны находиться на каких регистрах. Значит, можно построить алгоритм, являющийся комбинацией глобального распределения регистров и локального. Часть, отвечающая за глобальное распределение регистров, будет работать только с глобальными переменными и устанавливать условия на границах базовых блоков. Часть, отвечающая за локальное распределение регистров, будет осуществлять распределение регистров внутри базовых блоков с учетом условий на их границах.

4.1.1. Схема комбинированного алгоритма

Определение 1. Назовем условия на распределение регистров в начале базового начальными, а в конце — конечными. Введем следующие обозначения.

- Если есть базовый блок b, то начальные условия этого базового блока обозначим как b^{pre}, а конечные — как b^{post}.
- Все множество начальных условий для всех базовых блоков блока трансляции TB с графом потока управления $G = \langle B, E \rangle$ обозначим как C^{pre}, конечных — C^{post}, а всех граничных условий — C.

$$C^{pre} = \{b^{pre} : b \in B\} \tag{4.1}$$

$$C^{post} = \left\{b^{post} : b \in B\right\} \tag{4.2}$$

$$C = C^{pre} \cup C^{post} \tag{4.3}$$

Определение 2. Множество граничных условий C блока трансляции TB с графом потока управления $G = \langle B, E \rangle$ назовем корректным, если

$$\forall (b_1, b_2) \in E : b_1^{post} = b_2^{pre}.$$

Определение 3. Дуги e_1 и e_2 графа потока управления назовем родственными, если они выходят из одного и того же блока либо входят в один и тот же блок. Обозначим $e_1 \sim e_2$.

Определение 4. Точкой синхронизации назовем непустое множество J дуг графа потока управления такое, что

- для любых двух родственных дуг e_1 и e_2 выполнено соотношение

$$e_1 \in J \iff e_2 \in J,$$

- для любых двух дуг u и v графа потока управления, входящих в одну точку синхронизации, существует последовательность дуг $e_1, e_2, \ldots, e_k$ таких, что $e_1 = u$, $e_k = v$, $\forall i \in [1, k-1]$ $e_i \sim e_{i+1}$.

Теорема 3. *Пусть дуги $e_1 = (u_1, v_1)$ и $e_2 = (u_2, v_2)$ графа потока управления принадлежат одной и той же точке синхронизации J. Тогда для любого корректного множества граничных условий C выполнены равенства $u_1^{post} = u_2^{post}$ и $v_1^{pre} = v_2^{pre}$.*

Доказательство. Для доказательства данного утверждения достаточно доказать равенство $u_1^{post} = u_2^{post}$. Верность второго равенства будет следовать из определения корректности множества граничных условий. Докажем первое равенство от противного.

Предположим, что $u_1^{post} \neq u_2^{post}$. Так как $e_1 \in J$ и $e_2 \in J$, по определению точки синхронизации существует последовательность дуг $f_1 = (x_1, y_1), f_2 = (x_2, y_2), \ldots, f_k = (x_k, y_k)$ таких, что $f_1 = e_1$, $f_k = e_2$ и $\forall i \in [1, k-1]$ $f_i \sim f_{i+1}$. $x_1^{post} \neq x_k^{post}$, значит, $\exists j \in [1, k-1] : x_j^{post} \neq x_{j+1}^{post}$, что противоречит определению корректности множества граничных условий C, так как $f_j \sim f_{j+1}$. □

Данное утверждение, являющееся необходимым условием корректности множества граничных условий, послужит основой для комбинированного алгоритма: необходимо для каждой точки синхронизации выбрать граничные условия, которые будут записаны на обоих концах дуг, входящих в эту точку синхронизации.

Пусть дан блок трансляции TB с графом потока управления $G = \langle B, E \rangle$. Построим неориентированный граф $G_E = \langle E, F \rangle$, в котором $(e_1, e_2) \in F$ тогда и только тогда, когда $e_1 \sim e_2$.

Теорема 4. *Множество дуг $\{e_i\}$ графа потока управления $G = \langle B, E \rangle$ является точкой синхронизации тогда и только тогда, когда они образуют компоненту связности в графе G_E.*

Доказательство. Пусть множество $\{e_i\}$ является точкой синхронизации J. Покажем, что это множество является компонентой связности в графе G_E. Компоненту связности графа G_E, которой принадлежит дуга e_1, обозначим K.

$\forall e_i \in J$ существует последовательность $f_1, f_2, \ldots, f_k$ такая, что $e_1 = f_1$, $e_i = f_k$, $\forall j \in [1, k-1]$ $f_j \sim f_{j+1}$. Последнее значит, что $\forall j \in [1, k-1]$ f_j и f_{j+1}) являются смежными в графе G_E, а последовательность $f_1, \ldots, f_k$ задает путь в этом графе. Таким образом, e_1 и e_i принадлежат одной и той же компоненте связности. Значит, $J \subseteq K$.

$\forall u \in K$ существует путь $f_1, f_2, \ldots, f_k$ такой, что $f_1 = e_1$, $f_k = u$. Так как $\forall j \in [1, k-1]$ f_j и f_{j+1} являются смежными в графе G_E, то $f_j \sim f_{j+1}$ (в графе потока управления G). То, что $u = f_k$ принадлежит J, можно доказать по индукции:

- (база) $f_1 = e_1 \in J$,
- (переход) $f_j \in J$ и $f_j \sim f_{j+1}$, значит, $f_{j+1} \in J$.

Получаем, что $K \subseteq J$. Значит, $J = K$.

Теперь докажем утверждение в обратную сторону. Пусть $\{e_i\}$ образуют компоненту связности K в графе G_E. Покажем, что данное множество является точкой синхронизации.

Для любых двух родственных дуг u и v графа потока управления G соответствующие им вершины графа G_E будут смежными. Значит, они либо обе входят в K, либо ни одна из них туда не входит. То есть $u \in J \iff v \in J$.

$\forall u, v \in K$ существует путь $f_1, f_2, \ldots, f_k$ такой, что $f_1 = e_1$, $f_k = u$. Так как $\forall j \in [1, k-1]$ f_j и f_{j+1} являются смежными в графе G_E, $f_j \sim f_{j+1}$.

Таким образом, по определению точки синхронизации множество K является точкой синхронизации. □

У доказанного выше утверждения есть ряд важных следствий.

1. Любые две точки синхронизации либо совпадают, либо не пересекаются.
2. Каждая дуга принадлежит ровно одной точке синхронизации.

3. Эффективно находить и обходить точки синхронизации можно с помощью поиска в ширину либо поиска в глубину в графе G_E.

Далее приведем псевдокод алгоритма и докажем его корректность. Все множество точек синхронизации блока трансляции TB обозначим $\mathfrak{J}(TB)$, множество всех базовых блоков, входящих в его поток управления,— $B(TB)$.

Алгоритм 5 Комбинированный алгоритм

```
procedure COMBINED-REG-ALLOC(TB)
    for all b ∈ B(TB) do                          ▷ Начальная инициализация
        b^pre ← ∅
        b^post ← ∅
    end for
    for all J ∈ 𝔍(TB) do                          ▷ Основной цикл
        regmap ← COMPUTE-REGISTER-MAPPING(J)
        for all e ∈ J do
            src(e)^post ← regmap
            dst(e)^pre ← regmap
        end for
    end for
end procedure
```

Теорема 5. *Множество граничных условий, полученное в ходе работы алгоритма 5, корректно.*

Доказательство. Предположим обратное. Пусть существует дуга e такая, что $src(e)^{post} \neq dst(e)^{pre}$. Обозначим точку синхронизации, которой принадлежит дуга e, как J. Граничные условия $src(e)^{post}$ и $dst(e)^{pre}$ были установлены как минимум один раз в основном цикле алгоритма (во время обработки точки синхронизации J).

Пусть итоговые конечные условия $src(e)^{post}$ были установлены в ходе обработки точки синхронизации J_1, а начальные $dst(e)^{pre}$ — в ходе обработки точки синхронизации J_2. Тогда существует дуга $e_1 \in J_1$ такая, что $src(e_1) = src(e)$. Но это значит, что $e \sim e_1$. Следовательно, $e \in J_1$, а $J \cap J_1 \neq \varnothing$. Получаем, что $J = J_1$. Аналогично, $J = J_2$. Значит, $J_1 = J_2$.

Заметим, что все граничные условия, установленные в ходе обработки одной точки синхронизации в основном цикле алгоритма, одинаковы. Тогда из неравенства $src(e)^{post} \neq dst(e)^{pre}$ следует, что $J_1 \neq J_2$. Однако выше было доказано, что $J_1 = J_2$. Полученное противоречие завершает доказательство теоремы. □

4.1.2. Выбор количества регистров для использования под граничные условия

Для выбора граничных условий в точке синхронизации сначала необходимо ответить на вопрос, сколько регистров можно занять под граничные переменные.

Предположим, что нам известно точное количество регистров, которые необходимы для генерации кода для базового блока b — обозначим его Registers-Needed(b). Общее количество регистров целевой архитектуры обозначим Total-Registers. Тогда, если выполняется условие

$$\left|b^{post}\right| + |b^{pre}| + \textsc{Registers-Needed}(b) \leqslant \textsc{Total-Registers}, \tag{4.4}$$

то множества регистров $Regs(b^{pre})$, $Regs(b^{post})$ и множество регистров, используемых внутри базового блока, можно выбрать непересекающимися. Иными словами, сокращение числа свободных регистров из-за их использования в пред- и постусловиях не ухудшит код, сгенерированный для базового блока.

Теперь немного ослабим ограничение (4.4) за счет того, что возьмем множества $Regs(b^{pre})$ и $Regs(b^{post})$ пересекающимися. Для этого опишем, как эффективно из распределения b^{pre} получить распределение b^{post}. Далее, если вы-

полнено условие

$$\left|b^{post}\right| + \textsc{Registers-Needed}(b) \leqslant \textsc{Total-Registers}, \tag{4.5}$$

то можно вставить соответствующий код в начало базового блока b, а затем выбрать множества регистров $Regs(b^{post})$ и используемых при генерации кода блока b непересекающимися. Аналогично можно поступить в случае, когда выполнено условие

$$|b^{pre}| + \textsc{Registers-Needed}(b) \leqslant \textsc{Total-Registers}. \tag{4.6}$$

Определение 5. Задачей переупорядочивания регистров назовем задачу генерации кода для пустого базового блока b с граничными условиями b^{pre} и b^{post}. Алгоритм, который решает эту задачу, назовем алгоритмом переупорядочивания регистров.

Теорема 6. *Существует алгоритм переупорядочивания регистров, генерирующий код, который использует только регистры из множества* $R \supseteq Regs(b^{pre}) \cup Regs(b^{post})$.

Доказательство. Проведем доказательство конструктивно, то есть опишем, как получить код, удовлетворяющий указанным условиям. Для этого будем использовать три операции:

- $\textsc{Spill}(v, r)$ — сохранить содержимое регистра r в переменную v,
- $\textsc{Load}(v, r)$ — загрузить значение переменной v в регистр r,
- $\textsc{Move}(r_1, r_2)$ — скопировать содержимое регистра r_2 в регистр r_1.

Алгоритм будет выполняться в два этапа. На первом этапе все регистры освобождаются с помощью операции Spill. На втором нужные переменные загружаются в нужные регистры. □

Алгоритм 6 Наивный алгоритм переупорядочивания регистров

procedure REG-REORDER(b^{pre}, b^{post})
  for all $(v, r) \in b^{pre}$ **do**
    SPILL(v, r)
  end for
  for all $(v, r) \in b^{post}$ **do**
    LOAD(v, r)
  end for
end procedure

Приведенный алгоритм достаточен для доказательства утверждения, однако он генерирует избыточное количество сохранений в память и чтений из нее. Далее приведем более эффективный алгоритм генерации кода для такого базового блока, поскольку он будет использован в дальнейшем как часть комбинированного алгоритма распределения регистров.

Обозначим текущее распределение регистров b^{cur}. Изначально оно совпадает с b^{pre}. Построим ориентированный граф G_r, вершинами которого будут являться регистры целевой архитектуры. Дуга (r_1, r_2) присутствует в графе тогда и только тогда, когда существует переменная v такая, что $(v, r_1) \in b^{cur}$, $(v, r_2) \in b^{post}$ и $r_1 \neq r_2$. То есть содержимое регистра r_1 необходимо переместить в регистр r_2.

По определению граничных условий в графе G_r в каждую вершину входит не более одной дуги и выходит также не более одной дуги. Значит, граф представляет собой совокупность цепочек и циклов.

Рассмотрим, как различные операции изменяют граф.

- Операция SPILL может быть применена только к регистру, в котором хранится некоторая переменная. При этом дуга, выходящая из соответствующей вершины, исчезает, если она была.

- Операция LOAD может быть применена только к регистру, в котором не

хранится никакая переменная. При этом появится дуга, выходящая из соответствующей вершины. Случай загрузки переменной, не входящей в множество $Var(b^{post})$, рассматриваться не будет.

- Операция MOVE может быть применена только к паре регистров (r_1, r_2) таких, что в r_1 не хранится никакая переменная, а в r_2 хранится некоторая переменная. При этом если из вершины r_2 выходила дуга $e = (r_2, r)$, то она исчезнет, а вместо нее добавится дуга $e' = (r_1, r)$.

Перейдем к описанию алгоритма. Он будет состоять из нескольких шагов.

1. Все регистры, содержащие переменные, не входящие во множество $Vars(b^{post})$, освобождаются с помощью операций SPILL. После этого шага все регистры, из которых в графе G_r не выходит дуги, являются свободными.

2. До тех пор, пока существует пара регистров (r_1, r_2) такая, что в G_r есть дуга из r_2 в r_1 и нет дуги, исходящей из r_1, к ним применяется операция MOVE(r_1, r_2). В результате этой операции исчезает дуга (r_2, r_1). После завершения данного шага в графе G_r не останется цепочек.

3. До тех пор, пока в графе G_r существует цикл, он разрывается, а шаг 2 повторяется. Разорвать цикл можно двумя способами:

   - переместив с помощью операции MOVE содержимое одного из регистров, входящих в цикл, в свободный;
   - сбросив содержимое одного из регистров, входящих в цикл, в память с помощью операции SPILL.

   Второй способ имеет смысл применять только в том случае, если свободного регистра не нашлось (то есть на регистрах из множества R хранятся $|R|$ различных переменных). Заметим, что сбрасывать переменную на этом шаге алгоритма потребуется не более одного раза, поскольку после

этого на регистрах останется $|R| - 1$ переменная, и свободный регистр всегда найдется.

4. После завершения предыдущего шага в графе G_r не осталось ни одной дуги. Осталось загрузить на регистры недостающие переменные (то есть переменные из множества $Vars(b^{post}) \setminus Vars(b^{cur})$). Все нужные регистры уже свободны, так как в графе G_r к этому моменту нет ни одной дуги.

Ни один из шагов алгоритма не увеличивает в графе G_r количество дуг. Каждая итерация шага 2 уменьшает количество дуг на 1. После каждой итерации шага 3 выполняется хотя бы одна итерация шага 2. Значит, алгоритм конечен.

Приведем псевдокод полученного в ходе доказательства теоремы алгоритма. Будем считать, что граф G_r уже построен и что операции Spill, Load и Move корректно его обновляют. Операция Break-Cycle разрывает цикл в графе одним из приведенных в описании шага 3 способов.

Посчитаем, сколько операций загрузки (L), сохранения (S) и пересылки (M) регистров генерирует данный алгоритм. Для этого сначала определим, в каком случае на шаге 3 алгоритма приходится прибегать к сбросу содержимого регистра в память. В момент выполнения шага 3 на регистрах могут находиться только переменные из множества $Vars(b^{pre}) \cap Vars(b^{post})$. Значит,

$$|Vars(b^{pre}) \cap Vars(b^{post})| \geqslant |R| \geqslant |Regs(b^{pre}) \cup Regs(b^{post})|.$$

С другой стороны,

$$|Vars(b^{pre}) \cap Vars(b^{post})| \leqslant |Vars(b^{pre})| = |Regs(b^{pre})| \leqslant |Regs(b^{pre}) \cup Regs(b^{post})|,$$

$$|Vars(b^{pre}) \cap Vars(b^{post})| \leqslant |Vars(b^{post})| = |Regs(b^{post})| \leqslant |Regs(b^{pre}) \cup Regs(b^{post})|.$$

Такое возможно, только если во всех нестрогих неравенствах достигается равенство. То есть

$$|Vars(b^{pre}) \cap Vars(b^{post})| = |Vars(b^{pre})| = |Vars(b^{post})| \Rightarrow$$

Алгоритм 7 Алгоритм переупорядочивания регистров

procedure REG-REORDER(b^{pre}, b^{post})
  for all $(v, r) \in b^{pre} : v \in Vars(b^{pre}) \setminus Vars(b^{post})$ **do** ▷ Шаг 1
    SPILL(v, r)
  end for
  while $\exists (u, v) \in Edges(G_r) : u \neq v$ **do**
    ▷ Пока в графе G_r есть цепь или цикл
    for all $(r_2, r_1) \in Edges(G_r) : \nexists r_3 : (r_1, r_3) \in Edges(G_r)$ **do** ▷ Шаг 2
      MOVE(r_1, r_2)
    end for
    if $\exists c \in Cycles(G_r)$ **then** ▷ Шаг 3
      BREAK-CYCLE(c)
    end if
  end while
  for all $(v, r) \in b^{post} : v \in Vars(b^{post}) \setminus Vars(b^{pre})$ **do** ▷ Шаг 4
    LOAD(v, r)
  end for
end procedure

$$Vars(b^{pre}) = Vars(b^{post}),$$

$$|Regs(b^{pre}) \cup Regs(b^{post})| = |R| = |Regs(b^{pre})| = |Regs(b^{post})| \Rightarrow$$

$$Regs(b^{pre}) = Regs(b^{post}) = R.$$

Возможны два случая.

- Если $b^{pre} = b^{post}$, то алгоритм 7 на шаге 3 не будет генерировать дополнительных сбросов в память.
- Если $b^{pre} \neq b^{post}$, то граф G_r будет представлять собой совокупность циклов, и алгоритм 7 на шаге 3 сгенерирует один дополнительный сброс в память.

Таким образом, сброс содержимого регистра в память на шаге 3 алгоритма 7 происходит тогда и только тогда, когда

$$b^{pre} \neq b^{post} \wedge Vars(b^{pre}) = Vars(b^{post}) \wedge Regs(b^{pre}) = Regs(b^{post}) = R. \quad (4.7)$$

Вернемся к вычислению величин L, S и M.

- На первом шаге алгоритма произойдет $|Vars(b^{pre}) \setminus Vars(b^{post})|$ операций Spill.
- На втором шаге алгоритма произойдет $|Edges(G_r)| - 1$ операций Move в случае выполнения условия (4.7) либо $|Edges(G_r)|$ в противном случае.
- На третьем шаге алгоритма произойдет $|Cycles(G_r)| - 1$ операций Move и одна операция Spill в случае выполнения условия (4.7) либо $|Cycles(G_r)|$ операций Move в противном случае.
- На третьем шаге алгоритма произойдет $|Vars(b^{post}) \setminus Vars(b^{pre})| + 1$ операций Load в случае выполнения условия (4.7) либо $|Vars(b^{post}) \setminus Vars(b^{pre})|$ в противном случае.

Значит, если выполнено условие (4.7), то

$$L = |Vars(b^{post}) \setminus Vars(b^{pre})| + 1 = 1,$$

$$S = |Vars(b^{pre}) \setminus Vars(b^{post})| + 1 = 1,$$

$$M = |Edges(G_r)| + |Cycles(G_r)| - 2.$$

Иначе

$$L = |Vars(b^{post}) \setminus Vars(b^{pre})|,$$

$$S = |Vars(b^{pre}) \setminus Vars(b^{post})|,$$

$$M = |Edges(G_r)| + |Cycles(G_r)|.$$

Определение 6. Пусть есть два алгоритма переупорядочивания регистров. Алгоритм A_1 генерирует L_1 операций Load, S_1 операций Spill и M_1 операций Move. Алгоритм A_2 генерирует L_2 операций Load, S_2 операций Spill и M_2 операций Move. Алгоритм A_1 эффективнее алгоритма A_2 тогда и только тогда, когда $L_1 + S_1 < L_2 + S_2$ либо $L_1 + S_1 = L_2 + S_2$ и $M_1 < M_2$.

Определение 7. Алгоритм переупорядочивания регистров является оптимальным, если не существует другого алгоритма переупорядочивания A', эффективнее данного.

Теорема 7. *Алгоритм 7 является оптимальным среди алгоритмов переупорядочивания регистров, использующих только регистры из множества* $R \supseteq Regs(b^{pre}) \cup Regs(b^{post})$.

Доказательство. От противного. Пусть существует алгоритм A', который эффективнее описанного. Алгоритм 7 генерирует L операций Load, S операций Spill и M операций Move. Алгоритм A' генерирует L' операций Load, S' операций Spill и M' операций Move. Возможны два случая: если условие (4.7) выполняется и если оно не выполняется.

Предположим, что условие (4.7) не выполняется. Поскольку переменные из множества $Vars(b^{pre}) \setminus Vars(b^{post})$ должны быть сохранены, то

$$S' \geqslant |Vars(b^{pre}) \setminus Vars(b^{post})| = S$$

Аналогично,

$$L' \geqslant |Vars(b^{post}) \setminus Vars(b^{pre})| = L$$

Поскольку $L' + S' \leqslant L + S$, то во всех неравенствах достигается равенство. Значит, в алгоритме A' никаких сохранений, кроме сохранений переменных из множества $Vars(b^{pre}) \setminus Vars(b^{post})$, нет. Данные сохранения не меняют граф G_r. Значит, количество дуг и циклов в графе G_r может уменьшаться только за счет операций Move. Так как целевой регистр операции Move должен быть свободным, каждая операция может либо уменьшать количество циклов в графе на 1, либо уменьшать количество ребер в графе на 1. Значит,

$$M' \geqslant |Edges(G_r)| + |Cycles(G_r)| = M.$$

Это противоречит тому, что алгоритм A' эффективнее алгоритма 7.

Осталось рассмотреть второй случай. Пусть условие (4.7) выполняется.

Поскольку все регистры из множества R в начале работы алгоритма заняты, первой инструкцией сгенерированного кода может быть только операция Spill, примененная к одной из переменных из множества $Vars(b^{post})$. Значит, $S' \geqslant 1$. Однако эта переменная в конце должна располагаться на регистре. Значит, она будет загружена с помощью Load, то есть $L' \geqslant 1$. Так как $L' + S' \leqslant L + S = 2$, то $L' = 1$, $S' = 1$ и $L' + S' = L + S$.

Каждая операция Spill может уменьшить количество циклов графа G_r не более чем на 1. Аналогично она может уменьшить количество дуг не более чем на 1. Тогда аналогично предыдущему случаю

$$M' \geqslant |Edges(G_r)| - 1 + |Cycles(G_r)| - 1 = |Edges(G_r)| + |Cycles(G_r)| - 2 = M.$$

Это противоречит тому, что алгоритм A' эффективнее алгоритма 7. □

Величина Registers-Needed(b), используемая в условиях (4.5) и (4.6), априори не известна и не может быть легко вычислена. Оценим ее приближенно. Для этого введем понятие регистрового давления.

Определение 8. Регистровым давлением в инструкции I из базового блока b называется минимальное количество регистров, необходимых для генерации кода данной инструкции в предположении, что все переменные, которые живы в данной точке базового блока и используются в нем в инструкции I или после нее, располагаются на регистрах.

$$\text{Reg-Pressure}(I, b) = |\text{Live-Variables}(I, b)| + |\text{Extra-Registers}(I)|$$

В этом определении множество живых переменных Live-Variables(I, b) может быть взято из результатов анализа времени жизни переменных. Множество дополнительных регистров Extra-Registers(I), которые нужны инструкции I, зависит только от типа самой инструкции. Так, например, для вызова функции это множество будет состоять из регистров, которые могут быть испорчены вызываемой функцией, регистра, в котором хранится возвращаемое значение, и регистров, которые будут использованы для передачи параметров.

Определение 9. Регистровым давлением в базовом блоке b назовем максимальное среди регистровых давлений во всех его инструкциях.

$$\text{Reg-Pressure}(b) = \max_{I \in b} (\text{Reg-Pressure}(I, b))$$

Перепишем условия (4.5) и (4.6), используя новое определение:

$$\left|b^{post}\right| \leqslant \text{Total-Regs} - \text{Reg-Pressure}(b), \tag{4.8}$$

$$|b^{pre}| \leqslant \text{Total-Regs} - \text{Reg-Pressure}(b). \tag{4.9}$$

Теперь, если использовать для граничных условий в точке синхронизации J не более

$$\text{Total-Regs} - \max_{e \in J} (\text{Reg-Pressure}(src(e)), \text{Reg-Pressure}(dst(e))) \tag{4.10}$$

регистров, то одно из условий условий (4.8) и (4.9) будет обязательно выполнено во всех прилегающих к J базовых блоках.

4.1.3. Выбор переменных для граничных условий

Включение переменной в граничные условия позволяет ей пересекать границы базовых блоков на регистрах и избежать лишнего сброса этой переменной в память с последующей загрузкой ее из памяти, если она используется в нескольких базовых блоках.

Введем функцию полезности включения переменной x в граничные условия точки синхронизации J: USEFULNESS(x, J). Функция полезности будет вычисляться по следующей формуле:

$$\text{USEFULNESS}(x, J) = |e : e \in J \wedge x \in Vars(src(e)) \wedge x \in Vars(dst(e))| \quad (4.11)$$

Данная формула описывает, сколько ребер входит в точку синхронизации таких, что переменная x используется как в базовом блоке, из которого данное ребро исходит, так и в базовом блоке, в которое данное ребро входит.

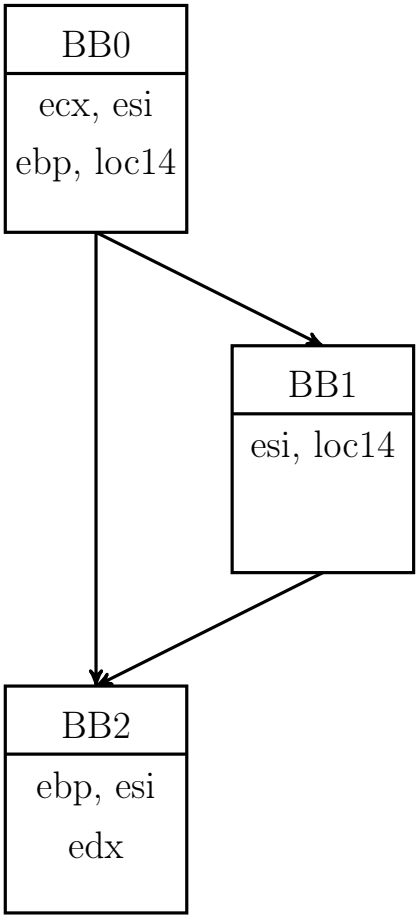
Соответственно, для включения в граничные условия выбираются переменные с максимальным значением функции полезности.

4.2. Реализация комбинированного алгоритма

Реализация комбинированного алгоритма может быть разбита на несколько подзадач:

- построение графа потока управления для блока трансляции,
- вычисление точек синхронизации,
- вычисление граничных условий в точках синхронизации,

Рис. 4.1. Пример вычисления функции полезности



Переменная	Usefulness
ecx	0
edx	0
ebp	1
loc14	1
esi	3

- модификация алгоритма локального распределения регистров таким образом, чтобы он учитывал граничные условия.

4.2.1. Построение графа потока управления

Во внутреннем представлении QEMU все инструкции блока трансляции хранятся в массиве, причем инструкции, входящие в один базовый блок, идут подряд. Базовый блок будет описываться индексами своей первой и последней инструкций, а дуга графа потока управления — указателями на базовые блоки, из которого она выходит и в который она входит.

Во внутреннем представлении QEMU бывают только условные и безусловные переходы на фиксированную метку либо выходы из блока трансляции. Таким образом, из каждого базового блока может исходить не более двух дуг. Запишем эти дуги в массив из двух элементов, хранящийся в базовом блоке. Все входящие в один базовый блок дуги свяжем в однонаправленный список. В каждом базовом блоке запишем указатель на голову соответствующего списка.

Базовый блок $Exit$, соответствующий выходу из блока трансляции, в явном виде заводить не будем. Вместо этого установим на дугах, идущих в него, специальный флаг, указывающий на то, что данная дуга должна вести в него.

Граф потока управления строится за два просмотра кода в прямом порядке. Первый просмотр вычисляет количество базовых блоков и какому базовому блоку соответствует каждая метка. Затем выделяется массив базовых блоков нужного размера. Второй проход заполняет все поля структур.

Границы базовых блоков определяются по типу инструкций: последовательные инструкции I_k и I_{k+1} лежат в разных базовых блоках тогда и только тогда, когда либо I_k является операцией перехода, либо I_{k+1} является меткой.

4.2.2. Вычисление точек синхронизации и граничных условий в них

Как было показано ранее, точки синхронизации могут быть эффективно найдены с помощью обхода в глубину в графе $G_E = \langle E, F \rangle$, где $(e_1, e_2) \in F$ тогда и только тогда, когда $e_1 \sim e_2$. Этот же поиск может быть использован для получения необходимых при вычисления по формуле (4.11) данных и для записи полученных граничных условий в нужные базовые блоки.

Для проведения обхода в глубину строить граф G_E в явном виде нет необходимости. Все вершины, смежные с вершиной e графа G_E, можно найти, просмотрев все исходящие из вершины $src(e)$ графа G дуги и все входящие в $dst(e)$.

4.2.3. Модификация алгоритма локального распределения регистров

При обработке базового блока алгоритмом локального распределения регистров начальная инициализация расположения переменных должна быть установлена в соответствии с начальным условием данного блока.

При выборе регистра для переменной, используемой в вычислениях внутри базового блока, предпочтение отдается регистру из конечного условия, если эта переменная туда входит.

В конце базового блока текущее распределение регистров должно быть приведено к конечному условию. Это может быть эффективно сделано с использованием алгоритма 7.

Отдельно необходимо рассмотреть случай последней инструкции в базовом блоке. В случае инструкции перехода все переупорядочивание регистров должно быть произведено до нее, при этом сама инструкция может требовать размещение некоторых переменных, не входящих в конечное условие базового блока, в регистрах. Для этих переменных выделяются регистры, не входящие в множество $Regs(b^{post})$. Множество регистров, доступных алгоритму 7, затем задается так, чтобы не включать эти регистры.

Необходимое количество регистров всегда найдется, так как

$$r \leqslant \textsc{Reg-Pressure}(b) \leqslant \textsc{Total-Regs} - |b^{post}|,$$

где r — необходимое для работы инструкции перехода количество регистров. Первое неравенство в цепочке следует из определения регистрового давления, второе — из неравенства (4.8).

Алгоритм 8 Выбор граничных условий для точки синхронизации

function COMPUTE-REGISTER-MAPPING(J)
  $p \leftarrow \max\limits_{e \in J} (\textsc{Reg-Pressure}(src(e)), \textsc{Reg-Pressure}(dst(e)))$
  $n \leftarrow \textsc{Total-Regs} - p$
  $result \leftarrow \varnothing$
  $priority \leftarrow \textsc{Compute-Usefullness}(J)$
  for $i \leftarrow 1, n$ **do**
    $result \leftarrow result \cup \{(priority[i], register[i])\}$
  end for
  return $result$
end function

4.3. Результаты

Данный алгоритм был реализован в QEMU версии 1.0. Тесты запускались в режиме эмуляции приложения. Тестирование проводилось на модельном примере и на тестах из набора SPEC CINT2000.

В качестве модельного примера была взята последовательность условных и безусловных инструкций архитектуры ARM, приведенная в листинге 4.1. Данная последовательность выполнялась в цикле $b0000000_{16}$ раз. Условные инструкции обеспечивают наличие нескольких базовых блоков внутри одного блока трансляции. Гостевые регистры $r1$ и $r2$ станут глобальными переменными, используемыми в нескольких базовых блоках. Выполнение данного фрагмента

Листинг 4.1. Модельный пример для глобального распределения регистров

```
addlt    r1, r1, r2
sub      r2, r2, r1
subgt    r1, r1, r2
add      r2, r2, r1
```

Без глобального распределения регистров (секунд)	С глобальным распределением регистров (секунд)	Ускорение
16.644	11.715	29.6%

Таблица 4.1. Результаты тестирования глобального распределения регистров на модельном примере

большое количество раз в цикле позволяет провести измерение времени. Результаты тестирования приведены в таблице 4.1. Ускорение получается за счет того, что переменные, соответствующие гостевым регистрам $r1$ и $r2$, в случае глобального распределения регистров загружаются на регистры основной машины один раз в начале обработки инструкции `addlt`, а в случае локального — перед обработкой каждой инструкции. Таким образом, экономится по 6 загрузок и сохранений на каждом выполнении приведенного фрагмента.

Для того, чтобы определить изменение производительности на реальных программах, были использованы тесты из набора SPEC CINT2000. Тестирование на них показало небольшое падение производительности на большинстве из тестов. Результаты тестирования приведены в таблице 4.2. В ходе анализа удалось установить две существенные причины падения производительности.

Первая из них связана с применимостью данного алгоритма. Для того, чтобы он был применен, необходимо, чтобы

- блок трансляции состоял из нескольких базовых блоков,

Тест	Без глобального распределения регистров (секунд)	С глобальным распределением регистров (секунд)	Ускорение
164.gzip	81.004	81.840	-1%
175.vpr	144.560	148.164	-2.5%
256.bzip2	42.496	40.580	4.5%
300.twolf	36.508	36.544	0%

Таблица 4.2. Результаты тестирования глобального распределения на тестах из набора SPEC CINT2000

- некоторые глобальные переменные использовались в нескольких базовых блоках.

Как удалось выяснить с помощью профилирования, таких блоков всего около 10% от общего числа блоков трансляции. Результаты профилирования для трех тестов приведены в таблице 4.3. Данная проблема может быть устранена за счет увеличения блоков трансляции таким образом, чтобы они могли включать несколько гостевых базовых блоков.

Вторая причина связана с недостаточным учетом граничных условий при локальном распределении регистров. Инициализации начальными условиями и обеспечения выполнения конечных условий недостаточно. Необходимо также отдавать предпочтение регистру из граничных условий при выборе регистра для переменной, а также по возможности не занимать регистры из постусловий под переменные в них не входящие. Для устранения данной причины необходимо внести дополнительные модификации в существующий алгоритм локального распределения регистров, однако без увеличения блоков трансляции данная модификация не окажет существенного влияния на производительность.

Тест	Всего блоков трансляции	Блоков трансляции, состоящих из нескольких базовых блоков	Блоков трансляции, на которых произошло глобальное распределение регистров
164.gzip	3526	651	365
175.vpr	8016	1447	801
256.bzip2	3003	689	388
300.twolf	8221	1631	911

Таблица 4.3. Результаты профилирования глобального распределения регистров

Глава 5

Трансляция векторных инструкций во время динамической двоичной трансляции

В данной главе рассматривается процесс динамической двоичной трансляции векторных инструкций одной процессорной архитектуры в векторные инструкции другой, в общем случае отличной, архитектуры. Реализация предложенных алгоритмов выполнялась в рамках эмулятора QEMU, однако выделенные требования к системе динамической двоичной трансляции являются общими и применимы к любой другой системе.

5.1. Описание метода

Для обеспечения поддержки трансляции векторных инструкций гостевой архитектуры в векторные инструкции основной архитектуры необходимо:

- добавить соответствующий тип данных во внутреннее представление,
- добавить операции, которые работают с переменными данного типа,
- обеспечить работу операций загрузки и сохранения со значениями данного типа, выполняющихся в виртуальных адресах гостевой системы,
- добавить использование новых операций во фронтенд,
- добавить генерацию кода для новых операций в бэкенд.

Операции загрузки и сохранения, выполняющиеся в виртуальных адресах гостевой системы, были выделены отдельно, потому что с ними связана нетривиальная логика преобразования адресов и кэширования результатов данного преобразования, которая может зависеть от размера операнда. Минимальный набор операций должен включать в себя операции загрузки, сохранения и пересылки.

Все остальные операции могут быть сэмулированы через поддерживаемые или с помощью скалярных операций над элементами векторов.

Для возможности использования векторных инструкций основной архитектуры векторные значения должны быть распределены на векторные регистры этой архитектуры, то есть должны являться глобальными переменными внутреннего представления. С другой стороны, в общем случае не все векторные инструкции гостевой архитектуры смогут быть выражены через векторные инструкции основной архитектуры, то есть необходимо будет в ряде случаев генерировать код, который обращается к отдельным элементам вектора, расположенным в памяти. Таким образом, два ранее несовместимых способа работы с состоянием процессора должны будут одновременно работать с одними и теми же полями структуры, описывающей состояние гостевого процессора.

Большое количество комбинаций гостевых и основных архитектур делает процесс выбора способа эмуляции для каждой конкретной инструкции нетривиальным. Поэтому необходимо предусмотреть удобную в использовании функцию, которая будет для требуемой операции выбирать и генерировать оптимальный способ эмуляции в зависимости от возможностей основной архитектуры. Без такой вспомогательной функции практическое применение разрабатываемого метода будет слишком трудозатратным.

Как минимум в одной процессорной архитектуре присутствует существенное перекрытие векторных регистров различной длины между собой. В расширении NEON архитектуры ARM 32-битные регистры с `s0` по `s31` объединены по парам и образуют 64-битные регистры с `d0` по `d15`, которые, в свою очередь, объединены в пары и образуют 128-битные регистры с `q0` по `q8`. Среди скалярных регистров такая ситуация тоже встречается: например, регистр `ax` архитектуры `x86` состоит из регистров `al` и `ah` и является частью регистра `eax`. В данный момент все эти регистры считаются как одна глобальная переменная, соответствующая самому большому из них, и при необходимости идут обращения к ее фрагментам. Это создает дополнительные зависимости по данным

между непересекающимися фрагментами, однако из-за редкого использования регистров меньшей длины не дает существенного ухудшения производительности. Использование такого же подхода к векторным регистрам нежелательно, так как сокращение количества независимых 32-битных векторных регистров в 4 раза приведет к существенному падению производительности приложений, их использующих.

5.1.1. Анализ указателей

Для того, чтобы с одними и теми же полями состояния процессора можно было работать и как с глобальными переменными, и как с ячейками памяти, необходимо добавить в эмулятор механизм отслеживания ситуаций, когда обращение к памяти перекрывается с некоторой глобальной переменной. Данная задача хорошо известна под названием анализ указателей [9] и для случая произвольной программы с произвольными вычислениями над указателями не существует способа решать ее с достаточно высокой точностью. К счастью, в рассматриваемом случае можно сделать дополнительные предположения, которые существенно упрощают решение задачи.

Рассмотрим эти дополнительные предположения. Все они вытекают из принципа работы эмулятора.

1. Все переменные располагаются в памяти эмулятора, и обращения к памяти гостевой системы не могут с ними пересекаться. Для случаев, когда регистры гостевой системы могут быть отображены в адресное пространство, в эмуляторе существует механизм `iomem`, который обеспечивает корректность работы обращений через адресное пространство памяти в ущерб производительности. В этом случае при всех операциях с соответствующими адресами происходят вызовы вспомогательных функций, которые обеспечат корректность сохранения всех переменных.

2. Все глобальные переменные имеют ячейку памяти, являющуюся полем

`CPUState`, и ко всем из них обращение происходит по адресу начала данной структуры с добавлением константного смещения.

3. Подавляющее большинство обращений к памяти эмулятора будут также адресоваться относительно начала структуры `CPUState`.

Таким образом, необходимо для каждого обращения к памяти определить, адресуется ли оно относительно начала структуры `CPUState` и чему равно смещение. Далее, зная смещения и размеры всех переменных, можно определить, какие из них пересекаются с этим обращением. Для обозначения адреса начала структуры `CPUState` используется специальная переменная внутреннего представления `env`, которая всегда находится на специально выделенном под нее регистре и никогда не меняет значения.

Граф потока управления блока трансляции является ациклическим, значит, данная задача анализа потока данных может быть решена за один просмотр кода в прямом порядке. Для каждой переменной в каждой точке программы определим:

- является ли она константой в данной точке программы, и если да, то чему равно значение этой константы;
- может ли она быть выражена как `env + C`, где `C` — некоторая константа, и если да, то чему равна эта константа.

Если ни одно из этих утверждений не выполнено, то будем считать, что переменная содержит некоторое значение, про которое ничего не известно.

Все обращения к памяти во внутреннем представлении записывают адрес в форме *база + смещение*, где и *база*, и *смещение* — некоторые переменные. Если одна из этих переменных может быть представлена как `env + C`, а вторая является константой, то можно однозначно определить, с какими переменными внутреннего представления пересекается данная операция. В противном случае можно действовать консервативно и считать, что данная операция может

пересекаться с любой переменной. На практике данный случай никогда не реализуется.

Собранная информация о перекрытии обращений к памяти с глобальными переменными используется в анализе времени жизни переменных, результаты которого используются при распределении регистров. Необходимо различать два типа перекрытия:

- *полное*, когда обращение к памяти полностью покрывает все байты переменной,
- *частичное*, когда обращение к памяти пересекается с переменной лишь по некоторым байтам данной переменной.

При чтении из памяти все переменные, с которыми данное обращение пересекается, должны быть сохранены в память. При записи в память только переменные, с которыми данное обращение перекрывается частично, должны быть сохранены в память, а переменные, которые полностью покрываются данной записью, должны быть помечены как мертвые.

Рассмотрим пример. Пусть в эмуляторе выполнялся фрагмент кода, написанного для архитектуры ARM, которому соответствует ассемблерный код, приведенный на листинге 5.1. Предположим, что для операции векторного сложения (`vadd`) поддерживается трансляция в векторную операцию основной архитектуры, а для операции попарного векторного сложения (`vpadd`) — нет. Тогда на промежуточном представлении для операций векторного сложения будут сгенерированы соответствующие векторные операции, а для попарного векторного сложения — эмулирующий код, который читает данные поэлементно. Полученное таким образом промежуточное представление приведено в листинге 5.2. Смещение всех обращения к памяти относительно `env` является константным, данные обращения перекрываются с переменной `q0`, которая имеет смещение `0x858`, и никак не затрагивают переменную `q1`, имеющую смещение `0x868`. Поэтому значение переменной `q0` должно быть сохранено в память после первого

Листинг 5.1. Анализ указателей — исходный ассемблер ARM

```
vadd.i32    q0, q0, q1
vpadd.i32   d0, d0, d1
vadd.i32    q0, q0, q1
```

векторного сложения и загружено обратно перед последним. Переменная `q1` может быть загружена на регистр один раз в начале блока трансляции и сохранена только в его конце. Соответствующий ассемблерный код архитектуры x86_64 приведен на листинге 5.3. Регистр `%r14` используется для хранения значения переменной `env`.

5.1.2. Перекрытие переменных

Случай полного или частичного перекрытия переменных обрабатывается аналогично перекрытию обращения к памяти с переменными. Для каждой переменной необходимо построить два списка: список переменных, которые полностью содержатся в данной, и список переменных, которые частично перекрываются с данной. Данные списки могут быть составлены вручную либо вычислены автоматически.

Для автоматического вычисления перекрытия переменных следует отметить, что речь идет только о глобальных переменных. Все глобальные переменные являются также полями структуры `CPUState` и могут быть описаны своим размером и смещением соответствующего им поля структуры от ее начала. Таким образом, для проверки пары переменных на перекрытие достаточно проверить два известных диапазона адресов на пересечение. Общее количество глобальных переменных небольшое, к тому же они создаются один раз при инициализации эмулятора. Таким образом, перебор всех пар переменных квадратичной сложности является вполне допустимым с точки зрения производительности.

Листинг 5.2. Анализ указателей — внутреннее представление

```
——— vadd.i32   q0, q0, q1
add_i32x4 q0,q0,q1

——— vpadd.i32 d0, d0, d1
ld_i32 tmp5,env,$0x858
ld_i32 tmp6,env,$0x85c
add_i32 tmp5,tmp5,tmp6
st_i32 tmp5,env,$0x858
ld_i32 tmp5,env,$0x860
ld_i32 tmp6,env,$0x864
add_i32 tmp5,tmp5,tmp6
st_i32 tmp5,env,$0x85c

——— vadd.i32   q0, q0, q1
add_i32x4 q0,q0,q1
```

Листинг 5.3. Анализ указателей — результирующий ассемблер (x86)

```
;---- vadd.i32 q0, q0, q1
    movdqu 0x858(%r14),%xmm0
    movdqu 0x868(%r14),%xmm1
    paddd  %xmm1,%xmm0
    movdqu %xmm0,0x858(%r14)

;---- vpadd.i32 d0, d0, d1
    mov    0x858(%r14),%ebp
    mov    0x85c(%r14),%ebx
    add    %ebx,%ebp
    mov    %ebp,0x858(%r14)
    mov    0x860(%r14),%ebp
    mov    0x864(%r14),%ebx
    add    %ebx,%ebp
    mov    %ebp,0x85c(%r14)

;---- vadd.i32  q0, q0, q1
    movdqu 0x858(%r14),%xmm0
    paddd  %xmm1,%xmm0
    movdqu %xmm0,0x858(%r14)
    movdqu %xmm1,0x868(%r14)
```

5.2. Реализация метода

5.2.1. Обеспечение работы операций qemu_st и qemu_ld

Операции `qemu_st` и `qemu_ld` обеспечивают загрузку данных из памяти гостевой системы в переменные внутреннего представления. Загрузка производится в виртуальных адресах гостевой системы, которые должны быть преобразованы в виртуальные адреса основной системы, поскольку реальная память адресуется в них. Для улучшения производительности данной операции преобразованные адреса страниц сохраняются в виртуальном TLB (Translation Lookaside Buffer). Кроме того, данные операции могут приводить к исключениям, если память по запрошенному адресу недоступна.

Таким образом, каждая из операций `qemu_st` и `qemu_ld` раскрывается в большой фрагмент машинного кода основной системы. Сначала данный код проверяет наличие страницы в TLB. Если страница присутствует, ее адрес берется оттуда, и с его использованием производится прямая загрузка. Какая инструкция будет использована для загрузки, зависит от размера запрашиваемого фрагмента памяти. Если страница не была найдена в TLB, то вызывается вспомогательная функция, которая производит все сложные вычисления, связанные с преобразованием адресов и вызовом исключения. Для каждого размера обращения к памяти существует отдельная такая функция.

Для работы `qemu_ld` и `qemu_st` с переменными нового размера необходимо:

- добавить поддержку генерации прямой загрузки в бэкенд,
- добавить вспомогательные функции для загрузок соответствующего размера.

Поддержка генерации прямой загрузки не представляет труда — необходимо рассмотреть требуемый случай и сгенерировать инструкцию с соответствующим кодом операции (opcode).

Вспомогательные функции для разных размеров имеют очень много сходств и, во избежание повторения больших фрагментов кода, генерируются по шаблону с помощью препроцессора. Функции для записи в память принимают записываемую величину как один аргумент по значению, а функции для чтения из памяти возвращают прочитанное значение. Соглашения о вызовах для векторных аргументов могут существенно отличаться в зависимости от используемого компилятора и операционной системы либо могут попросту отсутствовать, если основная система не поддерживает векторных инструкций. Значит, при реализации вспомогательных функций для загрузки и сохранения векторных значений требуется модифицировать подход. Функции для записи в память будут принимать записываемое значение как массив байт. Функции для чтения из памяти будут принимать указатель на массив байт, куда требуется записать прочитанное значение. Для генерации этих функций были созданы отдельные шаблоны, отличающиеся от имеющихся.

5.2.2. Анализ указателей и перекрытие переменных

Анализ указателей был реализован как отдельный проход по блоку трансляции, который выполняется в самом начале его обработки. Информация, собранная во время этого прохода, записывается в каждой инструкции блока трансляции. Данная информация используется во время анализа жизни переменных. Если про инструкцию было установлено, что она обращается к памяти, которая потенциально может пересекаться с памятью, отведенной под хранение глобальных переменных, то перебираются все глобальные переменные, и их расположение в памяти сравнивается с адресами памяти, с которыми работает данная инструкция. Такая реализация допустима с точки зрения производительности потому, что в коде встречается относительно мало инструкций, работающих с памятью эмулятора, и число глобальных переменных также невелико.

Для учета перекрытия переменных в код анализа времени жизни переменных было добавлено распространение пометок:

- когда переменная помечается для синхронизации в память, все переменные, с которыми она пересекается, помечаются для синхронизации в память;

- когда переменная помечается как мертвая, все переменные, с которыми имеет место быть полное перекрытие, также помечаются как мертвые, а все переменные, с которыми данная перекрывается частично, помечаются для синхронизации в память.

5.2.3. Функции-обертки

Различные основные архитектуры могут поддерживать различные наборы векторных инструкций, поэтому при трансляции кода гостевой архитектуры при каждой потенциальной возможности использования векторной инструкции необходимо проверить, поддерживается ли данная инструкция. Чтобы это можно было использовать на практике, необходимо реализовать функции-обертки, которые проверяют возможность использования соответствующей векторной инструкции. В случае, если она поддерживается, на внутреннем представлении генерируется векторная операция, иначе генерируется эмулирующий код, который работает с элементами вектора по отдельности.

Эмулирующий код может быть организован несколькими способами:

- как серия вызовов вспомогательной функции, каждый из которых производит операцию над одним элементом вектора,

- как один вызов вспомогательной функции, которая выполняет преобразование над всем вектором,

- как последовательность скалярных операций на внутреннем представлении.

Последний способ является предпочтительным с точки зрения производитель-

ности, однако в случае операций, которые сложно выразить на внутреннем представлении, другие способы тоже являются приемлемыми.

5.3. Результаты

Описанный способ трансляции векторных инструкций гостевой архитектуры в векторные инструкции основной архитектуры был реализован в эмуляторе QEMU версии 2.9.50 (коммит 9964e96dc9999cf7f7c936ee854a795415d19b60). Были реализованы векторные операции обращения к памяти и векторные операции сложения 64-х и 128-битных векторов. В качестве гостевой архитектуры использовалась архитектура `ARM`, в качестве основной — `x86_64`. Никакие другие операции, кроме сложения реализованы пока не были, но даже имеющаяся частичная реализация подхода демонстрирует заметный прирост производительности. Тестирование производилось на нескольких типах тестов:

- искусственный пример, на который разработанный метод должен оказать наиболее существенное влияние,
- результат работы алгоритма автоматической векторизации компилятора GCC,
- искусственные примеры, которые должны проверить корректность работы анализа указателей и учета перекрытия переменных,
- реальная программа для сжатия `x264`, активно использующая векторные инструкции, написанные разработчиками вручную.

Искусственный пример для проверки потенциального улучшения производительности представляет собой цикл с фиксированным количеством итераций, в котором многократно происходит сложение векторных регистров. Ассемблерный код данного цикла приведен в листинге 5.4.

Листинг 5.4. Листинг искусственного примера

```
    mov         r0, #0xb0000000
loop:
    vadd.i32    q0, q0, q1
    vadd.i32    q0, q0, q1
    vadd.i32    q0, q0, q1
    vadd.i32    q0, q0, q1
    subs        r0, r0, #1
    bne         loop
```

Второй тип тестов был получен в результате работы алгоритма автоматической векторизации [58] на цикле, производящем сложение элементов двух массивов и записывающем результат в третий. Исходный код цикла на языке Си приведен в листинге 5.5. Варьируя типы элементов массивов, удалось получить четыре разных теста, в которых фигурировали различные типы сложения. Для сложения однобайтовых чисел количество элементов массива пришлось увеличить вдвое, иначе компилятор производит разворачивание цикла. Для компиляции тестов использовался компилятор GCC [59] версии 4.7.3 20130102 (prerelease) с опциями `-O3 -ftree-vectorize -mfpu=neon -mfloat-abi=hard -mcpu=cortex-a8 -fno-unroll-loops`.

Искусственные примеры для проверки корректности работы анализа указателей и учета перекрытия переменных были получены из тестов двух предыдущих категорий. В ассемблерный код для этих тестов вносились дополнительные инструкции, оперирующие с перекрывающимися переменными, либо неподдерживаемые инструкции, которые будут оттранслированы в обращения к памяти в эмуляторе.

Корректность работы всех вышеперечисленных категорий тестов проверялась выводом результирующих значений.

Листинг 5.5. Цикл для автоматической векторизации

```
int a[256], b[256], c[256];
void foo (void) {
  int i;

  for (i=0; i<256; i++){
    a[i] = b[i] + c[i];
  }
}
```

Тест	QEMU	QEMU-vect	Ускорение
artificial	25.304	7.748	3.27x
autovect.i8	1.604	0.616	2.60x
autovect.i16	3.648	1.304	2.80x
autovect.i32	5.392	2.248	2.40x
autovect.i64	6.296	4.280	1.47x
x264	204.356	183.172	1.12x

Таблица 5.1. Результаты измерения производительности на тестах

Для проверки на реальном приложении была взята программа `x264` [60] (коммит 3f5ed56d4105f68c01b86f94f41bb9bbefa3433b), и с ее помощью проведено сжатие небольшого видеоролика. Корректность работы в данном случае проверялась сравнением `md5`-хэша полученного файла с хэшем ожидаемого файла. Детерминированность данной реализации видеокодека позволяет сделать такую проверку.

Результаты измерения производительности приведены в таблице 5.1. Столбец «QEMU» соответствует запуску QEMU без модификаций, «QEMU-vect» — запуску QEMU с описанными в статье модификациями. Строка «artificial» со-

ответствует искусственному тесту. Строки «autovect.i8», «autovect.i16», «autovect.i32» и «autovect.i64» соответствуют тестам, полученным автовекторизацией. Число на конце соответствует размеру элемента массива в битах. Строка «x264» соответствует запуску программы для кодирования видео.

На искусственном примере достигается ускорение более чем в 3 раза. На автоматически векторизованных циклах ускорение немного меньше, так как меньше вклад векторных инструкций в общее время работы. Ускорение при сложении 64-битных чисел существенно меньше, так как существующая реализация использует меньше операций по сравнения с реализацией для 32-битных и чисел. Существенной разницы между 8-ми, 16-ти и 32-битными числами не наблюдается из-за особенностей существующей реализации. В ней чтения всегда делаются 32-битными фрагментами, а далее за счет маскирования старших битов элементов вектора производится векторное сложение этих 32-битных векторов с использованием скалярных операций. Таким образом, разница в количестве операций между новой реализацией и старой остается неизменной.

Реальное приложение, в отличие от остальных примеров, использует большое количество различных инструкций, не отдавая никакого предпочтения реализованным операциям сложения. На этой программе также наблюдается вполне заметное ускорение на 12%, которое объясняется влиянием операций копирования массивов данных. При использовании векторных регистров происходит меньше операций загрузки из памяти и сохранения обратно в память.

Заключение

В диссертационной работе рассмотрены возможности проведения оптимизаций кода во время динамической двоичной трансляции. Основные результаты диссертационной работы заключаются в следующем.

1. Разработан и реализован алгоритм решения задачи анализа потока данных для ациклических графов потока управления для применения во время динамической двоичной трансляции.

2. Разработан и реализован однопроходный алгоритм локального распределения регистров для применения во время динамической двоичной трансляции. Данный алгоритм учитывает имеющуюся информацию о времени жизни переменных для более эффективного выбора регистров для сброса их содержимого в память.

3. Разработан и реализован комбинированный (глобальный и локальный) алгоритм распределения регистров для применения во время динамической двоичной трансляции. Данный алгоритм позволяет выполнять глобальное распределение регистров, совмещенное с генерацией кода, без введения дополнительного внутреннего представления. Применение данного алгоритма дало увеличение производительности программ до 30%.

4. Разработан метод выражения векторных инструкций одной процессорной архитектуры через векторные инструкции другой процессорной архитектуры. Реализация этого метода обеспечила увеличение производительности до 3 раз на программах, активно использующих векторные инструкции.

В дальнейших работах планируется исследовать влияние размера и структуры блока трансляции на возможности проведения оптимизаций во время динамической двоичной трансляции. В данный момент блоки трансляции имеют

ациклический граф потока управления, что делает невозможным проведение оптимизаций циклов. Кроме того, небольшой размер блока трансляции делает многие другие оптимизации неэффективными. С другой стороны, увеличение блоков трансляции приведет к увеличению времени трансляции и более частой перетрансляции некоторых участков кода. Определение оптимального размера и структуры блока трансляции является важным направлением исследований.

Список литературы

1. *Батузов К.*, *Меркулов А.* Оптимизация динамической двоично трансляции // Труды Института системного программирования РАН. — 2011. — Т. 20. — С. 37—50.

2. *Батузов К.* Задача локального распределения регистров во время динамической двоичной трансляции // Труды Института системного программирования РАН. — 2012. — Т. 22. — С. 67—76.

3. *Батузов К.* Задача глобального распределения регистров во время динамической двоичной трансляции // Труды Института системного программирования РАН. — 2016. — Т. 28, № 5. — С. 199—214.

4. *Батузов К.* Использование векторных инструкций одной процессорной архитектуры для эмуляции векторных инструкций другой процессорной архитектуры // Программирование. — 2017. — № 6. — С. 45—54.

5. *Cifuentes C.*, *Malhotra V. M.* Binary Translation: Static, Dynamic, Retargetable? // Proceedings of the 1996 International Conference on Software Maintenance. — Washington, DC, USA : IEEE Computer Society, 1996. — С. 340—349. — (ICSM '96). — ISBN 0-8186-7677-9.

6. *Horspool R.*, *Marovac N.* An approach to the problem of detranslation of computer programs // The Computer Journal. — 1979. — Т. 23, № 3. — С. 223—229.

7. *Horspool R. N.*, *Marovac N.* An approach to the problem of detranslation of computer programs // The Computer Journal. — 1980. — Т. 23, № 3. — С. 223—229.

8. *Smith J.*, *Nair R.* Virtual Machines: Versatile Platforms for Systems and Processes (The Morgan Kaufmann Series in Computer Architecture and

Design). — San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 2005. — ISBN 1558609105.

9. Компиляторы: принципы, технологии и инструментарий (Второе издание) / А. В. Ахо [и др.]. — Москва : Издательский дом “Вильямс”, 2008. — ISBN 978-5-8459-1349-4.

10. *Muchnick S. S.* Advanced compiler design implementation. — Morgan Kaufmann, 1997.

11. *Chipounov V.*, *Candea G.* Dynamically Translating x86 to LLVM using QEMU: тех. отч. — 2010. — EPFL-TR-149975.

12. HQEMU: A Multi-threaded and Retargetable Dynamic Binary Translator on Multicores / D.-Y. Hong [и др.] // Proceedings of the Tenth International Symposium on Code Generation and Optimization. — San Jose, California : ACM, 2012. — С. 104—113. — (CGO ’12). — ISBN 978-1-4503-1206-6. — DOI: 10.1145/2259016.2259030. — URL: http://doi.acm.org/10.1145/2259016.2259030.

13. LnQ: Building High Performance Dynamic Binary Translators with Existing Compiler Backends / C.-C. Hsu [и др.] // Proceedings of the 2011 International Conference on Parallel Processing. — Washington, DC, USA : IEEE Computer Society, 2011. — С. 226—234. — (ICPP ’11). — ISBN 978-0-7695-4510-3. — DOI: 10.1109/ICPP.2011.57. — URL: http://dx.doi.org/10.1109/ICPP.2011.57.

14. The LLVM Compiler Infrastructure. — Дата обращения: 14.12.2017. http://llvm.org/.

15. *Schlansker M. S.*, *Rau B. R.* EPIC: An Architecture for Instruction-Level Parallel Processors: тех. отч. — 2000. — HPL-1999-111.

16. Intel Itanium Architecture Software DEveloper’s Manual. Volume 3: Intel Itanium Instruction Set Reference / Intel. — 420 с. — May 2010.

17. PQEMU: A Parallel System Emulator Based on QEMU / J.-H. Ding [и др.] // Proceedings of the 2011 IEEE 17th International Conference on Parallel and Distributed Systems. — Washington, DC, USA : IEEE Computer Society, 2011. — С. 276—283. — (ICPADS '11). — ISBN 978-0-7695-4576-9. — DOI: 10.1109/ICPADS.2011.102. — URL: http://dx.doi.org/10.1109/ICPADS.2011.102.

18. Features/tcg-multithreading — QEMU. — Дата обращения: 07.02.2018. https://wiki.qemu.org/Features/tcg-multithread.

19. Multi-threaded emulation for QEMU. — Дата обращения: 07.02.2018. https://lwn.net/Articles/697265/.

20. Микропроцессорные вычислительные комплексы с архитектурой «Эльбрус» и их программное обеспечение / А. К. Ким [и др.] // Вопросы радиоэлектроники. — 2009. — Т. 4, № 3. — С. 5—37.

21. Система динамической двоичной трансляции X86→«ЭЛЬБРУС» / Н. В. Воронов [и др.] // Вопросы радиоэлектроники. — 2012. — Т. 4, № 3. — С. 89—108.

22. AMD64 Architecture Programmer's Manual Volume 4: 128-Bit and 256-Bit Media Instructions / Advanced Micro Devices. — 1025 с. — October 2013.

23. ARM Architecture Reference Manual. ARMv7-A and ARMv7-R edition / ARM Limited. — 2158 с. — April 2008.

24. Eltechs ExaGear Desktop. Run x86 applications on ARM-based devices. — Дата обращения: 03.07.2017. https://eltechs.com/.

25. ExaGear Desktop System Requirements (Updated for v2.1). — Дата обращения: 03.07.2017. http://forum.eltechs.com/viewtopic.php?f=4&t=4& sid=.

26. *Nethercote N.*, *Seward J.* Valgrind: a framework for heavyweight dynamic binary instrumentation // SIGPLAN Not. — New York, NY, USA, 2007. — Июнь. — Т. 42, № 6. — С. 89—100. — ISSN 0362-1340.

27. *Farach M.*, *Liberatore V.* On local register allocation // Proceedings of the ninth annual ACM-SIAM symposium on Discrete algorithms. — San Francisco, California, United States : Society for Industrial, Applied Mathematics, 1998. — С. 564—573. — (SODA '98). — ISBN 0-89871-410-9.

28. *Liberatore V.*, *Farach-Colton M.*, *Kremer U.* Evaluation of Algorithms for Local Register Allocation. — 1999.

29. *Hsu W.-C.*, *Fisher C. N.*, *Goodman J. R.* On the Minimization of Loads/Stores in Local Register Allocation // IEEE Trans. Softw. Eng. — Piscataway, NJ, USA, 1989. — Окт. — Т. 15, № 10. — С. 1252—1260. — ISSN 0098-5589.

30. Register allocation via coloring / G. J. Chaitin [и др.] // Computer languages. — 1981. — Т. 6, № 1. — С. 47—57.

31. *Chaitin G. J.* Register allocation & spilling via graph coloring // Proceedings of the 1982 SIGPLAN symposium on Compiler construction. — Boston, Massachusetts, United States : ACM, 1982. — С. 98—105. — (SIGPLAN '82). — ISBN 0-89791-074-5.

32. *Kolte P.*, *Harrold M. J.* Load/store range analysis for global register allocation // ACM SIGPLAN Notices. Т. 28. — ACM. 1993. — С. 268—277.

33. *Chow F.*, *Hennessy J.* Register allocation by priority-based coloring // ACM Sigplan Notices. — 1984. — Т. 19, № 6. — С. 222—232.

34. *Callahan D.*, *Koblenz B.* Register allocation via hierarchical graph coloring // ACM SIGPLAN Notices. Т. 26. — ACM. 1991. — С. 192—203.

35. *Gupta R.*, *Soffa M. L.*, *Steele T.* Register allocation via clique separators // ACM SIGPLAN Notices. Т. 24. — ACM. 1989. — С. 264—274.

36. *Gupta R.*, *Soffa M. L.*, *Ombres D.* Efficient register allocation via coloring using clique separators // ACM Transactions on Programming Languages and Systems (TOPLAS). — 1994. — Т. 16, № 3. — С. 370—386.

37. *Poletto M.*, *Sarkar V.* Linear scan register allocation // ACM Transactions on Programming Languages and Systems. — New York, NY, USA, 1999. — Сент. — Т. 21, № 5. — С. 895—913.

38. *Traub O.*, *Holloway G.*, *Smith M. D.* Quality and speed in linear-scan register allocation // Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation. — Montreal, Quebec, Canada : ACM, 1998. — С. 142—151. — (PLDI '98). — ISBN 0-89791-987-4.

39. The Transmeta Code Morphing™ Software: using speculation, recovery, and adaptive retranslation to address real-life challenges / J. C. Dehnert [и др.] // Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization. — San Francisco, California : IEEE Computer Society, 2003. — С. 15—24. — (CGO '03). — ISBN 0-7695-1913-X.

40. Pin: building customized program analysis tools with dynamic instrumentation / C.-K. Luk [и др.] // Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation. — Chicago, IL, USA : ACM, 2005. — С. 190—200. — (PLDI '05). — ISBN 1-59593-056-6.

41. Pin - A Dynamic Binary Instrumentation Tool. — Дата обращения: 14.12.2017. https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrum

42. *Bellard F.* QEMU, a fast and portable dynamic translator // Proceedings of the annual conference on USENIX Annual Technical Conference. — Anaheim, CA : USENIX Association, 2005. — С. 41—46. — (ATEC '05).

43. QEMU: the FAST! processor emulator. — Дата обращения: 14.12.2017. https://www.qemu.org/.

44. Valgrind. — Дата обращения: 14.12.2017. http://valgrind.org/.

45. *Cmelik R. F.*, *Keppel D.* Shade: A Fast Instruction-Set Simulator for Execution Profiling: тех. отч. — 1993. — SMLI 93-12, UWCSE 93-06-06.

46. Instruction-level parallel processors / W.-M. W. Hwu [и др.] // / под ред. H. C. Torng, S. Vassiliadis. — Los Alamitos, CA, USA : IEEE Computer Society Press, 1995. — Гл. The superblock: an effective technique for VLIW and superscalar compilation. С. 234—253. — ISBN 0-8186-6527-0.

47. An Efficient Method of Computing Static Single Assignment Form / R. Cytron [и др.] // Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. — Austin, Texas, USA : ACM, 1989. — С. 25—35. — (POPL '89). — ISBN 0-89791-294-2. — DOI: 10.1145/75277.75280. — URL: http://doi.acm.org/10.1145/75277.75280.

48. Efficiently computing static single assignment form and the control dependence graph / R. Cytron [и др.] // ACM Transactions on Programming Languages and Systems (TOPLAS). — 1991. — Т. 13, № 4. — С. 451—490.

49. *Poletto M.*, *Sarkar V.* Linear scan register allocation // ACM Trans. Program. Lang. Syst. — New York, NY, USA, 1999. — Сент. — Т. 21, № 5. — С. 895—913. — ISSN 0164-0925.

50. Система динамической двоичной трансяции/МЦСТ. — Дата обращения: 24.08.2017. http://www.mcst.ru/dvoichnyy_translyator.

51. *Chetverina O. A.* Procedures classification for optimizing strategy assignment // Труды Института системного программирования РАН. — 2015. — Т. 27, № 3. — С. 87—99.

52. *Василец П. С.* Методология повышения производительности мультимедийных приложений в процессе оптимизирующей двоичной трансляции. Диссертация на соискание степени к. т. н. // Институт микрпроцессорных вычислительных систем РАН. — 2006.

53. *Рыбаков А. А.* Методы а алгоритмы оптимизации переходов в компиляторе базового уровня системы двоичной трансляции для архитектуры «Эль-

брус». Диссертация на соискание степени к. ф.-м. н. // ЗАО «МЦСТ». — 2013.

54. *Alpern B.*, *Wegman M. N.*, *Zadeck F. K.* Detecting equality of variables in programs // Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. — ACM. 1988. — С. 1—11.

55. *Rüthing O.*, *Knoop J.*, *Steffen B.* Detecting equalities of variables: Combining efficiency with precision // International Static Analysis Symposium. — Springer. 1999. — С. 232—247.

56. *Briggs P.*, *Cooper K. D.*, *Simpson L. T.* Value numbering // Software-Practice and Experience. — 1997. — Т. 27, № 6. — С. 701—724.

57. SPEC CPU2000. — http://www.spec.org/cpu2000.

58. Auto-vectorization in GCC — GNU Project — Free Software Foundation (FSF). — Дата обращения: 03.07.2017. https://gcc.gnu.org/projects/tree-ssa/vecto

59. GCC, the GNU Compiler Collection. — Дата обращения: 03.07.2017. https://gcc.gnu.org/.

60. x264, the best H.264/AVC encoder — VideoLAN. — Дата обращения: 03.07.2017. http://www.videolan.org/developers/x264.html.