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

АВТОРЕФЕРАТ

диссертации на соискание ученой степени

кандидата физико-математических наук

Москва – 2018

Работа выполнена в Федеральном государственном бюджетном учреждении науки Институте системного программирования им. В.П. Иванникова Российской академии наук.

Научный руководитель: **Белеванцев Андрей Андреевич**,
кандидат физико-математических наук

Официальные оппоненты: **Ильин Вячеслав Анатольевич**,
доктор физико-математических наук,
начальник отдела Курчатовского комплекса НБИКС-технологий Национального исследовательского центра «Курчатовский институт»

Волконский Владимир Юрьевич,
кандидат технических наук,
начальник отделения «Системы программирования» Публичного акционерного общества «Институт электронных управляющих машин им. И. С. Брука»

Ведущая организация: Федеральный исследовательский центр «Информатика и управление» Российской академии наук

Защита состоится 24 мая 2018 г. в 15 часов на заседании диссертационного совета Д 002.087.01 при Федеральном государственном бюджетном учреждении науки «Институт системного программирования им. В.П. Иванникова Российской академии наук» по адресу: 109004, Москва, ул. А. Солженицына, 25.

С диссертацией можно ознакомиться в библиотеке и на сайте Федерального государственного бюджетного учреждения науки «Институт системного программирования им. В.П. Иванникова Российской академии наук».

Автореферат разослан «____» _____ 2018 г.

Ученый секретарь
диссертационного совета Д 002.087.01,
кандидат физико-математических наук Зеленов С.В.

Общая характеристика работы

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.

Методология и методы исследования. Результаты диссертационной работы получены с использованием методов и моделей, используемых при трансляции и оптимизации кода программ. Математическую основу данной работы составляют теория графов, теория множеств, теория алгоритмов.

Положения, выносимые на защиту.

1. Алгоритм решения задачи анализа потока данных для ациклических гра-

фов 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-независи-

мых 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.

Содержание работы

Во Введении обоснована актуальность диссертационной работы, сформулирована цель и аргументирована научная новизна исследований, показана практическая значимость полученных результатов, представлены выносимые на защиту научные положения.

В первой главе вводятся основные понятия и определения, описывается процесс динамической двоичной трансляции и приводится обзор работ, имеющих отношение к теме диссертации.

В начале главы вводятся термины, которые используются далее в тексте диссертации. Двоичная трансляция — это процесс получения по заданной программе P программы Q, удовлетворяющей заданным требованиям, если обе программы записаны в виде машинных кодов.

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. Трансляция гостевого кода происходит небольшими участками, называемыми *блоками трансляции*.

В разделе 1.2 описывается общая схема динамической двоичной трансля-

ции. В разделе 1.3 ставится задача анализа потока данных и приводятся итеративные алгоритмы 1 и 2 ее решения. Данные алгоритмы ищут решение задачи потока данных, начиная с фиксированного начального приближения. До тех пор, пока в текущее приближение вносятся изменения, они просматривают все базовые блоки программы и перевычисляют значение потока данных на основании уравнений потока данных.

В разделе 1.4 описываются возможности повышения производительности эмулятора за счет параллельного выполнения части вычислений. В разделе 1.5 вводится понятие векторных инструкций процессора и разбираются сложности по трансляции данного типа инструкций из одной процессорной архитектуры в другую. В разделе 1.6 формулируется задача распределения регистров и приводятся основные алгоритмы ее решения. В разделе 1.7 приведен обзор существующих программ, использующих динамическую двоичную трансляцию в своей работе: системы динамической трансляции для процессоров «Эльбрус», Valgrind, Pin, Shade, Transmeta Code Morphing, QEMU и другие.

В разделе 1.8 подводится итог, какие методы оптимизации имеет смысл применить в программных эмуляторах, и описывается дальнейшая структура диссертации. В качестве машинно-независимых оптимизаций были выбраны продвижение констант и копий и сворачивание константных выражений. Удаление мертвого кода уже реализовано в выбранном для экспериментов эмуляторе QEMU, а примеров возникновения недостижимого кода или общих подвыражений обнаружить на существующих примерах не удалось. Алгоритмы распределения регистров предлагается делать однопроходными и совмещенными с генерацией кода. Они также должны минимизировать накладные расходы, так как они будут выполняться во время работы программы. Для более эффективного использования вычислительных ресурсов процессора предлагается транслировать векторные инструкции одной архитектуры в векторные инструкции другой архитектуры.

Во второй главе строится алгоритм 3 продвижения констант и копий и

сворачивания константных выражений, использующий решение частного случая задачи анализа потока данных для ациклических графов потока управления и приводятся его использования для продвижения копий и констант, а также сворачивания константных выражений. Рассматриваемый алгоритм опирается на свойство ацикличности графа потока управления блока трансляции и тем самым отличается от алгоритмов, используемых в компиляторах.

Избыточности, устраняемые данными машинно-независимыми оптимизациями, могут возникать из разных источников. В случае динамической двоичной трансляции это обычно является следствием различия гостевой и целевой процессорных архитектур либо неоптимальностью самого процесса трансляции.

В работе показано, что итеративные алгоритмы 1 и 2 сойдутся за одну итерацию в случае, если граф потока управления ациклический и все базовые блоки просматриваются в определенном порядке, а именно, доказываются следующие теоремы:

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.*

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.*

В разделе 2.2 происходит построение алгоритма оптимизации. Для каждой переменной внутреннего представления предлагается хранить её текущее состояние, которое может принимать одно из трёх значений: *является константой*, *является копией*, *не является ни копией, ни константой*. Для переменных, яв-

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

В любой заданной точке кода отношение «переменная A является копией переменной B» является отношением эквивалентности на множестве переменных и задаёт классы эквивалентности в этом множестве. В ходе оптимизаций в каждой точке кода в каждом таком классе выбирается представитель и обеспечивается, чтобы все остальные переменные данного класса считались копиями этого одного представителя.

Далее описывается сама оптимизация. Код блока трансляции просматривается в прямом порядке, и вычисляются состояния всех переменных в каждой точке программы. Изначально все переменные находятся в состоянии *не является ни копией, ни константой*. Операции пересылки переменных из одного класса эквивалентности удаляются. Арифметические операции, аргументы которых являются константами, вычисляются на этапе трансляции.

Алгоритм 3 является консервативным: эквивалентность всех изменений может быть легко доказана на основе имеющейся у оптимизации информации на момент внесения изменений в инструкцию. Сложность алгоритма 3 — O(количество_инструкций · количество_переменных).

В конце главы приводятся результаты тестирования данной оптимизации, реализованной в эмуляторе QEMU. При тестировании эмулировалась архитектура ARM в режиме приложения на целочисленных тестах из набора SPEC CPU2000. Тесты 254.gap и 255.vortex не работали ожидаемым образом в этом режиме. Небольшое увеличение производительности наблюдалось на всех тестах, кроме 176.gcc и 186.crafty. При этом изменения скорости работы теста 176.gcc находятся в пределах точности измерений. На тесте 186.crafty наблюдается ухудшение производительности.

В третьей главе описывается новый алгоритм локального распределения регистров для применения во время динамической двоичной трансляции.

Данный алгоритм строится на основе базового алгоритма распределения, схожего с уже реализованным в QEMU, и описанных в разделе 1.6 эвристик для локального распределения регистров.

Базовый алгоритм распределения регистров просматривает код в прямом порядке (от начала к концу). Для каждого аргумента очередной инструкции, который еще не распределен на регистр, выбирается произвольный свободный регистр. Если свободного регистра не нашлось, то произвольный не используемый в данной инструкции регистр освобождается, а его содержимое сбрасывается в память.

Процесс распределения регистров 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 в последствии уже невозможно, и алгоритм распределения регистров является однопроходным. Распределение регистров локальное, границы базовых блоков переменные пересекают в памяти.

В динамической двоичной трансляции производится распределение регистров во время выполнения программы, поэтому время работы самого алгоритма также важно учитывать. По соотношению времени работы и качества получаемого кода одним из лучших является эвристический алгоритм FF (Furthest-First). Этот алгоритм аналогичен существующему в QEMU, однако при освобождении регистра в случае их нехватки выбирается не произвольный, а тот, следующее использование содержимого которого находится как можно дальше в коде программы.

Однако наличие во внутреннем представлении QEMU вызовов функций, которые не рассматриваются в алгоритме FF, накладывает два дополнительных ограничения: вызываемая функция может читать или изменять глобальные переменные, поэтому все они должны быть сохранены из регистров в память; содержимое некоторых реальных регистров может быть испорчено вызываемой функцией в соответствии с соглашениями о вызовах. В существующем в QEMU

алгоритме эти проблемы решены лишь частично.

В диссертационной работе предлагается улучшенный алгоритм 4, который сохраняет глобальные переменные в память не перед самим вызовом функции, а сразу после их последнего использования до этого вызова. На границах базовых блоков алгоритм 4 поступает 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 эту запись как можно раньше, освобождая регистр для других нужд.

В случае динамической двоичной трансляции большинство вызовов являются вызовами небольших вспомогательных функций самой системы трансляции. Определить, какая именно функция вызывается, можно по ее адресу. Про них возможно получить дополнительную информацию про использование этими функциями глобальных переменных. Эту информацию можно использовать в дальнейшем при распределении регистров.

В разделе 3.3 приводятся детали реализации алгоритма 4 и результаты тестирования.

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.

При распределении регистров для каждого регистра в каждый момент времени 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 перемен-

ную 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.

Алгоритм 3 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.

Алгоритм 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, когда описание части вспомогательных функций отсутствуют.

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.

Тестирование показало существенное снижение количества 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 алгоритме 4 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 этого момента они могут быть сброшены по причине нехватки регистров. Тем не менее, общее количество сбросов регистров уменьшилось на 1%.

Кроме того, тестирование показало, что существенная часть сбросов регистров в память приходится на границы базовых блоков (более 33%) и на вызовы функций (почти 60%). Из этого делается вывод, что переход к глобальному распределению регистров может увеличить производительность эмулятора.

В четвертой главе описывается комбинированный алгоритм 5 распределения регистров, состоящий из алгоритма локального распределения регистров и алгоритма глобального распределения регистров. Алгоритм 5 решает задачу глобального распределения регистров.

Описание алгоритма 5 начинается с рассмотрения возможностей по применению существующих алгоритмов глобального распределения регистров. Алгоритм раскраски графа отличается высокими накладными расходами, что делает его применение нежелательным в рамках динамической двоичной трансляции. Алгоритм линейного сканирования, напротив, имеет низкие накладные расходы и решает задачу глобального распределения регистров за линейное время. Однако данный алгоритм требует второго прохода для устранения противоречий на входах в базовые блоки с несколькими предшественниками. Для этого требуется промежуточное представление с уже распределенными регистрами. В случае его отсутствия добавление такого представления также повлечет за собой дополнительные накладные расходы.

Далее изучаются особенности промежуточного представления, получаемого во время динамической двоичной трансляции, и делается наблюдение, что количество переменных, интервалы жизни которых пересекают границы базовых блоков, существенно меньше общего числа переменных.

Алгоритм 4 может быстро и эффективно распределять регистры в рамках одного базового блока. Его можно модифицировать 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 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.

Далее рассматривается вопрос эффективного построения непротиворечивого набора граничных условий для всех базовых блоков блока трансляции. Условия на распределение регистров в начале базового называются начальными, а в конце — конечными. Начальные условия базового блока b обозначаются как b^{pre}, а конечные — как b^{post}. Для корректности всех граничных условий необходимо, чтобы для каждой дуги e графа потока управления выполнялось равенство $src(e)^{post} = dst(e)^{pre}$.

Определение 1. Точкой синхронизации назовем непустое множество J дуг графа потока управления такое, что

- для любых двух смежных 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 и e_{i+1} — смежные.

Теорема 3. *Множество дуг $\{e_i\}$ графа потока управления $G = \langle B, E \rangle$ явля-*

ется точкой синхронизации тогда и только тогда, когда они образуют компоненту связности в графе $G_E = \langle E, F \rangle$, $\forall e_1, e_2 \in E : (e_1, e_2) \in F \iff e_1$ *и* e_2 *— смежные.*

У данной теоремы 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.

Последнее следствие является основой для алгоритма 5. Алгоритм 5 строит все точки синхронизации блока трансляции и затем для каждой точки синхронизации устанавливает одинаковые граничные условия на обоих концах всех дуг графа потока управления, входящих в эту точку синхронизации.

Теорема 4. *Множество граничных условий, построенное вышеописанным способом, корректно.*

Следующая подзадача, решаемая в работе, — определение количества регистров для использования под граничные условия. Пусть точное количество регистров, которые необходимы для генерации кода для базового блока b, — Registers-Needed(b). Общее количество регистров целевой архитектуры — Total-Registers. Тогда для возможности эффективного распределения регистров внутри базового блока достаточно потребовать выполнения условий:

$$\left|b^{post}\right| + \text{Registers-Needed}(b) \leqslant \text{Total-Registers}, \tag{1}$$

$$\left|b^{pre}\right| + \text{Registers-Needed}(b) \leqslant \text{Total-Registers}. \tag{2}$$

Величина Registers-Needed(b), используемая в условиях (1) и (2), априори неизвестна и не может быть легко вычислена. Ее можно оценить приближенно через количество одновременно живых переменных внутри блока и количество дополнительных регистров, которые необходимы для служебных нужд

кода. Полученную таким образом оценку предлагается называть *регистровым давлением* и обозначать Reg-Pressure(b). Данная оценка является консервативной. Если использовать для граничных условий в точке синхронизации J не более

$$\text{Total-Regs} - \max_{e \in J} (\text{Reg-Pressure}(src(e)), \text{Reg-Pressure}(dst(e))) \quad (3)$$

регистров, то условия (1) и (2) будут обязательно выполнены во всех базовых блоках.

Для выбора переменных для включения в граничные условия вводится следующая функция полезности:

$$\text{Usefulness}(x, J) = |e : e \in J \wedge x \in Vars(src(e)) \wedge x \in Vars(dst(e))| \quad (4)$$

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.

Также в диссертационной работе описывается алгоритм 6, который добавляет в конец базового блока код, который обеспечивает выполнение конечных условий. Эта подзадача может быть сведена к задаче переупорядочивания регистров, которая формулируется следующим образом.

Определение 2. 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.

Определение 3. 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 $M_1 < M_2$.

В работе описывается алгоритм 7 переупорядочивания регистров, использующий только регистры из множества $R \supseteq Regs(b^{pre}) \cup Regs(b^{post})$, и доказывается теорема 7, что данный алгоритм является оптимальным.

В разделе 4.2 приводятся детали реализации комбинированного алгоритма 5. Реализация алгоритма 5 может быть разбита на несколько подзадач: построение графа потока управления для блока трансляции, вычисление точек синхронизации, вычисление граничных условий в точках синхронизации, модификация алгоритма локального распределения регистров таким образом, чтобы он учитывал граничные условия.

Точки синхронизации могут быть эффективно найдены с помощью обхода в глубину в графе $G_E = \langle E, F \rangle$. Этот же поиск может быть использован для получения необходимых для вычисления по формуле (4) 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 и все входящие в $dst(e)$ дуги.

При обработке базового блока алгоритмом локального распределения регистров начальная инициализация расположения переменных должна быть установлена в соответствии с начальным условием данного блока. В конце базового блока текущее распределение регистров должно быть приведено к конечному условию. Это может быть эффективно сделано с использованием алгоритма 7.

В разделе 4.3 приводятся результаты тестирования алгоритма 5. Он был реализован в QEMU. Тесты запускались в режиме эмуляции приложения. Тестирование проводилось на модельном примере и на тестах из набора SPEC CINT2000. На модельном примере в результате применения алгоритма 5 производительность эмулятора выросла на 29.6%. На большинстве реальных примеров наблюдается небольшое замедление.

В работе формулируются две причины замедления на реальных тестах. Первая состоит в том, что лишь 10% блоков трансляции имеют такую структу-

ру, что алгоритм глобального распределения к ним применим. Все остальные блоки трансляции имеют либо один базовый блок в своем составе, либо никакие глобальные переменные не используются более чем в одном базовом блоке.

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 конечных условий под переменные в них не входящие.

В пятой главе рассматривается процесс динамической двоичной трансляции векторных инструкций одной процессорной архитектуры в векторные инструкции другой, в общем случае отличной, архитектуры.

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 работать с одними и теми же полями структуры `CPUState` одновременно.

В некоторых процессорных архитектурах присутствует существенное перекрытие векторных регистров различной длины между собой. Среди скалярных регистров такая ситуация тоже встречается: например, регистр `ax` архитектуры `x86` состоит из регистров `al` и `ah`, а также 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, например, в архитектуре `ARM`, приведет к существенному падению производительности приложений, их использующих.

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, 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.
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.

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 и *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.

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 либо 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. 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 за $O(N^2)$ является вполне допустимым с точки зрения производительности.

Описанный способ трансляции векторных инструкций гостевой архитектуры в векторные инструкции основной архитектуры был реализован в эмуляторе QEMU версии 2.9.50 (коммит 9964e96d). 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, реа-

лизованы в данной работе 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.

Второй тип тестов был получен в результате работы алгоритма автоматической векторизации на цикле, производящем сложение элементов двух массивов и записывающем результат в третий. Варьируя типы элементов массивов, удалось получить четыре разных теста, в которых фигурировали различные типы сложения.

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.

Для проверки на реальном приложении была взята программа `x264` (коммит 3f5ed56d) и с ее помощью проведено сжатие небольшого видеоролика. Кор-

ректность работы в данном случае проверялась сравнением `md5`-хэша полученного файла с хэшем ожидаемого файла. Детерминированность данной реализации видеокодека позволяет сделать такую проверку.

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 на данных типах тестов находится в пределах от 1.5 до 2.8 раз.

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.

В Заключении формулируются результаты диссертационной работы и рассматриваются направления дальнейших исследований. Основным фактором, ограничивающим эффективность алгоритмов оптимизации, является маленький размер блоков трансляции.

Основные результаты диссертационной работы.

1. Разработан и реализован алгоритм, решающий задачу 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 в память.

3. Разработан и реализован комбинированный (глобальный и локальный) алгоритм распределения регистров для эффективного распределения ре-

гистров 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.

Основные публикации по теме диссертации

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.