Оптимизация динамической двоичной трансляции

Кирилл Батузов, Алексей Меркулов
batuzovk@ispras.ru, steelart@ispras.ru

Аннотация. Двоичная трансляция — это процесс получения по заданной программе P программы Q, удовлетворяющей заданным требованиям, если обе программы записаны в виде машинных кодов. Если двоичная трансляция производится во время выполнения программы, то она называется динамической двоичной трансляцией. В данной статье рассматриваются возможности применения различных оптимизаций во время динамической двоичной трансляции, а именно: мы улучшили алгоритм поиска блоков трансляции в кэше трансляций в QEMU, проанализировали влияние алгоритма распределения регистров на быстродействие кода в QEMU, реализовали простые машинно-независимые оптимизации в QEMU и планировщик инструкций в Valgrind. Изменения алгоритма поиска блоков трансляций в кэше дали наибольший эффект, планировщик инструкций также является многообещающей оптимизацией.

Ключевые слова: оптимизации программы, динамическая двоичная трансляция, QEMU, Valgrind.

1. Введение

Двоичная трансляция — это процесс получения по заданной программе P программы Q, удовлетворяющей заданным требованиям, если обе программы записаны в виде машинных кодов. Примерами требований могут быть «программа Q должна делать то же, что и программа P, но выполняться на процессоре с другой архитектурой» или «программа Q должна делать все то же, что и программа P, но дополнительно должна проверять корректность всех операций работы с памятью». Если двоичная трансляция производится во время выполнения программы, то она называется динамической двоичной трансляцией.

Различают два подхода к динамической двоичной трансляции: *копирование и аннотирование* и *дизассемблирование и перегенерация*. Первый подход подразумевает, что код транслируемой программы копируется в том виде, в котором он уже существует, и к нему добавляется некоторый дополнительный, анализирующий код. Во втором подходе программа дизассемблируется в некоторое внутреннее представление. В этом

представлении в неё вносятся изменения, после чего, для полученного внутреннего представления генерируется машинный код, который и будет исполняться [1]. В данной статье будет рассматриваться только второй подход.

Целью данной работы является исследование возможностей применения различных оптимизаций во время динамической двоичной трансляции.

Динамическая двоичная трансляция имеет несколько особенностей, которые необходимо учитывать во время разработки алгоритмов оптимизаций:

- оптимизации производятся во время выполнения, 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).

Для практических экспериментов в данной работе используются два инструмента, использующих динамическую двоичную трансляцию: QEMU [2] и Valgrind [3]. Оба эти инструмента имеют открытый исходный код.

2. Оптимизация двоичной трансляции в QEMU

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. QEMU также поддерживает использование аппаратной виртуализации, однако она не будет рассматриваться в рамках данной работы.

Единицей двоичной трансляции в QEMU является расширенный базовый блок — ациклический участок кода с одним входом. Трансляция происходит при первом обращении к соответствующему коду. Полученные оттранслированные блоки кэшируются и затем используются при повторных выполнениях того же самого кода.

Профилирование QEMU при эмуляции процессора ARM на архитектуре x86 с помощью OProfile [4] выявило, что оттранслированный код гостевой программы и код самого QEMU выполняются сравнимое количество времени. Поэтому в данной работе будут рассмотрены оба аспекта: оптимизации кода

во время динамической двоичной трансляции и улучшение кода самого QEMU.

2.1. Оптимизация поиска в кэше оттранслированных блоков.

Первая рассматриваемая в данной работе оптимизация повышает эффективность поиска оттранслированных блоков в кэше.

В QEMU поддерживаются два уровня кэша для оттранслированных блоков. Оба уровня реализованы с помощью хэш-таблиц. В первом, более быстром, уровне хэш берётся от значения счётчика команд. На этом уровне каждому хэшу может соответствовать не более одного блока, а при возникновении коллизий один блок вытесняет другой. В случае отсутствия нужного блока в первом уровне кэша поиск начинается во втором. На втором уровне хэш берётся от физического адреса блока трансляции. Каждому хэшу соответствует список блоков, среди которых ведётся линейный поиск нужного блока. Если нужный блок не был найден, то вызывается процедура двоичной трансляции, которая его генерирует. После того, как блок найден или сгенерирован, оба уровня кэша обновляются соответствующим образом.

Использование счётчика команд для хэширования на первом уровне кэша позволяет быстро выполнять поиск без необходимости преобразования виртуальных адресов в физические. Это является весьма дорогой операцией, поскольку выполняется полностью программно. В свою очередь, использование физических адресов на втором уровне кэша позволяется нескольким приложениям использовать одну и ту же разделяемую библиотеку без необходимости несколько раз транслировать код этой библиотеки.

Нами было экспериментально установлено, что первый уровень кэша успешно обрабатывает от 75% запросов, для сложных систем с большим количеством активных графических приложений и задействованных библиотек, до почти 100% для небольших программ, занимающихся какими-либо вычислениями без привлечения стороннего кода из разделяемых библиотек. В результате загрузки и работы операционной системы на базе ядра Linux с большим набором графических приложений, количество элементов списка, просматриваемых на втором уровне кэша, на каждом запросе в среднем превосходило 20. Профилирование показало, что функция поиска в кэше второго уровня при этом работала более 60% общего времени работы.

Предложенная нами оптимизация переносит только что найденный на втором уровне кэша блок в начало соответствующего списка. Таким образом, если данный блок в скором времени опять будет запрошен, он будет найден быстрее. В сочетании с тем, что выполняющийся код обладает очень высокой степенью локальности, можно ожидать, что такая оптимизация снизит количество просматриваемых в среднем элементов списка и, тем самым, уменьшит время поиска в кэше второго уровня.

Эксперименты показали, что среднее количество просмотренных элементов списка при загрузке и последующей работе той же самой операционной

системы в результате применения данной оптимизации снизилось на порядок и стало равно 1.5. Кэш первого уровня при этом остался без изменений. Время загрузки используемой в эксперименте операционной системы снизилось в 2 раза. Также было заметно существенное улучшение времени отклика графических приложений. Время работы небольших вычислительных программ остались без изменений.

Мы отправили патч, добавляющий данную оптимизацию разработчикам QEMU, и 5 декабря 2010 года он был включен в основную ветку разработки [5].

2.2. Распределение регистров в QEMU.

В QEMU используется чрезвычайно простой и быстрый алгоритм распределения регистров. Если в некоторый момент требуется поместить некоторое значение на регистр, алгоритм выбирает «первый» свободный регистр из допустимых. Допустимость регистров определяется из ограничений самой инструкции, а также условия несовпадения выбранного регистра с зарезервированными. Если свободного регистра не нашлось, то «первый» среди допустимых сбрасывается в память и, затем, используется для хранения нового значения.

В данной части мы рассмотрим влияние порядка просмотра регистров целевой архитектуры на эффективность данного алгоритма распределения регистров. В частности, от этого порядка будет зависеть, какой регистр будет находиться как «первый» допустимый регистр при генерировании сброса регистра в память.

В QEMU версии 0.12.50, в которой начинались данные эксперименты, при генерации кода для процессоров архитектуры x86, регистры просматривались в следующем порядке при распределении: EAX, EDX, ECX, EBX, ESI, EDI, EBP. Глядя на этот порядок можно ожидать, что регистры EAX, EDX и ECX будут использоваться и сбрасываться в память значительно чаще, чем остальные, поскольку они не только являются приоритетными для алгоритма распределения регистров, но также участвуют в соглашениях о вызовах функций и к ним жёстко привязаны многие инструкции.

Действительно, эксперименты показали, что за время загрузки операционной системы на базе ядра Linux и пользовательской графической оболочки было сгенерировано 1818624 сброса регистров в память. Из них на EAX приходятся 1129590 сбросов, на EDX — 521281, на ECX — 110387, а на все остальные — 57366.

Легко видеть существенный перекос в сторону регистров EAX, ECX и EDX. Пытаясь исправить этот перекос и, за счет этого, снизить общее количество сбросов регистров в память, мы понизили приоритет регистров EAX, ECX и EDX. Таким образом, был получен следующий порядок просмотра регистров: EBX, ESI, EDI, EBP, ECX, EDX, EAX.

Эксперименты выявили следующее: за время загрузки той же самой операционной системы было сгенерировано 1327104 сброса регистров в память, из них EBX — 677100 сбросов, ESI — 406587, EDI — 140361, EAX — 49363, ECX — 38612, EDX — 15081. Результаты показывают, что общее количество сбросов регистров в память уменьшилось на 27%, а также, что количество сбросов значительно выровнялось между регистрами по сравнению с тем, что было изначально. С точки зрения производительности, наибольший эффект данная оптимизация даёт на рекурсивных программах — до 20% уменьшения времени работы. Дальнейшие изменения порядка просмотра регистров не привели к заметным улучшениям генерируемого кода.

Этот результат также был независимо получен Ричардом Хендерсоном. Сделанные им изменения были включены в основную ветку разработки и попали в QEMU версии 0.13 [6].

2.3. Машинно-независимые оптимизации над внутренним представлением в QEMU.

В результате профилирования QEMU удалось выяснить, что весь процесс динамической двоичной трансляции занимает очень мало времени по сравнению со временем, затрачиваемым на выполнение полученного кода. Это является следствием практически полного отсутствия оптимизаций во время двоичной трансляции и быстрым, но неэффективным алгоритмом распределения регистров. В QEMU присутствуют только две оптимизации: удаление мёртвого кода и подстановка констант, которая выполняется во время распределения регистров.

В данной работе мы опишем наши эксперименты по добавлению оптимизаций по продвижению констант и копий, а также сворачивания константных выражений в QEMU. Прежде чем перейти к деталям реализации данных оптимизаций, опишем внутреннее представление, используемое в QEMU, на котором эти оптимизации производятся, и источники неэффективности в генерируемом коде.

Несмотря на то, что код гостевой программы в большинстве случаев был хорошо оптимизирован компилятором, код, получаемый из него при двоичной трансляции, может быть улучшен машинно-независимыми оптимизациями. Причиной этому служат два факта:

- гостевая программа может компилироваться для одной архитектуры, а выполняться после динамической трансляции на другой,
- трансляция в QEMU производится с помощью замены каждой инструкции гостевой программы на последовательность инструкций машины, на которой программа будет выполняться.

На стыке инструкций как раз и возникает 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
```

Рассмотрим теперь особенности внутреннего представления QEMU. Код, содержащийся в блоке трансляции, представляется в виде двух массивов: в первом хранятся коды операций, а во втором — аргументы. Коды операций всегда 16-битные числа, аргументы — 32-битные или 64-битные целые числа, в зависимости от того, для какой архитектуры генерируется код. Как трактуются числа, записанные в массиве аргументов, полностью зависит от кода операции, к которому они относятся. Набор кодов операций тоже зависит от целевой архитектуры. Однако есть некоторый набор операций, которые всегда присутствуют во внутреннем представлении. Этот набор включает в себя основную целочисленную 32-битную арифметику (сложение, вычитание, умножение, сдвиги, побитовые логические операции; деление может отсутствовать), операции загрузки и сохранения значений, а также операции перехода.

При такой структуре внутреннего представления важно уметь отвечать на следующие вопросы:

- К какой операции относится данный аргумент?
- Что означает записанное в нем число?

Основным вопросом является первый, так как ответ на второй может быть получен из семантики операции. Ответ на первый получается из свойства упорядоченности: коды операции упорядочены так, как они будут выполняться, аргументы упорядочены в соответствии с порядком операций, к которым они относятся, — аргументы первой операции идут первыми в порядке, заданном семантикой этой операции, затем идут аргументы второй операции и так далее.

Таким образом, при последовательном просмотре кода в прямом или обратном порядке легко определить, к какой операции относится данный аргумент. Проблема операций с переменным числом параметров (например, вызов вспомогательной функции) решается введением фиктивных первого и последнего аргумента, в которых записано общее количество аргументов данной операции.

В таблице 1 приведён пример внутреннего представления кода, который вычисляет значение выражения r11 + 12, где r11 — регистр гостевой системы, а 12 — константа. В первой строке таблицы записаны коды операций в мнемоническом виде, во второй строке записаны их аргументы также в мнемоническом виде, а в третьей — те же самые аргументы, но в том виде, в

котором они хранятся во внутреннем представлении QEMU. Аргументы и коды операций выровнены так, чтобы аргументы операции всегда находились под её кодом. Легко видеть, что аргумент, равный 12, в случае операции mov_i32 интерпретируется как r11, а в случае операции movi_i32 — как константа 12.

опкод	mov_i32		movi_i32		add_i32		
Аргу-менты	tmp8	r11	tmp9	12	tmp8	tmp8	tmp9
	30	12	31	12	30	30	31

Таблица 1. Пример внутреннего представления QEMU.

Основным недостатком такого внутреннего представления является сложность внесения локальных изменений, поскольку они будут приводить к сдвигу обоих массивов.

Опишем детали реализации оптимизаций по сворачиванию константных выражений и продвижению копий. Обе эти оптимизации будут производиться в рамках одного базового блока. Они будут выполнены в результате одного линейного просмотра кода в прямом порядке.

Для каждой переменной внутреннего представления будем хранить её текущее состояние, которое будет 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.

Очевидно, что в любой заданной точке кода 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. Если в классе эквивалентности существуют внутренние переменные, не соответствующие регистрам эмулируемой системы, то в качестве представителя класса берётся одна из таких переменных. В противном случае берётся любая переменная из данного класса. Такие приоритеты вызваны тем, что работа с переменными, соответствующими регистрам эмулируемой системы, приводит к операциям чтения и записи структуры, содержащей значения этих регистров. Эта структура расположена в памяти реальной машины и доступ к ней может быть достаточно медленным в случае промаха мимо кэша данных первого уровня.

Остальные же переменные чаще всего располагаются на регистрах реальной машины и попадают в память только при нехватке регистров. Доступ к ним, как правило, быстрый.

Опишем саму оптимизацию. 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). 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 операция является операцией пересылки «A = B», где B — константа, то A также помечается как константа, операция заменяется на операцию пересылки константы. Если A была представителем своего класса эквивалентности, то в этом классе эквивалентности выбирается новый представитель и состояния всех переменных этого класса обновляются.

Если все аргументы арифметической операции «A = B op 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 операции, как сдвиги, необходимо обнулять старшие биты входных констант.

Сворачивание константных выражений было реализовано для основных 32-битных арифметических операций и их 64-битных аналогов.

Во всех остальных случаях, все входные переменные заменяются на представителей их классов, 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.

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

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 просто строить новый массив аргументов. Поскольку операции просматриваются в прямом порядке, это не представляет никакой алгоритмической сложности. Далее заметим, что в ходе данной оптимизации число аргументов текущей операции либо остаётся неизменным, либо уменьшается (вместе с изменением самой операции). Значит, новый массив аргументов можно строить в том же физическом массиве, что и имеющиеся аргументы, затирая уже прочитанные и, следовательно, ненужные аргументы.

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.

Мы протестировали оптимизации на целочисленных тестах из набора SPEC CPU2000 [7]. 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. 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 тесте 186.crafty наблюдается ухудшение производительности.

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

3. Оптимизация двоичной трансляции в Valgrind

Valgrind — инструментальное программное обеспечение, предназначенное для отладки и обнаружения ошибок в программе, а также для профилирования. Под названием Valgrind скрывается целое множество различных инструментов по анализу программ. Например, инструмент Memcheck анализирует утечки памяти и обращения к невыделенной памяти.

Valgrind сначала транслирует тестируемую программу в промежуточное представление (Intermediate Representation, сокращённо IR), которое машинно-независимо и находится в SSA-форме. Затем Valgrind инструментирует код дополнительными инструкциями (в зависимости от инструмента) и на последней стадии транслирует это представление обратно в машинный код.

В этой части данной работы мы опишем оптимизацию по планированию инструкций, которая была добавлена в Valgrind.

Как и в случае с QEMU, единицей двоичной трансляции является расширенный базовый блок. До и после аннотирования кода отладочными

инструкциями Valgrind производит машинно-независимые оптимизации на уровне промежуточного представления.

Из промежуточного представления в фазе выбора инструкций генерируется низкоуровневое промежуточное представление, очень близкое к ассемблеру целевой машины, но использующее виртуальные регистры в большинстве случаев. Реальные регистры используются в тех случаях, когда не предусмотрено альтернативы. Например, в архитектуре ARM, по соглашениям о вызовах, параметры в функцию передаются через первые четыре регистра (r0, r1, r2, r3). Поэтому инструкция передачи аргумента в функцию транслируется как копирование значения из виртуального регистра в соответствующий реальный.

Виртуальные регистры имеют только одно определение (впрочем, это определение может занимать несколько ассемблерных инструкций, которые обязательно выполняются до первого использования). Реальные регистры могут иметь сколько угодно определений.

После фазы выбора инструкций идёт фаза распределения регистров, которая заменяет виртуальные регистры реальными. Между этими двумя фазами нами была добавлена оптимизация по планированию инструкций. Она является реализацией алгоритма списочного планирования (list scheduling [8]), рассмотренного также в [9,10]. Эта оптимизация строит граф зависимостей между инструкциями, после чего присваивает им веса так, чтобы вес был тем больше, чем больше длина цепочки зависимости от этой инструкции до последней. После этого планировщик, в соответствии с моделью процессора, потактово выдаёт одну или несколько инструкций. Для этого он на каждом такте строит список доступных инструкций, то есть инструкций, вычисление всех аргументов которых уже завершилось. Далее из этого списка планировщик выбирает нужное количество инструкций с наивысшим приоритетом.

Опишем модель процессора ARM Cortex-A8, на котором мы тестировали нашу оптимизацию. На данных процессорах отсутствует аппаратное переупорядочивание инструкций, поэтому эффект от планирования должен быть существенен. В процессорах Cortex-A8 имеется два конвейера и, следовательно, на один такт можно помещать до двух инструкций, если они удовлетворяют ограничениям: две инструкции работы с памятью, две инструкции перехода не могут быть выданы на одном такте, а также операция умножения всегда выполняется на первом конвейере. Если между инструкциями на разных конвейерах возникает зависимость, то это может приводить к простою конвейера. В архитектуре ARM возможно адресовать 16 регистров общего назначения. Из них занято 4 регистра: r15 — счётчик команд, r14 — адрес возврата, r13 — указатель стека, r8 использует для адресации своих внутренних данных Valgrind. Помимо них, регистры r0-r3 участвуют в соглашениях о вызовах. Остальные 8 регистров могут быть использованы произвольным образом.

Поскольку количество свободных регистров ограничено, слишком агрессивное переупорядочивание инструкций приводит к сбросу регистров в память, что негативно сказывается на производительности. Поэтому планировщик оценивает на каждом такте регистровое давление (количество живых регистров в данной точке программы) [8] и старается планировать инструкции так, чтобы это давление не превышало 6 регистров. Данная оценка была получена экспериментально.

Однако эффект от одного лишь планировщика снижался по двум причинам. Во-первых, в силу архитектурных причин, в низкоуровневом внутреннем представлении в Valgrind вместе с обычными командами ассемблера ARM использовались также виртуальные команды, состоящие из нескольких реальных команд. Это снижало эффективность планировщика из-за того, что он не всегда мог оперировать реальными инструкциями и поэтому рассчитать латентность и такты процессора (см. также схожие проблемы в [11]). К тому же, виртуальные команды оперировали фиксированными реальными регистрами для промежуточных вычислений, что мешало планировщику оценивать текущее давление регистров. Поэтому пришлось реализовать дополнительное преобразование, которое работает перед планировщиком инструкций и транслирует виртуальные команды в эквивалентную последовательность реальных, заменяя эти лишние фиксированные регистры на виртуальные.

Во-вторых, алгоритм распределения регистров был устроен так, что если требовалось назначить на место очередного виртуального регистра реальный, он искал свободный реальный регистр в массиве, просматривая с начала до конца. Из-за этого, если регистр умирал на одной инструкции, он мог быть тут же распределён в инструкции, непосредственно следующей за ней. В результате между этими инструкциями возникает зависимость, и они не могут быть выполнены параллельно на двух разных конвейерах. Однако планировщик работает с виртуальными регистрами, и считает, что между этими инструкциями зависимостей нет. В сумме, это приводит к неоптимальному планированию. Поэтому, алгоритм распределения регистров был модифицирован таким образом, чтобы освобождающийся реальный регистр попадал в конец списка поиска.

Мы протестировали получившийся планировщик инструкций на целочисленных тестах из набора SPEC CPU2000 [7]. На тестах 164.gzip, 181.mcf и 256.bzip2 было получено ускорение (3.30%, 2.07% и 2.32% соответственно), а на тестах 176.gcc, 186.crafty, 253.perlbmk, 300.twolf было получено замедление (4.31%, 5.45%, 1.47% и 5.24% соответственно). Это говорит о том, что данная оптимизация выглядит многообещающей, но требует дополнительной, более тщательной настройки.

4. Заключение

В данной статье были рассмотрены некоторые направления оптимизации динамической двоичной трансляции в QEMU и Valgrind, реализованные в ИСП РАН. Наилучшего результата удалось добиться за счёт увеличения быстродействия инфраструктуры двоичной трансляции. Простые машинно-независимые оптимизации в QEMU не приносят больших результатов, но и накладные расходы на их выполнение минимальны, поэтому они полезны в среднем. Также многообещающими выглядят оптимизации по улучшению распределения регистров в QEMU и планированию инструкций в Valgrind. Данные оптимизации заслуживают более тщательного исследования и настройки.

Литература

[1] Nicolas Nethercote, Julina Seward. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. Proceedings of ACM SIGPLAN 2007 Conference on Programming Languages Design and Implementation, 2007.

[2] QEMU – Open Source Processor Emulator. http://wiki.qemu.org/Main_Page. Дата обращения: 09.12.2010.

[3] Valgrind. http://www.valgrind.org/. Дата обращения: 09.12.2010.

[4] Oprofile – A System Profiler For Linux. http://oprofile.sourceforge.net/. Дата обращения: 10.12.2010.

[5] qemu.git – the QEMU master repository. http://git.qemu.org/qemu.git/commit/?id=2c90fe2b71df2534884bce96d90cbfcc93aeedb8. Дата обращения 16.12.2010.

[6] qemu.git – the QEMU master repository. http://git.qemu.org/qemu.git/commit/?id=6648e29608ce17f6109d5696fb01f056238e2628. Дата обращения 16.12.2010.

[7] SPEC CPU2000. http://www.spec.org/cpu2000. Дата обращения 21.12.2010.

[8] Steven Muchnick. Advanced compiler design and implementation. Morgan Kaufmann Publishers Inc., 1997.

[9] А.Белеванцев, Д.Журихин, Д.Мельник. Компиляция программ для современных архитектур. Труды института системного программирования РАН, том 16, 2009, стр. 31-50.

[10] Andrey Belevantsev, Alexander Chernov, Maxim Kuvyrkov, Vladimir Makarov, Dmitry Melnik. Improving GCC instruction scheduling for Itanium. In Proceedings of GCC Developers' Summit 2005, Ottawa, Canada, June 2005, pp.1-13.

[11] Andrey Belevantsev, Maxim Kuvyrkov, Vladimir Makarov, Dmitry Melnik, Dmitry Zhurikhin. An interblock VLIW-targeted instruction scheduler for GCC. In Proceedings of GCC Developers' Summit 2006, Ottawa, Canada, June 2006, pp.1-12.