Задача локального распределения регистров во время динамической двоичной трансляции

Кирилл Батузов <batuzovk@ispras.ru>

Аннотация. QEMU — эмулятор аппаратного обеспечения, использующий динамическую двоичную трансляцию в своей работе. В данной статье рассматриваются возможные улучшения алгоритма локального распределения регистров в QEMU. Изменения были произведены на практике, и результаты показали, что дальнейшее улучшение алгоритма возможно только через изменение существующих ограничений на него.

Ключевые слова: распределение регистров, динамическая двоичная трансляция, QEMU.

1. Введение

QEMU — эмулятор аппаратного обеспечения, который позволяет запускать программы и операционные системы, написанные для одних машин, на других. QEMU имеет два режима работы: режим эмуляции системы и режим эмуляции приложения. В первом режиме эмулируется все аппаратное обеспечение, и в полученной виртуальной машине загружается своя операционная система. Во втором режиме эмулируется только процессор. Этот режим позволяет запускать отдельные пользовательские приложения, написанные для одной архитектуры, на другой [1].

В своей работе QEMU использует динамическую двоичную трансляцию. При этом гостевой код дизассемблируется в некоторое внутреннее представление, и все необходимые изменения производятся на уровне внутреннего представления. Затем из внутреннего представления генерируется машинный код для заданной архитектуры.

В данной статье рассматривается задача распределения регистров во время генерации кода по внутреннему представлению.

2. Обзор существующих алгоритмов локального распределения регистров

Задача распределения регистров состоит в назначении 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 [6]. В данной работе рассматривается только задача локального распределения регистров.

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 [3].

Задача локального распределения регистров является NP-трудной [5]. Известно несколько алгоритмов ее точного решения: через сведение к задаче линейного программирования [4] либо поиском кратчайшего пути в ациклическом взвешенном ориентированном графе [3]. 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 [4, 5]. 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. Для освобождения выбираются те регистры, ближайшее использование содержимого которых находится как можно дальше.
4. Алгоритм переходит к следующей инструкции и возвращается на шаг 1.

3. Анализ возможных улучшений существующего в QEMU алгоритма

QEMU обрабатывает гостевой код небольшими кусками — блоками трансляции. Код гостевой системы транслируется сначала во внутреннее представление, а затем из внутреннего представления в машинный код процессора, на котором запущен эмулятор. Внутри блок трансляции состоит из одного или нескольких базовых блоков и представляет собой ациклический участок кода с одним входом.

В качестве внутреннего представления в QEMU выступает массив инструкций и их аргументов. Аргументами могут быть временные, локальные, глобальные переменные или константы. Временные, локальные и глобальные переменные отличаются областью видимости: базовый блок для временных, блок трансляции для локальных и весь код для глобальных. Глобальные переменные используются, в частности, для эмуляции регистров гостевой архитектуры — каждому регистру ставится в соответствие глобальная переменная [1].

Прежде чем для инструкции внутреннего представления может быть сгенерирован код, ее операнды должны быть распределены на регистры. 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. Таким образом, переменная может находиться на регистре только внутри базового блока, а в его конце она должна быть записана обратно память.

Алгоритм распределения регистров в QEMU 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 подходящий (с учетом ограничений целевой архитектуры) свободный регистр. Если подходящих свободных регистров несколько — то берется наиболее приоритетный. Приоритеты регистров статические и выражаются в порядке их просмотра при выборе подходящего [2].
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.

Теперь рассмотрим возможности применения в QEMU алгоритмов, описанных в части 2 данной статьи.

QEMU производит 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 алгоритм FF. Для его реализации в рамках существующего алгоритма необходимо внести изменения в четвертый шаг алгоритма.

Однако наличие во внутреннем представлении QEMU 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 QEMU эти проблемы частично решены. Корректность генерируемого кода обеспечивается шагом 2 алгоритма. Неравноправие регистров, которое вносит вторая проблема, учитывается в приоритетах регистров при распределении на третьем шаге [2].

В данной статье 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 в QEMU — 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 описанным 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.

4. Изменения в алгоритме распределения регистров в QEMU

В ходе данной работы в алгоритме распределения регистров в QEMU были реализованы две эвристики, а также внесены изменения в анализ времени жизни переменных 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. Аналогичным образом для каждой инструкции внутреннего представления определяется ближайший следующий за ней вызов функции.

При распределении регистров для каждого регистра в каждый момент времени поддерживается информация, в какой инструкции внутреннего представления будет следующее использование содержимого данного регистра. Эта 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;
- 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.

5. Экспериментальные результаты

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 1 и 2 приведены результаты тестирования. Тестирование проводилось на архитектуре x86, в качестве гостевой архитектуры также использовалась x86. Первые три теста запускались в режиме эмуляции приложения, последний тест представляет собой загрузку операционной системы GNU/Linux в режиме эмуляции системы.

	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

Таб. 1. Результаты профилирования причин сброса регистров оригинального алгоритма распределения регистров в QEMU.

	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

Таб. 2. Результаты профилирования причин сброса регистров измененного алгоритма распределения регистров в QEMU.

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 этого 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 недетерменизма, поэтому 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 остальных причин приходится не более 3.5%. Таким образом, дальнейшие улучшения алгоритма, направленные на более удачный выбор регистров 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.

Снятие или ослабление ограничений в общем случае, когда про вызываемую функцию ничего не известно, невозможно. Однако в QEMU большое количество вызовов связано с небольшими вспомогательными функциями самого QEMU (например, функция helper_rdtsc, реализующая инструкцию rdtsc архитектуры x86). Если информацию про то, какие глобальные переменные используют данные функции, передать в алгоритм распределения регистров, то станет возможным сохранять только нужные глобальные переменные.

6. Заключение

В данной работе были рассмотрены возможные улучшения алгоритма локального распределения регистров в QEMU. Рассматривались как существующие алгоритмы, так и новые, специфичные для QEMU, эвристики. Выбранные эвристики были реализованы, и было проведено исследование их эффективности. По результатам этого исследования стало ясно, что дальнейшее улучшение выбора регистров не принесет результатов без снятия существующих в QEMU ограничений на алгоритм распределения регистров: он является локальным алгоритмом распределения регистров, и все глобальные переменные должны быть сохранены в память перед вызовом функции.

Список литературы

[1] QEMU – Open Source Processor Emulator. http://wiki.qemu.org/Main_Page. Дата обращения: 20.03.2012.

[2] К. Батузов, А. Меркулов. Оптимизация динамической двоичной трансляции. Труды Института системного программирования РАН, том 20, 2011 г.

[3] Wei-Chung Hsu, Charles N. Fisher, James R. Goodman. On the Minimization of Load/Stores in Local Register Allocation. IEEE Transactions on Software Engineering, vol 15, No. 10, October 1989.

[4] Vincenzo Liberatore, Martin Farach-Colton, Olrich Kremer. Evaluation of Algorithms for Local Register Allocation. Lecture notes in Computer Science, vol 1575,1999.

[5] Martin Farach, Vincenzo Liberatore. On Local Register Allocation. DIMACS Technical Report 97-33, July 1997.

[6] Альфред В. Ахо, Моника С. Лам, Рави Сети, Джеффри Д. Ульман. Компиляторы: принципы, технологии и инструментарий. Второе издание. Вильямс, 2008.

Local Register Allocation Problem in Dynamic Binary Translation

Kirill Batuzov
ISP RAS, Moscow, Russia
batuzovk@ispras.ru

Abstract. QEMU is a generic machine emulator based on a dynamic binary translation approach. This paper evaluates possible improvements to the local register allocation algorithm in QEMU. The local register allocation problem is NP-hard, but there are several approximate heuristic algorithms. Among them Furthest-First and Clean-First heuristics are known to produce a near optimal result in linear time.

Local register allocation algorithm in QEMU scans instructions in direct order. For each instruction operand that is not yet stored on a register, an empty register is allocated. If there are no empty registers, then a spill of an arbitrary register is generated. On basic block ends and helper functions' calls all global variables are also spilled.

Two improvements for the register allocation algorithm are proposed in this paper. The first is to improve spill choices by using the heuristic from the Furthest-First algorithm. The second is to aggressively spill global variables which will be spilled anyway because of helper function calls and basic block ends. Both improvements were implemented and tested on practical workloads. Profiling results show that the amount of generated spills was reduced by approximately 1%.

With these improvements, over 96% of spills are generated due to function calls and basic block ends. This implies that the further improvements to this algorithm are not possible without altering its existing constraints.

Keywords: register allocation, dynamic binary translation, QEMU

References

[1]. QEMU — Open Source Processor Emulator. http://wiki.qemu.org/Main_Page. Access date: 20.03.2012.

[2]. K. Batuzov, A. Merkulov. Optimizatsiya dinamicheskoj dvoichnoj translyatsii [Optimizations in Dynamic Binary Translation]. *Trudy ISP RAN [The Proceedings of ISP RAS]*, 2011, vol. 20, pp. 37-50 (in Russian).

[3]. Wei-Chung Hsu, Charles N. Fisher, James R. Goodman. On the Minimization of Load/Stores in Local Register Allocation. *IEEE Transactions on Software Engineering*, vol 15, No. 10, October 1989. doi: 10.1109/TSE.1989.559775

[4]. Vincenzo Liberatore, Martin Farach-Colton, Olrich Kremer. Evaluation of Algorithms for Local Register Allocation. *Lecture notes in Computer Science*, vol 1575, 1999. doi: 10.1007/978-3-540-49051-7_10

[5]. Martin Farach, Vincenzo Liberatore. On Local Register Allocation. *DIMACS Technical Report 97-33*, July 1997.

[6]. Alfred V. Aho , Monica S. Lam , Ravi Sethi , Jeffrey D. Ullman. Compilers: Principles, Techniques, and Tools (2nd Edition). Addison-Wesley Longman Publishing Co., Inc., Boston, MA, 2006.