Задача глобального распределения регистров во время динамической двоичной трансляции

К.А. Батузов <batuzovk@ispras.ru>
Институт системного программирования РАН,
109004, Россия, г. Москва, ул. А. Солженицына, д. 25

Аннотация. Распределение регистров оказывает существенное влияние на производительность генерируемого кода. В данной статье исследуется задача распределения регистров во время динамической двоичной трансляции. Так как существующие алгоритмы распределения регистров рассчитаны на использование в компиляторах, они плохо подходят для использования во время динамической двоичной трансляции. Был разработан новый алгоритм, который определяет, какие переменные должны располагаться на каких регистрах в начале и в конце базового блока (назовем эти ограничения пред- и постусловиями данного базового блока), а затем решает задачу локального распределения регистров в данных ограничениях. Для обеспечения корректности ограничений алгоритм должен работать так, чтобы бля любого базового блока b', предшествующего блоку b, постусловия блока b' совпадали с предусловиями блока b. Этого можно достичь, если выделить в графе потока управления группы дуг, на всех концах которых ограничения должны быть неизменны. Такие дуги называются точками синхронизации. Точки синхронизации являются связными компонентами в неориентированном графе, вершинами которого являются дуги графа потока управления, а ребрами соединены те дуги-вершины, которые либо входят, либо выходят из одного базового блока. Данное утверждение позволяет эффективно находить точки синхронизации. Для определения того, сколько переменных должно находиться на регистрах в точке синхронизации, количество доступных регистров оценивается с помощью регистрового давления. Затем выбираются конкретные переменные на их частоты использования в данном фрагменте кода. Алгоритм локального распределения регистров был модифицирован, чтобы использовать предусловия и обеспечивать постусловия. В статье приводится эффективный алгоритм для приведения существующего распределения в конце базового блока к требуемым постусловиям и доказывается оптимальность этого алгоритма. Применение описанного алгоритма распределения регистров привело к сокращению времени работы синтетического примера на 29.6%.

Ключевые слова: глобальное распределение регистров; динамическая двоичная трансляция; эмуляторы; QEMU.

DOI: 10.15514/ISPRAS-2016-28(5)-12

Для цитирования: К.А. Батузов. Задача глобального распределения регистров во время динамической двоичной трансляции. Труды ИСП РАН, том 28, вып. 5, 2016, стр. 199-214. DOI: 10.15514/ISPRAS-2016-28(5)-12

1. Введение

При разработке виртуальных машин возникает задача выполнения кода, написанного для одной процессорной архитектуры, на другой архитектуре. Для ее решения используется динамическая двоичная трансляция. Трансляция кода ведется небольшими фрагментами, которые называются блоками трансляции. Типичная реализация динамической двоичной трансляции предполагает сначала дизассемблирование блока трансляции в некоторое внутреннее представление, а затем генерацию машинного кода для целевой архитектуры из этого внутреннего представления. На втором этапе возникает необходимость распределения регистров. Данная статья посвящена вопросу решения задачи глобального распределения регистров во время динамической трансляции.

Распределение регистров оказывает существенное влияние на производительность полученного кода. При этом необходимо учитывать, что в случае динамической двоичной трансляции распределение регистров происходит во время работы программы, и время работы самого алгоритма должно быть как можно меньшим. Локальное распределение регистров может быть реализовано очень быстрым жадным алгоритмом [1], однако глобальное распределение регистров может давать более эффективный результирующий код. Существующие алгоритмы глобального распределения регистров (алгоритмы раскраски графов зависимостей [2], алгоритмы линейного сканирования [3]) не вполне учитывают ограничения на накладные расходы для выполнения самого алгоритма. Поэтому был разработан новый алгоритм, который выполняет глобальное распределение регистров с очень низкими накладными расходами. Данный алгоритм учитывает особенности кода, возникающего во время динамической двоичной трансляции. В своей работе он использует локальный алгоритм для распределения регистров внутри базового блока.

Полученный алгоритм был реализован в эмуляторе QEMU [4]. Этот эмулятор был выбран, так как он обладает открытым исходным кодом, а также широко применяется на практике.

В качестве внутреннего представления в QEMU используется последовательность инструкций, оперирующих с переменными трех видов и константами. Инструкции представляют собой язык ассемблера упрощенной абстрактной машины и содержат арифметические и логические операции, операции установки меток, условного и безусловного перехода на метку, вызова функций, загрузки из памяти и сохранения в память. Переменные внутреннего представления делятся на три вида: глобальные, которые

существуют все время работы эмулятора, локальные, которые существуют в рамках одного блока трансляции, и временные, которые существуют в рамках одного базового блока. Вид переменной явно указывается при ее создании.

В QEMU распределение регистров объединено с генерацией кода. Вариант промежуточного представления, в котором регистры были бы частично распределены, отсутствует. Кроме того, при разработке алгоритма время его работы является очень критичным, так как он выполняется во время выполнения программы. Таким образом, при введении любого нового внутреннего представления требуется учитывать необходимые накладные расходы на создание данного представления.

Алгоритмы глобального распределения регистров, основанные на раскраске графа зависимостей, плохо решают поставленную задачу, так как они имеют непредсказуемое время работы, что недопустимо в случае динамической двоичной трансляции. Алгоритмы линейного сканирования требуют введение еще одного полноценного внутреннего представления с распределенными регистрами, так как им требуется второй проход для устранения противоречий на границах базовых блоков.

Построим новый алгоритм, который будет минимизировать расходы на дополнительное внутреннее представление, но перед этим отметим одну особенность промежуточного представления, получаемого при динамической двоичной трансляции.

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

Алгоритм, описанный в статье [1], может быстро и эффективно распределять регистры в рамках одного базового блока. Не составляет труда немного модифицировать его так, чтобы он принимал во внимание условия на границах базового блока: какие глобальные переменные должны находиться на каких регистрах. Значит, можно построить алгоритм, являющийся комбинацией глобального и локального распределения регистров. Часть, отвечающая за глобальное распределение регистров, будет работать только с глобальными переменными и устанавливать условия на границах базовых блоков. Часть, отвечающая за локальное распределение регистров, будет осуществлять распределение регистров внутри базовых блоков с учетом условий на их границах. Условия на границах блока трансляции будут иметь вид множества пар *(v, r)*, означающих что переменная *v* должна в данной

точке находиться на регистре *r*. Если переменная не фигурирует ни в одной из пар, то считается что переменная должна располагаться в памяти.

2. Схема комбинированного алгоритма

Назовем условия на распределение регистров в начале базового блока начальными, а в конце — конечными, а совокупность начальных и конечных условий — граничными. Введем следующие обозначения.

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

Множество граничных условий C блока трансляции ТВ с графом потока управления $G = \langle B, E \rangle$ назовем корректным, если $\forall (b_1, b_2) \in E: b_1^{post} = b_2^{pre}$.

Дуги e_1и e_2 графа потока управления назовем родственными, если они выходят из одного и того же блока, либо входят в один и тот же блок. Обозначим $e_1 \sim e_2$.

Точкой синхронизации назовем не пустое множество J дуг графа потока управления, такое что

- для любых двух родственных дуг e_1 и e_2 выполнено соотношение $e_1 \in J \Leftrightarrow e_2 \in J$,
- для любых двух дуг u и v графа потока управления входящих в одну точку синхронизации существует последовательность дуг $e_{1,}e_{2,}\ldots, e_k$ таких, что $e_1 = u$, $e_k = v$, $\forall i \in [1, k-1] e_i \sim e_{i+1}$.

Утверждение Пусть дуги $e_1 = (u_{1,}v_1)$ и $e_2 = (u_{2,}v_2)$ графа потока управления принадлежат одной и той же точке синхронизации J. Тогда для любого корректного множества граничных условий C выполнены равенства $u_1^{post} = u_2^{post}$и $v_1^{pre} = v_2^{pre}$.

Это утверждение может быть легко доказано от противного на основе определения точки синхронизации.

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

Пусть дан блок трансляции ТВ с графом потока управления $G = \langle B, E \rangle$. Построим неориентированный граф $G_E = \langle E, F \rangle$, в котором $(e_{1,}e_2) \in F$ тогда и только тогда, когда $e_1 \sim e_2$.

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

Доказательство этого утверждения следует из определений точки синхронизации и компоненты связности графа.

У данного утверждения есть ряд важных следствий.

1. Любые две точки синхронизации либо совпадают, либо не пересекаются.
2. Каждая дуга принадлежит ровно одной точке синхронизации.
3. Эффективно находить и обходить точки синхронизации можно с помощью поиска в ширину, либо поиска в глубину в графе G_E.

Далее приведем псевдокод алгоритма распределения регистров и докажем его корректность. Алгоритм будет считать использовать множество точек синхронизации и функцию для вычисления требуемых граничных условий в точке синхронизации Compute-Register-Mapping(J). Реализация данной функции будет рассмотрена позже.

Все множество точек синхронизации блока трансляции TB обозначим J(TB), множество всех базовых блоков, входящих в его поток управления — B(TB).

```
procedure Combined-Reg-Alloc(TB)
      for all b ∈ B(TB) do
            b^pre ← ∅
            b^post ← ∅
      end for
      for all J ∈ J(TB) do
            regmap ←Compute-Register-Mapping(J)
            for all (src, dst) ∈ J do
                  src^post ← regmap
                  dst^pre ← regmap
            end for
      end for
end procedure
```

Утверждение Множество граничных условий, полученное в ходе работы комбинированного алгоритма распределение регистров, корректно.

Методом доказательства от противного легко показать, что для любого блока b его предусловия (и, аналогично, постусловия) не могут быть установлены при обработке двух различных точек синхронизации. Далее, из того, что при

обработке одной точки синхронизации всегда устанавливаются одни и те же граничные условия, следует корректность полученных граничных условий.

3. Выбор количества регистров для использования в граничных условиях

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

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

$$|b^{post}| + |b^{pre}| + \text{RegistersNeeded}(b) \leqslant \text{TotalRegisters}, \quad (1)$$

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

Теперь немного ослабим ограничение (1) за счет того, что возьмем множества $\text{Regs}(b^{pre})$ и $\text{Regs}(b^{post})$ пересекающимися. Для этого опишем, как эффективно из распределения b^{pre} получить распределение b^{post}. Далее, если выполнено условие

$$|b^{post}| + \text{RegistersNeeded}(b) \leqslant \text{TotalRegisters}, \quad (2)$$

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

$$|b^{pre}| + \text{RegistersNeeded}(b) \leqslant \text{TotalRegisters} \quad (3)$$

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.

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

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

- Spill(v, r) — сохранить содержимое регистра r в переменную v,

- Load(v, r) — загрузить значение переменной v в регистр r,
- Move(r1, r2) — скопировать содержимое регистра r2 в регистр r1.

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

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

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

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

Рассмотрим, как различные операции изменяют граф.

- Операция Spill может быть применена только к регистру, в котором хранится некоторая переменная. При этом дуга, выходящая из соответствующей вершины, исчезает, если она была.
- Операция Load может быть применена только к регистру, в котором не хранится никакая переменная. При этом появится дуга, выходящая из соответствующей вершины. Случай загрузки переменной, не входящей в множество $Var(b^{post})$, рассматриваться не будет.
- Операция Move может быть применена только к паре регистров (r_1, r_2) таких, что в r_1не хранится никакая переменная, а в r_2 хранится некоторая переменная. При этом если из вершины r_2 выходила дуга $e = (r_2, r)$, то она исчезнет, а вместо нее добавится дуга $e' = (r_1, r)$.

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

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

2. До тех пор, пока существует пара регистров (r_1, r_2), такая что в G_rесть дуга из r_2в r_1и нет дуги исходящей из r_1, к ним применяется операция $Move(r_1, r_2)$. В результате это операции исчезает дуга (r_2, r_1). После завершения данного шага в графе G_rне останется цепочек.

3. До тех пор, пока в графе G_r существует цикл, он «разрывается», а шаг 2 повторяется. «Разорвать» цикл можно двумя способами:
    - переместив с помощью операции Move содержимое одного из регистров, входящих в цикл, в свободный;
    - сбросив содержимое одного из регистров, входящих в цикл, в память, с помощью операции Spill.

1. Второй способ имеет смысл применять только в том случае, если свободного регистра не нашлось (то есть на регистрах из множества R хранятся |R| различных переменных). Заметим, что сбрасывать переменную на этом шаге алгоритма потребуется не более одного раза, поскольку после этого на регистрах останется |R|-1 переменная, и свободный регистр всегда найдется.

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

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

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

```
procedure Reg-Reorder(b^pre, b^post, G_r)
      for all (v, r) ∈ b^pre: v ∈ Vars(b^pre) \ Vars(b^post) do
            Spill(v, r)
      end for
      while ∃(u, v) ∈ Edges(G_r): u ≠ v do
            for all
            (r_2, r_1) ∈ Edges(G_r): ∄r_3: (r_1, r_3) ∈ Edges(G_r) do
                  Move(r_1, r_2)
            end for
            if ∃c ∈ Cycles(G_r) then
                  Break-Cycle(C)
            end if
      end while
```

```
        for all (v, r) ∈ b^post: v ∈ Vars(b^pre) \ Vars(b^pre)do
            Load(v, r)
        end for
end procedure
```

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

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

С другой стороны

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

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

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

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

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

Возможны два случая.

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

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

$$\mathrm{b}^{\mathrm{pre}} \neq \mathrm{b}^{\mathrm{post}} \wedge \mathrm{Vars}(\mathrm{b}^{\mathrm{pre}}) = \mathrm{Vars}(\mathrm{b}^{\mathrm{post}}) \wedge \mathrm{Regs}(\mathrm{b}^{\mathrm{pre}}) = \mathrm{Regs}(\mathrm{b}^{\mathrm{post}}) = \mathrm{R} \quad (4)$$

Вернемся к вычислению величин L, S и M.

На первом шаге алгоритма произойдет $|\mathrm{Vars}(\mathrm{b}^{\mathrm{pre}}) \setminus \mathrm{Vars}(\mathrm{b}^{\mathrm{post}})|$ операций Spill.

На втором шаге алгоритма произойдет $|\mathrm{Edges}(\mathrm{G_r})| - 1$ операций Move в случае выполнения условия (4), либо $|\mathrm{Edges}(\mathrm{G_r})|$ в противном случае.

На третьем шаге алгоритма произойдет $|\mathrm{Cycles}(\mathrm{G_r})| - 1$ операций Move и одна операция Spill в случае выполнения условия (2), либо $|\mathrm{Cycles}(\mathrm{G_r})|$ операций Move в противном случае.

На четвертом шаге алгоритма произойдет $|\mathrm{Vars}(\mathrm{b}^{\mathrm{post}}) \setminus \mathrm{Vars}(\mathrm{b}^{\mathrm{pre}})| + 1$ операций Load в случае выполнения условия (4), либо $|\mathrm{Vars}(\mathrm{b}^{\mathrm{post}}) \setminus \mathrm{Vars}(\mathrm{b}^{\mathrm{pre}})|$ в противном случае.

Значит, если выполнено условие (4), то

$$\mathrm{L} = |\mathrm{Vars}(\mathrm{b}^{\mathrm{post}}) \setminus \mathrm{Vars}(\mathrm{b}^{\mathrm{pre}})| + 1 = 1,$$
$$\mathrm{S} = |\mathrm{Vars}(\mathrm{b}^{\mathrm{pre}}) \setminus \mathrm{Vars}(\mathrm{b}^{\mathrm{post}})| + 1 = 1,$$
$$\mathrm{M} = |\mathrm{Edges}(\mathrm{G_r})| + |\mathrm{Cycles}(\mathrm{G_r})| - 2.$$

Иначе

$$\mathrm{L} = |\mathrm{Vars}(\mathrm{b}^{\mathrm{post}}) \setminus \mathrm{Vars}(\mathrm{b}^{\mathrm{pre}})|,$$
$$\mathrm{S} = |\mathrm{Vars}(\mathrm{b}^{\mathrm{pre}}) \setminus \mathrm{Vars}(\mathrm{b}^{\mathrm{post}})|, .$$
$$\mathrm{M} = |\mathrm{Edges}(\mathrm{G_r})| + |\mathrm{Cycles}(\mathrm{G_r})|$$

Пусть есть два алгоритма переупорядочивания регистров. Алгоритм $\mathrm{A_1}$ генерирует $\mathrm{L_1}$ операций Load, $\mathrm{S_1}$ операций Spill и $\mathrm{M_1}$ операций Move. Алгоритм $\mathrm{A_2}$ генерирует $\mathrm{L_2}$ операций Load, $\mathrm{S_2}$ операций Spill и $\mathrm{M_2}$ операций Move. Будем считать, что алгоритм $\mathrm{A_1}$эффективнее алгоритма $\mathrm{A_2}$тогда и только тогда, когда $\mathrm{L_1} + \mathrm{S_1} < \mathrm{L_2} + \mathrm{S_2}$ либо $\mathrm{L_1} + \mathrm{S_1} = \mathrm{L_2} + \mathrm{S_2}$ и $\mathrm{M_1} < \mathrm{M_2}$.

Утверждение Описанный алгоритм переупорядочивания регистров является наиболее эффективным среди всех алгоритмов переупорядочивания регистров, использующих только регистры из множества $\mathrm{R} \supseteq \mathrm{Regs}(\mathrm{b}^{\mathrm{pre}}) \cup \mathrm{Regs}(\mathrm{b}^{\mathrm{post}})$.

Доказательство (от противного)

Пусть существует алгоритм A', который эффективнее описанного. Описанный алгоритм генерирует L операций Load, S операций Spill и M операций Move. Алгоритм A' генерирует L' операций Load, S' операций Spill и M' операций Move. Отдельно рассмотрим два случая: когда условие (4) выполняется, и когда оно не выполняется.

Предположим, что условие (4) не выполняется. Поскольку переменные из множества $\mathrm{Vars}(\mathrm{b}^{\mathrm{pre}}) \setminus \mathrm{Vars}(\mathrm{b}^{\mathrm{post}})$ должны быть сохранены, то

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

Аналогично

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

Поскольку $\mathrm{L}' + \mathrm{S}' \leqslant \mathrm{L} + \mathrm{S}$, во всех неравенствах достигается равенство. Значит в алгоритме A' никаких сохранений, кроме сохранений переменных из множества $\mathrm{Vars}(\mathrm{b}^{\mathrm{pre}}) \setminus \mathrm{Vars}(\mathrm{b}^{\mathrm{post}})$ нет. Данные сохранения не меняют граф $\mathrm{G_r}$. Значит количество дуг и циклов в графе $\mathrm{G_r}$, может уменьшаться только за

счет операций Move. Так как целевой регистр операции Move должен быть свободным, каждая операция может либо уменьшать количество циклов в графе на 1, либо уменьшать количество ребер в графе на 1. Значит

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

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

Осталось рассмотреть второй случай. Пусть условие (4) выполняется.

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

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

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

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

Величина RegistersNeeded(b), используемая в условиях (2) и (3), априори неизвестна и не может быть легко вычислена. Оценим ее приближенно. Для этого введем понятие регистрового давления.

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

$$\mathrm{RegPressure}(I, b) = |\mathrm{LiveVariables}(I, b)| + |\mathrm{ExtraRegisters}(I)|$$

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

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

$$\mathrm{RegPressure}(b) = \max_{I \in b}(\mathrm{RegPressure}(I, b))$$

Регистровое давление является оценкой сверху на величину RegistersNeeded, поэтому перепишем условия (2) и (3) используя новое определение:

$$|b^{post}| \leqslant TotalRegs - RegPressure(b), \quad (5)$$

$$|b^{pre}| \leqslant TotalRegs - RegPressure(b) \quad (6)$$

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

$$TotalRegs - \max_{e \in J}(RegPressure(src(e)), RegPressure(dst(e)))$$

регистров, то одно из условий (5) и (6) будет обязательно выполнено во всех прилегающих к J базовых блоках.

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

Для выбора конкретных переменных для граничных условий введем функцию полезности включения переменной x в граничные условия точки синхронизации J: Usefullness(x,J). Функция полезности будет вычислять по следующей формуле:

$$Usefullness(x, J) = |e: e \in J \wedge x \in Vars(src(e)) \wedge x \in Vars(dst(e))|.$$

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.

Итоговый алгоритм вычисления граничных условий в точке синхронизации J выглядит следующим образом.

```
function Compute-Register-Mapping(J)
        p ← max_{e∈J}(RegPressure(src(e)), RegPressure(dst(e)))
        n ← TotelRegs − p
        result ← ∅
        priority ←Compute-Usefullness(J)
        for i ← 1, ndo
                result ← result ∪ {(priority[i], register[i])}
        end for
        return result
end function
```

4. Экспериментальные результаты

Описанный алгоритм был реализован в QEMU версии 1.0. При хранении графа потока управления учитывается такая особенность QEMU, что из базового блока может исходить не более двух дуг. Вычисление точек синхронизации, сбор информации о регистровом давлении в базовом блоке, вычисление функции полезности переменной в точке синхронизации и запись граничных условий делается с помощью обхода графа $G_E = \langle E, F \rangle$ в глубину. Строить его в явном виде не нужно. Все вершины, смежные с вершиной e=(src,dst) графа G_E, можно найти просмотрев все исходящие из вершины src графа G дуги, и все входящие в dst.

Тестирование данного алгоритма начнем с модельного примера. В качестве такого примера возьмем последовательность инструкций архитектуры ARM

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

выполненную в цикле $b0000000_{16}$ раз. Условные инструкции обеспечат наличие нескольких базовых блоков внутри одного блока трансляции. Гостевые регистры r1 и r2 станут глобальными переменными, используемыми в нескольких базовых блоках. Выполнение данного фрагмента большое количество раз в цикле позволит провести измерение времени. Результаты тестирования приведены в таблице 1. Ускорение получается за счет того, что переменные, соответствующие гостевым регистрам r1 и r2 в случае глобального распределения регистров загружаются на регистры основной машины один раз в начале обработки инструкции addlt, а в случае локального — перед обработкой каждой инструкции. Таким образом экономится по 6 загрузок и сохранений на каждом выполнении приведенного фрагмента.

Табл. 1. Результаты тестирования глобального распределения на модельном примере

Table 1. Global register allocation testing results for model example

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

Для того, чтобы определить изменение производительности на реальных программах были использованы тесты из набора SPEC CINT2000. Тестирование на них показало небольшое падение производительности на большинстве из тестов. Результаты тестирования приведены в таблице 2. На момент написания данной статьи удалось установить две существенные

причины падения производительности, которые планируется устранить в ходе дальнейших работ над данным алгоритмом.

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

- блок трансляции состоял из нескольких базовых блоков,
- некоторые глобальные переменные использовались в нескольких базовых блоках.

Табл. 2. Результаты тестирования глобального распределения на тестах из набора SPEC CINT2000

Table 2. Global register allocation testing results for SPEC CINT2000 benchmarks

Тест	Без глобального распределения регистров (секунд)	С глобальным распределением регистров (секунд)	Ускорение
164.gzip	81.004	81.840	-1%
175.vpr	144.56	148.164	-2.5%
256.bzip2	42.496	40.580	4.5%
300.twolf	36.508	36.544	0%

Табл. 3. Результаты профилирования глобального распределения решистров

Table 3. Profiling results for global register allocation

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

Как удалось выяснить с помощью профилирования, таких блоков всего около 10% от общего числа блоков трансляции. Результаты профилирования для трех тестов приведены в таблице 3. Данная проблема может быть устранена за

счет увеличения блоков трансляции таким образом, чтобы они могли включать несколько гостевых базовых блоков.

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

Список литературы

[1]. Батузов К. Задача локального распределения регистров во время динамической двоичной трансляции. Труды ИСП РАН, том 22, 2012 г., стр. 67-76. DOI: 10.15514/ISPRAS-2012-22-5

[2]. Chaitin G. J. Register allocation & spilling via graph coloring. Proceedings of the 1982 SIGPLAN symposium on Computer construction. SIGPLAN '82. New York, USA: ACM, 1982, pp. 98-105.

[3]. Poletto Massimiliano Sarkar Vivek. Linear san regsiter allocation. ACM Transaction on Programming Languages and Systems. 1999 Vol. 21, no 5, pp 895-913.

[4]. Bellard Fabrice. QEMU, a fast and portable dynamic translator. Proceedings of the annual conference on USENIX Annual Technical Conference. ATEC '05. Berkley, USA: USENIX Association, 2005, pp. 41-46.

Global register allocation during dynamic binary translation

K.A. Batuzov <batuzovk@ispras.ru>

[1] *Institute for System Programming of the Russian Academy of Sciences, 25, Alexander Solzhenitsyn st., Moscow, 109004, Russia.*

Abstract. Register allocation have a significant impact on performance of generated code. This paper explores the problem of global register alloction during dynamic binary translation. Since existing algorithms are designed for compilers and they are not always suitable for dynamic binary translation, a new global register allocation was developed. This algorithm decides which variables should reside on which registers before and after each basic block (called pre- and post- conditons of this basic block) and solves local register allocation problem in these constraints. To ensure that pre- and post- conditions of different basic blocks are consistent the algorithms must choose these conditions in such a way that for each basic block *b'* that precides arbitrary block *b* it's postconditions are the same as preconditions of *b*. This can be achieved by finding groups of arcs in control flow graph on which these conditions should remain the same (let's call them synchronisation points) and then choosing conditions for each such synchronisation point. Synchronization points are

connected components in graph G_E which is a graph where arcs of original CFG are vertices and edges connect arcs which start or end in the same basic block. This gives an efficient way to find synchronization points. To choose which variables should reside on registers in each synchronization point the amount of available register is estimated using register pressure in incident basic blocks. Then actual variables a picked based on how often they are used in incident basic blocks. Finally the local register allocation algorithm is modified to use precondition and ensure post conditions of the basic block. The efficient algorithm to convert existing allocation to the desired postcondition at the end of basick block is presented with the proof of that it's optimal in terms of generated spills, reloads and register moves. The described algorithm showed 29.6% running time decrease on the synthetic example. The needed modifications of the algorithm to efficiently run on real life application are explored.

Keywords: global register allocation, dynamic binary translation, QEMU..

DOI: 10.15514/ISPRAS-2016-28(5)-12

For citation: K.A. Batuzov. Global register allocation during dynamic binary translation. Trudy ISP RAN/Proc. ISP RAS, vol. 28, issue 5, 2016, pp. 199-214 (in Russian). DOI: 10.15514/ISPRAS-2016-28(5)-12

References

[1]. Batuzov K. Local register allocation during dynamic binary translation. Trudy ISP RAN/Proc. ISP RAS, vol. 22, 2012, pp. 67-76 (in Russian). DOI: 10.15514/ISPRAS-2012-22-5

[2]. Chaitin G. J. Register allocation & spilling via graph coloring. Proceedings of the 1982 SIGPLAN symposium on Computer construction. SIGPLAN '82. New York, USA: ACM, 1982, pp. 98-105.

[3]. Poletto Massimiliano Sarkar Vivek. Linear san regsiter allocation. ACM Transaction on Programming Languages and Systems. 1999 Vol. 21, no 5, pp 895-913.

[4]. Bellard Fabrice. QEMU, a fast and portable dynamic translator. Proceedings of the annual conference on USENIX Annual Technical Conference. ATEC '05. Berkley, USA: USENIX Association, 2005, pp. 41-46.