ТОЧНЫЙ ПОДХОД К ОДНОВРЕМЕННОМУ РЕШЕНИЮ ЗАДАЧ ВЫБОРА И ПЛАНИРОВАНИЯ ИНСТРУКЦИЙ В ГЕНЕРАТОРЕ КОДА

В.А. Галатенко
Научно-исследовательский институт системных исследований РАН
Россия, 117218, Москва, Нахимовский пр-т, 36, к. 1
E-mail: galat@niisi.msk.ru

С.В. Самборский
Научно-исследовательский институт системных исследований РАН
Россия, 117218, Москва, Нахимовский пр-т, 36, к. 1
E-mail: sambor @niisi.msk.ru

Н.И. Вьюкова
Научно-исследовательский институт системных исследований РАН
Россия, 117218, Москва, Нахимовский пр-т, 36, к. 1
E-mail: niva@niisi.msk.ru

Работа посвящена проблеме совместного решения задач выбора и планирования инструкций в оптимизирующем компиляторе. Предлагается использовать стандартные методы выбора инструкций с получением нескольких вариантов последовательностей команд; планирование инструкций может быть реализовано как решение задачи целочисленного линейного программирования (ЦЛП), в которой учтены зависимости по данным, ограничения по аппаратным ресурсам и ограничения, обеспечивающие выбор оптимального варианта набора команд.

EXACT APPROACH TO SIMULTANEOUS SOLUTION OF INSTRUCTION SELECTION AND SCHEDULING TASKS IN A CODE GENERATOR / V.A. Galatenko (Scientific Research Institute for System Analysis, Russian Academy of Sciences, 36-1 Nakhimovskiy pr., Moscow 117218, Russia, E-mail: galat@niisi.msk.ru), S.V. Samborskiy (Scientific Research Institute for System Analysis, Russian Academy of Sciences, 36-1 Nakhimovskiy pr., Moscow 117218, Russia, E-mail: sambor@niisi.msk.ru), N.I. Viukova (Scientific Research Institute for System Analysis, Russian Academy of Sciences, 36-1 Nakhimovskiy pr., Moscow 117218, Russia, E-mail: mailto:niva@niisi.msk.ru). The work is devoted to the problem of cooperative solution of the instruction selection and scheduling tasks in an optimizing compiler. The authors propose to use standard instruction selection methods with consideration of several variants of instruction sequences; scheduling is then implemented as solution of an Integer Linear Programming (ILP) task which takes into account data dependencies, restrictions on hardware resource and restrictions that provide selection of an optimal variant of set of instructions.

Актуальность и постановка задачи

Современные программно-аппаратные решения для встроенных систем нередко предполагают расширение системы команд процессора инструкциями, специфическими для некоторого класса задач, что может значительно увеличить быстродействие и сократить энергопотребление. Возможен и обратный вариант, когда из системы команд исключаются некоторые инструкции с целью упростить процессор и уменьшить его стоимость.

Характерными чертами подобных разработок является то, что, во-первых, система команд (по крайней мере ее специфическая часть) неоднократно меняется в процессе разработки программного обеспечения ("встречная оптимизация", [1]). Во-вторых, специфичные для области применения команды могут плохо соответствовать операциям языков высокого уровня (например, вычисление среднего арифметического, или вычитание половины, или определенная перестановка байт в слове).

Это означает, что разработка компилятора для подобного процессора является нетривиальной задачей. Отказ от использования новых специфических команд при генерации кода может быть невозможен (если из стандартного набора инструкций исключены незаменимые команды), либо нерационален, так как в этом случае большая часть прикладной программы должна будет реализована на ассемблере. Дополнительный аргумент за использование компилятором новых команд — то, что эти команды могут быть успешно использованы компилятором не только для той цели, для которой они были изначально предусмотрены.

Генерация специфичных для предметной области команд в компиляторе определяет два актуальных требования: во-первых, требуется простота модификации генератора кода при изменении в системе команд. Во-вторых, необходима возможность легкого подключения сложных и нестандартных инструкций, не соответствующих операциям языка высокого уровня. При этом остаются высокие требования к эффективности полученного программного кода.

Дополнительной проблемой является то, что на этапе выбора инструкций невозможно достоверно выяснить, какое из найденных решений окажется лучше, так как может оказаться, что дорогие инструкции могут быть запущены параллельно с другими инструкциями, а дешевые нет. Таким образом, окончательно качество выбора инструкций может быть оценено только после фазы планирования инструкций (code scheduling).

Рассмотрим простой пример: обнулить целочисленный регистр можно при помощи команды, пересылающей значение выделенного регистра:

```
mov r15, zero
```

Другой вариант — вычесть из регистра его значение:

```
sub r15, r15
```

В первом случае выполняется команда пересылки, а во втором арифметическая команда. Во многих процессорах возможно параллельно исполнять команды пересылки и арифметические команды; следовательно, выбор удачного варианта возможен только после того, как станут известны действия, которые могут выполняться параллельно с обнулением регистра, т.е. на стадии планирования команд.

Один из подходов к этим проблемам рассматривается в данном докладе.

Методы выбора инструкций в генераторе кода

В генераторах кода современных компиляторов подбор команд процессора для участка программы выделяется в отдельную подзадачу (instruction selection). Используются два основных подхода, которые условно можно назвать: "императивный" и "декларативный".

Императивный подход основан на внутреннем представлении компилируемой программы и состоит в наборе правил, которые сопоставляют возможным фрагментам этого внутреннего представления одну или несколько машинных команд. Более точно, правило может соответствовать не одной вершине графа внутреннего представления, а некоторому подграфу, кроме того, применение правила способно затрагивать окрестности обрабатываемого узла или подграфа. Возможно наличие нескольких правил для одного фрагмента с разными условиями применения и ценами, что позволяет выбирать более эффективный вариант, но только среди заранее заготовленных.

Для императивного выбора инструкций характерна прагматичность: правила заводятся только для тех узлов или подграфов внутреннего представления, которые получаются при компиляции. Использованы могут быть только те команды процессора, которые явно указаны в каком-либо правиле. Примером компилятора, применяющего (в основном) императивный подход к выбору инструкций, является свободно распространяемый компилятор GCC [2].

Преимущество императивного подхода — скорость компиляции, так как задача выбора фактически решена заранее, на этапе построения компилятора. Кроме того, несложно ввести небольшие изменения, например, отменить использование инструкции, в реализации которой выявлена ошибка. Можно также сразу генерировать готовую последовательность команд для операций над типами, которые напрямую не соответствуют архитектуре процессора (например, для действий в арифметике повышенной точности).

Очевидные недостатки императивного подхода: необходимо большое количество правил, которые отражают не только свойства целевого процессора, но и детали архитектуры конкретного компилятора, связанные со структурой внутреннего представления. При ручном или полуавтоматическом составлении этих правил неизбежны ошибки, которые сложно выявить тестированием. С одной стороны, редко применяемое правило может быть ошибочно, а с другой, для редко встречающегося фрагмента внутреннего представления может не найтись правила трансляции. В любом случае задача выбора инструкций решается не самым эффективным способом, так как эффективные правила заготавливаются только на часто встречающиеся фрагменты внутреннего представления. К тому же, подобный подход практически исключает генерацию нестандартных инструкций, поскольку сложно предусмотреть правила на все случаи, когда эти инструкции могли бы быть эффективно использованы.

Альтернативой императивному является декларативный подход, основанный на описании системы команд процессора. Для каждой команды процессора формулируется ее действие и автоматически создается правило, содержащее условие применения и сокращающее внутреннее представление оставшейся программы при генерации данной команды.

При декларативном подходе достаточно внести изменения в описание системы команд процессора, чтобы компилятор начал автоматически генерировать новый код. Не важно, насколько сложное и нестандартное действие совершает новая инструкция, — она будет использована там, где это окажется выгодно. Также можно рассчитывать на более удачный выбор инструкций, так как компилятор не ограничен заранее составленными шаблонами.

С другой стороны, декларативный подход предполагает необходимость в компиляторе сводить операции над сложными типами данных к операциям над типами, непосредственно поддерживаемыми целевой архитектурой (нельзя просто вставить заготовку из десятка инструкций).

Минус декларативного подхода — меньшая скорость компиляции, так как задача нахождения оптимального набора команд в общем случае является NP-полной [3]. Следовательно, при практической реализации потребуется отказаться от поиска точного оптимума и использовать приближенные методы и эвристики. На самом деле и не имеет смысла искать в данном случае точное решение, поскольку, как замечено во введении, оценить качество выбора инструкций можно только при планировании кода.

В декларативном выборе инструкций используются два основных подхода. В рамках первого подхода — BUMP (bottom-up pattern matching) — как правило, используется частный случай, основанный на методах синтаксического разбора (BURG, [4]). В этих алгоритмах командам процессора соответствуют правила вывода в некоторой особого вида контекстно-свободной грамматике (без однозначности синтаксического разбора). Применение каждого правила имеет свою цену, цена дерева разбора — сумма цен примененных правил, что дает возможность находить оптимальный (самый дешевый) разбор методом динамического программирования.

Второй подход — BURS (bottom-up rewrite system, [5]) — похож на BURG, но вместо правил контекстно-свободной грамматики в нем применяются произвольные правила переписывания. Это, с одной стороны, позволяет учитывать контекст преобразования, а также использовать алгебраические тождества, такие как коммутативность и ассоциативность. С другой стороны, алгоритм сильно усложняется, а на правила переписывания приходится накладывать ограничения, чтобы гарантировать завершение процедуры. Также приходится ограничивать перебор путем использования эвристик.

В обоих (BURG и BURS) подходах алгоритм распадается на две фазы: разметка, соответствующая первой части алгоритма динамического программирования, и выделение оптимального решения. На первой фазе в каждом узле для каждого возможного результата применения правил грамматики запоминается наиболее дешевый способ получения данного результата. Во второй фазе на основе разметки выделяется оптимальное дерево разбора и формируется соответствующий набор инструкций.

Заметим, что дополнительные возможности, которые обеспечиваются в BURS-подходе гибкими правилами переписывания, можно привнести в BURG-подход, сохранив при этом его преимущества: гарантированное завершение и разумное время работы. Учет контекста можно эмулировать, используя грамматику с атрибутами, а алгебраические тождества эффективнее вводить не в виде дополнительных правил переписывания, а включить непосредственно в механизм применения правила. Хорошей аналогией может послужить то, как в системах автоматического доказательства [6] вместо явных аксиом коммутативности и ассоциативности используют специальный алгоритм унификации выражений, учитывающий эти свойства определенных бинарных операций.

Отложенный выбор инструкций

Как уже отмечалось, рассмотренная выше постановка задачи выбора инструкций не вполне корректна, поскольку достоверно определить, какое из возможных решений окажется лучше, можно только после фазы планирования инструкций (code scheduling). В настоящей работе предлагается подход, при котором на первой фазе алгоритма выбора инструкций запоминается не самый дешевый способ получения определенного результата, а все варианты последнего шага синтаксического разбора или переписывания, порождающие данный результат. Это не ведет к экспоненциальному росту хранимых данных, так как запоминаются только варианты последнего шага, а не последовательности шагов. Таким образом, получается не одно решение задачи выбора инструкций, а все возможные (с некоторыми ограничениями) решения, так что выбор наилучшего из них откладывается. Количество решений может экспоненциально зависеть от длины программы, но они кратко закодированы в структуре полиномиального (по крайней мере для BURG) размера.

Можно рассматривать варианты, запоминаемые на первой стадии, как номера ветвей в системе хранения версий подобной CVS [7]. Это позволяет указать для каждой команды один или несколько номеров альтернатив (с подвариантами и т.д.), в которых эта команда присутствует. После этого остается решать задачу планирования для набора команд, согласованных по версиям.

Более привлекательным, с нашей точки зрения, является рассматриваемый далее подход к планированию, основанный не на нумерации версий, а только на зависимостях по данным между командами.

Задача планирования команд может решаться для линейных участков кода или для тел циклов, в последнем случае она называется программной конвейеризацией.

Планирование инструкций работает с графом зависимостей по данным (DFG — Data Flow Graph). При отложенном выборе можно построить "DFG с альтернативами". Его вершинами будут все команды (в одном экземпляре), которые попали хотя бы в один вариант при выборе инструкций. Естественно, что одинаковые команды, возникающие в разных частях программы, не отождествляются, совпадение команд определяется не только их природой, но и аргументами. Все обычные DFG, соответствующие разным выборам инструкций, окажутся подграфами этого графа.

Отметим, что можно забыть о происхождении включенных в DFG команд, и рассматривать команды из разных альтернатив как "конкурирующих производителей" в том случае, если их результаты совпадают как функции входных данных (и также совпадает тип и место хранения). То естественное ограничение, что входные данные для запланированной команды должны быть заранее вычислены, автоматически гарантирует, что будет выбрано согласованное множество команд.

Проиллюстрируем вышесказанное одним очень важным примером, связанным с использованием инструкции "сложения с умножением" (`madd`). Эта команда выполняется над тремя регистрами (`r1`, `r2` и `r3`). Ее действие эквивалентно следующему выражению языка C:

```
r1 += r2*r3
```

Время исполнения этой инструкции меньше чем суммарное время исполнения двух отдельных команд умножения и сложения. Поэтому возникает предположение, что компилятор обязан безусловно предпочитать команду `madd` паре отдельных команд.

Оказывается, что подобная стратегия не всегда оптимальна, причем она может не быть оптимальна для важнейшей и простейшей прикладной задачи — скалярного умножения действительных векторов. Рассмотрим цикл:

```
for (i=0; i<1000; i++)
  res += A[i]*B[i];
```

Игнорируя проверку условия цикла и не рассматривая инициализацию, получаем следующее тело цикла с использованием команды `madd` (ассемблерные команды записаны псевдокодом, чтобы не отвлекаться на конкретный синтаксис):

```
{
  1  a = A[i]
  2  b = B[i]
  3  res += a*b
  4  i++
}
```

Построим для тела цикла граф зависимостей по данным (DFG). Он очень простой: команда номер 3 зависит от команд 1 и 2, команды 1 и 2 зависят от команды 4 предыдущей итерации, и, самое главное, команда 3 (`madd`) зависит от самой себе предыдущей итерации, так как регистр `res` для нее и входной и выходной. Команда 4 (`add`) также зависит от себя самой в предыдущей итерации, по тем же причинам для регистра `i`. Таким образом, в DFG есть два цикла, причем оба состоят из одной дуги, отвечающей зависимости команды от самой себя в предыдущей итерации. Это означает, что нельзя сделать "интервал запуска" (время за которое исполняется одна итерация цикла) меньше времени выполнения любой из команд 3 (`madd`) и 4 (`add`) (см. [8]). Но инструкция `madd` содержит умножение и может выполняться существенно дольше чем все остальные команды загрузки и сложения в этом цикле.

Заметим, что возможность исполнять несколько операций умножения параллельно (на одном конвейере или на различных АЛУ) в данном случае нисколько не исправляет ситуацию. Команды `madd` должны исполняться последовательно по причине зависимости друг от друга (считаем что процессор не содержит механизмов трансляции команд в набор микроопераций).

Что изменится, если вместо одной команды `madd` использовать две отдельные команды умножения и сложения:

```
{
  1  a = A[i]
  2  b = B[i]
  3  tmp = a*b
  4  res += tmp
  5  i++
}
```

В DFG тела этого цикла также присутствуют два цикла единичной длины — команды сложения 4 и 5 зависят от самих себя в предыдущей итерации. Для большинства процессорных архитектур время выполнения команды сложения существенно меньше чем "умножения со сложением", следовательно, возможно лучше конвейеризовать цикл, за счет перекрытия по времени операций умножения из соседних итераций цикла.

Предлагаемый подход заключается в том, чтобы отложить выбор между использованием одной команды `madd` и пары команд (умножение и сложение), включив в тело цикла оба альтернативных варианта:

```
{
  1  a = A[i]
  2  b = B[i]
  3  res += a*b
  4  tmp = a*b
  5  res += tmp
  6  i++
}
```

Разумеется, если запланировать выполнение всех этих команд, результат получится неверный (получим удвоенное скалярное произведение). Поэтому приходится отказываться от стандартного условия при программной конвейеризации, что все команды должны попасть в тело конвейеризованного цикла ровно в одном экземпляре. В данном примере достаточно указать что должна быть запланирована в точности одна из двух команд 3 и 5. Команды 1, 2 и 6 будут гарантировано включены в выходной цикл, так от них косвенно зависит и команда 3 и команда 5, а команда 4 будет запланирована если запланирована команда 5. (Если запланирована команда 3, а не 5, то команда 4 может быть включена — код останется корректным, но скорее всего это не произойдет, так как лишнее умножение не может присутствовать в оптимальном расписании.)

Для планирования инструкций обычно применяются эффективные эвристические методы такие как планирование по списку [9] (list scheduling) для линейного участка, планирование по модулю (modulo scheduling) [8] для тел циклов. Они не очень хорошо подходят для планирования, совмещенного с выбором инструкций. Возможно, для целей данной работы были бы более удачны их варианты с возвратами или с перебором вариантов оформленным в стиле эволюционных алгоритмов, в котором могла бы явно отражаться конкуренция команд из разных альтернатив, производящих одно и то же значение.

С другой стороны, для отложенного выбора инструкций хорошо подходят точные методы, основанные на сведении задачи планирования к некоторой задаче математического программирования: к целочисленному линейному программированию (ЦЛП) или к определению возможной истинности логической формулы без кванторов (SAT). В частности, хорошо исследовано сведение задачи планирования инструкций к ЦЛП [10], при этом возможно одновременно учитывать ограничения на число используемых регистров, что гарантирует успех их последующего распределения. Это важно, потому что альтернативные варианты решения задачи выбора инструкций могут требовать разного количества временных регистров разного типа.

Оказалось, что наличие альтернатив в DFG не очень сильно меняет сведение задачи планирования к ЦЛП. Главное, чтобы выходные значения были вычислены какими-то командами, а для каждой команды, запуск которой планируется на определенном такте, были заранее вычислены (не важно как) входные значения. Более того, не обязательно даже, чтобы некоторое необходимое значение вычислялось только один раз — иногда может быть удобно вычислить его повторно, вместо того чтобы занимать регистр для его хранения, причем повторное вычисление может быть сделано альтернативным способом.

Заключение

Декларативные методы выбора инструкций в генераторе кода позволяют эффективно получить не только одно решение данной задачи, но и все множество решений в компактном виде.

Точный подход к планированию инструкций можно распространить на планирование наборов инструкций с альтернативами. При этом нет необходимости хранить информацию о происхождении инструкции, т.е. о том, к какому варианту она принадлежит, достаточно ее зависимостей по данным.

Тем самым возможно одновременное решение задач выбора инструкций, планирования инструкций и резервирования аппаратных регистров, что позволяет построить лучший исполняемый код, чем при их последовательном решении. В частности, совместное решение этих задач позволяет точнее определить, в каких случаях оправдано применение специфических инструкций процессора с нетривиальными условиями планирования.

Список литературы

1. Методы встречной оптимизации. М.:НИИСИ РАН, 2005.
2. GCC, the GNU Compiler Collection. http://gcc.gnu.org.
3. М. Гэри, Д. Джонсон. Вычислительные машины и труднорешаемые задачи. — Москва, "Мир", 1982.
4. Fraser, C. W., Henry, R. R., and Proebsting, T. A. BURG — Fast optimal instruction selection and tree parsing. SIGPLAN Notices 27, 4 (Apr. 1992), 68-76.
5. A. Nymeyer, J.-P. Katoen. Code generation based on formal burs theory and heuristic search. Acta Informatica, Vol. 34, pages 597-635, 1997.
6. Ч. Чень, Р. Ли. Математическая логика и автоматическое доказательство теорем. М.: Наука, 1983.
7. CVS. http://ru.wikipedia.org/wiki/CVS.
8. Н. И. Вьюкова, В.А. Галатенко, С.В. Самборский. Программная конвейеризация циклов методом планирования по модулю. — Программирование 6, 2007г.
9. List Scheduling. http://en.wikipedia.org/wiki/List_scheduling.
10. С.В. Самборский. Формулировка задачи планирования линейных и циклических участков кода. — Программные продукты и системы. 3(79), стр. 12-16, 2007г.