Поддержка команд с условным выполнением в селективном планировщике команд

Дмитрий Мельник <dm@ispras.ru>,
Александр Монаков <amonakov@ispras.ru>
Арутюн Аветисян <arut@ispras.ru>

Аннотация. Условное выполнение — аппаратная возможность, реализованная в некоторых процессорах, позволяющая аннотировать команды условным предикатом, при этом команда исполняется только в случае истинности предиката. В данной работе предлагается метод для поддержки условного выполнения во время планирования команд, а также рассматриваются преимущества данного подхода по сравнению с отдельной оптимизацией, работающей до планирования команд. Предложенный метод был реализован в селективном планировщике в компиляторе GCC. Тестирование реализации показало рост производительности на тестах SPECFP набора SPEC CPU2000 в среднем почти на 2% (и до 16% на отдельных тестах).

Ключевые слова: планирование команд, условное выполнение, GCC.

1. Введение

Современные микропроцессоры используют параллелизм на различных уровнях архитектуры. Одним из видов параллелизма является параллелизм на уровне команд: несколько независимых команд программы могут быть выполнены на отдельных функциональных устройствах процессора. Чтобы использовать такой вид параллелизма, необходимо уметь выбирать из потока программы независимые команды, и назначать их на функциональные устройства процессора. Для этого применяется либо аппаратное динамическое переупорядочение команд во время выполнения, либо порядок команд определяется во время компиляции, либо комбинация этих методов. Достоинством первого метода является то, что на этапе выполнения доступна более полная информация о допустимости переупорядочения команд и фактической загрузке функциональных устройств, однако он увеличивает энергопотребление процессора за счет работы дополнительной логики в процессоре, которая при этом должна быть сравнительно простой из соображений энергоэффективности. Именно такой подход используется в процессорах семейств x86 и x86_64. Соответственно, использование второго метода позволяет снизить энергопотребление процессора, в то время как на

этапе компиляции статическая оптимизация может быть сколь угодно сложной (но и менее точной). Именно этот метод используется в архитектурах с очень длинным командным словом, например, в *архитектуре с явным параллелизмом команд* — EPIC (Explicitly Parallel Instruction Computing [1]), реализованной в процессорах семейства Itanium. EPIC-программа содержит явные указания на то, какие инструкции должны выполняться параллельно, и на каких функциональных устройствах. Идея EPIC состоит в том, чтобы переложить задачу по поиску независимых инструкций на компилятор, при этом предоставив ему дополнительные средства для выполнения оптимизации, такие как спекулятивное выполнение, условное выполнение, вращающиеся регистровые окна. Также архитектура предоставляет большое число функциональных устройств и большое число регистров.

В виду перечисленных особенностей архитектуры EPIC особую важность в компиляторе приобретает *планирование команд*. Основной задачей этой оптимизации является выбор независимых команд из потока программы и упаковка их машинные слова («пакеты») в определенном порядке, которые выполняются EPIC-процессором параллельно.

В качестве основы для нашей работы был выбран метод селективного планирования [2], основными особенностями которого являются:

- Поддержка произвольных ациклических регионов графа потока управления (регион может иметь несколько входов и выходов). Поддержка больших регионов со сложными структурами управления позволяет находить больше команд, которые могут выполняться параллельно;
- Поддержка конвейеризации циклов с неизвестным заранее числом итераций (non-counted loops);
- Поддержка ряда преобразований команд, позволяющих устранять некоторые зависимости по данным и по управлению, а также отождествлять команды с разных путей;
- Поддержка создания новых команд в процессе планирования, например, в качестве компенсационного кода при перемещении только по одной дуге или дополнительных команд восстановления при спекулятивных загрузках.

В данной работе мы описываем, как поддержка условного выполнения, одной из возможностей архитектуры EPIC, может быть реализована в планировщике команд, а также рассматриваем преимущества данного подхода по сравнению с отдельной оптимизацией, работающей до планирования команд. Мы также приводим результаты тестирования нашего подхода на наборе тестов SPEC 2000.

2. Описание алгоритма селективного планирования

В данном разделе мы в общих чертах даем описание алгоритма селективного планирования, уделяя основное внимание особенностям, существенным с

точки зрения добавления поддержки условного выполнения. Детальное описание исходного алгоритма может быть найдено в работе [2].

2.1. Основные понятия и этапы селективного планирования

Областью работы алгоритма является произвольный ациклический участок графа потока управления[1]. Такой участок мы будем называть *регионом*. Регион может иметь несколько входов и выходов — дуг, входящих в регион из других участков, или исходящих из него.

Параллельная группа команд — группа независимых команд, которые на данной архитектуре могут быть выполнены одновременно. Каждая группа может иметь несколько *границ* (boundaries) — дуг графа потока управления, соединяющих команды, находящиеся внутри группы, с командами за ее пределами. Границы одной параллельной группы образуют *барьер* (fence), который перемещается по мере включения команд региона в параллельную группу. Во время планирования может существовать одновременно несколько параллельных групп (барьеров), на границах которых происходит планирование, но в каждый момент только один из барьеров является активным. На рис. 1 проиллюстрированы эти понятия, а также изменение границ параллельной группы по мере включения в нее новых команд. Заметим, что архитектура EPIC допускает одновременное исполнение до 3-х команд условного перехода за один такт, при этом передачу управления выполняет только первая из команд с истинным предикатом.

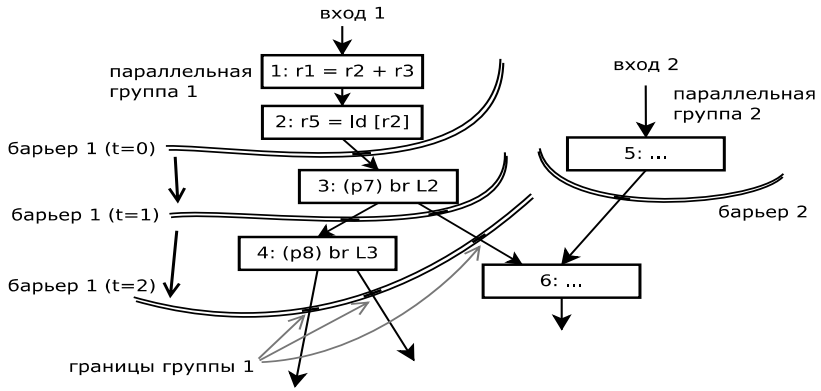

Рис. 1. Перемещение барьера и появление новых границ при последовательном включении команд в параллельную группу.

[1] Под графом потока управления программы будем понимать граф, вершинами которого являются отдельные команды программы, а дуги определяют допустимую последовательность их выполнения.

На самом верхнем уровне, процедура селективного планирования выглядит следующим образом. Сначала на каждом входе в регион создаются пустые параллельные группы команд, в которые в процессе планирования будут переноситься команды. Затем планировщик циклически выполняет следующие шаги:

- Построение множества доступных команд — кандидатов для переноса в текущую параллельную группу.
- Определение множества доступных регистров. Для каждой команды из множества доступных определяется множество регистров назначения, с которыми она может быть запланирована.
- Выбор «наилучшей» команды из множества доступных для планирования с помощью эвристик. Например, учитывается вероятность исполнения дуг, по которым доступна команда, длина критического пути в графе зависимостей, доступна ли команда только в спекулятивной форме, и может ли быть запланирована с исходным регистром. Более подробно эти эвристики рассматриваются в работе [2].
- Фактическое перемещение команды в параллельную группу. Команда перемещается из исходного места в регионе к границам параллельной группы, при этом выполняются все необходимые преобразования, а также создается дополнительный компенсационный код там, где это необходимо для сохранения семантики программы.
- Перемещение границы планирования. Если текущая параллельная группа заполнена (в нее не может быть включено больше команд из-за ограничений по ресурсам), либо множество доступных команд исчерпано, то на границах текущей параллельной группы создаются новые параллельные группы (и барьеры), и планирование продолжается со следующего барьера.

В нашей работе наиболее существенные изменения были внесены в первые два шага алгоритма, поэтому рассмотрим их более подробно.

2.2. Построение множества доступных выражений

Для того чтобы заполнить параллельную группу, прежде всего необходимо определить множество команд, которые потенциально могут быть включены в нее. При решении этой задачи рассматриваются не только машинные команды целиком, но и отдельно правые части выражений, соответствующих этим командам. Такой подход позволяет отождествлять правые части выражений, доступных вдоль различных путей в графе потока управления, а также выбирать любой доступный регистр в точке планирования, если исходный регистр назначения используется на пути переноса команды к точке планирования. Правая часть может быть выделена, если соответствующее ей

выражение не имеет побочных эффектов, как, например, у команды $a = b * c$. В то же время, команды из ассемблерных вставок, команды с автоинкрементом и т.п. не могут быть разделены.

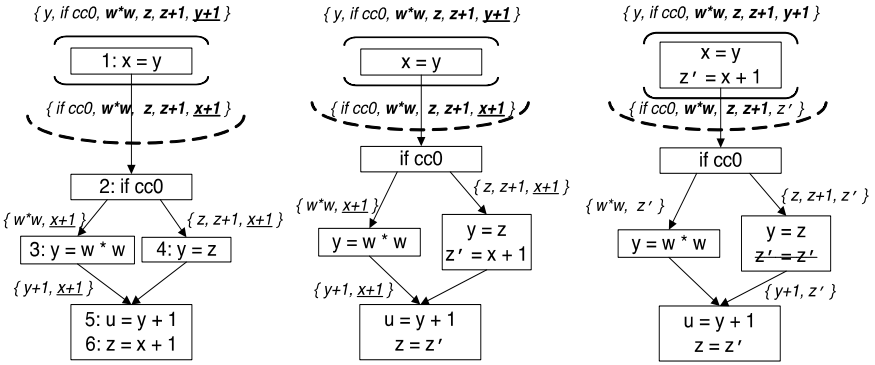

Рис. 2. Пример работы алгоритма выборочного планирования. а) исходный граф; б) команда z' =y+1 перемещается вдоль левой ветви; в) команда перемещается вдоль правой ветви.

Множество доступных команд в точке программы, предшествующей команде *i*, содержит команды, которые доступны для планирования в этой точке, и обозначается как *av_set(i)*. Здесь и далее под командами, в качестве элементов этих множеств, понимаются либо команды целиком, либо их правые части без регистра назначения. Последние становятся полноценными командами только после выбора целевого регистра, который происходит непосредственно перед планированием команды.

Множества доступных команд вычисляются для каждой команды в регионе в обратном топологическом порядке, начиная с «листовых» команд. Сначала вычисляется объединение множеств потомков команды *i*, а затем из него удаляются команды, недоступные над *i* (т.е. команды, имеющие зависимости с *i*). К результирующему множеству добавляется сама команда *i*, либо ее правая часть (ниже она обозначена как *av_op(i)*):

$$av\_set(i) = moveup\_set(i, \bigcup_{x \in Succ(i)} av\_set(x)) \bigcup av\_op(i)$$

Здесь функция *moveup_set(i, S)* удаляет из множества *S* команды, имеющие зависимости с командой *i*. Кроме того, она может выполнять преобразование команд в форму, в которой они не зависят от *i*. В частности, поддерживаются подстановка через присваивание, условное выполнение, а также спекулятивное выполнение.

Рассмотрим построение множеств *av_set* на примере рис. 2(а), где они приведены в фигурных скобках для первых команд базовых блоков. Так, команда *z + 1* из в множества *av_set(4)* была получена из команды *y + 1*, принадлежащей множеству *av_set(5)*, с помощью подстановки через присваивание *y = z*. Без выполнения этого преобразования поменять местами команды 4 и 5 было бы невозможно, т.к. они имеют истинную зависимость по *y*. В то же время, команда 5 ни в какой форме не содержится в *av_set(3)* из-за зависимости с командой 3, которая не может быть устранена с помощью подстановки, т.к. как она не является простой пересылкой, в отличие от команды 4.

После того как вычислены множества доступных команд на границах параллельной группы, строится множество доступных команд для параллельной группы в целом. Оно вычисляется аналогичным образом, распространяя множества *av_set* от границ параллельной группы к ее вершине через уже запланированные операции, при этом также выполняется подстановка аргумента при прохождении на пути вверх через операции копирования. Это позволяет в одной параллельной группе планировать операции, которые в противном случае имели бы истинные зависимости по данным. На рис. 2 выражение *x + 1*, доступное на границе параллельной группы, преобразуется в *y + 1*, которое теперь может быть выполнено параллельно вместе с уже запланированной командой *x = y*.

2.3. Определение множества доступных целевых регистров для выражения

Так как некоторые команды в множестве доступных присутствуют в нем только в виде своих правых частей без указания целевого регистра, то сначала необходимо выбрать целевой регистр, с которым команда будет запланирована. Для этого необходимо определить множество регистров, доступных на пути перемещения команды от ее исходного места в регионе до точки планирования.

Критерием доступности регистра является выполнение следующих условий:

1. Регистр не читается и не записывается ни на одном из путей, вдоль которых доступна выбранная команда;
2. Регистр не жив непосредственно после одной из первоначальных команд (за исключением целевого регистра соответствующей команды);
3. Регистр не жив в начале лежащих вне путей переноса ветвях условных переходов, через которые эта команда была перенесена

Для того чтобы проверить эти условия, для каждого выражения выполняется обход тех путей графа потока управления, по которым доступно исходное выражение, при этом выполняются преобразования, обратные тем, которые выполнялись при подъеме данного выражения к границе планирования.

2.4. Конвейеризация циклов при селективном планировании

С помощью селективного планировщика можно также выполнять конвейеризацию циклов [3]. Т.к. планировщик может работать только с ациклическими регионами, для представления тела цикла в виде ациклического региона применяется специальная техника. Она заключается в том, что в каждый момент подразумевается, что цикл «разорван» по текущим барьерам планирования, что делает возможным перемещение команд через обратную дугу, и невозможным — через текущий барьер. После планирования текущей параллельной группы барьер перемещается, что делает только что запланированные команды доступными для повторного планирования. Таким образом, эффект конвейеризации достигается за счет перемещения команд через обратную дугу, а пролог цикла создается в качестве компенсационного кода в момент такого перемещения команды.

2.5. Реализация селективного планировщика в GCC

Ранее алгоритм селективного планирования был реализован нами в свободном компиляторе GCC [4]. Реализованный планировщик включен в основную ветвь разработки компилятора, начиная с версии 4.4, и используется по умолчанию с уровнем оптимизации `-O3` на архитектуре Itanium. В нашей реализации также была добавлена поддержка спекулятивного выполнения по данным и управлению [5, 6], а также выполнен ряд усовершенствований алгоритма , прежде всего касающихся его быстродействия [7]. Среди них стоит отметить оптимизацию вычисления доступности регистров, сохранение истории преобразований команд, изменение процедуры вычисления приоритетов команд, а также ряд других эвристик. Реализация поддержки команд с условным выполнением, рассматриваемая в данной работе, была выполнена в селективном планировщике GCC.

3. Условное выполнение команд

Одним из факторов, ограничивающих эффективность конвейеризации циклов, является наличие зависимостей по управлению внутри тела конвейеризуемого цикла. Поскольку, вообще говоря, невозможно заранее вычислить, в каком направлении будет выполнен переход на очередной команде передачи управления, чтобы не прекращать наполнение конвейера до того момента, как эта команда будет полностью выполнена, в процессорах применяются схемы предсказания переходов. Их функционирование основывается на эвристиках и сборе статистики о переходах во время выполнения программы; также в некоторых процессорах (в т.ч. в Itanium) существует возможность программно влиять на поведение модуля предсказания переходов. Однако, если предсказание перехода было выдано неверно, происходит сброс конвейера — выполнение всех команд, выданных после неверно предсказанной команды условного перехода, останавливается, и начинается выполнение команды с противоположной ветви условия.

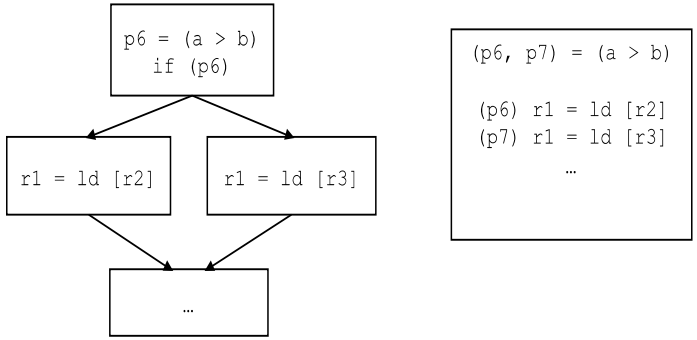

Рис. 3. Преобразование ветвления в программе в последовательный код
а) исходное ветвление; б) код после преобразования

Чтобы избежать потерь в производительности при неправильном предсказании условного перехода, в некоторых процессорах реализована поддержка условного выполнения команд. В процессоре Itanium существует возможность аннотировать любую из команд одним из 64 парных предикатных регистров, при этом команда будет выполнена в том и только в том случае, если указанный предикатный регистр имеет значение «истина». Вместо регистра флагов, команды сравнения устанавливают пару соседних предикатных регистров в противоположные значения; таким образом, можно одновременно хранить результат 31 сравнения (значение нулевого предикатного регистра фиксировано и равно логической единице). В коде каждой команды есть шестибитное поле, в котором записан номер предикатного регистра, контролирующего её выполнение (наличие всегда установленного в единицу предикатного регистра позволяет единообразно записывать условно и безусловно выполняющиеся команды). Условные переходы записываются как безусловные, защищённые соответствующим предикатом.

Возможность условного выполнения в процессоре требует соответствующей поддержки со стороны компилятора. В простейшем случае, она будет заключаться в удалении условных переходов внутри тела цикла, при этом все команды в базовых блоках, выполнение которых зависело от этого перехода, защищаются соответствующим предикатом (рис. 3). Чтобы избежать избыточного увеличения размера кода, обычно применяются эвристики. Так, в компиляторе GCC максимальное число команд для преобразованного блока для архитектуры EPIC задано равным 12 (что соответствует максимальному числу команд, которое может быть выполнено за два такта), а в работе [8]

преобразование применяется только к тем базовым блокам, которые не содержат вызовов функций и длинных цепочек зависимостей по данным. В работе [9] предлагается производить полное преобразование ветвлений для всех базовых блоков программы сравнительно рано в процессе оптимизации кода. Это позволяет упростить граф потока управления для последующих оптимизаций, но для эффективной работы требует добавления учёта условного выполнения в анализ зависимостей по данным. Чтобы отменить негативный эффект слишком агрессивного преобразования команд в условную форму, во время первого прохода планировщика команд производится обратное преобразование ветвлений. В работе [10] рассматривается подход к оптимизации ветвлений на основе минимизации логических схем, описывающих зависимости условий переходов. Это позволяет упростить граф потока управления и оптимизировать вычисление предикатов.

Общим недостатком всех рассмотренных подходов является то, что выгодность преобразования оценивается только исходя из количества преобразованных команд, а их требования к ресурсам процессора не учитываются (например, на процессорах архитектуры Itanium 2, 12 команд с плавающей точкой не могут быть выполнены быстрее чем за 6 тактов, так как процессор имеет только два устройства для вещественных вычислений, но в то же время 12 простых арифметических команд могут быть выполнены за 2 такта). Кроме того, в некоторых случаях имеет смысл преобразовывать в условную форму не целые блоки, а их части, например, чтобы иметь возможность начать загрузки из памяти раньше, чем произойдёт переход на содержащий её базовый блок (при условии что предикат, контролирующий этот переход, может быть вычислен достаточно рано).

Описанные недостатки могут быть устранены, если осуществлять преобразование ветвлений в процессе планирования команд, т.к. в планировщике известна наиболее точная информация о распределении команд на функциональные блоки процессора, и её можно использовать для принятия решений о переносе команд через ветвления. В данной работе рассматривается, как поддержка условного выполнения может быть реализована в селективном планировщике [2].

4. Поддержка условного выполнения в селективном планировщике

Добавление поддержки команд с условным выполнением в селективный планировщик включает следующие подзадачи:

1. Вычисление предиката условно выполняющейся команды, сохранение его в атрибутах команды. Добавление предиката к команде.
2. Добавление условно выполняющихся команд в множество готовых инструкций.
3. Обеспечение корректности при переносе выбранной в условно выполняющейся форме команды к границе планирования.
4. Обеспечение корректности преобразований, применяемых селективным планировщиком к команде в условно выполняющейся форме.

В данном разделе мы рассмотрим подробно каждую из этих задач.

4.1. Обработка условий в планировщике

Для реализации поддержки команд с условным выполнением потребовалось внести ряд изменений в структуры данных и функции планировщика. В частности, в множестве готовых команд вместе с самой командой должен храниться также и предикат, с которым она доступна. Этот предикат вычисляется на этапе инициализации структур данных планировщика в функции, выполняющей анализ команды в форме внутреннего представления компилятора GCC. Среди других изменений можно отметить добавление кэша преобразований, позволяющего хранить различные условные формы для команд, доступных с разных ветвей условного перехода (и получающие предикат условного перехода или его отрицание, соответственно), а также поддержку разделения инструкций в условной форме на левую и правую части на этапе анализа, что позволяет в таких командах выполнять переименование регистров.

4.2. Добавление условно выполняющихся команд в множество готовых инструкций

Преобразование команд в форму с условным выполнением отличается от других преобразований в селективном планировщике, таких как подстановка через присваивание и спекулятивное выполнение. Во-первых, для выполнения данного преобразования необходимо знать, на какой ветви условного перехода доступна команда: к командам с ветви, на которую осуществляется переход, должен быть присоединён предикат, контролирующий переход, а для команд с противоположной ветви — обратный ему. Поэтому присоединение предикатов необходимо выполнять во время объединения множеств доступных команд от обеих ветвей на команде условного перехода, а не при вычислении множества доступных команд, как это делается, например, при спекулятивном выполнении. Во-вторых, при преобразовании команд в условную форму, исходные команды также остаются в множестве доступных, таким образом сохраняется возможность выдать исходную команду в спекулятивной форме, например, еще до того, как вычислен предикат.

Пусть *cond* — условие, контролирующее переход, $avset_1$ и $avset_2$ — множества доступных команд, вычисленные для команд на ветвях условного перехода.

Тогда множество доступных команд непосредственно после перехода J с добавленными командами в условной форме имеет вид:

$$\text{avset\_below}(J) = \bigcup_{I' \in \text{Succ}(I)} \text{avset}(I') \cup$$

$$\bigcup_{I' \in \text{avset}_1} \{apply\_predicate(I', cond)\} \cup \bigcup_{I' \in \text{avset}_2} \{apply\_predicate(I', \neg cond)\}$$

Здесь функция *apply_predicate(I, cond)* присоединяет предикат *cond* к команде *I*.

Чтобы при последующем подъёме *avset_below(J)* через условный переход *J* команды в условной форме не были удалены из множества готовых инструкций, необходимо явно разрешать перенос условных команд через переходы с тем же предикатом в функции *move_up*. Исключение составляют команды, модифицирующие предикатный регистр, контролирующий этот условный переход.

4.3. Перенос к границе планирования

Реализация поддержки условного выполнения требует следующих изменений в процедуре перемещения выбранной команды к текущей границе планирования (функция *move_op* [2]). На этапе поиска изначальных команд при спуске через команду условного перехода необходимо преобразовывать команды, контролируемые тем же предикатом, из условной формы в обычную, удаляя соответствующий предикат у команд из текущего множества изначальных команд *orig_ops*. При этом поиск необходимо продолжать только на одной ветви, соответствующей предикату выбранной команды. Поиск изначальных команд на ветви условного оператора, соответствующей противоположному значению предиката, даже если они там есть в какой-либо форме, не должен производиться из соображений корректности: в противном случае условная команда, которая не могла бы выполниться на данном пути (т.к. он содержит условный переход по противоположному предикату), будет поднята выше условного перехода. В этом случае команда может быть исполнена при соответствующем значении предиката, что противоречит семантике исходной программы.

Еще одно изменение касается создания компенсационного кода. Во время поиска изначальных команд в функции *move_op* планировщик удаляет все найденные формы перемещаемой к границе планирования команды, включая и созданный ранее компенсационный код. Так, на рис. 4 показано, как при планировании обычной команды *a = a + b* сначала будет найдена по левой ветви и удалена изначальная команда 1, что на обратном пути приведет к созданию компенсационного кода 3, а затем и кода 2. После этого, уже при проходе по правым ветвям, сначала будет найдена и удалена команда 2, а затем команда 3, после чего команда *a = a + b* будет перемещена в верхний блок.

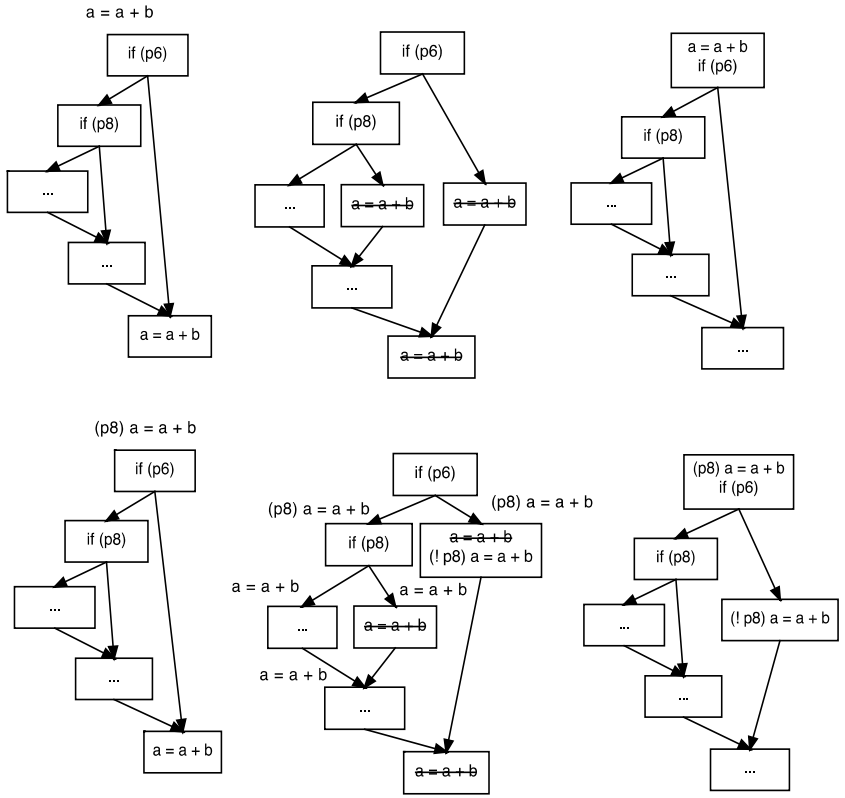

Рис. 4. Создание компенсационного кода при планировании: а) обычной команды; б) команды с условным предикатом

В случае планирования команд с условным предикатом, такое удаление созданного ранее компенсационного кода может привести к созданию некорректного кода. На рис. 4(б) показано планирование того же участка программы, но для планирования выбрана команда с условным выполнением *(p8) a = a + b*. Если бы мы поступили прежним образом и просто удалили компенсационный код, то переменной *a* не присваивалось бы правильное значение в случае, когда предикат *p8* не установлен. Для сохранения корректности кода на всех путях выполнения при нахождении изначальной команды необходимо вместо ее удаления добавить к ней предикат, противоположный тому, который эта команда имеет в множестве изначальных команд *orig_ops*. Удаление команды происходит только в том случае, если команда в этом множестве находится в безусловной форме.

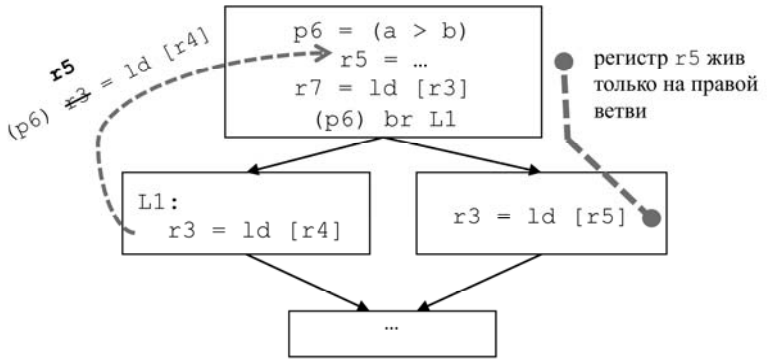

Рис. 5. Выбор регистра назначения для команды с условным выполнением при переименовании регистров

4.4. Взаимодействие с другими преобразованиями

4.4.1 Подстановка через присваивание

В дополнение к подстановке регистра при перемещении команды в условной форме через безусловное присваивание, добавляется возможность осуществлять подстановку через условно выполняемое присваивание, если выполнение команды, в которой выполняется подстановка, контролируется тем же предикатом (табл. 1).

Код до преобразования	Код после преобразования
(p6) movl r32=r33 (p6) ld4 r34=[r32]	(p6) ld4 r34=[r33] (p6) movl r32=r33

4.4.2 Переименование регистров

Заметим, что перенос команды присваивания через условный переход с преобразованием её в условную форму не влияет на значение её целевого регистра на ветке перехода, на которой эта команда не доступна. Например, на рис. 5 для команды `r3 = ld [r4]`, которую можно преобразовать в условную форму и выдать сразу после вычисления предиката, в качестве регистра назначения может быть выбран `r5`, несмотря на то, что его значение используется на другой ветви. Соответственно, условие (3) из критерия доступности регистра в разделе 2.3 можно уточнить следующим образом:

(3). Регистр не жив в начале лежащих вне путей переноса ветвях условных переходов, через которые эта команда была перенесена без преобразования в форму с условным выполнением по предикату, контролирующему этот переход.

Соответственно, механизм вычисления множества недоступных целевых регистров и атрибута *target_available* [7] был модифицирован с учётом этого уточнения. Для атрибута *target_available* достаточно пропускать его обновление при объединении множеств доступных команд на команде ветвления для команд, получивших при переносе через ветвление условную форму.

4.4.3 Использование спекулятивных загрузок

Условное выполнение может также комбинироваться со спекулятивным выполнением команд в произвольном порядке. Например, перенесённая через два ветвления и ставшая спекулятивной загрузка может быть защищена предикатом одного из переходов, что выгодно отличается от простой спекулятивной формы тем, что при ложном предикате обращения к памяти не происходит.

4.5. Конвейеризация циклов с использованием условного выполнения

Поддержка условного выполнения позволяет улучшить конвейеризацию циклов, выполняемую селективным планировщиком.

Во-первых, перенос команд через обратную дугу цикла с добавлением условия, контролирующего выход из цикла, не приводит к увеличению количества команд, которые будут выполнены при запуске конвейеризованного цикла (на последней итерации цикла предикаты команд, перенесённых через обратную дугу, будут ложными).

Во-вторых, поскольку доступность целевого регистра не сбрасывается при переносе через условный переход в конце цикла (даже если этот регистр жив на выходе из цикла), необходимость в использовании переименования регистров возникает реже. Это позволяет сократить увеличение размера кода.

Наконец, появляется возможность переносить команды загрузки из памяти через обратную дугу цикла без использования спекулятивного чтения, что позволяет обойтись без команды проверки спекулятивного чтения. Это позволяет не только сократить размер кода, но и дополнительно оказывает положительный эффект на производительность за счет того, что увеличивается расстояние между командами спекулятивной загрузки и проверки, т.к. если при планировании спекулятивной команды на место исходной вставляется проверка спекулятивного чтения, не всегда оказываются разнесены на достаточное расстояние, чтобы первая закончила свое выполнение до того, как второй понадобится ее результат, что может привести к задержке.

Кроме того, появляется возможность переносить команды, зависящие от этой загрузки (в случае, если используются спекулятивные чтения, это требует генерирования компенсирующего кода в блоке, на который осуществляется переход командой проверки; некоторые зависимые команды, например команды записи в память, не могут быть перенесены, так как не допускают использование непроверенных результатов спекулятивных загрузок).

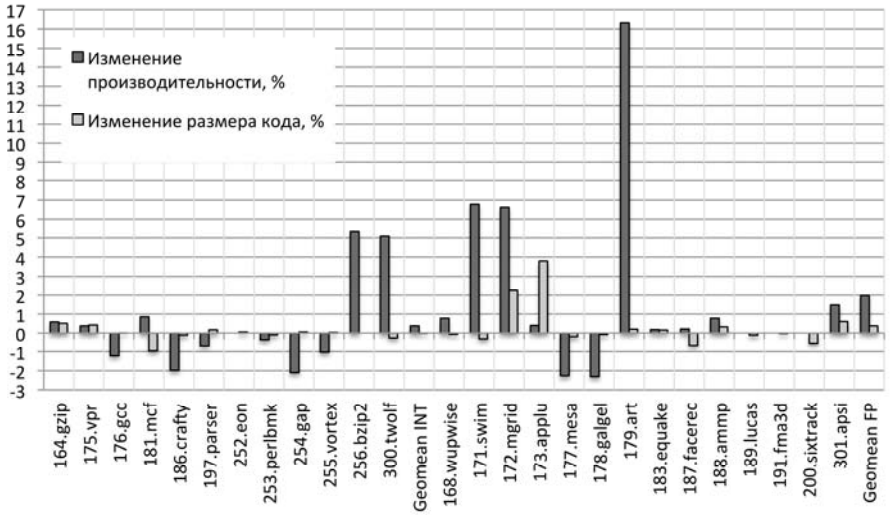

Рис. 6. Изменение производительности и размера кода приложений SPEC CPU 2000 для селективного планировщика с поддержкой условного выполнения

5. Экспериментальные результаты

Разработанный алгоритм был опробован на наборе SPEC CPU 2000. При компиляции для базовых измерений использовались ключи компиляции `-O3 -ffast-math -fno-auto-inc-dec`; на рис. 6 показаны процентные изменения производительности и размера кода тестов при включении поддержки условного выполнения в селективном планировщике.

Пять тестов показали заметный рост производительности (на 5% и более), несколько тестов замедлились, но не более, чем на 2.5%. В целом, производительность набора тестов вещественных вычислений увеличилась на 2%, изменение производительности тестов целочисленных вычислений находится в пределах погрешности измерений. Размер кода тестовых приложений практически не изменился.

6. Заключение

В данной работе был предложен новый подход к поддержке команд с условным выполнением во время планирования команд. Был разработан алгоритм, осуществляющий преобразование команд в условную форму во время работы селективного планировщика. Алгоритм был реализован в селективном планировщике компилятора GCC, и подан для включения в основную ветку разработки компилятора начиная с версии 4.7.

Реализация была протестирована на наборе тестов SPEC CPU 2000. Рост производительности на тестах SPECFP набора SPEC CPU 2000 составил почти 2%, при этом отдельные тесты ускорились на 7 и 16%, а также на 5% выросли два теста из SPECINT. Размер кода при этом изменился незначительно.

Список литературы

[1] Intel Itanium 2 Processor Reference Manual for Software Development and Optimization,

[2] http://www.intel.com/design/itanium/manuals/iiasdmanual.htm. Дата обращения: 01.05.2011.

[3] Soo-Mook Moon and Kemal Ebcioglu. Parallelizing Nonnumerical Code with Selective Scheduling and Software Pipelining. ACM TOPLAS, Vol 19, No. 6, pp. 853—898, November 1997.

[4] S. Muchnick. Advanced compiler design and implementation. Morgan Kaufmann, 3rd ed., 1997.

[5] GCC, the GNU Compiler Collection, http://gcc.gnu.org/. Дата обращения: 01.05.2011.

[6] Andrey Belevantsev, Maxim Kuvyrkov, Vladimir Makarov, Dmitry Melnik, and Dmitry Zhurikhin. An interblock VLIW-targeted instruction scheduler for GCC. Proceedings of GCC Developers' Summit 2006.

[7] Белеванцев, М. Кувырков, Д. Мельник. Использование параллелизма на уровне команд в компиляторе для Intel Itanium. Труды Института системного программирования РАН, 2006.

[8] Andrey Belevantsev, Maxim Kuvyrkov, Alexander Monakov, Dmitry Melnik, Dmitry Zhurikhin. Implementing an instruction scheduler for GCC: progress, caveats, and evaluation. Proceedings of GCC Developers' Summit 2007.

[9] Scott A. Mahlke, Richard E. Hank, James E. McCormick, David I. August, Wen-mei W. Hwu. A Comparison of Full and Partial Predicated Execution Support for ILP Processors. Proceedings of the 22nd Annual International Symposium on Computer Architecture 1995. No. 22. P. 138—149.

[10] David I. August, Wen-mei W. Hwu, Scott A. Mahlke. A Framework for Balancing Control Flow and Predication International Symposium on Microarchitecture 1997. P. 92—103.

[11] David I. August, John W. Sias, Jean-Michel Puiatti, Scott A. Mahlke, Daniel A. Connors, Kevin M. Crozier, Wen-mei W. Hwu. The Program Decision Logic Approach to Predicated Execution. Proceedings of the 22nd Annual International Symposium on Computer Architecture 1999. P. 208-219.