Программная конвейеризация циклов на платформе ARM

Роман Жуйков <zhroma@ispras.ru>, Дмитрий Мельник <dm@ispras.ru>
Рубен Бучацкий <ruben@ispras.ru>

Аннотация. В данной работе описывается проделанная с компилятором GCC работа по адаптации имеющегося алгоритма поворотного модульного планирования для архитектуры ARM. Имеющийся в компиляторе алгоритм может работать только с циклами определенного вида, а на платформе ARM нет подходящей инструкции для организации циклов этого вида. Были выполнены предварительные оценки производительности с помощью реализации шаблона псевдо-инструкции для платформы ARM. По результатам этой проверки было решено расширить возможности алгоритма для оптимизации более общего вида циклов. Разработана и реализована поддержка циклов, счетчик которых меняется как арифметическая прогрессия. Для добавления такой поддержки были внесены значительные изменения в алгоритм поворотного модульного планирования. По результатам тестирования выявлено ускорение части тестовых приложений до 3-4%.

Ключевые слова: оптимизация программ; программная конвейеризация циклов; поворотное модульное планирование; архитектура ARM.

1. Введение

В современном мире широкое распространение получили различные встраиваемые системы. В связи с этим все больше возрастают требования к качеству кода для встраиваемых процессоров. Одним из способов получения более качественного кода является адаптация имеющихся многоплатформенных компиляторов к архитектурным особенностям встраиваемых платформ.

Цель данной работы – адаптировать имеющуюся в компиляторе GCC[1] программную конвейеризацию циклов для процессоров архитектуры ARM. Сложный оптимизационный алгоритм в данный момент работает лишь на небольшом классе циклов, причем такие циклы встречаются на очень ограниченном наборе платформ. Планируется расширить класс циклов, рассматриваемый оптимизацией поворотного модульного планирования.

Дальнейшее изложение построено следующим образом. В разделе 2 описывается программная конвейеризация циклов. Раздел 3 рассказывает об имеющейся реализации алгоритма поворотного модульного планирования.

Раздел 4 посвящен адаптации данной оптимизации для платформы ARM. В разделе 5 описаны полученные экспериментальные результаты.

2. Программная конвейеризация циклов

Программная конвейеризация циклов преобразует тело цикла таким образом, что на выполнение могут выдаваться команды из разных итераций исходного цикла, которые выполняются параллельно, подобно конвейеру. Рассмотрим пример:

```
for i = 1 to N
  A(i);
  B(i);
  C(i);
  D(i);
End
```

В этом примере A(i), B(i), C(i), D(i) обозначают инструкции, которые обрабатывают элемент данных i, зависящие друг от друга. Другими словами, A(i) должно быть выполнено перед тем, как может начать выполняться B(i). Например, A может загружать данные из памяти в регистр, B и C – выполнять арифметические операции над данными, а D – записывать полученный результат обратно в память. Предположим, что не существует зависимости между операциями для различных значений i. То есть, A(2) может начинаться до завершения D(1). Без программной конвейеризации, операции будут выполняться в следующем порядке:

```
A(1); B(1); C(1); D(1); A(2); B(2); C(2); D(2); A(3);…
```

Допустим, что на выполнение каждой инструкции уходит один такт процессора (без учета увеличения счетчика цикла). Также допустим, что в процессоре есть одно функциональное устройство, которое выполняет инструкции A и D (в примере – устройство работы с памятью) и одно функциональное устройство, которое выполняет инструкции B и C (в примере – арифметическое устройство). Тогда в рассмотренном случае на выполнение каждой итерации уходит четыре такта (при этом на каждом такте одно из функциональных устройств процессора простаивает, так как нет готовых для выполнения инструкций).

Теперь рассмотрим преобразованный цикл, который полностью использует все функциональные устройства процессора:

```
A(1);
B(1);
for i = 2 to N
  A(i), C(i - 1);
  B(i), D(i - 1);
```

```
end
C(N);
D(N);
```

Для него порядок выполнения инструкций будет следующим:

```
A(1); B(1); A(2), C(1); B(2), D(1); A(3), C(2); B(3), D(2); …
```

Здесь инструкции, разделенные ‘;’, выполняются на разных тактах процессора, а инструкции, разделенные ‘,’, – на одном. При этом можно легко проверить, что в среднем на выполнение каждой итерации тратится по два такта процессора.

Из этого примера видно, что при наличии возможности параллельного выполнения большого числа инструкций применение этой оптимизации может оказаться очень выгодным. Однако оно может быть связано с многими трудностями. В частности, для корректного преобразования цикла часто необходимо отдельно обрабатывать часть инструкций с нескольких первых и последних итераций – это ведет к увеличению размера кода (иногда в разы превышая размер тела цикла). Также, время жизни некоторых регистров может превышать длину одной итерации (например, какая-то инструкция использует значение переменной, вычисленное на предыдущей итерации), что означает необходимость использования временных регистров для хранения таких значений и, следовательно, создания новых операций для заполнения этих регистров. Эти дополнительные операции могут свести на нет все преимущества от применения конвейеризации циклов.

Было доказано[2], что проблема конвейеризации циклов за оптимальное количество тактов и с использованием минимального количества дополнительных регистров – NP-трудная. Поэтому на практике используются различные эвристические подходы. Один из таких подходов реализован в селективном планировщике команд[3], разработанном в отделе компиляторных технологий ИСП РАН. Реализация планировщика настроена главным образом на использование на процессорах Intel Itanium, однако может применяться и на других платформах (ARM и *x86-64*).

Еще одним вариантом алгоритма конвейеризации циклов является поворотное модульное планирование, реализация которого в компиляторе GCC описана в следующем разделе. Этот подход одновременно ищет такую расстановку инструкций с нескольких итераций цикла, чтобы минимизировать число тактов процессора, проведенных внутри цикла, и стремится минимизировать количество дополнительных регистров, пытаясь расставить инструкции цикла так, чтобы инструкции, устанавливающие и использующие значения регистров, были как можно ближе друг к другу.

3. Поворотное модульное планирование

Модульное планирование – это класс алгоритмов программной конвейеризации циклов, который упрощает задачу программной конвейеризации, используя одно общее расписание для всех итераций цикла и фиксированную частоту выдачи новых итераций.

Поворотное модульное планирование (Swing Modulo Scheduling) – это эффективный алгоритм из класса алгоритмов модульного планирования, целью которого является максимизация частоты выдачи новых итераций, а также минимизация регистрового давления, т.е. количества одновременно живущих регистров в одной итерации цикла.

В конвейеризованном цикле расписание для одной итерации цикла поделено на несколько стадий таким образом, что выполнение всех стадий по одной из последовательных итераций происходит одновременно. То есть, если, к примеру, тело цикла поделено на три стадии, то одновременно будут выполняться первая стадия i-й итерации, вторая стадия (i–1)-й итерации и третья стадия (i–2)-й итерации. Количество тактов между началом выполнения одной стадии и началом выполнения следующей, то есть количество тактов в одной стадии, называется интервалом запуска. На рисунке 1 для примера показан конвейеризованный цикл из пяти итераций, в теле которого три инструкции. Тело конвейеризованного цикла состоит из трех стадий и занимает четыре такта.

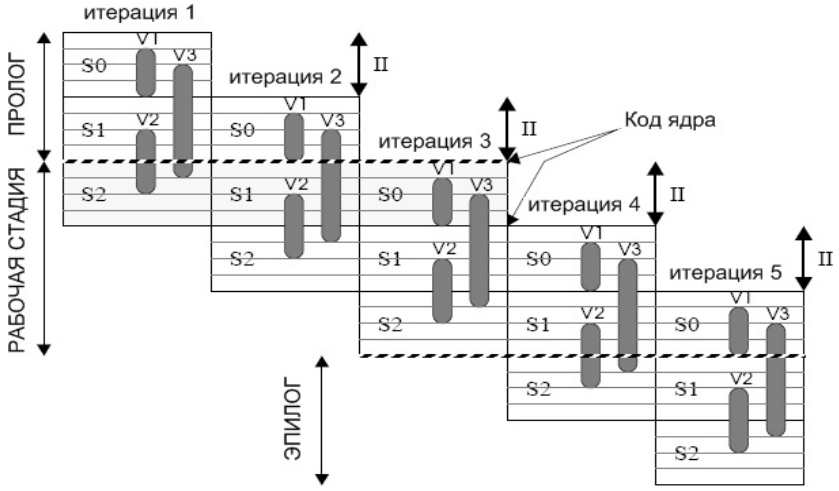

Рис. 1. Пример конвейеризованного цикла.

Очевидно, что чем меньше интервал запуска, тем чаще выдаются новые итерации, тем меньше времени выполняется цикл, то есть тем эффективнее

конвейеризация. Поэтому основной задачей алгоритма ПМП является нахождение такого минимального интервала запуска, чтобы ему соответствовало правильное расписание тела цикла. Проход, реализующий данную оптимизацию в компиляторе GCC, включается опцией –fmodulo-sched и работает перед проходом распределения регистров. Суть алгоритма заключается в последовательном переборе увеличивающихся значений интервала запуска, начиная с минимального и до максимального, и попытках нахождения правильного расписания для цикла при каждом значении интервала запуска.

Более формально, алгоритм выглядит следующим образом:

1. Построение графа зависимостей по данным.
2. Определение минимального и максимального интервалов запуска.
3. Определение порядка следования узлов.
4. Планирование тела цикла.
5. В случае успешного планирования всех узлов расписание готово, иначе необходимо увеличить размер интервала запуска и повторить фазу планирования. Если значение интервала запуска достигает своего максимального предела, то считается, что применение модульного планирования к данному циклу не даст выигрыша, и работа алгоритма заканчивается. В этом случае цикл будет обработан обычным планировщиком.
6. Генерация копий переменных.
7. Генерация пролога и эпилога.

Полное описание алгоритма приводится в работе [4]. Далее дано краткое описание фаз алгоритма для обозначения особенностей реализации поворотного модульного планирования в компиляторе GCC.

3.1. Построение графа зависимостей по данным

Для тела цикла строится *граф зависимостей по данным*, учитывающий как зависимости внутри тела цикла, так и зависимости между итерациями. При анализе зависимостей по данным используется анализ алиасов, а также данные о стоимости (латентности) инструкций из модели процессора и обычного алгоритма планирования ациклических участков кода.

3.2. Определение минимального и максимального интервалов запуска

Интервал запуска ограничивается снизу двумя факторами [5]: рекуррентными соотношениями в графе зависимостей (в графе зависимостей есть рекуррентное соотношение, если в нем существует простой цикл, то есть в исходном цикле используются значения, вычисленные на предыдущих итерациях) и ограничениями на ресурсы процессора.

Для максимального значения интервала запуска подходит длина любого правильного расписания для тела цикла. То есть, алгоритм пытается конвейеризовать цикл до тех пор, пока выполнять полученный цикл может получиться выгоднее, чем оригинальный цикл.

3.3. Фаза вычисления порядка следования узлов

Фаза вычисления порядка следования узлов использует построенный граф зависимостей и дополнительные атрибуты узлов и создает упорядоченный список, содержащий все узлы графа. Этот порядок указывает, в какой очередности узлы графа зависимостей будут анализироваться фазой планирования.

Фаза вычисления порядка следования узлов стремится учесть два фактора в процессе работы:

1. Давать больший приоритет инструкциям, расположенным на критических путях. Под критическим путем здесь подразумевается путь максимальной длины в ациклическом графе, который получается из графа зависимостей отбрасыванием зависимостей между итерациями. Тот факт, что на инструкции, которые окажутся в списке последними, будет больше ограничений, учитывается тем, что предпочитаются инструкции с меньшей мобильностью.
2. Уменьшать максимальное количество одновременно живых регистров. Для этого каждый узел помещается как можно ближе и к своим предкам, и к своим потомкам. Однако от порядка, в котором рассматриваются инструкции, может зависеть возможность такого планирования.

Для достижения обоих этих целей фаза упорядочивания узлов графа использует специальный обход графа зависимостей[6].

3.4. Фаза планирования

Фаза планирования инструкций рассматривает инструкции в порядке, заданном фазой упорядочивания и пытается вставить инструкции в частичное расписание цикла как можно ближе к соседним инструкциям, которые уже были запланированы. Инструкции планируются по-разному в зависимости от того, какие из их соседей уже присутствуют в частичном расписании.

Если для какого-либо узла не найдено подходящее место в расписании, то интервал запуска увеличивается на единицу, и фаза планирования повторяется заново с этим увеличенным значением интервала запуска, которое предоставит больше возможностей для составления правильного расписания. Одной из особенностей реализации алгоритма поворотного модульного планирования в GCC, влияющей на время его работы, является то, что фаза упорядочивания выполняется всего один раз, даже в том случае, когда фаза планирования выполняется несколько раз.

3.5. Генерация копий переменных

Этот этап необходим для того, чтобы из расписания тела цикла, сгенерированного на предыдущей фазе, получить программу, которая будет корректно выполняться. Например, если существуют регистры, срок жизни которых превышает длину интервала запуска, на которой было достигнуто успешное планирование, это означает, что инструкция, определяющая значение регистра, будет выполнена более одного раза до момента, когда это значение понадобится инструкции-потребителю, а, следовательно, правильное значение будет потеряно. Для таких регистров определяется количество копий, которые необходимо сгенерировать, и генерируются сами копирующие инструкции, после чего каждое использование исходного регистра заменяется копией, содержащей правильное значение. Создание копий переменных включается отдельной опцией компилятора GCC –fmodulo-sched-allow-regmoves. Когда данная опция выключена, расписание цикла составляется таким образом, чтобы избежать необходимости создания инструкций копирования.

3.6. Генерация пролога и эпилога

На этой фазе генерируется пролог и эпилог для получения окончательного расписания и корректной программы. Когда количество итераций цикла не известно на этапе компиляции есть несколько подходов. Например, можно выполнять отдельный эпилог в зависимости от количества итераций в, либо сохранить первоначальный цикл и выполнять его в случае, если количество итераций слишком мало, чтобы поток выполнения дошел до тела цикла (другими словами, данных недостаточно, чтобы загрузить конвейер). В реализации GCC, на которой основывается эта работа, применяется второй подход, так как он не приводит к резкому росту размера исполняемого кода. Предположим, при планировании тело цикла было поделено на три стадии. Это означается, что одновременно будут выполняться первая стадия i-й итерации, вторая стадия (i–1)-й итерации и третья стадия (i–2)-й итерации. Соответственно, в пролог цикла будут вынесены (скопированы) инструкций из первой и второй стадии первой итерации, а также дополнительно из первой стадии второй итерации. Аналогично в эпилог будет выноситься третья стадия предпоследней итерации, а также вторая и третья стадия последней итерации. В общем случае, если есть n стадий и они нумеруются с единицы, то инструкции i-й стадии будут n-i раз скопированы в пролог и i-1 раз скопированы в эпилог.

4. Адаптация алгоритма оптимизации для платформы ARM

Поворотное модульное планирование в компиляторе GCC использует шаблон *do-loop*, для распознавания циклов, в которых управляющая часть содержит все инструкции, изменяющие или использующие определенный регистр-

счетчик, а число итераций может быть изменено с помощью изменения значения этого регистра до цикла.

Do-loop – это цикл с уменьшающимся на единицу счетчиком, условие выхода из которого выглядит как “счетчик цикла равен нулю (или единице)”. На разных архитектурах этот шаблон определяется разными способами. На процессорах семейства *x86* есть инструкция “loop <метка>”, которая делает все вместе – уменьшение, сравнение и ветвление. Она использует регистр cx для хранения числа итераций. Похожая ситуация на процессорах Intel Itanium архимектуры *ia64*: существует специальный регистр ar.lc, который может быть использован только для этой цели, и инструкция br.cloop. Все другие поддерживаемые GCC платформы не имеют готовой инструкции, выполняющей все три действия, но компилятором допускается другой вариант организации *do-loop* цикла с инструкцией декремента отдельно от последней инструкции в теле цикла. Последняя инструкция, соответственно, должна выполнять сравнение и ветвление. Примером такой архитектуры является платформа Cell SPU. На ней есть инструкция “biz <регистр>, <*метка*>”, что означает “branch if zero”, то есть переход на метку происходит только при равенстве регистра нулю.

В алгоритме поворотного модульного планирование к *do-loop* циклам еще несколько требований. Счетчик цикла не должен использоваться в других инструкциях, вне управляющей части цикла. Цикл должен состоять из одного базового блок и не содержать вызовов других функций и процедур, а также других инструкций с побочными эффектами.

Возвращаясь к платформе ARM, на ней нет подходящей аппаратной инструкции для создания шаблона *do-loop* циклов. Однако прежде чем реализовывать поддержку алгоритма ПМП для более общего вида циклов, было решено сначала оценить возможности конвейеризации на процессорах ARM. Для этого необходимо было создать в машинно-зависимой части компилятора псевдо-инструкцию *do-loop* для платформы ARM.

4.1. Создание шаблона псевдо-инструкции

Были опробованы два способа организации шаблона. Оба подразумевают эмуляцию ассемблерной инструкции, которая состоит из двух реальных инструкций процессора. Пусть *reg* — регистр для счетчика цикла. Первый способ – использовать “sub *reg*, *reg*, 1” для декремента и пару “cmp *reg*, 0; bne <метка>” как инструкцию ветвления. Второй способ – просто взять пару инструкций “subs reg, reg, 1; bne <метка>” в качестве последней инструкции цикла. Второй способ должен работать быстрее, но при его реализации возникла следующая проблема с распределением регистров: в некоторых случаях при распределении регистров регистр-счетчик сбрасывается на стек, после чего не находится подходящего регистра для восстановления. Такая ситуация допускается алгоритмом распределения регистров, причем для правильного корректирования программы компилятором в этом случае

необходимо описать в машинно-зависимой части компилятора процесс замены инструкции на другие, использующие оперативную память вместо регистра.

Поскольку нашей целью была лишь предварительная оценка возможностей алгоритма поворотного модульного планирования на платформе ARM, реализация такого процесса замены инструкций была слишком трудоемкой. Поэтому был выбран для реализации первый способ создания шаблона псевдо-инструкции с помощью инструкций sub (для декремента на 1), cmp и bne. Данный вариант удалось запустить после внесения необходимых изменений в работу конечного автомата, описывающего состояние процессора в архитектурно-зависимой части GCC для платформы ARM. При тестировании оказалось, что полученная реализация генерирует неверный код для библиотеки *eina* (одна из библиотек для набора тестов *expedite*[7]). В процессе исследования этой проблемы было решено отказаться от использования псевдо-регистра и выбрать один конкретный аппаратный регистр на ARM (например, r5), чтобы использовать только этот регистр для *do-loop* циклов.

Использование одного аппаратного регистра допустимо, так как поворотное модульное планирование поддерживает работу только с *do-loop* циклами, являющимися самыми внутренними в группе вложенных циклов. Идея использования одного выбранного аппаратного регистра позволила избежать проблемы с распределением регистров и вернуться к использованию второго способа организации шаблона *do-loop* с помощью пары инструкций subs и bne. Дополнительно, было добавлено условие для генерации псевдо-инструкций с *do-loop* шаблоном только в случае, если выставлен уровень оптимизации не ниже первого (-O1) и включена опция –fmodulo-sched, отвечающая за ПМП в GCC.

После этих изменений, алгоритм ПМП заработал на платформе ARM следующим образом. В первую очередь, когда оптимизационный проход *do-loop* обрабатывает циклы, где число итераций может быть вычислено до начала цикла, такие циклы переформировываются для использования *do-loop* шаблона. До начала цикла, в пролог, добавляется инструкция “mov r5, <*1+число_итераций*>”. А две последние инструкции в цикле становятся “subs r5, r5, 1; bne <*метка–начало цикла*>” вместо исходной проверки условий выхода из цикла. После такого преобразования цикл становится подходящим для анализа с помощью имеющейся реализации ПМП, если он удовлетворяет другим обычным ограничениям - состоит из одного базового блока, который не содержит вызовов. Оптимизационный проход ПМП происходит при первой работе планировщика, до распределения регистров. Если найдено расписание команд, лучшее, чем исходное, цикл помечается, чтобы не быть перепланированным после распределения регистров. Соответственно, в успешно запланированных с помощью ПМП циклах аппаратный регистр r5 всегда оказывался занят для управляющей части цикла.

Экспериментально сравнивалась производительность базовых GCC оптимизаций уровня 2 (-O2) и этих же оптимизаций с опцией –fmodulo-sched. Измерялись результаты на тестовой плате OMAP3EVM. Тестовый набор для *sqlite*[8] показал замедление на 2%: один тест на 3% ускоряется, но другие замедляются на 1-4%. Тестовый набор *expedite* для библиотеки растеризации *libevas*[7] ускоряется на 0.25%. При этом отдельные тесты показывают результаты от замедления на 4% до ускорения на 5%.

Дополнительно было проведено тестирование набора небольших тестов *aburto* и исследованы полученные результаты. Тест *heapsort* ускоряется на 7%. Также увеличивается производительность простого умножения матриц и умножения с транспонированной второй матрицей на 2% и 4% соответственно. Тест *flops* замедляется на 2.5%. На других тестах производительность не изменилась.

Исследование ассемблерного кода и журналов компилятора GCC позволило сделать вывод, что ускорение программы происходит, если планировщик успешно составил новое улучшенное расписание для «горячего» цикла. Поворотное модульное планирование лучше распределяет нагрузку между итерациями, старается разнести максимально далеко друг от друга зависимые по данным инструкции с соседних итераций. Напротив, когда цикл «горячий» и планирование оказалось неуспешным, получается замедление, связанное с вставкой инструкций *do-loop* шаблона до прохода планировщика. Эти инструкции остаются в коде цикла независимо от того, было ли успешно составлено расписание. И, поскольку между этой парой инструкций есть дополнительная зависимость по данным, (ветвление зависит от флага, выставляемого сравнением), они дают небольшую задержку и приводят к замедлению. Один из возможных способов исправить эту ситуацию – реализация ПМП без требования *do-loop* шаблона, что требует больших изменений в платформо-независимой части ПМП. Логика алгоритма планирования во многом опирается на тот факт, что счетчик цикла не используется внутри цикла, кроме управляющих инструкций цикла.

4.2. Циклы с декрементом счетчика

Следующим этапом адаптации была реализация работы алгоритма ПМП на циклах, не использующих *do-loop* инструкций, счетчик которых ведет себя аналогично случаю *do-loop* циклов. Предполагается, что счетчик уменьшается каждую итерацию ровно на единицу, а цикл заканчивается при достижении счетчиком нуля. Также в этом случае необходимо отказаться не только от самого требования *do-loop* шаблона, но и разрешить использование (чтение) счетчика в инструкциях вне управляющей части цикла. Как следует из описания алгоритма планирования, в итоговом запланированном цикле число итераций уменьшается, часть инструкций копируется в пролог и эпилог. И если в *do-loop* циклах можно просто соответствующим образом поменять значение счетчика перед циклом, то в данном варианте такой подход оказывается недопустим.

Было выбрано простое решение - запланировать инструкцию декремента вместе с другими инструкциями, как будто она не относится к управляющей части цикла, а в конце проверить корректность полученного расписания. В силу особенностей реализации достаточно проверить, что инструкция декремента попала первую стадию в новом расписании, и отменить преобразование цикла в противном случае. Такой вариант реализации позволяет выиграть до 20% производительности на простейших тестовых примерах, однако не влияет на производительность настоящих тестов, поскольку циклы с таким поведением счетчика встречаются в реальных программах крайне редко.

Итак, исследования показали, что на тестовых программах планирование циклов с помощью ПМП позволяет значительно улучшить производительность на процессорах ARM. Поэтому было решено реализовать поддержку в алгоритме ПМП циклов более общего вида, чем *do-loop* циклы.

4.3. Поддержка циклов нового вида

Еще раз отметим, что ПМП оптимизация в компиляторе GCC работает до распределения регистров. Поэтому далее в данном разделе под словом “регистр” будет подразумеваться псевдо-регистр на этапе работы алгоритма ПМП, а не аппаратный регистр.

Опишем новые требования к циклу, под которые попадает значительно большее количество циклов в реальных приложениях. Цикл по-прежнему должен содержать только один базовый блок, и не иметь внутри вызовов и других инструкций с побочными эффектами. Однако для управляющей части и целочисленного регистра-счетчика цикла допускается большая свобода, чем в *do-loop* циклах.

Выход из цикла должен происходить при достижении равенства в сравнении счетчика с инвариантной в цикле величиной, то есть, либо с константой, либо с регистром, значение которого установлено до цикла и не меняется в теле цикла. Внутри цикла счетчик должен изменяться ровно одной инструкцией, которая прибавляет или вычитает ненулевую константу. Другие инструкции могут использовать счетчик, но не могут менять его значение. Получается, что счетчик цикла всегда соответствует элементам некоторой арифметической прогрессии.

Для поддержки циклов нового вида были внесены значительные изменения в оптимизацию поворотного модульного планирования. Необходимо отметить, что циклы, подходящие под новую модель, могли встретиться не только на платформе ARM, но и на других платформах. Примером такой платформы является распространенная на персональных компьютерах архитектура *x86-64*, которая является одной из приоритетных платформ для компилятора GCC. В связи с этим, в процессе нижеописанных работ проводились дополнительные проверки корректности и качества кода при компиляции для *x86-64*.

4.4. Корректировка конечного значения счетчика в цикле

В первую очередь опишем, как была решена проблема, возникшая в пункте 4.2. Часть инструкций с помощью копирования выносится в пролог и эпилог. После этого преобразования необходимо изменить управляющую часть цикла так, чтобы количество итераций уменьшилось, но все инструкции, использующие счетчик, получили его корректное значение. Будем считать, что построено новое расписание цикла, все инструкции поделены на стадии, стадии нумеруются с нуля. Пусть STAGE — номер стадии, на которую в новом расписании попала инструкция сравнения счетчика с финальным значением FIN, которое может быть как регистром, так и константой. Инструкции со стадии STAGE будут скопированы в эпилог ровно STAGE раз. Сама инструкция сравнения не должна копироваться в эпилог, ее результат не будет там использоваться, поскольку в эпилог не выносится инструкция ветвления. Тем не менее, ровно на STAGE итераций раньше необходимо выполнить выход из нового цикла, чтобы инструкции в эпилоге получили корректное значение счетчика.

Для корректирования управляющей части цикла необходимо сделать следующее: в инструкции сравнения, нужно заменить FIN на новое значение по формуле FIN2 = FIN — STAGE * STEP, где STEP — шаг изменения счетчика на каждой итерации. В случае, когда FIN представляет собой регистр, для нового значения FIN2 необходимо перед циклом завести новый регистр, в котором произвести соответствующие вычисления. Если же FIN — константа, которая записывается в виде непосредственного значения в инструкции сравнения, необходимо заменить эту константу на новую. Иногда новая константа FIN2 не может быть записана в виде непосредственного значения, так как на ARM набор таких констант ограничен специфическим образом. В этом случае для хранения такой константы лучше также выделить отдельный регистр, вместо того, чтобы на каждую итерации цикла получать константу с помощью нескольких инструкций.

4.5. Подсчет числа итераций цикла во время выполнения

Как уже было описано ранее, в случае, когда нет возможности определить количество итераций цикла во время компиляции, алгоритм поворотного модульного планирования копирует исходную версию цикла, прежде чем изменить ее. Получается, что в коде есть две версии цикла, кроме того, в код добавляется проверка условия на число итераций. Если итераций оказывается недостаточно, чтобы заполнить пролог и эпилог, и хотя бы один раз выполнить тело оптимизированного цикла – происходит переход на исходную версию цикла. В обычном случае, когда итераций много, будет выполняться оптимизированная версия цикла. Рассмотрим, как выглядит это условие, при котором должна выполняться оптимизированная версия цикла. Число итераций цикла должно быть строго больше чем число стадий в построенном расписании цикла. В случае *do-loop* цикла число итераций можно легко

узнать, зная начальное значение счетчика цикла. Тем самым, остается добавить инструкцию сравнения этого регистра с числом стадий (константа, известная в момент компиляции) и переходить на оптимизированную версию цикла только при достаточном количестве итераций.

Для нового типа циклов с постоянным шагом счетчика, ситуация несколько сложнее. Для правильного подсчета числа итераций необходимо учитывать возможные переполнения счетчика цикла, которые могут усложнить способ подсчета количества итераций. Кроме того, переполнения могут сделать цикл бесконечным. В случае, когда переполнений не происходит, число итераций можно подсчитать, поделив разность конечного и начального значения счетчиков цикла на шаг счетчика. Приведем два примера циклов, к которым не применимо такое решение.

```
for (unsigned int x = 0x0; x != 0x6F80919A; x += 0xEDCBA987) ...; //этот цикл имеет 22 итерации
for (i = 0; i <= n; i++) …;
```

Если в теле цикла не изменяются беззнаковые целочисленные переменные i и n, то число итераций в нем равняется n+1. Однако, в случае, когда n представляет собой максимально возможное для заданного типа беззнаковое число, состоящее из всех единиц в двоичной записи, цикл становится бесконечным.

Конечно, подобные ситуации встречаются редко, но необходима их корректная обработка в оптимизации поворотного модульного планирования. Можно было бы просто отбросить циклы, в которых происходит переполнение целочисленных регистров. В процессе изучения того, как отличить такие циклы от обычных, было найден имеющийся в GCC механизм для анализа циклов - функция get_simple_loop_desc, которая позволяет получить выражение, вычисляющее число итераций через значения регистров во время выполнения программы. Для циклов, анализ которых невозможен, компилятор будет отказываться от оптимизации. Оказалось, что на платформах с *do-loop* инструкцией данная функция неприменима для циклов, на которых было выполнено *do-loop* преобразования. Данный случай был рассмотрен отдельно, чтобы не уменьшать множество циклов, для которых применима оптимизация поворотного модульного планирования.

Функция get_simple_loop_desc может определить условия, при которых цикл становится бесконечным. Соответственно, в этом случае до подсчета количества итераций цикла в код добавляется проверка этих условий. В получившемся коде оптимизированная версия цикла выполняется только в случае, когда количество итераций конечно и превосходит число стадий в расписании. В других случаях происходит переход на исходную версию цикла.

4.6. Учет зависимостей по регистру флагов

Некоторые изменения были внесены так же в построение графа зависимостей по данным. Поскольку в *do-loop* циклах регистр-счетчик использовался только в управляющей части цикла, условие выхода из цикла проверялось до начала оптимизации, и далее инструкция сравнения исключалось из анализа. Для поддержки циклов нового вида необходимо было вычислять зависимости, как для инструкции сравнения, так и для инструкции ветвления – последней в теле цикла. Это связано с тем, что теперь было необходимо принимать во внимание другие инструкции, которые могли читать значение регистра-счетчика, или даже менять значение регистра флагов, который используется инструкцией ветвления.

На данном этапе было обнаружено, что для некоторых тестов на платформе *x86-64* происходит генерация некорректного кода. В отличие от ARM, где многие арифметические инструкции имеют две версии — с выставлением флагов и без (например, sub и subs), на *x86-64* почти все арифметические инструкции изменяют флаги. Иногда в новом расписании цикла между сравнением регистра счетчика с финальным значением и инструкцией ветвления, организующей переход в начало цикла, вставлялась другая арифметическая инструкция, которая меняла значение флага нулевого результата. Для исправления данной ошибки была добавлена реализация генерации необходимых ребер в графе зависимостей по данным.

Кроме того, была выявлена и исправлена еще одна ошибка, касающаяся регистра флагов. В некоторых случаях неверно определялся регистр, который порождает зависимость по данным между двумя инструкциями, из-за чего неверно генерировались копии переменных. Стоит отметить, что регистр флагов на платформах ARM и *x86-64* не может быть скопирован, поэтому для зависимостей по этому регистру важно правильно расставить нужные анти-зависимости в графе, чтобы в итоговом расписании не возникало необходимости копировать регистр флагов.

5. Результаты тестирования

Все проделанные работы были выполнены на компиляторе GCC. Использовалась текущая разрабатываемая сообществом версия компилятора 4.7. Сравнивалась производительность тестов на обычной оптимизации –O2 с результатами, полученными с разработанной поддержкой нового вида циклов при включенных опциях –O2, –fmodulo-sched и –fmodulo-sched-allow-regmoves.

Тестирование на платформе ARM выполнялось на тестовых платах EBV Beagle Board и Samsung C110. Обе платы с процессором ARM Cortex-A8. Тестовый набор для *sqlite* показал ускорение на 3%. Тестовый набор *expedite* для библиотеки растеризации *libevas* в среднем показывает такие же результаты, при этом отдельные тесты показывают результаты от замедления

на 3% до ускорения на 4%. Дополнительно было проведено тестирование умножения матриц из целых и вещественных чисел с помощью различных алгоритмов. Для матриц из целых чисел среднее ускорение составило 3%, а для случая чисел с плавающей точкой производительность увеличилась на 1%.

Тестирование на платформе x86-64 проводилось на персональном компьютере с процессором Intel Pentium IV, поддерживающим 64 битное расширение EM64T. Для умножения матриц из чисел с плавающей точкой с помощью различных алгоритмов ускорение составило в среднем 1%. При этом прирост производительности составил 1.5% для самой простой реализации умножения, 2.5% для реализации с временной переменной, и около 3% для алгоритма с транспонированной второй матрицей и алгоритма, где поменяна вложенность внутренних циклов. В то же время реализация методами Maeno Toshinori и Dan Warner замедлилась на 3% и 5% соответственно.

Было проведено тестирование на промышленном наборе тестов SPEC CPU2000[9], в среднем производительность изменилась незначительно. Наибольшее ускорение получено на тесте 300.twolf , оно составило 1.7%. На тестах 252.eon и 178.galgel производительность снизилась на 2%.

6. Заключение

В рамках данной работы был проведен анализ имеющегося в компиляторе GCC алгоритма программной конвейеризации циклов ПМП для выявления возможностей его использования на платформе ARM. По результатам этой работы было обнаружено, что планировщик может работать только с циклами очень узкого класса. Были исследованы возможности отказа от этого ограничения и реализована поддержка имеющимся алгоритмом циклов нового типа. Проделанная работа позволяет применять данную оптимизацию на таких распространенных платформах, как ARM и *x86-64*.

Были сделаны необходимые улучшения, позволяющие алгоритму конвейеризации циклов корректно работать на циклах нового вида. В частности, был улучшен алгоритм создания пролога и эпилога, а также значительно усложнен механизм генерации проверки условий для выполнения оптимизированной версии цикла. Также реализовано внесение в граф зависимостей по данным необходимых изменений для генерации компилятором корректного кода.

По результатам тестирования получено значительное увеличение производительности на некоторых приложениях. Среди дальнейших работ планируется проверить работу алгоритма на других платформах, а также исследовать причины снижения производительности некоторых тестов при конвейеризации циклов.

Список литературы

[1] Веб-сайт Gnu Compiler Collection. http://gcc.gnu.org/

[2] M.R. Garey, D.S. Johnson. “Computers and Intractability: A Guide to the Theory of NP-completeness”. San Francisco: W. H. Freeman & Company Publishers. 1979.

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

[4] J. Llosa, E. Ayguade, A. Gonzalez, M. Valero, J. Eckhardt. “Lifetime-sensitive modulo scheduling in a production environment”. Computers, IEEE Transactions on. Volume 50, Issue 3, pp.234-249. 2001.

[5] B.R. Rau. “Iterative modulo scheduling: An algorithm for software pipelining loops”. In Proc. of the 27th Annual International Symposium on Microarchitecture, pp. 63-74. November 1994.

[6] Mostafa Hagog and Ayal Zaks. Swing Modulo Scheduling in GCC. In Proceedings of the GCC Developer's Summit 2004, pp 55-64, Ottawa, Canada.

[7] Веб-сайт Enlightenment Foundation Libraries http://www.enlightenment.org/p.php?p=about/efl

[8] Веб-сайт SQLite http://www.sqlite.org/about.html

[9] Веб-сайт Standard Performance Evaluation Corporation. http://www.spec.org/cpu2000/

Loops software pipelining on ARM platform

Roman Zhuykov, ISP RAS, Moscow, Russia zhroma@ispras.ru
Dmitry Melnik, ISP RAS, Moscow, Russia dm@ispras.ru
Ruben Buchatskiy, ISP RAS, Moscow, Russia ruben@ispras.ru

Abstract. This article describes improvements we made in the implementation of swing modulo scheduling (SMS), a well-known software pipelining technique, in the GNU Compiler Collection (GCC) for ARM platform. Prior GCC implementation required a loop being pipelined to conform to the do-loop pattern, which needs a special hardware instruction. However, such hardware instruction is absent on ARM. First we implemented a "fake" do-loop instruction in the ARM backend, which helped us to verify whether GCC SMS implementation is profitable on ARM. Then we designed and implemented support for loops which loop counter varies as an arithmetic progression. In do-loops the loop counter must be used only in control part of the loop, and we allow reading loop counter register by other loop instructions. For such loops we improved the algorithm for creating prologue and epilogue as well as implemented much more complex algorithm of verification conditions for entering performance-optimized version of the loop. Also we made necessary changes in data dependency graph to generate correct code. When dependency graph is built we create additional anti-dependencies between instructions which use flag register. The resulting performance improvement is 3-4% for selected test applications on ARM platform. For x86-64 platform, performance results are mostly neutral, with exception of 2-3% improvement on matrix multiplication tests.

Keywords: program optimizations; loop software pipelining; swing modulo scheduler; ARM.

References

[1]. Gnu Compiler Collection website. http://gcc.gnu.org/
[2]. M.R. Garey, D.S. Johnson. "Computers and Intractability: A Guide to the Theory of NP-completeness". San Francisco: W. H. Freeman & Company Publishers. 1979.
[3]. A. Belevantsev, D. Zhurikhin, D. Melnik. Kompilyatsiya programm dlya sovremennykh arkhitektur [Program compilation for modern architectures], Trudy ISP RAN [The Proceedings of ISP RAS], 2009, vol. 16, pp. 31-50 (in Russian).
[4]. J. Llosa, E. Ayguade, A. Gonzalez, M. Valero, J. Eckhardt. "Lifetime-sensitive modulo scheduling in a production environment". Computers, IEEE Transactions on. Volume 50, Issue 3, pp.234-249. 2001. doi: 10.1109/12.910814
[5]. B.R. Rau. "Iterative modulo scheduling: An algorithm for software pipelining loops". In Proc. of the 27th Annual International Symposium on Microarchitecture, pp. 63-74. November 1994. doi: 10.1145/192724.192731
[6]. Mostafa Hagog and Ayal Zaks. Swing Modulo Scheduling in GCC. In Proceedings of the GCC Developer's Summit 2004, pp 55-64, Ottawa, Canada.
[7]. Enlightenment Foundation Libraries website http://www.enlightenment.org/p.php?p=about/efl
[8]. SQLite website http://www.sqlite.org/about.html
[9]. Standard Performance Evaluation Corporation website. http://www.spec.org/cpu2000/