Компиляция программ для современных архитектур

А. Белеванцев, Д. Журихин, Д. Мельник

Аннотация. Настоящая статья посвящена обзору некоторых работ по оптимизации программ для современных вычислительных архитектур, проводимых в отделе компиляторных технологий Института системного программирования РАН. Работы включают в себя выявление параллелизма на уровне команд для архитектуры Intel Itanium, исследование и разработку энергосберегающих оптимизаций для архитектуры ARM, а также исследования по динамическим оптимизациям для языков общего назначения, выполняемым на машине пользователя. Большинство приведенных работ выполнялось в рамках компилятора GCC с открытыми исходными кодами, являющегося стандартным компилятором для Unix-подобных систем.

1. Введение

Развитие вычислительной техники за последние годы приводит к появлению большого количества процессорных архитектур, для использования возможностей которых необходимы новые технологии компиляции. Например, архитектуры с явно выраженным параллелизмом команд требуют от компилятора наличия оптимизаций, направленных на выявление и использование такого параллелизма, а именно – агрессивного планирования команд и конвейеризации циклов. Популярность встраиваемых архитектур, повсеместно использующихся в мобильных устройствах самого разнообразного назначения, влечет необходимость разработки компиляторных технологий, обеспечивающих не только высокую производительность программ, но и небольшой размер исполняемых файлов, а также низкое энергопотребление системы. Многоядерные архитектуры и гетерогенные архитектуры с несколькими акселераторами, получившие широкое распространение, нуждаются в разработке новых методов компиляции, позволяющих программисту в полуавтоматическом режиме указать желаемое распределение вычислений и потоков данных по компонентам таких архитектур. Наконец, актуальной является задача об оптимизации программы для конкретной реализации некоторой архитектуры, а также для конкретных наборов входных данных пользователя. Эта задача частично решена для динамических языков типа Java, но не для языков общего назначения.

Настоящая статья посвящена обзору некоторых работ, проводимых по этим направлениям в отделе компиляторных технологий Института системного программирования РАН. Работы включают в себя выявление параллелизма на уровне команд для архитектуры Intel Itanium, исследование и разработку энергосберегающих оптимизаций для архитектуры ARM, а также исследования по динамическим оптимизациям для языков общего назначения, выполняемым на машине пользователя. Большинство приведенных работ выполнялось в рамках компилятора GCC [[10]] с открытыми исходными кодами, являющегося де-факто стандартом для UNIX-систем и поддерживающего широкий набор входных языков (Си/Си++, Фортран, Java, Ада) и целевых архитектур (x86, PowerPC, SPARC, ARM, Itanium и множество других). Каждой из этих работ далее посвящена один раздел, включающий обзор существующих исследований по соответствующему направлению, описание работ, выполненных в ИСП РАН, полученные результаты и планы на ближайшие годы. Наконец, в заключение статьи приводятся и обсуждаются выводы из проведенных исследований.

2. Компиляция для архитектур с явно выраженным параллелизмом команд

Современные процессорные архитектуры обладают большим количеством параллельно работающих конвейерных функциональных устройств. Для достижения высокой производительности на этих архитектурах требуется обеспечить непрерывную загрузку этих функциональных устройств, максимально используя *параллелизм на уровне команд*, имеющийся в программе. Основным способом в выявлении такого параллелизма является переупорядочивание команд, выполняемое при планировании команд либо конвейеризации циклов.

Суперскалярные архитектуры (x86, PowerPC) планируют команды аппаратно во время выполнения программы, т.е. порядок выдачи команд на выполнение может отличаться от порядка, диктуемого программой. Архитектуры с явным параллелизмом команд (EPIC) требуют, чтобы окончательный порядок выполнения команд определялся при компиляции: сама архитектура точно следует заданному порядку, не выполняя никакого динамического переупорядочивания. Это позволяет отказаться от аппаратных устройств, реализующих это переупорядочивание, в пользу других свойств, предоставляющих компилятору больше возможностей по выявлению параллелизма на уровне команд. Рассмотрим кратко наиболее важные из этих свойств, реализованных в архитектуре Itanium.

2.1. Особенности архитектур с явно выраженным параллелизмом

Опережающее выполнение команд прежде, чем становится известно, что их выполнение необходимо, принято называть *спекулятивным выполнением* (speculative execution). В суперскалярной архитектуре поддержка спекулятивного выполнения, необходимая для предсказания переходов, обеспечивается специальным буфером переупорядочивания, хранящим промежуточные результаты выполнившихся спекулятивно команд, а также раздельными механизмами фиксации и выдачи спекулятивных и обычных команд. В EPIC-архитектуре компилятор обязан выбрать команду для спекулятивного выполнения, переместить её в новое место и пометить, как спекулятивную. При этом для корректной обработки исключений архитектура обеспечивает их подавление при выполнении спекулятивной команды и выброс исключения на специальной команде проверки, также вставляемой компилятором. Пример спекулятивного выполнения команд на архитектуре Itanium показан на рис. 1.

а)

```
r2 = ld[r3]
;;
p6 = cmp r2, 0
;;
(p6) jmp label
mul r4, r4, r1
;;
add r5, r5, r4
  ;;
r6 = ld [r5]
```

б)

```
r2 = ld[r3]
mul r4, r4, r1
;;
add r5, r5, r4
;;
p6 = cmp.lt r2, 0
r6 = ld.s [r5]
;;
(p6) jmp label
```

Рис. 1. Спекулятивное выполнение на процессоре Itanium: а) – первоначальный код, б) – измененный компилятором

Другим примером осооенности, направленнои на выявление параллелизма на уровне команд, является поддержка *условного выполнения* через предикатные регистры. При условном выполнении практически любую команду можно аннотировать одним из предикатных регистров, при этом команда выполняется только в том случае, если значение предикатного регистра равно единице. Это позволяет выражать достаточно длинные ветвления без переходов, но лишь командами сравнения и командами с условным выполнением, что в свою очередь уменьшает количество зависимостей по управлению, мешающих выявлению параллелизма.

а)

```
cmpl %edx,
%eax jle L
movl %edx,
%eax
```

б)

```
cmp4.gt
p6,p7=r32,r33 (p6)
movl r32=r33
```

Рис. 2. Вычисление минимума двух чисел на ассемблере x86 (а) и с помощью условного выполнения на процессоре Itanium (б).

Рассмотрим пример реализации условного выполнения в процессорах Itanium. Предикатные регистры в Itanium хранятся в 64-битном слове, при этом 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 6-битное 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 переходы, защищённые соответствующим предикатом. На рис. 2 показан пример вычисления минимума из двух целых чисел на ассемблере x86 (без использования команды условной пересылки) и на ассемблере Itanium с применением условного выполнения.

а)

```
L1:
ld4 r32 = [r5], 4
add r7 = r4, r9
st4 [r6] = r7, 4
      br.cloop L1;;
```

б)

```
L1:
ld4 r32 = [r5], 4
// N+2
add r35 = r34, r9
// N+1
st4 [r6] = r36, 4;;
// N
r34 = r33
r33 = r32
r36 = r35
br.cloop L1;;
```

в)

```
L1:
(p16) ld4 r32 =
[r5], 4    //N+2
(p18) add r35 =
r34, r9  //N+1
(p19) st4 [r6] =
r36, 4   //N
br.ctop L1;;
```

Рис. 3. Исходный цикл (а) и ядро конвейеризованного цикла с использованием явных пересылок между регистрами (б) и вращающихся регистров (в).

Наконец, последним рассмотрим поддержку в EPIC-архитектуре *вращающихся регистров*. Важной оптимизацией для выявления параллелизма на уровне команд является *программная конвейеризация* циклов, целью которой является такое планирование команд тела цикла, что итерации цикла выстраиваются в «конвейер», образуя пролог цикла, ядро из команд с различных итераций, и эпилог. В том случае, когда в ядре перекрываются сразу несколько итераций, часто необходимо выполнять переименование регистров, чтобы устранить ложные зависимости по регистрам между итерациями. Такое переименование обычно требует дополнительных операций пересылок между регистрами (см. рисунок 3(б), где такие пересылки показаны курсивом), причем если результат, записанный на предыдущей итерации в копируемый регистр, еще не готов, то обращение к этому регистру вызовет останов конвейера до завершения вычисления этого результата.

Избавиться от лишних пересылок регистров помогает механизм вращающихся регистров, представляющий из себя аппаратно поддерживаемое переименование регистров. Команды цикла используют виртуальные номера регистров, `r32-r127`, а команда `br.ctop` выполняет сдвиг окна отображения виртуальных регистров в физические таким образом, что происходит циклическое переименование: `r[i]=r[i-1]`, `i=1..N-1`, `r[0]=r[N-1]`, где `N` – размер вращающегося регистрового окна. Никаких физических пересылок значений между регистрами при этом не происходит. Более того, за счет использования вращающихся предикатных регистров автоматически генерируется пролог и эпилог цикла (см. рис. 3 (в)).

2.2. Алгоритм планирования команд и конвейеризации циклов для Intel Itanium

В ИСП РАН было выполнено несколько работ по улучшению производительности компилятора GCC для платформы Intel Itanium, в ходе которых разрабатывалась и реализовывалась поддержка в GCC рассмотренных выше свойств этой архитектуры. Первыми были закончены работы по добавлению поддержки спекулятивного выполнения в планировщик команд компилятора GCC, описанные в [[5]]. По результатам тестирования реализации на пакете тестов SPEC CPU 2000 [[23]] было получено ускорение в 2.5%, а на отдельных тестах – до 20%. Это позволило включить реализованную поддержку в официальные релизы компилятора GCC, начиная с версии 4.2.0. Кроме этого, были выполнены работы по улучшению точности низкоуровневого анализа алиасов, используемого в компиляторе GCC, и использованию более точных данных при планировании команд.

По результатам изначальных исследований по улучшению планирования команд было принято решение о разработке и реализации нового планировщика команд и конвейеризации циклов для EPIC-архитектур, основанного на подходе селективного планирования [[19]]. Алгоритм

селективного планирования был разработан для архитектур с очень длинным командным словом и хорошо подходит для экспериментов по увеличению производительности для EPIC-архитектур. Он поддерживает ряд полезных преобразований команд, позволяющих избавляться от части зависимостей по данным (переименование регистров, подстановка через копии), а также делает простым добавление новых преобразований.

2.2.1. Базовый алгоритм селективного планирования

Селективный планировщик является классическим итеративным планировщиком, обходящим регион планирования сверху вниз. Обрабатываются произвольные ациклические регионы графа потока управления программы, возможно, с несколькими входами. Поддерживается несколько точек планирования, к которым собираются доступные команды, называемых *барьерами*. Каждая итерация планировщика четко делится на этап *сбора доступных команд*, этап *выбора лучшей команды для планирования* и этап *перемещения выбранной команды*, при этом корректность программы обеспечивается этапом сбора и перемещения, а получаемая производительность полностью зависит от этапа выбора лучшей команды, который обычно является набором эвристик. После того, как на текущем цикле планирования для данного барьера невозможно выполнить больше команд, либо нет доступных для выполнения команд, обрабатывается следующий барьер. После обработки всех барьеров происходит передвижение барьеров через запланированные команды, и цикл планирования повторяется. Планировщик останавливается по достижении конца региона.

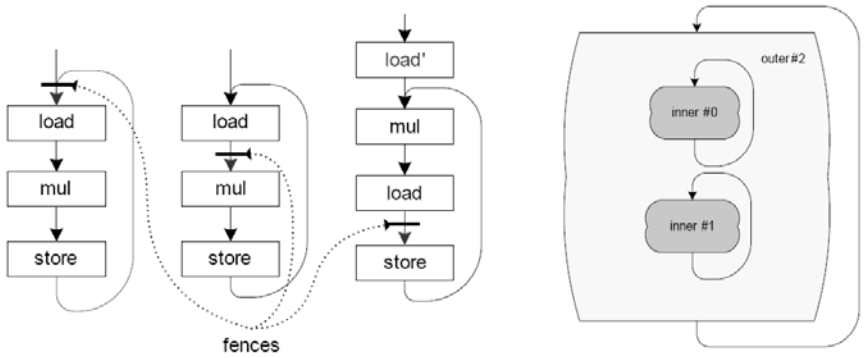

Рис. 4. Конвейеризация циклов в селективном планировании: продвижение барьеров во внутреннем цикле (слева), образование регионов для всего гнезда циклов (справа).

При сборе команд для планирования регион обходится в обратном топологическом порядке; при этом текущее множество собранных команд «протаскивается» через обрабатываемую команду на пути «наверх», и все команды, имеющие неустранимые зависимости по управлению либо по

данным с обрабатываемой, удаляются из множества. Все преобразования команд, ведущие к устранению зависимостей, могут быть реализованы на этом этапе. В точках разделения потока управления текущее множество доступных команд предварительно получается как объединение всех множеств, доступных на потомках обрабатываемой команды. Дополнительно в процессе сбора могут быть вычислены некоторые атрибуты команд (доступность вдоль разных путей, вероятность выполнения и т.п.), которые могут использоваться в дальнейшем при выборе лучшей команды для планирования. Промежуточные *множества доступных команд* сохраняются в начале каждого базового блока.

На этапе перемещения выбранной команды регион обходится аналогичным образом сверху вниз в поиске команд, которые могли быть преобразованы в выбранную, при этом используются сохраненные множества доступных команд – если искомая команда не содержится в сохраненном множестве, то её нет смысла искать ниже текущего места региона. Если команда найдена, то она удаляется из региона, а на обратном пути вверх обновляются сохраненные множества доступных команд и в точках слияния потока управления на путях, не лежащих на текущем пути обхода, создаются компенсационные копии выбранной команды. После окончания перемещения выбранная команда в преобразованном виде добавляется в поток команд в точке планирования, а все промежуточные множества доступных команд оказываются верными, что значительно ускоряет этап сбора команд для следующей итерации планирования.

Важным достоинством селективного планировщика является возможность конвейеризации циклов, вытекающая из поддержки перемещений команд с созданием компенсационных копий и из того, что перемещение команд через барьер запрещено. При планировании внутреннего цикла из гнезда циклов текущим регионом считается ациклический регион, получающийся из цикла разрывом тех дуг, на которых в данный момент стоят барьеры. В начале планирования разрывается обратная дуга цикла. На этапе сбора команд разрешается собирать уже запланированные команды – при планировании обычного региона такая ситуация запрещена. При перемещении уже запланированной команды вдоль обратной дуги на входе в цикл наблюдается слияние потока управления, и поэтому на дуге перед циклом будет создана компенсационная копия, образующая пролог конвейеризованного цикла (см. рис. 4). При планировании внешнего цикла пролог внутреннего цикла добавляется к региону планирования, а тело внутреннего цикла обрабатывается как «черный ящик», при этом перемещения команд через него запрещены.

2.2.2. Усовершенствования базового алгоритма

После реализации вышеописанного базового алгоритма для компилятора GCC и первоначальных экспериментов нами был разработан и реализован ряд

усовершенствований, улучшивших как показатели производительности алгоритма, так и время его работы. Во-первых, нами были реализованы дополнительные преобразования команд: спекулятивное выполнение команд и условное выполнение команд. Для поддержки обоих преобразований необходимо модифицировать этап сбора доступных команд, а также поиск и перемещение выбранной команды наверх к точке планирования. Спекулятивные команды для Intel Itanium создаются при протаскивании команды загрузки наверх либо через условный или безусловный переход (спекулятивность по управлению), либо через возможно зависимую команду записи в память (спекулятивность по данным). Для спекулятивных команд отслеживается вероятность выполнения зависимостей (одной или нескольких), нарушенных при превращении команды в спекулятивную форму. При обнаружении команды загрузки, породившей спекулятивную форму, помимо ее удаления создается команда проверки результата спекулятивного выполнения и код восстановления, а при создании компенсационной копии такой команды эта копия обязательно преобразуется в спекулятивную форму. Более детально поддержка спекулятивного выполнения в нашем планировщике команд описана в работах [[4], [7], [8]].

Команды для условного выполнения создаются при слиянии промежуточных множеств доступных команд в точке разделения потока управления: в зависимости от направления, с которого поступила команда, если она еще не была аннотирована предикатным регистром, то она аннотируется либо регистром, контролирующим условный переход в точке разделения потока, либо его отрицанием. При поиске выбранной в условной форме команды необходимо преобразовать эту команду в обычную форму ровно на том условном переходе, на котором к команде был добавлен предикат при сборе команд. Остальные этапы алгоритма, в том числе создание компенсационных копий, при обработке команд в условной форме не меняются.

Во-вторых, был выполнен ряд улучшений этапа выбора лучшей команды. В первую очередь, для выбора стал использоваться существующий механизм компилятора GCC, заключающийся в отслеживании конфликтов конвейера процессора через конечный автомат, описывающий функциональные устройства процессора [[15]]. Интерфейс автомата позволяет узнать необходимую задержку в тактах для выдачи данной команды в данном состоянии автомата. С помощью этого интерфейса в GCC реализован механизм локального перебора команд из множества готовых к выдаче на данном такте для поиска такой команды, выдача которой позволит выдать на данном такте наибольшее количество других готовых команд. Данный механизм был адаптирован нами для работы с вычисленным планировщиком множеством готовых команд.

Далее, в ходе этапа сбора доступных команд также вычисляется *полезность* команды, отражающая вероятностей выполнения тех путей графа потока управления, вдоль которых доступна эта команда. Полезность команды в

промежуточном множестве доступных команд умножается на вероятность перехода по дуге при протаскивании команды вверх вдоль этой дуги, а при объединении множеств в точке слияния потока управления полезности одинаковых команд, пришедших в эту точку по разным путям, складываются. Полезность готовой команды используется при сортировке готовых команд для выделения более приоритетных. Другими эвристиками при этой сортировке являются длина критического пути, начинающегося от команды, ее спекулятивность, а также вероятность выполнения зависимостей, нарушенных ее перемещением, если она спекулятивна. Кроме того, незапланированные команды предпочитаются запланированным, чтобы гарантировать окончание алгоритма при конвейеризации циклов.

Наконец, было выполнено большое количество исправлений реализации планировщика и кодогенератора GCC (более 30), которые явились результатом анализа производительности скомпилированных программ из пакета тестов SPEC CPU 2000. Приведем наиболее важные примеры. Переименование регистров применялось только к тем инструкциям, чья латентность превышает время выполнения инструкции копирования регистра в регистр. Это отсекает переименования, которые никогда не дадут выигрыша. Другим улучшением является запрет на применение преобразования переименования регистров (и спекулятивного выполнения команд по управлению) в тех случаях, когда результирующая инструкция будет запланирована на последнем такте цикла, и можно показать, что такое преобразование будет невыгодным. Далее, перепланирование конвейеризованного кода для достижения более плотного расписания в тех местах кода, из которых были перемещены инструкции, позволило нам улучшить ряд тестов SPEC на 0.5-1%. Этот дополнительный проход особенно полезен для маленьких циклов, в которых создаваемые конвейеризацией «дырки» имеют значение.

В-третьих, алгоритм планирования был ускорен по сравнению с базовым. Из основных улучшений, приведших к уменьшению времени работы алгоритма, можно перечислить следующие:

- кэширование результатов проноса команды через другую команду;
- сохранение полной «истории» преобразований, которым подверглась команда при проносе наверх, для быстрого «отката» этих изменений;
- применение переименования регистров только к самым приоритетным инструкциям;
- ограничение количества обновлений множества доступных команд так, чтобы множества обновлялись только после планирования нескольких команд на данном барьере;
- ограничение длины «окна» команд, которое просматривает планировщик в поисках кандидатов на выдачу, для прохода, на

котором выполняется перепланирование кода после конвейеризации.

По результатам тестирования усовершенствованного алгоритма планирования на платформе Intel Itanium было получено среднее ускорение в 3-4% на пакете тестов SPEC CPU FP 2000 (для разного набора базовых опций получено разное ускорение), а на отдельных тестах – до 10%. Часть результатов представлена в таблице 1. Мелким шрифтом выделен тест, который работает некорректно с текущей реализацией поддержки условного выполнения.

	База	Сел	Сел +Усл	Сел +Зав	Сел+Зав +Усл
168.wupwise	553	-2,35%	-2,35%	1,27%	1,45%
171.swim	754	1,46%	4,91%	0,93%	5,04%
172.mgrid	574	3,66%	3,83%	7,49%	8,01%
173.applu	531	3,95%	3,95%	3,77%	4,33%
177.mesa	774	1,42%	*1,42%*	2,58%	*1,42%*
178.galgel	856	2,45%	2,22%	3,50%	3,50%
179.art	2025	1,14%	6,17%	1,23%	6,22%
183.equake	509	8,64%	8,64%	6,88%	7,07%
187.facerec	959	-0,31%	0,52%	0,00%	0,42%
188.ammp	739	3,79%	4,19%	3,79%	4,19%
189.lucas	898	0,33%	-0,33%	-0,11%	0,00%
191.fma3d	549	-1,28%	-1,28%	0,55%	0,00%
200.sixtrack	325	0,00%	1,23%	8,92%	8,92%
301.apsi	538	1,30%	2,04%	4,65%	5,02%
SPEC FP Geo Mean	687,7963	1,70%	2,47%	3,21%	3,93%

Таблица 1. Результаты тестов SPEC FP для планировщика команд.

Исходные коды реализованного алгоритма планирования команд и конвейеризации циклов был включен в специальную ветвь компилятора GCC, доступную с официального сайта разработчиков. Кроме того, по результатам настройки алгоритм планирования был включен в основную ветвь разработки компилятора GCC, как планировщик по умолчанию для платформы Itanium, и будет доступен в следующем релизе компилятора версии 4.4.0.

Мы продолжаем работы над улучшением алгоритма, в первую очередь – над добавлением поддержки полного графа зависимостей по данным, что позволит как ускорить сам алгоритм, так и реализовать более эффективные эвристики для этапа выбора наилучшей команды. Также будут вестись работы над настройкой реализованного алгоритма на другие архитектуры, в частности, IBM Power6.

3. Оптимизации энергопотребления встраиваемых систем, управляемые компилятором

Исследования по оптимизации энергопотребления встраиваемых систем активно ведутся в последнее десятилетие. Из наиболее популярных направлений можно отметить динамическое изменение напряжения на процессоре и его частоты; оптимизации доступа к памяти, в том числе отключение неактивных банков памяти; оптимизацию энергопотребления на стадии разработки новых чипов и т.д. (хорошие обзоры можно найти в работах [[9], [20]]). В данном разделе рассматриваются программные оптимизации энергопотребления, управляемые компилятором. Мы исследовали несколько направлений таких оптимизаций с использованием компилятора GCC для архитектуры ARM: динамическое изменение напряжения, основанное на данных профиля программы; влияние оптимизаций работы с памятью на энергопотребление; оптимизацию переключения битов на шине команд через модификации планировщика команд. Тестирование оптимизаций проводилось с помощью пакетов Aburto [[2]], MediaBench [[16]] и MiBench [[18]] на платах OMAP2430 [[21]] и MV320 [[20]], содержащие процессоры ARM 11-го поколения. Рассматриваемые тестовые пакеты состоят из небольших приложений, представляющих из себя обработку изображений и звука, а также другие вычисления, типичные для встраиваемых систем.

В целом, проведенные исследования показали, что наиболее интересным подходом является динамическое изменение напряжения. В настоящее время мы развиваем прототипную реализацию этого метода в компиляторе GCC. Цикловые оптимизации, ускоряющие работу программы и снижающие энергопотребление, также являются многообещающими, однако в GCC мощная инфраструктура для таких оптимизаций появится лишь в версии 4.4.0, которая выходит в январе 2009 года.

3.1. Динамическое изменение напряжения

Основной идеей динамического изменения напряжения на процессоре (далее ДИН) является такое изменение напряжение на элементе питания чипа в некоторых точках программы (называемых *точками управления напряжением,* ТУН), что энергопотребление системы сокращается, при этом сохраняя (либо незначительно снижая) производительность. Возможность такой оптимизации обеспечивается тем, что потребляемая энергия квадратично зависит от подаваемого напряжения, тогда как частота процессора (а, следовательно, и производительность) зависит от напряжения лишь линейно.

Существует несколько классов алгоритмов ДИН, известные в литературе как статические (offline), динамические (online) и смешанные (mixed). Разница между этими классами заключается в моменте, в который принимается решение, во-первых, о местонахождении точек управления напряжением, и

во-вторых, о величине, на которую изменяется напряжение. Динамические алгоритмы ДИН принимают все эти решения во время работы программы (например, в планировщике ОС); статические алгоритмы определяют как точки, так и величины изменения напряжения во время компиляции (хотя непосредственно изменение напряжения также происходит во время работы программы); наконец, смешанные алгоритмы обычно вычисляют возможные точки изменения напряжения во время компиляции, а величина изменения определяется динамически.

Нами была выполнена реализация статического алгоритма ДИН, основанная на [[13]]. Выбранный алгоритм вставляет точки изменения напряжения в тех местах программы, основное время выполнения которых тратится на работу с памятью. Если в такой области кода понизить напряжение на процессоре, то снижения производительности не произойдет, так как процессор все равно вынужден ждать данных из памяти. Необходимым условием для этого является раздельное питание процессора и памяти, что обычно и бывает в современных системах. Как точки изменения, так и величины изменения напряжения вычисляются алгоритмом статически на основании данных профиля программы, при этом учитывается время, затрачиваемое на смену напряжения.

Мы рассматривали и другие статические алгоритмы ДИН в качестве кандидатов для исследований, но они либо тестировались только на симуляторах (а не на реальных встраиваемых системах либо ноутбуках), либо заключались в комбинировании классических цикловых оптимизаций с понижением напряжения, что может быть выполнено и независимо.

3.1.1. Реализованный алгоритм ДИН

Алгоритм обрабатывает т.н. *базовые* и *комбинированные* регионы. Базовым регионом является либо базовый блок, либо гнездо циклов. Комбинированный регион – это объединение базовых регионов, имеющее один вход и один выход, при этом вход доминирует, а выход постдоминирует регион. Это определение предоставляет больше возможностей по созданию регионов, чем поиск по набору шаблонов графа потока управления, как предлагается в [[13]]. Тем не менее, существует ряд дополнительных ограничений на регионы. Во-первых, в первоначальной реализации не рассматривались регионы, содержащие вызовы функций, так как алгоритм был внутрипроцедурным (в текущей реализации это ограничение снято). Во-вторых, регионы с «нетипичным» потоком управления (например, несколько дуг пересекают границы цикла) не обрабатываются. В-третьих, небольшие регионы также исключаются из рассмотрения, так как затраты на переключение напряжения наверняка превысят возможный выигрыш на таком регионе.

Алгоритм состоит из следующих основных шагов:

- Построение базовых и комбинированных регионов для данной функции.

- Профилирование времени выполнения, `T(R,v)`, и количества раз, `N(R)`, которое выполнился регион, для каждого базового региона на каждом доступном уровне напряжения.
- Вычисление этих величин для комбинированных регионов. Время выполнения считается как сумма времен по всем базовым регионам, составляющим данный комбинированный регион; количество выполнений берется из базового региона, находящегося на входе в комбинированный.
- Поиск такого региона, на котором понижение напряжения минимизирует энергопотребление системы во время выполнения программы, а сама программа замедляется не больше, чем на `p%`. Потребленная энергия оценивается по времени работы региона на данном уровне напряжения с учетом затрат на выполнение команд переключения напряжения.
- Вставка команд изменения напряжения в начале и конце выбранного региона.

Описанный алгоритм, как и многие другие алгоритмы ДИН, полагается на результаты профилирования программы. В нашей реализации для компилятора GCC используются уже имеющиеся в компиляторе механизмы, позволяющие профилировать количество выполнений базовых блоков и дуг графа потока управления. Дополнительно мы реализовали профилирование времен выполнения базовых блоков и циклов, входящих в комбинированные регионы (с помощью аппаратных счетчиков, если они есть в системе).

Исходная реализация алгоритма рассматривает лишь регионы внутри одной функции и только для двух уровней напряжения, а также понижает напряжение только для одного региона из имеющихся, что существенно упрощает поиск необходимого минимума энергопотребления. Интерфейс переключения напряжения реализован через встроенные функции компилятора GCC (builtins) и системные вызовы ОС Linux.

Тестирование реализации проводилось на пакете тестов Aburto и тестовой плате MV320. Из пакета предварительно было удалены тесты, калибрующиеся автоматически, так как они выполняют разный объем вычислений на разных частотах. В качестве базового использовался уровень оптимизации `-O2`.

Из 196 функций, содержащихся в программах пакета Aburto, наша реализация алгоритма нашла 144 функции, которые подходят для динамического изменения напряжения. Для значения параметра `p` допустимого замедления программы от 10% до 40% было найдено от 3 до 14 подходящих регионов соответственно. При запуске оптимизированной версии время работы составило 8 минут, а потребленная энергия – 750 мВч. Неоптимизированные программы работали 7 минут 30 секунд, требуя 720 мВч. При этом потребление незагруженной системы составило 59 мВч за 45 секунд. Вычитая это потребление из обоих результатов, получаем, что при замедлении системы

на 6.6% сокращение потребления энергии только процессором составило 7%. Если же принять за ограничение времени работы системы 8 минут, то за это время неоптимизированные версии программ потребили бы 759.3 мВч, что соответствует сокращению потребления оптимизированной версией на 1.24%.

В настоящий момент ведутся работы по реализации межпроцедурного алгоритма, в котором регионы могут содержать вызовы функций, а также вход и выход региона могут принадлежать разным функциям. Кроме того, разрабатывается эвристический алгоритм, понижающий напряжение на множестве регионов. По результатам предварительного тестирования, количество регионов, на которых происходит понижение напряжения, выросло в два раза, что позволяет предположить об увеличении эффективности алгоритма.

3.1.2. Оптимизация переключения битов (bit-switching)

Переключение битов, происходящее на шинах команд и данных, ответственно за значительную долю потребляемой процессором энергии [[24]]. Переключение происходит тогда, когда процессором обрабатывается очередная команда. Если битовые кодирования последовательных команд отличаются в некоторых битах, то на переключение дорожек шины для этих битов тратится энергия. Оптимизация переключения битов заключается в такой организации команд и их кодировок, что переключения на шине случаются как можно реже.

Мы исследовали вопрос о том, можно ли минимизировать переключения влиянием на порядок команд через планировщик команд компилятора. Во-первых, были выяснена верхняя оценка на количество энергии, которое можно сохранить через минимизацию переключения битов. Были подготовлены тесты, использующие команды с как можно более различающимся кодированием. Так, в битовой кодировке команд `ands r6,r8,r0` и `bicne r9,r7,#0x3FC` только 3 из 32 битов одинаковы. Из двух тестовых программ, первая содержала цикл из 1000 команд: 500 команд первого типа, за которыми следовали 500 команд второго типа; вторая содержала цикл из 500 пар команд первого и второго типа. Оба цикла выполнялись достаточное количество раз для того, чтобы имелась возможность замерить энергопотребление. Эксперименты с выполнением этих двух тестах показали, что разница в энергопотреблении составляет 1-2% для одной тестовой платы и около 5% для второй платы. Учитывая, что энергопотребление процессора является лишь частью энергопотребления всей системы, можно было утверждать, что экономия в энергопотреблении процессора составила около 10%.

Для минимизации переключения битов в компиляторе необходимо знать, как команда во внутреннем представлении компилятора будет закодирована в битовой форме. В случае компилятора GCC, результатом компиляции является ассемблерный листинг программы, а информации о кодировании команд нет, так как этим занимается ассемблер. Для преодоления этого

препятствия мы реализовали машинно-зависимую функцию (т.н. target hook), «предсказывающую» финальную кодировку команды во внутреннем представлении компилятора в той части, в которой это известно компилятору (то есть, за исключением вычисления адресов, неизвестных на этапе компиляции). При сравнении предсказанных кодировок с реально получившимися на ряде тестов обнаружилось практически полное совпадение, за исключением случаев, когда из данной команды во внутреннем представлении можно было сгенерировать несколько вариантов машинной команды, и в итоге был выбран менее вероятный вариант.

С помощью полученной функции была реализована новая эвристика для планировщика команд GCC, которая дает предпочтение командам, образующим меньшее количество переключений битов с предыдущей запланированной командой. Эвристика использует параметр, изменяющийся от 0 до 32, который может рассматриваться как количество одинаковых битов на шине команд, которые увеличивают приоритет этой команды на 1. Так, если параметр установлен в 5, и планировщик выбирает между двумя командами с приоритетами 3 и 4, которые оцениваются как переключающие 7 и 22 бита на шине команд соответственно, то приоритет первой команды составит 3+(32-7)/5=8, а приоритет второй команды – 4+(32-22)/5=6, и будет выбрана первая команда вместо второй.

При тестировании данной эвристики на пакете тестов Aburto максимальное сокращение переключений битов было зафиксировано на тесте `sim` и составило 7%, а в среднем – около 3%. К сожалению, этого недостаточно, чтобы значительно повлиять на энергопотребление. Возможно, одной из причин было то, что большое количество операций с плавающей точкой, реализованных через библиотечные вызовы, не позволяло достаточно точно предсказать кодирование этих операций. Аналогичные эксперименты с оптимизацией, комбинирующей несколько команд в одну, показали, что переключение битов меняется еще меньше, чем для планирования. Вообще говоря, видно, что для изменения энергопотребления на 1% необходимо изменить количество переключений битов как минимум на порядок больше, чего не получается достигнуть в рамках компилятора.

3.1.3. Оптимизация работы с памятью

Подсистема работы с памятью является одной из самых потребляющих компонентов встраиваемых систем. Мы проанализировали ряд оптимизаций доступа к памяти, имеющихся в компиляторе GCC. Так, префетчинг данных поддерживается для некоторых реализаций процессора ARM через команду `pld`, и, в частности, поддерживается на тестовой плате OMAP2430. Тестирование реализации префетчинга массивов в циклах в компиляторе GCC версий 4.2 и 4.3 показало, что некоторые тесты ускоряются при использовании префетчинга, а некоторые замедляются – общая картина получается достаточно противоречивой, чтобы не рекомендовать использовать

префетчинг по умолчанию для компиляции программ для данной тестовой платы. Другие машинно-независимые оптимизации, улучшающие производительность и, как следствие, уменьшающие энергопотребление, не дают большого эффекта в текущих версиях GCC для архитектуры ARM (автоматическая векторизация, преобразования циклов). Мы предполагаем, что в будущем, с появлением в GCC инфраструктуры Graphite для оптимизации циклов [[11]], можно будет разрабатывать цикловые оптимизации, имеющие своей целью, в том числе, уменьшение энергопотребления.

Кроме этого, известен ряд машинно-зависимых оптимизаций работы с памятью, направленных исключительно на энергопотребление. Например, скрэтч-память (scratch-pad memory) является по сути дополнительным кэшем, контролируемым компилятором. Использование такой памяти во встраиваемых системах позволяет экономить энергию, если память более эффективна, чем главная память, либо просто ускорять программу. К сожалению, в имеющихся у нас тестовых платах скрэтч-память присутствовала только в OMAP2430, и ее предназначение не позволяло использовать ее для этих целей. Оптимизация, отключающая неиспользуемые банки памяти, также возможна на тестовой плате OMAP2430, однако размер банка памяти в ней достаточно велик, и более разумным представляется распределять банки памяти по процессам в операционной системе вместо контроля распределения памяти компилятором.

4. Динамические оптимизации для языков общего назначения

При компиляции программы необходимо учитывать конкретные наборы входных данных компилируемой программы и особенности аппаратуры, на которой она будет выполняться. Практика применения современных оптимизирующих компиляторов показывает, что это способно ускорить выполнение программы на десятки процентов. В современных компиляторах для языков общего назначения (Си/Си++) не существует приемлемого решения этих задач.

Для учета наборов входных данных производится сбор профилей на заданном множестве наборов входных данных и учет полученной статистики. Отметим, что статистика на разных наборах данных может значительно отличаться, что в некоторых случаях приводит к замедлению программы. Такой подход связан со значительными накладными расходами на сбор профилей и подбор параметров компилятора.

Параметры архитектуры целевой машины (размер кэша, соотношение между частотой памяти и процессора, наличие специальных векторных инструкций) влияют на оптимизации обращений к памяти (префетчинг, оптимизации локальности), векторизацию, встраивание функций, развертку циклов и др. В

настоящее время проблема учета деталей архитектуры решается только за счет генерации многочисленных версий кода программы (даже в рамках одной аппаратной платформы имеется десятки версий), что неудобно и приводит к дополнительным накладным расходам.

Для оптимизации программы с учетом профиля пользователя планируется рассмотреть следующие подходы:

1. Динамическая оптимизация во время работы программы (JIT). Имеет то преимущество, что программа оптимизируется на конкретном наборе входных данных для данного конкретного запуска. Собранная статистика используется только для оптимизации данного запуска. Разные запуски программы могут приводить к различным оптимизациям. Необходим баланс между уровнями оптимизации «холодного» и «горячего» кода.

   JIT-оптимизации на языке Java, учитывающие профиль пользователя, подробно исследованы. Максимальный эффект в этом случае дают: оптимизация встраивания функций, развертка циклов, оптимизация обращений к памяти и распределение регистров. Эти оптимизации могут быть применены и в JIT-компиляторе для Си/Си++. Меньшая эффективность от этих оптимизаций из-за необходимости сложного анализа алиасов для Си/Си++ не уменьшает их актуальности.
2. Статическая оптимизация между запусками программы. Статистика накапливается между запусками, во время остановки программы выполняется оптимизация. Этот подход ближе к обычной оптимизации с учетом профиля программы, однако, не требует наличия JIT-компилятора.
3. Оптимизация выполняется динамически, однако данные статистики и принятые решения по оптимизации сохраняются между запусками. Позволяет уменьшить расходы на JIT-оптимизацию при условии того, что похожий набор данных уже встречался и был оптимизирован.

Для оптимизации программы с учетом конкретной архитектуры пользователя будут рассмотрены следующие подходы:

1. Динамическая оптимизация во время работы программы, применяемая только к «горячим» участкам кода (аналогично пункту 1 для оптимизаций с учетом профиля).
2. Статическая оптимизация во время установки программы. Для этого требуется лишь распространение программы во внутреннем представлении, компилятор и компоновщик на стороне пользователя, а виртуальная машина и JIT-компилятор не требуются. Этот подход используется при развертывании .NET-программ (оптимизатор NGEN от Microsoft).

В качестве основы для проведения работ мы выбрали систему LLVM (Low Level Virtual Machine) [[14]] с открытыми исходными кодами на языке Си++,

поддерживаемый компанией Apple. Все необходимые компоненты – внутреннее представление достаточно высокого уровня, компоновщик, виртуальная машина, JIT-компилятор – представлены или разрабатываются в рамках проекта LLVM. Из-за модульной организации и высокоуровневого языка реализации LLVM является популярным исследовательским компилятором. В LLVM была предложена концепция “lifelong optimization”, представляющая из себя компоненты для оптимизации программы на всем жизненном цикле ее существования, включая оптимизацию на машине пользователя. Кроме того, LLVM поддерживает межмодульные оптимизации и JIT-компиляцию, но не оптимизацию на стороне пользователя. В компании Apple реализован JIT-компилятор для OpenGL программ с помощью LLVM, позволивший отказаться от специализированного JIT-компилятора, использовавшегося до этого, и значительно улучшить производительность графических операций.

Следовательно, ожидаемым результатом работ для нас является система на базе LLVM, функционирующая как на машине разработчика, так и на целевой машине, и использующая динамические оптимизации для учета конкретных входных данных пользователя и специализации под машину пользователя. Для выполнения этой цели по вышеперечисленным направлениям нами были выделены следующие работы:

- исследование и разработка системы поддержки времени выполнения для LLVM, позволяющей осуществлять динамический мониторинг и профилирование работы программы – необходимо организовать интерпретацию программы во внутреннем представлении LLVM, динамическое малозатратное профилирование программы, сохранение результатов профилирования в промежуточных файлах;
- исследование и разработка динамических оптимизаций, которые применимы к языкам общего назначения C/Си++, а также реализация выбранных оптимизаций с учетом профиля программы в JIT-компиляторе LLVM;
- исследование и разработка подсистемы оптимизации программы во внутреннем представлении LLVM с учетом параметров целевой машины.
- сравнительный анализ возможностей статического компилятора с возможностями JIT-компилятора LLVM с использованием пакета тестов SPEC CPU2006 и на реальных приложениях.

Необходимо также отметить, что с развитием инфраструктуры для оптимизаций времени компоновки в компиляторе GCC часть разработанных технологий можно будет перенести в GCC – например, выполнять оптимизации на машине пользователя над внутренним представлением, сохраненным в объектных файлах. Выполнение этой работы позволит сделать доступным эти технологии для более широкого круга пользователей.

5. Заключение

Мы выполнили краткий обзор части работ, которые проводятся по компиляторным технологиям для современных архитектур в Институте системного программирования РАН. Завершенные работы по оптимизациям для архитектуры Intel Itanium, проводившиеся в течение последних трех лет, привели к среднему ускорению тестов SPEC CPU FP 2000 около 10%. При этом большинство реализаций, в том числе новый планировщик команд и конвейеризатор циклов, были включены в официальную версию компилятора GCC.

Наиболее важными для нас текущими работами являются разработка энергосберегающих оптимизаций для архитектуры ARM, выполняемая по контракту с компанией Samsung, и разработка методов динамической оптимизации для языков общего назначения. Первые результаты по энергосберегающим оптимизациям уже получены и позволяют утверждать, что динамическое изменение напряжения, управляемое компилятором, может быть полезным для встраиваемых систем на базе процессора ARM. В качестве основы для этих работ мы используем популярный компилятор GCC с открытыми исходными кодами, а также планируем использовать исследовательский компилятор LLVM.

Литература

[1] Arutyun Avetisyan, Andrey Belevantsev, and Dmitry Melnik. GCC instruction scheduler and software pipelining on the Itanium platform. 7th Workshop on Explicitly Parallel Instruction Computing Architectures and Compiler Technology (EPIC-7). Boston, MA, USA, April 2008. http://rogue.colorado.edu/EPIC7/avetisyan.pdf

[2] Alfred Aburto's system benchmarks. ftp://gd.tuwien.ac.at/perf/benchmark/aburto

[3] Andrey Belevantsev, Alexander Chernov, Maxim Kuvyrkov, Vladimir Makarov, Dmitry Melnik. Improving GCC instruction scheduling for Itanium. In Proceedings of GCC Developers' Summit 2005, Ottawa, Canada, June 2005, pp.1-14.

[4] Andrey Belevantsev, Maxim Kuvyrkov, Vladimir Makarov, Dmitry Melnik, Dmitry Zhurikhin. An interblock VLIW-targeted instruction scheduler for GCC. In Proceedings of GCC Developers' Summit 2006, Ottawa, Canada, June 2006, pp.1-12.

[5] А. Белеванцев, М. Кувырков, Д. Мельник. Использование параллелизма на уровне команд в компиляторе для Intel Itanium. Труды ИСП РАН, т.9, 2006, с.9-22.

[6] Andrey Belevantsev, Maxim Kuvyrkov, Alexander Monakov, Dmitry Melnik, and Dmitry Zhurikhin. Implementing an instruction scheduler for GCC: progress, caveats, and evaluation. In Proceedings of GCC Developers' Summit 2007, Ottawa, Canada, July 2007, pp. 7-21.

[7] Andrey Belevantsev, Dmitry Melnik, and Arutyun Avetisyan. Improving a selective scheduling approach for GCC. GREPS: International Workshop on GCC for Research in Embedded and Parallel Systems, Brasov, Romania, September 2007. http://sysrun.haifa.il.ibm.com/hrl/greps2007/

[8] А.А.Белеванцев, С.С.Гайсарян, В.П.Иванников. Построение алгоритмов спекулятивных оптимизаций. Журнал Программирование, N3 2008, с. 21-42.

[9] L. Benini and G. Micheli. System-level power optimization: Techniques and tools. ACM Transactions on Design Automation of Electronic Systems, 5:115–192, April 2000.

[10] GCC, GNU Compiler Collection. http://gcc.gnu.org

[11] Graphite GCC framework. http://gcc.gnu.org/wiki/Graphite

[12] K. Flautner, S. Reinhardt, T. Mudge. Automatic performance setting for dynamic voltage scaling. Proceedings of the 7th Annual international Conference on Mobile Computing and Networking, pp.260-271, 2001.

[13] C. Hsu. Compiler-Directed Dynamic Voltage and Frequency Scaling for CPU Power and Energy Reduction. Doctoral Thesis, Rutgers University, 2003.

[14] LLVM Compiler. http://llvm.net

[15] Vladimir Makarov. The finite state automaton based pipeline hazard recognizer and instruction scheduler in GCC. In Proceedings of GCC Developers' Summit, Ottawa, Canada, June 2003.

[16] MediaBench Test Suite. http://euler.slu.edu/~fritts/mediabench/

[17] Dmitry Melnik, Sergey Gaissaryan, Alexander Monakov, Dmitry Zhurikhin. An Approach for Data Propagation from Tree SSA to RTL. GREPS: International Workshop on GCC for Research in Embedded and Parallel Systems, Brasov, Romania, September 2007.

[18] MiBench Test Suite. http://www.eecs.umich.edu/mibench/

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

[20] MV320 ARM Board. http://mvtool.co.kr/products/product.php?query=list&code=100101&lv=3&lang=

[21] OMAP2430 Development Board. http://focus.ti.com/general/docs/wtbu/wtbugencontent.tsp?contentId=14645&navigationId=12013&templateId=6123

[22] H. Saputra, M. Kandemir, N. Vijaykrishnan, M.J. Irwin, J. Hu, C.-H. Kremer. Energy conscious compilation based on voltage scaling. In ACM/SIGPLAN Joint Conference on Languages, Compilers, and Tools for Embedded Systems Software and Compilers for Embedded Systems, pp. 2-11, June 2002.

[23] SPEC CPU 2000. http://spec.org/cpu2000/

[24] Ching-Long Su, Chi-Ying Tsui, and A.M. Despain. Low power architecture design and compilation techniques for high-performance processors. Compcon Spring '94, Digest of Papers, pp.489-498, 1994.

[25] V. Venkatachalam and M. Franz. Power reduction techniques for microprocessor systems. ACM Comput. Surv. 37, 3 (Sep. 2005), pp. 195-237.