Автоматическая настройка оптимизационных преобразований компилятора GCC для платформы ARM

Роман Жуйков <zhroma@ispras.ru>,
Дмитрий Плотников <dplotnikov@ispras.ru>,
Мамикон Варданян <mamikon@ispras.ru>

Аннотация. В данной работе описывается созданная в ИСП РАН система для автоматической настройки параметров компиляции, разработанная для использования на встраиваемых платформах. Система включает в себя средства анализа полученных результатов. С помощью этих средств были выявлены недочеты в работе компилятора GCC, приводящие к генерации неоптимального кода для платформы ARM. Были выявлены причины генерации неоптимального кода, а также рассмотрены и реализованы различные подходы к улучшению оптимизаций. Также была проведена работа по улучшению других оптимизаций компилятора GCC, недостатки которых были обнаружены при ручном анализе ассемблерного кода. Получено значительное увеличение производительности выбранных тестовых приложений на платформе ARM.

Ключевые слова: генетические алгоритмы; оптимизация программ; векторизация; предварительная загрузка данных; 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 – рассмотреть потенциальные возможности такой адаптации на примере компилятора GCC[1] с открытым исходным кодом и процессоров архитектуры ARM. Данное исследование выполняется в рамках проводимых в отделе компиляторных технологий ИСП РАН работ по оптимизации программ для современных вычислительных архитектур[2]. Планируется добиться генерации более качественного кода для встраиваемых систем без внесения значительных изменений в инфраструктуру компилятора GCC.

GCC является многоплатформенным компилятором и содержит множество оптимизаций и их параметров, которые могут влиять на итоговый машинный код. В основном, большинство оптимизаций настроено на получение качественного кода для платформ x86 и x86-64. Учитывая сложность GCC, существуют большие возможности для его улучшения, путем настройки компилятора для одной конкретной архитектуры. Улучшения могут быть нескольких видов. Во-первых, могут быть существенные недостатки при генерации кода для менее популярных архитектур. Исправления для них обычно требуют добавления новых шаблонов инструкций или настройки платформозависимых параметров. Во-вторых, машинно-независимые оптимизации могут быть не оптимально настроены для определенной архитектуры, в этом случае для достижения лучшего результата может быть изменено их поведение по умолчанию. Например, при распределении регистров могут использоваться разные ограничения для различных архитектур. Наконец, есть возможность написать новую оптимизацию для учета особенностей конкретной архитектуры. Например, в ИСП РАН была реализована поддержка спекуляции в планировщике в соответствии с особенностью Intel Itanium. Однако, такой вид улучшения может потребовать большое количество времени и ресурсов.

Производительность сгенерированного кода имеет первостепенное значение для компиляторов. Для достижения хорошей производительности обычно недостаточно реализовать платформозависимые оптимизации, также необходима настройка уже существующих платформонезависимых оптимизаций. Настройка для архитектуры ARM выполнялась с использованием нескольких тестовых приложений, для каждого из которых была сделана попытка добиться максимально возможной производительности на целевой платформе. Для улучшения работы компилятора с тестовыми приложениями можно использовать ручной анализ ассемблерного кода. Такой анализ требует большого количества времени, поэтому мы создали инструмент, который позволяет выявить часть недостатков в оптимизационных платформонезависимых преобразованиях, приводящих к генерации неоптимального кода на выбранных тестовых приложениях.

APTT (Automatic Performance Tuning Tool, Инструмент для Автоматической Настройки Производительности) позволяет автоматически подбирать лучшие опции и параметры компилятора для заданной программы и наборов входных данных. Разработанный инструмент основан на ACOVEA[3] (Analysis of Compiler Optimizations via an Evolutionary Algorithm), которая позволяет искать лучшие опции при помощи генетических алгоритмов, но имеет ряд недостатков. Они были исправлены одновременно с расширением функциональности.

В данной статье описываются найденные в компиляторе GCC недочеты и выявляются причины генерации неоптимального кода, а также рассматриваются различные подходы к улучшению оптимизаций.

Дальнейшее изложение построено следующим образом. В разделе 2 описываются используемые тестовые приложения. Раздел 3 рассказывает о разработанной системе для автоматического подбора опций компиляции. Раздел 4 посвящен проделанной работе над оптимизацией преобразования ветвлений в GCC. В разделе 5 описываются найденные в оптимизации предварительной загрузки данных недочеты и способы их исправления. Раздел 6 содержит описание изменений, касающихся автоматической векторизации циклов.

2. Тестовые приложения

Для тестирования оптимизаций компилятора GCC были выбраны три библиотеки с открытым исходным кодом. Эти библиотеки широко используется на современных встраиваемых системах с процессорами ARM. Примерами таких устройств являются мобильные телефоны и планшеты, пользовательские оболочки которых реализованы с использованием функциональности нижеописанных библиотек.

Первая библиотека – *libevas*. По результатам профилирования, на выполнение кода этой библиотеки уходит более 95% времени выполнения набора тестов *expedite* для EFL. Enlightenment Foundation Libraries [4] - набор свободно распространяемых графических библиотек для оконного менеджера Enlightenment. Используемый нами набор тестов *expedite* был урезан для оптимизации времени тестирования и получения большей стабильности результатов, без уменьшения покрытия всех главных функции *libevas*, для которых наиболее важно улучшение производительности.

Еще одна библиотека для тестирования – *sqlite* [5]. Она представляет собой легковесную встраиваемую реализацию реляционной базы данных, которая может использоваться для хранения пользовательской информации во встраиваемых системах. В качестве тестов для *sqlite* использовался набор из 20 тестов, содержащий большую часть операций языка SQL – включая добавление, изменение, поиск и удаление данных, и разнообразное выполнение комбинаций этих операций в виде единой транзакции. Для уменьшения влияния накладных расходов при работе с базой данных использовалась виртуальная файловая система в оперативной памяти.

В качестве третьего тестового приложения использовался *webkit* [6]. Это инфраструктура с открытым исходным кодом для отображения веб-страниц. Сейчас он используется в нескольких браузерах и других программах, нуждающихся в отображении и редактировании содержимого веб-страниц. Для тестирования производительности использовался набор тестов SunSpider для JavaScript, включенный в репозиторий *webkit*. К сожалению, рассмотренные нами тесты, измеряющие время визуализации HTML-кода, оказались очень нестабильными. Время различалось до 10-15% от запуска к запуску, несмотря на все попытки уменьшить накладные расходы, например организацию чтения страницы из оперативной памяти.

Мы адаптировали все библиотеки вместе с их тестами для использования с разработанным инструментом APTT, который описан в следующем разделе.

3. Реализация автоматизированного инструмента настройки

Для автоматического выявления недостатков в оптимизационных платформонезависимых преобразованиях, была решено создать систему, которая позволяет автоматически подбирать лучшие опции и параметры компилятора для заданной программы и наборов входных данных. Такая система позволяет улучшать работу компилятора следующим образом. Предположим, какие-то опции или параметры компиляции увеличивают производительность некоторого набора приложений на данной платформе. Можно оценить влияние найденных опций на генерируемый код в сравнении с оптимизацией по умолчанию, и соответствующим образом изменить поведение компилятора для этой платформы. Например, если из результатов тестирования следует, что отключение какой-либо машиннонезависимой оптимизации влечет увеличение производительности, то необходимо отключить данный оптимизационный проход на используемой платформе, или выяснить и устранить причины, по которым этот проход приводит к генерации неоптимального кода.

В инструменте ACOVEA реализован поиск лучшего набора опций компиляции с использованием генетического алгоритма. Основная идея генетического алгоритма использует схему естественного отбора. Сначала создается первое поколение особей – множество произвольных случайных наборов опций, для которых измеряется производительность. Далее среди наборов, которые позволили получить лучшие результаты тестов, производится скрещивание — обмен случайной частью набора опций. Из полученного таким образом второго поколения особей для скрещивания вновь выбираются только наборы опций, показавшие лучшую производительность. После создания заданного пользователем количества поколений поиск считается завершенным и в качестве результата эволюции предлагается тот набор опций, который дает самую лучшую производительность.

Разработанная нами система APTT выполняет эволюционный поиск параметров компилятора с использованием модифицированной версии ACOVEA и обеспечивает удобную инфраструктуру для сборки, запуска приложений, проверки и анализа результатов настройки. С помощью сбора дополнительной информации об использовании опций компилятора во всех поколениях генетического алгоритма получаем более детальные данные для анализа. Кроме того, была организована поддержка кросс-компиляции, что позволяет производить работу генетического алгоритма отбора и компиляцию тестовых приложений на одном компьютере, а запуск тестов на другом компьютере, например, тестовой плате с процессором ARM.

Инструмент APTT состоит из двух основных скриптов - aptt-host и aptt-target. Первый скрипт выполняет основные функции по поиску и фильтрации опций: подготовка структуры каталогов, запуск генетического алгоритма, анализ полученных результатов. Для реализации возможности кросс-компиляции, исходные тексты ACOVEA были модифицированы так, чтобы не производилась сборка и запуск тестового приложения напрямую. Теперь строка опций компиляции передается специальному скрипту host-acovea-proxy. Это скрипт выполняет кросс-компиляцию приложения с указанными опциями и посылает сообщение, что тестовое приложение готово к запуску. Сообщение обрабатывается скриптом aptt-target, запущенным на тестовой плате. После выполнения набора тестов aptt-target отсылает сообщение с результатами скрипту host-acovea-proxy, он в свою очередь передает результаты генетическому алгоритму.

Далее в этом разделе описывается еще несколько особенностей системы APTT, которые были реализованы для удобства ее использования при настройке компилятора GCC для платформы ARM.

3.1. Единая структура для развертывания приложений

Система для автоматической настройки состоит из одного многоядерного x86 компьютера для кросс-компиляции необходимых приложений и нескольких тестовых плат, связанных таким образом, что все устройства имеют один общий каталог через сетевую файловую систему NFS. APTT использует определенную структуру каталогов для хранения оригинальных источников приложения, общих ресурсов (библиотек/инструментов, которые сами не являются предметом для оптимизации, но необходимы целевому приложению для запуска), скриптов и каталогов для сборки, запуска приложения и хранения информации о его профиле (если включен запуск с профилированием). По своей структуре данная организация каталогов имеет аналогию с наследованием в объектно-ориентированных языках программирования, позволяя выносить общие действия и настройки на верхние уровни. Для того, чтобы добавить новые приложения для настройки, необходимо скопировать структуру каталогов из шаблона, после чего настроить файлы конфигурации и сценариев для конкретного приложения. Такая структура облегчает добавление новых тестов для настройки и позволяет реализовать часто выполняемые задачи в системе настройки.

3.2. Параллельная сборка и выполнение

Поддержка параллельной компиляции и выполнения значительно ускоряет процесс работы генетического алгоритма. Параллелизм используется на двух уровнях. Во-первых, APTT позволяет компилировать приложения во время ожидания результатов выполнения ранее скомпилированного приложения. Во-вторых, допускается использовать несколько тестовых плат в процессе работы, запуская целевые приложения одновременно на всех платах.

Еще одной особенностью является то, что тестовые платы, используемые для настройки, не обязательно должны быть одной и той же модели, одного производителя или с одинаковой скоростью процессора. Параллелизм на уровне плат осуществляется на уровне популяций генетического алгоритма, и напрямую сравниваются друг с другом только результаты производительности, полученные в одной популяции, то есть на одной тестовой плате. Эволюция происходит независимо в каждой популяции. Однако заложена возможность миграции между популяциями, которые позволяют лучшим комбинациям опций GCC из каждой популяции распространять себя в другие популяции, таким образом, продолжая их конкуренцию с "местными видами" из этой популяции. Обычно рекомендуется использовать от 2 до 4 популяций для оптимизации. Для оптимизации нагрузки количество популяций должно быть кратно числу используемых тестовых плат. В наших тестах мы провели настройку приложений на трех различных тестовых платах (TI OMAP 3540, EBV Beagle Board и Samsung C110, все с процессором ARM Cortex A8) и результаты, полученные по итогам оптимизации, были корректны и воспроизводимы на любой из трех плат.

3.3. Поддержка компиляции с профилированием

Структура APTT позволяет с его помощью настраивать приложения с учетом оптимизаций с профилированием, а не только со статическими оптимизациями. Оптимизации с профилированием включаются в GCC следующим образом: сначала приложение компилируется с опцией -fprofile-generate, затем при исполнении оно сохраняет информацию о профиле приложения (например, считает выполнения циклов, вероятности выполнения базовых блоков/путей потока управления и т.д.) в .gcda файлы, которые, как правило, находятся в каталоге для сборки приложения. После этого приложение перекомпилируется с опцией -fprofile-use, используя ранее собранную информацию о профиле. В автоматических инструментах оптимизация с профилем приводит к затратам двойного времени, необходимого для оценки того же количества особей. Задача системы управления очередью в APTT —позволить чередование выполнения двух этапов (сбора профиля и окончательной оценки) в различных каталогах сборки так, чтобы свести к минимуму время простоя тестовых плат, потому что обычно время запуска теста превышает время сборки приложения.

3.4. Быстрая перекомпиляция

Чтобы перекомпилировать приложение от запуска к запуску, допускается создание отдельного скрипта rebuild-pool, который должен перекомпилировать приложение с новыми опциями. Иногда тестовые приложения содержат большое количество кода и их компиляция с нуля с новыми опциями занимает длительное время. Если это время превышает время запуска теста на тестовой плате, для ускорения процесса тестирования

можно изучить профиль выполнения тестового приложения и определить файлы, содержащие функции, выполнение которых занимает большую часть времени работы приложения. Обычно количество таких файлов достаточно мало по сравнению с размером всего приложения. После чего можно внести в скрипт rebuild-pool только необходимые файлы. Остальные файлы приложения при этом будут всегда скомпилированы с опциями по умолчанию. Такой подход для тестового приложения *webkit* позволяется сократить время перекомпиляции с 15 до 3 минут, обеспечивая практически те же результаты оптимизации по итогам работы генетического алгоритма.

3.5. Сокращение итоговой строки опций компилятора

Набор флагов, полученных в результате работы генетического алгоритма, может быть избыточным: некоторые флаги могут в контексте других флагов не изменять поведение компилятора, но они будут по-прежнему присутствовать в результирующей строке опций. Мы написали программу, которая пытается уменьшить количество флагов для лучших особей за счет удаления тех опций, присутствие которых не влияет на итоговые бинарные файлы. Она пытается достичь этого путем последовательного удаления опций по одной, этот процесс может занять достаточно длительное время.

После использования инструмента на наших тестах мы смогли сократить лучшие по производительности наборы опций до не более чем 40 значимых опций, несмотря на то, что полный набор опций GCC для настройки состоит более чем из 200 опции. Существует потенциальная возможность еще сильнее сократить полученную строку опций за счет исключения из нее опций, которые мало влияют на производительность. Однако данный метод требует множества запусков целевого приложения на тестовых платах для определения точного вклада каждой из опций в итоговую производительность, а также, в некоторых случаях, ручной оценки существенности данного вклада.

3.6. Инструменты анализа результатов

Для облегчения анализа результатов при автоматической настройке мы разработали несколько скриптов – например, анализатор результатов эволюции и графический инструмент для проверки процесса эволюции. Скрипты используют записанные в виде файлов результаты запуска, в которых для каждой строки опций компиляции хранится результат ее производительности, а также номер текущего поколения и популяции.

Первый скрипт направлен на изучение булевых опций, которые могут иметь только два состояния - включена либо выключена. Скрипт по файлу с результатами всех запусков строит таблицу с опциями и параметрами GCC, упорядоченными по их полезности, которая рассчитывается как средняя производительность запусков, при компиляции которых использовалась данная. Кроме того, он показывает, сколько раз встречается каждая опция и

вычисляет ее среднее положение в логе запуска. Первое значение показывает, насколько часто встречается данная опция у особей, а второе показывает, насколько "зрелой" является каждая опция. Большее среднее значение позиции означает, что опция имеет тенденцию чаще появляться в конце эволюционного процесса. Чем больше эти два значения для опции, тем значимее выгода при использовании этой опции. Кроме того, этот инструмент помогает диагностировать те опции или параметры, которые вызывают ошибки при компиляции, либо приводят к генерации некорректного кода. При неудачном выполнении тестовой программы ставится заведомо самая плохая оценка производительности у данной особи (например, время выполнения 10^9 секунд). Опции, вызывающие ошибки, могут быть легко идентифицированы по их низкому положению в таблице результатов, отсортированной по среднему времени выполнения особей с такой опцией. Необходимо заметить, что такие опции или их комбинации незначительно влияют на работу генетического алгоритма, поскольку быстро исчезают в процессе эволюции.

Второй скрипт предназначен для проверки эволюции числовых параметров. Он представляет собой графический инструмент, который по информации о запусках тестовой программы генерирует некоторое количество веб-страниц с набором диаграмм. Они отображают, как распределение значений параметров варьируется между поколениями и популяциями для каждого настраиваемого параметра, что позволяет увидеть прогресс генетического алгоритма по поколениям. Наблюдение за процессом эволюции в такой визуальной форме помогает разработчику выбрать значения параметров, которые обеспечивают последовательное улучшение производительности.

Также, как часть APTT, мы разработали скрипт, который выбирает заданное число лучших строк опций и параметров компилятора, среди найденных в процессе эволюции, и перепроверяет их достоверность на указанной тестовой плате. После чего генерируется отчет в формате, совместимом с Excel, в котором показаны результаты работы тестов, в том числе относительный прирост производительности по сравнению с базовой оптимизацией –O2.

По результатам работы APTT с компилятором GCC на платформе ARM на используемых тестах были выбраны оптимизации компилятора GCC для дальнейшего исследования и ручного анализа. В следующих разделах рассматриваются самые важные улучшения, произведенные в оптимизациях компилятора GCC.

4. Преобразование ветвлений

Для тестового приложения *sqlite* инструмент APTT показал что опция -fno-if-conversion дает значительный прирост производительности. Данная опция выключает используемую по умолчанию оптимизацию преобразования ветвлений[7], при которой условные переходы удаляются, а все команды в базовых блоках, выполнение которых зависело от этого перехода, защищаются соответствующим предикатом. Преобразование применяется

только к тем базовым блокам, которые не содержат вызовов функций и длинных цепочек зависимостей по данным. Такое преобразование уменьшает количество сбросов конвейера и дает другим оптимизациям (например, планировщику инструкций) большую свободу в преобразовании кода и нахождении параллелизма на уровне инструкций.

Преобразование ветвлений требует поддержки со стороны процессора: необходимо наличие условной формы инструкций, которые выполняются, только если предикат выполнен, и работают как инструкция nop в противном случае. На платформе ARM любая инструкция может быть сделана условной по значению флагов. Например, moveq выполнит mov только если установлен флаг нулевого результата.

Были исследованы причины того, что эта оптимизация приводит к снижению производительности на ARM. Изучение получаемого ассемблерного кода показало, что проблему можно свести к следующему примеру:

```
int f(int a) {
 int z;

 if (a)
  z = 0x66667777;
 else
  z = 0x99998888;
 return z;
}
```

а)

```
ldr   r3, .L4
ldr   r2, .L4+4
cmp   r0, #0
moveq r0, r2
movne r0, r3
bx    lr
```

б)

```
cmp   r0, #0
ldrne r0, .L4
ldreq r0, .L4+4
bx    lr
```

в)

Рис 1. Пример неоптимального преобразования ветвления. а) Исходный код на C; б) ARM-ассемблер для -O2; в) ARM-ассемблер для -O2 -fno-if-conversion

В коде ассемблера на рис. 1*(б)* две загрузки из памяти выполняются всегда (без предикатов), хотя в дальнейшем используется результат только одной из них. В коде на рис. *1(в)* такой недостаток отсутствует, поэтому он выполняется быстрее. Причина создания данных двух загрузок – длинные целые константы в исходном коде на C.

Опция –fif-conversion включает в GCC оптимизационные проходы *ce1* и *ce2,* которые выполняются до распределения регистров. Аналогичный оптимизационный проход *ce3*, включаемый опцией –fif-conversion2, выполняется после распределения регистров. Причина изучаемой проблемы была выявлена в оптимизационном проходе *ce1*. Используется несколько методов преобразования ветвлений, и один из них – это замена присваиваний из базовых блоков ветвления в одно присваивание, которое использует RTL-выражение if-then-else. Впоследствии данное выражение преобразуется в

условную команду пересылки, как в примере выше. Компилятор пытается предсказать, будет ли возможно создать условную команду пересылки. Если аргумент – длинная константа, которая не может быть подставлена в инструкцию как непосредственное значение, простая условная команда пересылки не подходит, и используется безусловная загрузка длинной константы с последующей условной пересылкой из регистра. Для вышеописанного примера, RTL-код выглядит следующим образом:

```
r137:SI=0xffffffff99998888
r138:SI=0x66667777
cc:CC=cmp(r135:SI,0x0)
r133:SI={(cc:CC==0x0)?r137:SI
:r138:SI}
```

а)

```
r3:SI=0x66667777
r2:SI=0xffffffff99998888
cc:CC=cmp(r0:SI,0x0)
(!cc:CC) r0:SI=r2:SI
 (cc:CC) r0:SI=r3:SI
```

б)

Рис 2. RTL-код при преобразовании ветвлений а) после прохода ce1 ; б) после прохода split2

В этом коде, выражение “r133:SI={(cc:CC==0x0)?r137:SI:r138:SI}” описывает условную команду пересылки, которая преобразуется в RTL-код на рис. 2*(б)* оптимизационным проходом *split2*.

Если включен только один оптимизационный проход *ce3*, работающий после распределения регистров, то к моменту его работы компилятор уже создает необходимые инструкции загрузки длинных констант в регистр и может их преобразовать в условную форму.

Очевидным решением кажется отключение ранних оптимизационных проходов преобразования ветвлений и включением только *ce3* прохода с помощью опций –fno-if-conversion –fif-conversion2. Однако, данная комбинация отключит часть полезных преобразований, выполняемых оптимизационным проходом *ce1*. Например, *ce1* может преобразовать выражение “x > 0 ? x : 0”, которое обычно требует трех инструкций (сравнение, условное ветвление и пересылка) в инструкцию “bicle r8, r0, r0, asr #31” (ее семантика такова: очистить в регистре r0 те биты, которые равны 1 во втором аргументе, и сохранить результат в регистре r8; здесь второй аргумент равен r0 asr #31, который из определения арифметического сдвига равен 0 либо 0xffffffff, в зависимости от знака числа в регистре r0).

Решение о том, следует ли использовать промежуточный регистр при загрузке константы вне условной пересылки, принимается в функции emit_conditional_move. Но для архитектуры ARM, решение иногда оказывается неверным, например в вышеописанном случае. Данный недочет был исправлен, причем были сохранены преимущества оптимизаций выполненных проходом *ce1*, но с защитой от создания безусловных загрузок. До вызова функций emit_conditional_move производится проверка

необходимости использования промежуточного регистра, и если такая необходимость есть, то преобразование на оптимизационном проходе *ce1* отменяется, и только во время работы прохода *ce3* загрузки преобразуются в предикатную форму. Описанные изменения включаются с помощью опции -fif-conversion-no-unconditional-moves.

Результаты тестирования показали, что с введенной опцией компилятор для некоторых тестов создает улучшенный код, и не было найдено тестов, на которых наблюдается снижение производительности. Таким образом, мы улучшили работу на платформе ARM платформонезависимой оптимизации преобразования ветвлений в компиляторе GCC. Ускорение для тестовых приложений оказалось небольшим, для *sqlite* оно составило 0.5%. Причина этого в том, что подходящие условные инструкции не встречаются в горячих местах кода в исследуемых тестовых приложениях.

5. Предварительная загрузка данных

Предварительная загрузка данных (prefetching)[8] – технология, позволяющая ускорить работу программ за счет уменьшения количества промахов кэша. На многих современных архитектурах есть инструкция предварительной загрузки. Эта инструкция фактически является подсказкой процессору, она не влияет на логику работы программы. Предварительная загрузка работает почти как обычная инструкция загрузки из памяти, с той лишь разницей, что данные не попадают ни в один регистр. Запускается процесс передачи данных из памяти в кэш, и через некоторое число тактов (в это время выполняются другие инструкции) данные оказываются в кэше. При последующем обращении к этим данным пропадает необходимость длительного ожидания работы с памятью. Один из вариантов использования предварительной загрузки, предлагаемый в GCC, это использование функции __builtin_prefetch. На платформе ARM ее вызов будет заменен на инструкцию предварительной загрузки pld для адреса, переданного функции в качестве параметра.

Также в GCC есть оптимизационный проход, подключаемый с помощью опции -fprefetch-loop-arrays, который автоматически добавляет операции предварительной загрузки для обращений к элементам массива в циклах. В нем происходит анализ всех обращений в памяти происходящих на одной итерации цикла. В зависимости от множества параметров целевой платформы часть обращений выбирается для предварительной загрузки. Поскольку данные в кэше обновляются блоками по размеру линии кэша, обычно в оптимизационном проходе производится разворачивание цикла, чтобы для одного обращения к памяти в терминах исходного цикла было необходимо выполнить одну инструкцию предварительной загрузки на каждой итерации развернутого цикла.

5.1. Настройка параметров предварительной загрузки данных

Оптимизация предварительной загрузки данных в GCC имеет несколько параметров, контролирующих ее работу. Параметры определяют аппаратные характеристики кэша, такие как размер линии кэша, количество одновременно выполняемых загрузок, время задержки при передаче данных из памяти в кэш второго уровня и другие. Анализ результатов тестирования библиотеки *libevas* с помощью APTT показал, что эти параметры, и их интерпретация в компиляторе для генерации кода, сильно влияют на производительность.

В первую очередь, следует отметить два параметра оптимизации предварительной загрузки данных, появившиеся в GCC версии 4.5. Это prefetch-min-insn-to-mem-ratio и min-insn-to-prefetch-ratio. Они были созданы в первую очередь для более консервативной оптимизации предварительной загрузки данных на некоторых тестах набора SPEC CPU 2000 [9] на процессорах архитектуры x86. Первый параметр устанавливает ограничение на минимальное соотношение числа инструкции в цикле к числу обращений к памяти в нем, подразумевая, что если нет инструкций для выполнения на процессоре во время ожидания операций с памятью, то не будет улучшения производительности от предварительной загрузки. Но для многих циклов это не так, например обычный цикл копирования массива в памяти почти не содержит инструкций для долгого выполнения на процессоре, но может быть значительно ускорен за счет предварительной загрузки данных.

Второй параметр отключает предварительную загрузку в циклах с неизвестным числом итераций, если соотношение между числом предварительных загрузок и общим числом инструкций превосходит значение этого параметра. Поскольку без использования профилирования число итераций цикла редко оказывается известно, этот параметр успешно отключает предварительную загрузку на библиотеке *libevas,* устанавливая слишком высокое ограничение (значение по умолчанию равно 10). Можно сказать, что оба эти параметра оказываются неуместными, по крайней мере, на приложениях, исследованных в данной работе. Тестирование с помощью APTT показало, что оптимальным значением этих двух параметров на платформе ARM является ноль, это позволяет использовать все возможности оптимизации предварительной загрузки.

5.2. Оценка дистанции предварительной загрузки

С помощью APTT было проведено исследование влияния параметра prefetch-latency на производительность тестовых приложений. Данный параметр должен означать количество тактов процессора, которое необходимо для предзагрузки данных из памяти в случае, когда эти данные отсутствуют в кэше. Теоретически, оптимальное значение prefetch-latency должно зависеть от конкретного используемого аппаратного обеспечения, но не от запускаемых программ. Однако, результаты работы APTT показали, что

оптимальное значение данного параметра значительно отличается для выбранных тестовых приложений.

Оказалось, что в оптимизации предварительной загрузки prefetch-latency используется для вычисления дистанции предварительной загрузки. Дистанцией предварительной загрузки называется, в случае платформы ARM, число байт, добавляемое к аргументу инструкции предварительной загрузки pld по сравнению с обычной загрузкой ldr для обращения к той же области в памяти на той же итерации. Для подсчета дистанции предварительной загрузки компилятор пытается оценить время, требуемое для выполнения одной итерации цикла. Далее, зная шаг у каждого из обращений к памяти и задержку предварительной загрузки prefetch-latency, вычисляется дистанция предварительной загрузки. Ее максимально точное вычисление очень важно, ведь в случае, когда она слишком мала, данные не успеют оказаться в кэше к нужному моменту времени, а в случае, когда дистанция слишком велика, они могут быть стерты из кэша в результате записи туда других данных. Однако, реализованные в GCC вычисления не очень точны при определении времени выполнения одной итерации цикла. В частности, выполняется оценка времени на исходном (без развертывания итераций) теле цикла, после чего полученное значение умножается на коэффициент развертывания. Получается, что инструкции изменения счетчика цикла и ветвления подсчитываются несколько раз, хотя после развертывания большая часть из них будет удалена. В результате компилятор получает дистанцию предварительной загрузки большую, чем это требуется для архитектуры, особенно на коротких циклах. Это создает излишний дефицит кэша и не позволяет настраивать параметр prefetch-latency, поскольку его реальное влияние зависит от размера и шага конкретного цикла.

Механизм оценки дистанции предварительной загрузки был улучшен с помощью подсчета временной оценки для уже развернутого цикла вместо исходного. На простых тестовых циклах выигрыш производительности составляет до 10%. Тем не менее, даже после улучшения, возможно получение более высокой производительности тестов при выборе оптимального значения параметра prefetch-latency с помощью APTT.

5.3. Уменьшение регистрового давления

При анализе ассемблерного кода, сгенерированного при включенной оптимизации предварительной загрузки, было обнаружено, что дополнительный регистр используется как аргумент в каждой инструкции предварительной загрузки pld, в то время как требуемый адрес предварительно загружаемого места в памяти можно легко вычислять относительно аргументов инструкции загрузки ldr, для которой делается предварительная загрузка. Как следствие, требуется значительно больше регистров, причем разворачивание цикла усугубляет эту проблему.

Увеличение числа используемых регистров легко увидеть на следующем примере:

```
                    .L3:                .L3:
for (i=0; i<n; i++) mov r1, r2, asl     ldr ip, [r2, #0]
  s+=a[i]*b[i];     #2                  add r1, r1, #1
                    ldr r5, [r3, r4]    ldr r4, [r3, #0]
                    ldr r6, [r3, ip]    cmp r1, r5
                    add r2, r2, #1      pld [r2, #116]
                    cmp r2, r8          pld [r3, #116]
                    add r7, r1, r4      add r2, r2, #4
                    add r1, r1, ip      add r3, r3, #4
                    pld [r7, #116]      mla r0, r4, ip,
                    pld [r1, #116]      r0
                    add r3, r3, #4      bne .L3
                    mla r0, r6, r5,
                    r0
                    bne .L3
```

Рис 3. Пример излишнего регистрового давления.
а) Исходный код на C; б) ARM-ассемблер до исправления;
в) ARM-ассемблер после исправления

Удалось выяснить, что источник проблемы – оптимизация отслеживания изменений скалярных переменных (scalar evolution optimization pass), которая анализирует поведение скалярных переменных в циклах. Оказывается, данная оптимизация не позволяет разложить такое выражение, как &a[i] на сумму &a[0] и i, для дальнейшего анализа отдельно каждой части выражения. Этот недостаток является причиной того, что в дальнейшем при оптимизации индукционных переменных цикла оказывается невозможным вычисление адреса &a[i]с использованием того же регистра, что и для обращений к a[i].

Были разработаны улучшения для оптимизации скалярных переменных в циклах. Стало допустимым разложение адресного выражения. Эффект от внесенных изменений не ограничивается влиянием на оптимизацию предварительной загрузки данных, но также позволяет оптимизациям счетчиков цикла находить более полное множество индукционных переменных в циклах, где используются обращения к глобальным массивам или массивам внутри структур.

Улучшение оптимизации дает прирост производительности библиотеки *libevas* на процессоре ARM в среднем на 3.4% и до 11% на некоторых тестах. Кроме того, на 1% ускоряется набор тестов с использованием чисел с плавающей точкой SPEC FP 2000 на архитектуре x86 на процессоре Core 2 с использованием профилирования и следующих опций оптимизации: -O3

-ffast-math -march=native –fsched-pressure –fschedule-insns -fprefetch-loop-arrays. Для целочисленных тестов SPEC INT 2000 производительность не изменяется.

6. Автоматическая векторизация

Отдельно от предыдущих работ был выполнен ручной анализ кода, создаваемого компилятором GCC на платформе ARM при включенной автоматической векторизации[10]. GCC поддерживает автоматическую векторизацию для многих SIMD архитектур, включая технологию ARM NEON[11]. Векторизация не используется по умолчанию на уровнях оптимизации ниже третьего (-O3). Для включения оптимизации с помощью векторных инструкций NEON на платформе ARM необходимо указание следующих опций компилятора: -mfpu=neon –mfloat-abi=softfp –ftree-vectorize. В GCC оптимизационные проходы, производящие векторизацию, работают над внутренним представлением Tree-SSA. Для каждого цикла строится граф зависимостей по данным между инструкциями, в том числе с разных итераций цикла. В этом графе строятся компоненты сильной связности. Компоненты, состоящие из одной вершины-инструкции, означают что действия, выполняемые этой инструкцией на разных итерациях цикла, могут быть выполнены параллельно. В дальнейшем, компоненты сильной связности топологически сортируются. Это позволяет рассмотреть дополнительный параллелизм и возможности для векторизации, например, выполнив разделение цикла или перестановку вложенных циклов.

Были найдены и исправлены два недостатка в реализации автоматической векторизации в компиляторе GCC касающиеся генерации кода для векторной технологии NEON.

6.1. Выравнивание данных

При создании компилятором GCC векторизованной версии цикла, в код добавляются проверки условий, при которых будет выполняться эта векторизованная версия. Например, если цикл производит операцию копирования памяти memcpy(a, b, N), и он автоматически векторизуется для использования векторных инструкций NEON, обрабатывающих четыре четырехбайтовых слова за одну операцию, то для выбора выполняемой версии цикла проверяется условие, что (a | b) & 15 == 0. Данное условие разрешает выполнение векторизованной версии только в том случае, когда оба указателя выровнены по 16 байтам. На некоторых архитектурах (и даже некоторых версиях инструкций NEON, не поддерживаемых сейчас в GCC) на самом деле требуется, чтобы данные в векторных операциях были выровнены по размеру вектора, но инструкция загрузки vldr в NEON требует лишь выравнивания по размеру элемента (4 байта). Это условие выполняется на практике почти всегда, тем самым позволяя чаще выполнять более быструю версию цикла с векторными инструкциями. В таком варианте реализации для большей части циклов в библиотеке *libevas* лишь в редких случаях выполняется медленная

обычная версия цикла, тем самым давая прирост производительности 8.36% на наборе тестов *expedite*.

6.2. Поддержка непосредственных значений в векторных инструкциях сдвига

Оказалось, что GCC для архитектуры ARM не позволяет создание векторных инструкций сдвига с непосредственным значением последнего аргумента – величины сдвига, не смотря на наличие таких инструкций в NEON. Как следствие такого недостатка, компилятор вынужден сначала создавать вектор из четырех одинаковых значений в векторном регистре NEON, и потом использовать векторную инструкцию сдвига. Сгенерированный код автоматически векторизованного цикла получается неоптимальным.

В GCC в архитектурно зависимую часть для ARM был добавлен шаблон для поддержки векторных операций сдвига с непосредственно подставляемым значением величины сдвига. После этих изменений автоматическая векторизация оптимально обрабатывает операции сдвига. Полученные результаты не содержат замедлений, набор тестов *expedite* для библиотеки *libevas* ускоряется в среднем на 0.5%, и до 3% улучается производительность на конкретных тестах.

7. Заключение

Нами был разработан инструмент APTT на базе ACOVEA для автоматической настройки опций компилятора, который использует генетические алгоритмы. При этом функциональность ACOVEA была существенно доработана, что позволило сократить время работы инструмента, например при помощи параллельной компиляции и выполнения. Кроме того, доработки облегчили анализ итогового результата за счет фильтрации строки с опциями.

Используя результаты APTT и ручного анализа кода ассемблера, были разработаны исправления к нескольким оптимизациям компилятора GCC, включая автоматическую векторизацию, предварительную загрузку и преобразование ветвлений. Данные улучшения позволяют компилятору GCC генерировать более качественный код для платформы ARM и значительно увеличивают производительность некоторых тестовых приложений. Например, набор тестов *expedite* для библиотеки *libevas* ускоряется в среднем на 12% при использовании улучшенной версии компилятора GCC. Это означает, что более тонкая настройка компилятора для заданной встраиваемой платформы позволяет добиться генерации более качественного кода.

Следует также отметить, что созданный инструмент APTT полезен не только поиска плохо работающих оптимизаций компилятора, но может также использоваться простыми пользователями для повышения производительности различных приложений. APTT с помощью генетического алгоритма позволяет для заданного приложения и набора тестов получить

опции компиляции, существенно повышающие производительность приложения на используемой платформе. Мы сравнили на платформе ARM производительность выбранных нами тестов с использованием обычного компилятора GCC, а также нового улучшенного варианта GCC, причем с использованием лучшего набора опций, найденного с помощью APTT. Итоговое улучшение производительности для выбранных приложений без использования профилирования составило 30% для *libevas*, 9% для *sqlite*, и 14% для *webkit*. С использованием профилировщика, ускорение *sqlite* составило 28%.

В дальнейшем планируется реализация собственного инструмента для эволюционного поиска, для того, чтобы получить независимость от ACOVEA. Также необходимо дальнейшее усовершенствование инструментов для фильтрации значимых опций и построения отчетов.

Список литературы

[1] Веб-сайт компилятора GCC. http://gcc.gnu.org
[2] А. Белеванцев, Д. Журихин, Д. Мельник. Компиляция программ для современных архитектур. Труды Института системного программирования РАН, Том 16, 2009, стр. 31-50.
[3] Веб-сайт ACOVEA. http://www.coyotegulch.com/products/acovea/
[4] Веб-сайт Enlightenment Foundation Libraries. http://www.enlightenment.org/p.php?p=about/efl
[5] Веб-сайт SQLite. http://www.sqlite.org/about.html
[6] Веб-сайт WebKit. http://www.webkit.org
[7] S. Pop, R. Yazdani, Q. Neill “Improving GCC’s auto-vectorization with if-conversion and loop flattening”, Proceedings of the GCC Developers Summit 2010, pp. 89-96.
[8] C. Yang, C. Li, F. Wang “Performance Improvements for GCC Using Architecture Features on IA-64”, Proceedings of the GCC Developers Summit 2005, pp. 199-208.
[9] Веб-сайт Standard Performance Evaluation Corporation. http://www.spec.org/cpu2000
[10] D. Nuzman, A. Zaks. "Autovectorization in GCC - two years later", Proceedings of the GCC Developers Summit 2006, pp. 145-158.
[11] Веб-сайт ARM. Технология NEON. http://www.arm.com/products/processors/technologies/neon.php