Оптимизация приложений для заданных статических компиляторов и целевых архитектур: методы и инструменты

Дмитрий Мельник, Шамиль Курмангалеев, Арутюн Аветисян, Андрей Белеванцев Дмитрий Плотников, Мамикон Варданян
<dm@ispras.ru>, <kursh@ispras.ru>, <arut@ispras.ru>, <abel@ispras.ru>,<dplotnikov@ispras.ru>, <mamikon@ispras.ru>.

Аннотация. В статье рассматривается рабочий цикл оптимизации производительности приложений для заданной процессорной архитектуры на примере открытых компиляторов GCC и LLVM. Приводятся примеры выполненных оптимизаций и их результаты тестирования на известных наборах тестов. Также описывается инструмент TACT автоматической настройки компилятора на заданное приложение и его примерное использование как прикладным разработчиком, так и компиляторным инженером, приводятся результаты работы инструмента.

Ключевые слова: оптимизация программ; GCC; LLVM; автоматическая настройка компилятора.

1. Введение

Статическая компиляция по-прежнему является основным автоматическим способом повышения производительности программ на языках общего назначения (Си/Си++). Интересно, что актуальность наличия оптимизирующего статического компилятора не только не падает, но и возрастает. Во-первых, появление новых процессорных архитектур влечет за собой необходимость не только переноса на них существующих промышленных компиляторов (GCC, LLVM), но и разработки специфических и настройки имеющихся оптимизаций (распределения регистров, программной конвейеризации, использования особых возможностей набора команд архитектуры). Во-вторых, возрастающая сложность программ обеспечивает необходимость разработки масштабируемых инфраструктур межпроцедурных оптимизаций времени компоновки (LTO). В-третьих, возникает необходимость использовать оптимизации с учетом профиля программы без затрат на инструментирование, с помощью снятия профиля программы через взятие проб (sampling) во время выполнения на нагрузках промышленного типа. В-четвертых, необходима поддержка новых стандартов программирования для эффективного использования современных

многоядерных и гетерогенных архитектур (OpenMP, OpenCL, OpenACC, Cilk+). Наконец, все вышеперечисленное имеет смысл делать лишь в современных открытых компиляторных инфраструктурах типа GCC и LLVM, которые развиваются десятки лет и уже содержат все необходимые базовые машинно-зависимые и машинно-независимые оптимизации, а также средства для их создания.

Так как указанные компиляторы являются многоплатформенными, возникают естественные сложности при их использовании на архитектурах, не являющихся основным фокусом развития (ARM, MIPS и т.д.). Особенности архитектур могут приводить к тому, что машинно-независимые оптимизации, в основном разработанные и протестированные на Intel x64 или IBM Power, будут выполняться неоптимально или даже ухудшать код; то же тем более касается и машинно-зависимых оптимизаций. Имеющиеся трансформации (как правило, 100-200 оптимизаций) могут взаимодействовать ранее не учтенным способом, их настройка по умолчанию должна быть переделана.

В ходе работ сотрудников ИСП РАН над оптимизацией компиляторов GCC и LLVM для платформ Intel x86, ARM, Intel Itanium на примере конкретных наборов приложений выработался следующий подход:

1. Адаптация приложения для автоматического тестирования, или преобразование приложения в *бенчмарк*. Для этого требуется задать ряд наборов входных данных, обеспечить автоматический запуск приложения в пакетном режиме на заданных данных, сверку выходных данных с эталонными. Для известных пакетов тестов типа SPEC CPU 2006 эта работа уже выполнена авторами тестов.

2. Запустить приложение на заданных наборах тестов, получить профиль исполнения. Проанализировать ассемблерный код приложения в горячих местах профиля, выделить оптимизации, которые могли бы улучшить этот код.

3. Если выделенные оптимизации уже существуют в компиляторе, то нужно установить, почему они не отрабатывают в заданном случае с нужным качеством. Если нет, то нужные оптимизации требуется разработать и реализовать.

4. Обеспечить включение разработанных оптимизаций или исправлений в основную ветвь компилятора. Без этого шага самостоятельная поддержка изменений с учетом скорости разработки открытых компиляторов обходится слишком дорого.

В данном подходе исключительно ресурсоемким является второй шаг, требующий ручного анализа большого количества ассемблерного кода. Предложено несколько вариантов уменьшения количества необходимых для этого ресурсов:

- Исходя из общих знаний об используемом компиляторе и целевой

архитектуры, можно сразу сделать вывод о необходимости реализации новой для компилятора оптимизации. Например, работы по реализации нового планировщика команд GCC, учитывающего особенности архитектур с явным параллелизмом типа Itanium, были проведены в ИСП РАН в 2006-2008 годах на основе такого анализа.

- Можно провести автоматическую настройку компилятора на заданное приложение с помощью разработанного в ИСП РАН инструмента TACT (см. раздел 3). Инструмент может как найти опции компилятора, предоставляющие ускорение программы на 10-20% по сравнению со стандартным набором опций, так и подсказать, какие опции было бы оптимальнее исключить из стандартного набора. Оптимизации, соответствующие исключенным опциям, ухудшают код для заданного приложения и могут быть сразу подвергнуты анализу на третьем шаге.

- Можно организовать сравнение поведения компилятора на программе с другим известным компилятором и легче обнаружить недостающие оптимизации на примере более быстрого кода, полученного этим другим компилятором. Так, в оптимизациях LLVM (раздел 4) некоторые улучшения автоматической векторизации были получены как случаи, обрабатываемые компилятором GCC, но не обрабатываемые LLVM.

- Наконец, можно разработать и задействовать инструменты, позволяющие в автоматическом или полуавтоматическом режиме проверить, корректно ли использует приложение предлагаемые компилятором оптимизации. Например, для получения максимальной выгоды от межпроцедурных оптимизаций во время компоновки количество внешних функций приложения должно быть сведено к минимуму. В ИСП РАН разработан несложный инструмент, позволяющий проверить соответствие экспортируемых через заголовочные файлы интерфейсов реально видимым в двоичном файле библиотек функциям и исправить возможные неточности.

Настоящая статья посвящена примерам работ, выполненных для компиляторов GCC и LLVM в ИСП РАН согласно описанному подходу. В разделе 2 предлагается описание ряда выполненных исправлений компилятора GCC для платформы ARM. Раздел 3 кратко описывает инструмент автоматической настройки компилятора TACT и полученные с помощью него результаты. Раздел 4 содержит примеры оптимизационных работ в компиляторе LLVM, а раздел 5 завершает статью.

2. Оптимизации компилятора GCC

В данном разделе рассматриваются некоторые из разработанных нами оптимизаций для компилятора GCC: программная конвейеризация циклов [7] и ряд оптимизаций кодогенерации для архитектуры ARM. Первая из них является платформонезависимой, и разработанные улучшения применимы сразу для нескольких целевых архитектур. Вторая группа оптимизаций нацелена на архитектуру ARM, и их разработка, в основном, стала результатом анализа производительности приложений с помощью инструмента TACT, рассматриваемого в разделе 3.

2.1. Программная конвейеризация циклов

Программная конвейеризация циклов – оптимизация, которая преобразует тело цикла так, что на выполнение могут выдаваться команды из разных итераций исходного цикла, которые выполняются параллельно, подобно конвейеру.

В компиляторе GCC реализован один из вариантов программной конвейеризации циклов – *поворотное модульное планирование* [4]. Однако, реализация данного алгоритма в GCC обладает рядом ограничений, которые не позволяли использовать его на платформе ARM.

Во-первых, она поддерживает только циклы вида *do-loop*, а именно цикл с уменьшающимся на единицу счетчиком, условие выхода из которого выглядит как «счетчик цикла равен нулю (или единице)». Во-вторых, в наборе инструкций целевой архитектуры должна присутствовать команда, целиком реализующая семантику управляющей конструкции *do-loop* цикла (например, команда `loop` в процессорах *x86*). В-третьих, счетчик цикла не должен использоваться в других инструкциях, вне управляющей части цикла, а сам цикл должен состоять из одного базового блока и не содержать вызовов других функций и процедур, а также других инструкций с побочными эффектами.

Нами были исследованы возможности отказа от части этих ограничений и реализована поддержка имеющимся алгоритмом циклов более общего вида. Так, теперь поддерживаются циклы, в которых сравнение счетчика выполняется с произвольной константой, инвариантной относительно цикла, а также допускается чтение счетчика цикла командами внутри цикла. Эти изменения потребовали, в частности, улучшения алгоритма создания пролога и эпилога. Кроме того, был существенно усовершенствован механизм генерации проверки условий для выполнения оптимизированной версии цикла. Также было реализовано внесение в граф зависимостей по данным необходимых изменений для генерации компилятором корректного кода.

В результате было получено увеличение производительности на 1-4% на некоторых приложениях на процессоре ARM Cortex-A8. Тестовый набор для *sqlite* показал ускорение на 3%. Результаты для тестового набора *expedite* для

библиотеки растеризации *libevas* в среднем не изменились, при этом отдельные тесты показывали результаты от замедления на 3% до ускорения на 4%. Дополнительно было проведено тестирование умножения матриц из целых и вещественных чисел с помощью различных алгоритмов. Для матриц из целых чисел среднее ускорение составило 3%, а для случая чисел с плавающей точкой производительность увеличилась на 1%. На промышленном наборе тестов SPEC CPU2000 [5], в среднем производительность изменилась незначительно. Наибольшее ускорение получено на тесте *300.twolf*, оно составило 1.7%. На тестах *252.eon* и *178.galgel* производительность снизилась на 2%.

2.2. Улучшения поддержки набора инструкций NEON и Thumb-2

В результате анализа кода тестовых приложений, которые замедлялись при включении автоматической векторизации GCC для архитектуры ARM, было найдено несколько недостатков в поддержке команд векторного сопроцессора NEON.

Во-первых, в GCC не поддерживались векторные команды сдвига с непосредственным значением величины сдвига, несмотря на наличие таких команд в спецификации NEON. В результате цикл векторизовался, но компилятор был вынужден дополнительно выдавать команды для пересылки четырех одинаковых значений из регистров общего назначения в векторный регистр NEON, что приводило к замедлению даже по сравнению с невекторизованной версией. Кроме того, в компиляторе не была реализована поддержка команды vabd (векторный модуль разности), вместо которой последовательно выдавалось две команды для вычисления векторной разности и модуля.

Оба недостатка были устранены с помощью добавления в архитектурно-зависимую часть GCC для ARM шаблонов для поддержки соответствующих векторных операций. В результате этих изменений набор тестов *expedite* для библиотеки *libevas* ускорился в среднем на 0.5% (до 3% на конкретных тестах), а ускорение на приложении *x264* составило 2.5%.

Другой особенностью архитектуры ARM, в реализации которой в GCC были найдены недостатки, является условное выполнение в наборе команд Thumb-2. В отличие от 32-битного набора команд ARM, в Thumb-2 в целях экономии места предикаты не хранятся непосредственно в кодировке каждой команды, поэтому условным командам должна предшествовать специальная команда IT (*If-Then*), указывающая на начало IT-блока, определяющего условие исполнения до 4-х последующих команд. Ниже приводятся основные найденные недостатки в поддержке таких команд в GCC:

1. Оптимизация выбора кодировки команды происходит раньше преобразования в условную форму. В Thumb-2 команда, в зависимости от своих аргументов, может быть закодирована с

помощью 16 или 32 бит, причем большинство команд может быть представлено в 16-битной форме, только если они находятся внутри IT-блока, либо имеют суффикс *S*, т.е. записывают в регистр флагов по результатам своей работы. Чтобы представить команду в 16-битной форме, компилятор добавляет такой суффикс всегда, когда это допускается семантикой программы, причем соответствующая оптимизация происходит раньше, чем преобразование команд в условную форму, которое уже не может применяться к командам, записывающим результат в регистр флагов. Изменение порядка этих оптимизаций увеличивает количество IT-блоков на 2% в объектном коде для набора тестов SPEC 2000 INT.
2. Ограничение количества команд в условном блоке. Основная идея состоит в том, что на Thumb-2 преобразование в условную форму добавляет одну IT-команду на четыре преобразованных, при этом такое преобразование позволяет удалить одну или две команды перехода, в зависимости от наличия ветки *else* в условии. Таким образом, чтобы избежать увеличения размера кода, необходимо ограничить длину преобразуемого условного блока, соответственно, 4 или 8 командами.
3. Поддержка IT-блоков в планировщике команд. Так как на этапе планирования команд IT-блоков еще не создано, а условные предикаты во внутреннем представлении связаны с отдельными командами, планировщик может «перемешать» команды с разными предикатами, так что для каждой из них понадобится свой собственный IT-блок, что приводит к увеличению размера программы. Поэтому планировщик GCC был изменен так, чтобы отдавать приоритет тем командам, которые могут попасть в один IT-блок с последней выданной командой (т.е. имеющим точно такой же предикат, как последняя команда, или противоположный ей, например, eq/ne).

Рассмотренные улучшения обработки условных блоков существенно улучшили код для некоторых небольших тестов, однако в целом размер полученного кода для SPEC 2000 сократился незначительно.

3. Инструмент TACT автоматической настройки компилятора

Для достижения хорошей производительности сгенерированного кода в компиляторе обычно недостаточно реализовать платформозависимые оптимизации, но также необходима настройка уже существующих платформонезависимых оптимизаций. Такие оптимизации обычно имеют настройки, контролирующие применение тех или иных эвристик, параметры модели оптимизации, настраиваемые пороговые величины и т.п.

Стандартный подход к настройке компилятора под заданную архитектуру обычно состоит в следующем. Сначала определяется набор тестовых приложений (например, может быть использован промышленный тестовый набор SPEC CPU 2000). Для выбранных приложений собирается профиль выполнения, определяются «горячие» места, и вручную выполняется анализ полученного ассемблерного кода. Это довольно трудоемкий процесс, т.к. необходимо не только выделить те ассемблерные команды, которые больше всего влияют на производительность, но и определить, какие именно компиляторные оптимизации или их параметры привели к появлению этих команд. Для того чтобы упростить эту задачу, нами был разработан инструмент TACT [6] (Tool for Automatic Compiler Tuning – Инструмент для автоматической настройки компилятора), позволяющий автоматически находить оптимальный набор параметров компилятора для заданного приложения, а также определять те оптимизации (или их параметры), включенные по умолчанию, которые на данной архитектуре приводят к появлению неоптимального кода. Этот инструмент может быть использован как для непосредственного улучшения производительности приложений, так и для облегчения последующего ручного анализа кода и улучшения оптимизаций в компиляторе. Примеры таких улучшений в компиляторе GCC, сделанных по итогам анализа приложений с помощью TACT, были рассмотрены в разделе 2, а также в работе [8]. Далее в этом разделе мы рассмотрим основные особенности разработанного нами инструмента.

3.1. Генетический алгоритм

В инструменте TACT 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 параметров компилятора, 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 выдается набор параметров, с которым была получена самая лучшая производительность.

3.2. Общая схема работы TACT

Система для автоматической настройки состоит из одного управляющего *x86* компьютера, используемого также для кросс-компиляции настраиваемых приложений и нескольких тестовых плат (например, архитектуры ARM), доступных с управляющего компьютера по протоколу SSH и связанных таким образом, что все устройства имеют один общий каталог через сетевую

файловую систему NFS. Общая схема устройства инструмента TACT показана на Рис 1.

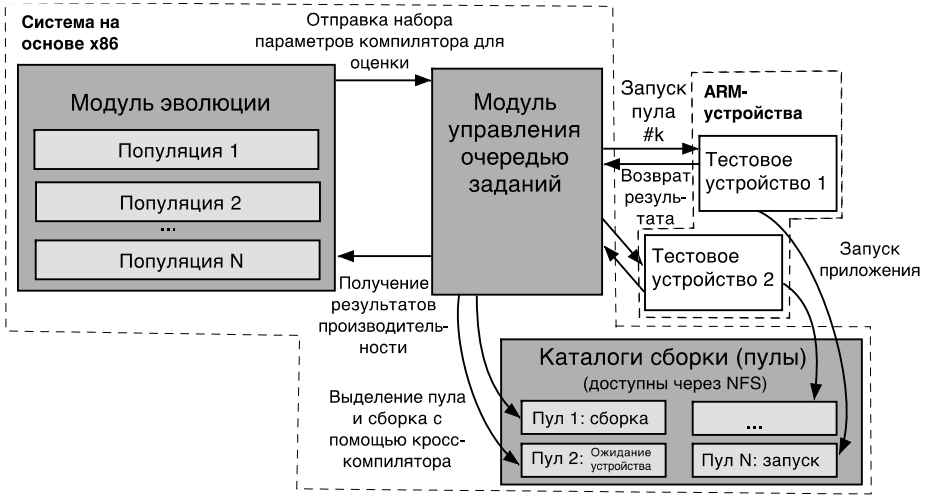

Рис 1. Компоненты TACT и общая схема работы

Модуль эволюции генерирует с помощью генетического алгоритма различные наборы параметров компилятора и отправляет их на выполнение на тестовые устройства, используя *модуль управления очередью заданий*. Данный модуль отвечает за сборку приложения с заданными параметрами компилятора, выбор свободного тестового устройства и запуск на нем скомпилированного приложения. Полученная на тестовых устройствах оценка производительности для заданного набора затем используется модулем эволюции для улучшения параметров компиляции в следующем поколении.

3.3. Определение наиболее значимых параметров компиляции

Набор опций командной строки компилятора, полученных в результате работы генетического алгоритма, может быть избыточным: некоторые опции могут в контексте других опций не изменять поведение компилятора, но они будут по-прежнему присутствовать в результирующей строке. После завершения работы генетического алгоритма, TACT пытается уменьшить количество опций компилятора в результирующих наборах опций за счет удаления тех из них, присутствие которых не влияет на итоговые бинарные файлы. Это происходит путем последовательного удаления опций по одной, и может занять достаточно длительное время. Данный процесс позволяет сократить количество опций в наборах в 3-6 раз.

После описанной процедуры итоговые наборы содержат только опции компиляции, влияющие на получившийся объектный код, но тем не менее большинство из них не оказывает значительного влияния на производительность. Поэтому на следующем этапе TACT уже пытается упорядочить опции в каждом наборе по их вкладу в производительность. Для этого на каждом шаге итерации из набора удаляется одна опция, такая, что ее удаление ведет к наименьшей потере производительности, а сама найденная на текущем шаге опция записывается в таблицу наиболее значимых, начиная снизу вверх. Процесс повторяется, пока в наборе не остается только одна опция базового уровня оптимизации (например, `-O2`), которая записывается в самую верхнюю строку таблицы. Таким образом, построенная таблица содержит все опции исходного набора в порядке убывания их значимости, причем вклад некоторых опций может оказаться отрицательным из-за погрешности измерения производительности на тестовых платах и неточностей, связанных с применением генетического алгоритма.

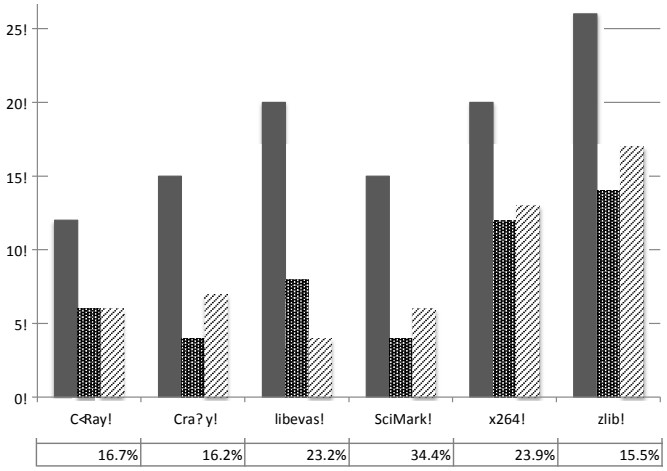

Рис2. Количество опций, существенных с точки зрения производительности приложения и величина прироста производительности

На Рис.2 для выбранных тестовых приложений левый столбец показывает количество опций из 200 исходных, положительно влияющих на производительность, и полученных описанных выше способом. Средний столбец показывает количество лучших опций, которые сохраняют 80% прироста производительности, полученного на лучшем наборе (итоговое улучшение производительности указано под названием приложения). Правый столбец показывает количество оптимизаций в полученном наборе, которые должны быть отключены по сравнению с настройками компилятора по

умолчанию в базовом уровне (`-O2`), а также количество параметров, значение которых отличается от базовых.

С точки зрения разработчика компилятора, именно опции из последней категории представляют наибольший интерес, т.к. могут указывать на недостатки в оптимизациях, используемых по умолчанию, и их исправление может дать прирост производительности сразу для широкого класса приложений. При этом число таких опций сравнительно невелико (5-15 шт. для данных приложений), что позволяет продолжить анализ производительности для найденных оптимизаций в ручном режиме.

4. Оптимизации компилятора LLVM

В ходе оптимизации компилятора LLVM были сделаны некоторые исправления кодогенератора LLVM для платформы ARM, улучшена оптимизация преобразования ветвлений (if-conversion), выполнен ряд оптимизаций с учетом профиля программы [3]. В настоящей работе далее приведены примеры наиболее существенных оптимизаций, сделанных за последние два года.

4.1. Выделение шаблонов перемежающегося доступа к данным

Рассмотрим пример:

```
int a[N/2], b[N+1], c[N+1];
for (i = 0; i < N/2; i++){
  a[i] = b[2i+1] * c[2i+1] - b[2i] * c[2i];
}
```

Листинг 4.1. Пример кода с перемежающимся доступом к данным

Заметим, что использование инструкции загрузки с устранением перемежения vld2.32 {d0, d1} [Rn] (где Rn указывает на ячейку памяти, хранящую элемент b[2i]) позволит загрузить на регистр d0 элементы массива b[2i] и b[2i+2], а на регистр d1 элементы массива b[2i+1] и b[2i+3].

```
int a[N/2], b[N+1], c[N+1];
for (i = 0; i < N/2; i+=2){
  even_b, odd_b = vector_load(&b[2i]);  //
even_b ={b[2i], b[2i+2]}
  even_c, odd_c = vector_load(&c[2i]);  //
odd_b = {b[2i+1], b[2i+3]}
  {a[i], a[i+1] } = odd_b * odd_c –
even_b * even_c;
}
```

Листинг 4.2. Векторизованный код с перемежающимся доступом к данным

Для нахождения шаблонов перемежающегося доступа внутри тела цикла в пределах одной итерации находились все инструкции загрузки данных из памяти. Среди них выделялись те, которые загружают данные из одного источника. С помощью анализа возможных значений скалярных выражений (Scalar Evolution) проводилось сравнение с шаблонами перемежающегося доступа. При совпадении с шаблоном проводилось преобразование промежуточного представления. Скалярные инструкции загрузок удалялись, вместо них вставлялась векторная инструкция загрузки с устранением перемежения.

В случае возможности векторизации цикла проводится сравнение стоимости выполнения векторизованной и скалярной версии цикла. Если стоимость векторной версии меньше, чем скалярной, то принимается решение о векторизации данного цикла.

4.2. Использование векторных загрузок с автоматическим увеличением адреса загрузки

Команды векторной загрузки ARM могут автоматически увеличивать значение базового регистра (по адресу из которого происходит загрузка) на величину загружаемых данных. Это позволяет объединять вычисление адреса и инструкцию векторной загрузки в одну операцию векторной загрузки с автоинкрементом.

Таблица 4.1. Использование команды автоинкрементной загрузки

загрузка и отдельное вычисление адреса	автоинкрементная загрузка
vld1.32 {d16, d17}, [r0] add r0, r0, #16	vld1.32 {d16, d17}, [r0]!

4.3. Использование выравненных векторных загрузок

Время перемещения данных из памяти в регистры инструкциями структурированной векторной загрузки из расширения Neon зависит от их выравнивания. Согласно документации количество циклов, затрачиваемых процессором на выполнение инструкции, больше в случае доступа к невыравненным данным. Например, инструкция vld1.64 при обращении к памяти с 64 битным выравниванием (3 последних бита в адресе равны нулю) затрачивает меньше процессорных циклов, чем при обращении к не выровненным данным.

Данную оптимизацию необходимо проводить на машинно-зависимом уровне. Для этого нужно для каждой встретившейся инструкции векторной загрузки провести анализ расположения данных в памяти

Применение описанных оптимизаций позволило ускорить используемый набор тестов (набор тестов векторизации компилятора GCC) в среднем на ~7%.

4.4. Генерация команд предвыборки при обработке массивов в цикле

Использование команд предвыборки при обработке массивов в цикле повышает эффективность использования кэша процессора во время последовательной загрузки данных.

Процессоры архитектуры ARM серии Cortex-A9 имеют встроенный автоматический механизм предвыборки данных, который загружает данные в кэш, учитывая промахи кэша, массив загружается в кэш после нескольких итераций и промахов кэша. Такое поведение неоптимально и может быть исправлено с помощью команды предвыборки “PLD”, которая указывает процессору, что вскоре будут использованы данные, на которые указывает команда, так что их желательно загрузить в кэш, если их там еще нет.

Предвыборка данных должна осуществляться заранее, до их непосредственного использования. На процессоре ARM Cortex-A9 требуется выполнение около 200 тактов после команды предвыборки, чтобы данные были загружены в кэш [2]. Тогда момент вставки инструкции упреждающей загрузки можно рассчитать, как отношение задержки загрузки данных в кэш после выполнения команды предвыборки к количеству команд в цикле.

Для того, чтобы команды предвыборки не выполнялись слишком часто и не указывали на участки памяти, которые уже загружены в кэш, используется развертывание циклов. Цикл развертывается столько раз, чтобы загружаемые данные за один проход развернутого цикла полностью заполняли одну строку кэша. Например, если размер строки кэша 32 байта (как на процессоре ARM Cortex-A9), а размер загружаемых каждую итерацию данных равен 4 байта, то цикл стоит развернуть 8 раз (32 / 4), и вставить команду предвыборки лишь в первую итерацию.

Тестирование на наборе тестов SPEC CPU 2000 показало, что прирост производительности составляет ~0.9%. На тестах SQLite, Expedite, Cray и Coremark прирост производительности составил 0,5 до 5 %, средний прирост составляет ~2.5%

4.5. Модификация алгоритма распределения регистров

Архитектура ARM поддерживает команды, осуществляющие множественную загрузку/сохранение по последовательным адресам - LDM/STM. Инфраструктура LLVM учитывает эту особенность архитектуры в оптимизирующем проходе "ARM load / store optimization pass", который осуществляет свертку последовательных операций (LDR/STR) в одну или несколько команд множественной загрузки/сохранения. Копирование структур в LLVM осуществляется посредством вызова функции `memcpy`, вызов которой в целях оптимизации заменяется серией команд, осуществляющих загрузку/сохранение. Но алгоритм распределения регистров не учитывает возможность такой оптимизации, поэтому регистры распределяются не в порядке строгого возрастания номеров. Алгоритм распределения регистров был модифицирован таким образом, чтобы обеспечить выбор следующего свободного регистра с учетом его номера, обеспечивая последовательное возрастание номера используемого регистра, что повысило качество работы оптимизации "ARM load / store optimization pass" и привело к росту быстродействия генерируемого кода.

5. Заключение

В статье описаны способы улучшения производительности программ при статической компиляции на примере работ, выполненных в ИСП РАН для компиляторов GCC и LLVM на платформе ARM. Получены результаты ускорения набора тестов в среднем на 1-5%, а конкретного приложения (при автоматической настройке) – на 10-20%. Работы по улучшению статических компиляторов будут продолжаться в направлении разработки новых инструментов, позволяющих уменьшить затраты на ручной анализ ассемблерного кода.

Список литературы

[1] The LLVM Compiler Infrastructure. http://LLVM.org/

[2] Справочное руководство по процессорной архитектуре ARM., http://infocenter.arm.com

[3] Ш.Ф. Курмангалеев. Методы оптимизации Си/Си++ - приложений распространяемых в биткоде LLVM с учетом специфики оборудования. Труды ИСП РАН, том 24, стр. 127-144, 2013 г. DOI: 10.15514/ISPRAS-2013-24-7.

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

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

[6] D. Plotnikov, D. Melnik, M. Vardanyan, R. Buchatskiy, R. Zhuykov, JH. Lee. Automatic Tuning of Compiler Optimizations and Analysis of their Impact. Procedia Computer Science Volume 18, pp.1312-1321. 2013.

[7] Р. Жуйков, Д. Мельник, Р. Бучацкий. Программная конвейеризация циклов на платформе ARM. Труды ИСП РАН. 2012. №22. С.33-48. DOI: 10.15514/ISPRAS-2012-22-3.
Р. Жуйков, Д. Плотников, М. Варданян. Автоматическая настройка оптимизационных преобразований компилятора GCC для платформы ARM. Труды ИСП РАН. 2012. №22. С.49-66. DOI: 10.15514/ISPRAS-2012-22-4.

Optimizing programs for given hardware architectures with static compilation: methods and tools

Dmitry Melnik dm@ispras.ru
ISP RAS, Moscow, Russia
Shamil Kurmangaleev <kursh@ispras.ru>
ISP RAS, Moscow, Russia
Arutyun Avetisyan <arut@ispras.ru>
ISP RAS, Moscow, Russia
Andrey Belevantsev <abel@ispras.ru>
ISP RAS, Moscow, Russia
Dmitry Plotnikov <dplotnikov@ispras.ru>
ISP RAS, Moscow, Russia

Abstract. The paper describes the workflow for optimizing programs for performance targeting the fixed hardware architecture with static compilation using GCC and LLVM compilers as examples. The workflow has gradually grown within ISP RAS Compiler Technology Team when working on GCC and LLVM compiler optimization. We start with preparing a benchmark using the given application as a source, and then proceed in parallel with manual analysis of generated assembly code and automatic compiler tuning tool. In general, a compiler optimization improvement produced by the manual analysis gives 1-5% speedup, while the automatic tuning results may give up to 10-20% speedup. However, the manual analysis results are usually valid for the broad class of applications and are contributed to the open source compilers, while the automatic tuning results make sense only for the given application.

We present some of the optimizations performed, e.g. improved NEON and Thumb-2 support for GCC, vectorization improvements for LLVM, register allocation improvements for LLVM, and the corresponding evaluation results. We also describe TACT, a tool for automatic compiler tuning for the given application mentioned above, and its example use cases both for an application developer and a compiler engineer. We give the sample of TACT optimization results.

Keywords: Program optimization, GCC, LLVM, automatic compiler tuning.

References

[1]. The LLVM Compiler Infrastructure. http://LLVM.org/
[2]. ARM Architecture Processor Manual, http://infocenter.arm.com
[3]. Kurmangaleev S.F. Metody optimizatsii Ci/Ci++ - prilozhenij rasprostranyaemykh v bitkode LLVM s uchetom spetsifiki oborudovaniya. [Machine-specific optimization methods for C/C++ applications that are distributed in the LLVM intermediate representation format]. Trudy ISP RAN [The Proceedings of ISP RAS], 2013, vol.24, pp. 127-144. DOI: 10.15514/ISPRAS-2013-24-7. (in Russian)
[4]. J. Llosa, E. Ayguade, A. Gonzalez, M. Valero, J. Eckhardt. “Lifetime-sensitive modulo scheduling in a production environment”. Computers, IEEE Transactions on. Volume 50, Issue 3, pp.234-249. 2001.
[5]. Standard Performance Evaluation Corporation. http://www.spec.org/cpu2000/
[6]. D. Plotnikov, D. Melnik, M. Vardanyan, R. Buchatskiy, R. Zhuykov, J.H. Lee. Automatic Tuning of Compiler Optimizations and Analysis of their Impact. Procedia of Computer Science Volume 18, pp.1312-1321. 2013.
[7]. Roman Zhuykov, Dmitry Melnik, Ruben Buchatskiy. Programmnaya konvejerizatsiya tsiklov na platforme ARM [Loop software pipelining on ARM platform]. Trudy ISP RAN [The Proceedings of ISP RAS], 2012, vol.22, pp. 33-48. DOI: 10.15514/ISPRAS-2012-22-3. (in Russian)
[8]. Roman Zhuykov, Dmitry Plotnikov, Mamikon Vardanyan. Avtomaticheskaya nastrojka optimizatsionnykh preobrazovanij kompilyatora GCC dlya platformy ARM [Automatic tuning of GCC optimization passes for ARM platform]. Trudy ISP RAN [The Proceedings of ISP RAS], 2012, vol.22, pp. 49-66. DOI: 10.15514/ISPRAS-2012-22-4. (in Russian)