Методы оптимизации Си/Си++ - приложений распространяемых в биткоде LLVM с учетом специфики оборудования[1]

Курмангалеев Ш.Ф.
kursh@ispras.ru

Аннотация. В статье рассматриваются методы оптимизации Си/Си++ приложений, применяемые в системе двухэтапной компиляции, позволяющей распространять такие приложения в промежуточном представлении LLVM [1]. Рассматривается методика замещения кода функций во время исполнения, реализованная в динамическом компиляторе LLVM, позволяющая осуществлять динамическую оптимизацию программ. Описывается метод статического инструментирования с неполным покрытием всех дуг потока управления программы и последующей его коррекцией для сбора профиля, позволяющий получать профиль сравнимый по качеству с классическим подходом, но при этом обеспечивающий существенное снижение накладных расходов на сбор профиля. Помимо этого описывается разработанная и реализованная методика конвертации динамически собираемого профиля, для статически скомпилированного приложения в формат, используемый LLVM для машинно-независимых оптимизаций. Описываются как существующие в LLVM оптимизации, модифицированные для использования профиля, так и предлагаются новые.

Предлагается метод позволяющий использовать команды предвыборки для повышения эффективности кода обработки массивов. Описывается подход к построению специализированного облачного хранилища приложений, позволяющего решать вопросы оптимизации и защиты программ, а также обеспечивающий снижение накладных расходов на компиляцию и оптимизацию приложений в облачной инфраструктуре.

[1]Работа выполнена при финансовой поддержке Минобрнауки по государственному контракту от 15.06.2012 г. № 07.524.11.4018 в рамках ФЦП "Исследования и разработки по приоритетным направлениям развития научно технологического комплекса России на 2007-2013 годы"

Ключевые слова: Двухэтапная компиляция, оптимизация, LLVM, облачное хранилище.

1 Введение

В связи с широким распространением мобильных платформ имеющих ограниченные вычислительные ресурсы и жесткие требования к энергопотреблению становится актуальной задача оптимизации программы для конкретного пользователя, а также задача оптимизации под конкретную реализацию архитектуры. Оптимизация программы с учетом профиля конкретного пользователя на его машине:

- Учитывает особенности поведения данного пользователя, что позволяет ускорить выполнение программы именно для тех наборов входных данных, которые важнее для этого пользователя.
- Позволяет применять «дорогие» оптимизации только к часто выполняющимся участкам кода программы с использованием компиляции во время выполнения программы. Это уменьшает затраты на компиляцию программы, которые становятся особенно существенны при проведении оптимизаций на машине пользователя, при сохранении производительности (так как важные участки оптимизируются).

Оптимизация программы для учета архитектуры машины пользователя:

- Учитывает детальные параметры архитектуры (размер кэша, соотношение между частотой памяти и процессора, наличие специальных векторных инструкций). Эти параметры необходимо учитывать при оптимизациях обращений к памяти; при векторизации; при встраивании функций и развертке циклов. Полученная производительность будет заведомо не хуже, чем при выполнении машинно-зависимых оптимизаций на стороне разработчика, где известна только часть параметров целевой машины.
- Позволяют сократить для разработчика затраты на распространение и поддержку программы. Разработчику достаточно поддерживать одну версию программы, при сборке которой применялись лишь машинно-независимые оптимизации.

Для языков общего назначения (Си/Си++) в настоящий момент не существует решения обеих задач (специализации под пользователя и под его целевую машину) в виде общей среды, удобной для использования, как на стороне разработчика, так и на стороне пользователя.

Предлагаемый метод распространения программ написанных на языках Си/Си++, в промежуточном представлении позволяет указанные задачи

переносимости программ в пределах одного семейства процессоров с учетом специфических особенностей каждого конкретного процессора, проводить адаптивную компиляцию, учитывая поведение пользователя и характер входных данных. Собирая информацию о профиле программы, поступающую от пользователей, можно применить к промежуточному представлению машинно-независимые оптимизации для повышения быстродействия программы для наиболее часто встречающихся вариантов использования. Помимо этого распространение программы в промежуточном представлении позволяет применять средства статического анализа программ для поиска уязвимостей, и производить запутывание программ для защиты от обратного проектирования. Все указанные операции могут происходить как на машине пользователя, но это может привести к дополнительным накладным расходам, что является важным фактором в случае работы программы на мобильных устройствах. Однако этого можно избежать, переместив второй этап компиляции, анализ и запутывание программ на сервер приложений [2].

Требования, предъявляемые к программному обеспечению для мобильных платформ, такие как производительность и энергопотребление справедливы и для гетерогенных кластеров. Таким образом, создание облачной инфраструктуры, которая будет позволять осуществлять развертывание программного обеспечения с учетом конкретного аппаратного обеспечения каждого узла. А также производить оптимизацию программного обеспечения между запусками заданий. В качестве примера такого программного обеспечения можно рассматривать пакет OpenFoam (открытая интегрируемая платформа для численного моделирования задач механики сплошных сред). Поскольку пакет написан на языке Си++ с использованием виртуальных функций, то имеет смысл оптимизировать вызовы виртуальных функций, для чего и была предложена оптимизация спекулятивной девиртуализации[2].

В настоящей статье в разделе 2 описывается система двухэтапной компиляции программ на основе LLVM и изменения, внесенные в динамический компилятор LLVM. Описывается метод статического инструментирования обеспечивающий снижение накладных расходов на сбор профиля программы, рассматривается подход позволяющий конвертировать динамически собираемый профиля в формат LLVM. Раздел 3 описывает реализованные оптимизации: открытую вставку функций, вынос участков кода в отдельные функции, спекулятивная девиртуализация и использование команд предвыборки при обработке массивов в цикле. В разделе 4 описывается функциональность сервера приложений и снижение расходов на компиляцию с помощью динамического выбора уровня оптимизаций. Раздел 5 завершает статью.

[2] Здесь и далее под девиртуализацией понимается генерация прямого вызова функции без обращения к таблице виртуальных функций.

2 Двухэтапная компиляция

Кратко опишем предложенный поход: На первом этапе приложение компилируется на машине разработчика специальным набором инструментов основанных на LLVM, на этом этапе выполняются машинно-независимые оптимизации и оптимизации времени связывания (LTO - link time optimizations). Затем происходит определение зависимостей между программными компонентами и происходит генерация программного пакета содержащего файлы с промежуточным представлением LLVM и информацию о схеме инсталляции. На втором этапе программа оптимизируется и инсталлируется на машине пользователя. Во время оптимизации возможен учет поведения пользователя и особенностей аппаратуры, на которой будет выполняться программа. Поддерживается два варианта инсталляции: 1) генерация и инсталляция бинарной программы; 2) инсталляция программы для динамической компиляции. В случае динамической компиляции программы поддерживается система легковесного сбора профиля на основе Oprofile [3],[4]. В обоих случаях специальная программа «демон» позволяет осуществлять оптимизацию программы во время простоя системы с учетом профиля пользователя.

В систему двухэтапной компиляции, рассматриваемую в работах [2] и [3], были внесены изменения позволяющие повысить производительность и совместимость системы. Была реализована поддержка нового “gold” линкера [5], что позволило отказаться от двух последовательных запусков компилятора – одного для получения бинарного кода, и второго для получения эквивалентного биткода. Новый линкер позволяет производить компиляцию программы, генерируя объектные файлы содержащие биткод. Причем генерация бинарного кода производится только на этапе генерации финального модуля программы. Помимо этого поддерживается прозрачное связывание между объектными файлами, содержащими бинарный код и промежуточное представление LLVM. Поскольку применяемый ранее компоновщик имел ряд ограничений, по сравнению с компоновщиком из пакета Binutils [5][6], например отсутствие поддержки скриптов линкера, поддержка стандартного компоновщика позволяет повысить совместимость и расширить спектр программ, сборка которых происходит автоматически с помощью предлагаемых инструментов. Важной особенностью предлагаемого подхода, является поддержка сборки программ использующих стандартные системы сборки, основанные на Autotools[7].

В систему сборки пользовательских программ были внесены изменения, позволившие сократить как время компиляции программы, так и время генерации инсталляционного пакета, а также снять ранее присутствующие ограничения на расположение каталогов сборки и инсталляции программы. Время генерации инсталляционного пакета уменьшилось из-за того, что вместо затратной по времени процедуры сопоставления контрольных сумм исходных файлов и файлов в после развертывания программы, для получения

списка файлов включаемых в пакет используется утилита installwatch (пакета Checkinstall [8]), перехватывающая системные вызовы при копировании файлов.

Помимо этого была реализована поддержка автоматического преобразования RPM-пакетов, содержащих исходный код в пакеты, содержащие биткод LLVM и скрипты для развертывания программы. Формат RPM используется в распространенных дистрибутивах на базе Linux, таких как Red Hat, OpenSUSE, Fedora и в открытой мобильной платформе Tizen.

2.1 Изменения в динамическом компиляторе LLVM

Для обеспечения непрерывного учета поступающего профиля программы, от динамического компилятора требуется поддержка механизма замещения на стеке. Замещение на стеке – механизм, позволяющий изменять код часто исполняемых функций во время исполнения программы. Применяется во многих динамических компиляторах (HotSpot, Jikes RVM, CACAO и др.), но пока не реализован в динамическом компиляторе LLVM.

Динамическая компиляция в LLVM организована следующим образом: трансляция в целевой код происходит непосредственно в памяти машины по функциям. Если функция исполняется дольше или большее количество раз, чем остальные, то, вероятнее всего, если заново оптимизировать ее более агрессивными оптимизациями, то можно получить прирост производительности.

На базе системы двухэтапной компиляции была реализована поддержка замены на стеке. В качестве профилировщика используется модифицированный Oprofile, сбор и обработка поступающей от профилировщика информации происходит в реальном времени. Компиляция и обработка информации от профилировщика осуществляется в разных потоках: в одном осуществляется динамическая компиляция и запуск исполняемого кода, а второй осуществляет сбор информации от профилировщика и, в зависимости от полученных данных, принимая решение о том, необходимо ли компилировать функцию заново.

Решение о повторной компиляции принимается в зависимости от «температуры» функции: «горячие» функции должны будут быть повторно скомпилированы с более глубокими оптимизациями, «теплые» останутся без изменений, а код «холодных» функций можно удалить из памяти[9]. Вызов функции осуществляется через специальный переходник, который производит одно из следующих действий: если функция еще скомпилирована, компилирует ее, если для функции доступна оптимизированная версия, вызывается она, даже если первоначальная функция все еще находится в памяти. Для освобождения памяти занимаемой различными вариантами функций в начало и конец каждой из функций вставляется атомарная операция инкремента/декремента, что позволяет оценить количество потоков

использующих данную функцию, после обнуления количества активных пользователей, функцию можно безопасно удалить из памяти. Кроме того, для горячих функций имеющих небольшой размер, происходит удаление таких счетчиков, для сокращения накладных расходов.

Температурные характеристики выделяются методом скалярной кластеризации. Необходимо построить три кластера: «горячий», «теплый» и «холодный». Известно, что «горячему» будет принадлежать элемент с максимальной числовой характеристикой, «холодному» - с минимальной.

2.2 Статический профиль с неполным покрытием всех дуг потока управления программы

Было проведено исследование влияния статического профиля с неполным покрытием всех дуг потока управления программы и последующей его коррекцией, что позволило сократить накладные расходы, обуславливаемые инструментированием.

В LLVM применяется два подхода к инструментации. Первый метод вставляет счетчики на каждое ребро в программе (опция - insert-edge-profiling). Он собирает полный профиль, но замедление оказывается порядка 100%. Второй метод основан на законе Кирхгофа - общее вес входящих в блок ребер, равен общему весу исходящих, инструментируются только вершины входящие в максимальное остовное дерево графа потока управления (опция -insert-optimal-edge-profiling). Замедление инструментированной программы порядка 50%.

Предложенный метод состоит в том, чтобы инструментировать ребра входящие в максимальное остовное дерево с некоторой вероятностью и восстановлением на этапе обработки собранного профиля, для снижения накладных расходов при минимальном снижении качества профиля.

Во время тестирования на легковесной СУБД Sqlite было выявлено, что инструментация с помощью алгоритма «optimal-edge-profiling» замедляет программу примерно на 46%, а предложенный подход с 50% вероятностью вставки счетчика на 9%. При этом при использовании полученных профилей для оптимизации программы показывает, что производительность обоих версий различается ~1%. Из чего можно сделать вывод, что качество собираемого профиля примерно совпадает.

2.3 Конвертирование динамического профиля в формат LLVM

Также для обеспечения возможности сбора профиля для статически скомпилированной программы с помощью динамический профилировщиков был разработан и реализован алгоритм сопоставления базовых блоков в

промежуточном представлении программы с бинарным образом исполняемого файла, что позволило конвертировать такой профиль в формат LLVM.

Для этого, во время генерации кода, смещения базовых блоков относительно начала функции их размер сохраняются в специальный файл с расширением «.pmap». Каждый такой файл состоит из заголовка и блоков, идущих друг за другом. Заголовок состоит из полей:

слово «AMAP» | размер файлы
имя модуля
имя компилятора
версия компилятора
строка аргументов компилятору.

Каждый блок состоит из полей:

размер имени функции
имя функции
размер функции в базовых блоках
номер блока | смещение | размер
номер блока | смещение | размер
….
номер блока | смещение | размер.

Поскольку профиль, собираемый Oprofile, привязан к виртуальным адресам, для его корректной загрузки, после завершения генерации кода и компоновки модулей при помощи утилиты objdump извлекаются виртуальные адреса функций. Имя виртуальные адреса функций, и смещения базовых блоков относительно начала функции мы можем конвертировать профиль в формат LLVM.

3 Оптимизации учитывающие информацию о профиле программы

3.1 Открытая вставка функций

Оптимизация открытой вставки функции – оптимизирующее преобразование компилятора, вставляющее код функции на место его вызова в тело вызывающей функции.

Для реализации оптимизации открытой вставки функций была рассмотрена существующая реализация – оптимизация компилятора GNU GCC[10]. Данная оптимизация использует данные профилирования для решения вопроса о

вставке функции: относительная частота вызова функций задается параметром frequency в пределах от 0 до 1, количество вызовов задается параметром calls, а потенциальный рост общего количества инструкций задается параметром growth. Таким образом, решение о том, вставить ли малую функцию вычисляется по формуле:

$$growth < 0 \rightarrow growth$$

growth >=0 → calls/growth или growth/frequency – в зависимости от того, включен глобальный или локальный профиль соответственно.

Для вычисления веса функции применяется следующая эвристика:

FunctionWeight= NumofInstructions*InstrucctionPenalty-NumArgguments*AllocaPenalty-NumofConstInstruction*ConstantPenalty, где

InsrucitonPenalty=2 – штраф за каждую инструкцию в функции

AllocaPenalty=2 – штраф за локальную переменную

ConstantPenalty=2 – штраф за константу

Вес функции с учетом информации из профиля вычисляется следующим образом:

NewWeight= (NumCallFromProfileInfo)/FunctionWeight

После присвоения весов, функции сортируются по возрастанию веса и встраиваются до тех пор, пока суммарный вес не превысит некоторого значения (по умолчанию 1000).

На тестах SQLite, Expedite, Cray и Coremark дала прирост скорости в ~2%.

3.2 Визуализация графа потока управления программы

Среди проходов LLVM имеется набор для визуализации графов потока управления функций, которые сохраняют каждый граф в формате Graphviz[11] в отдельный файл, либо выводят их на экран. Для удобства отладки оптимизаций использующих информацию о профиле программы была реализована возможность выбора нужной функции, а также аннотирование базовых блоков и ребер выводимого графа весами из профиля.

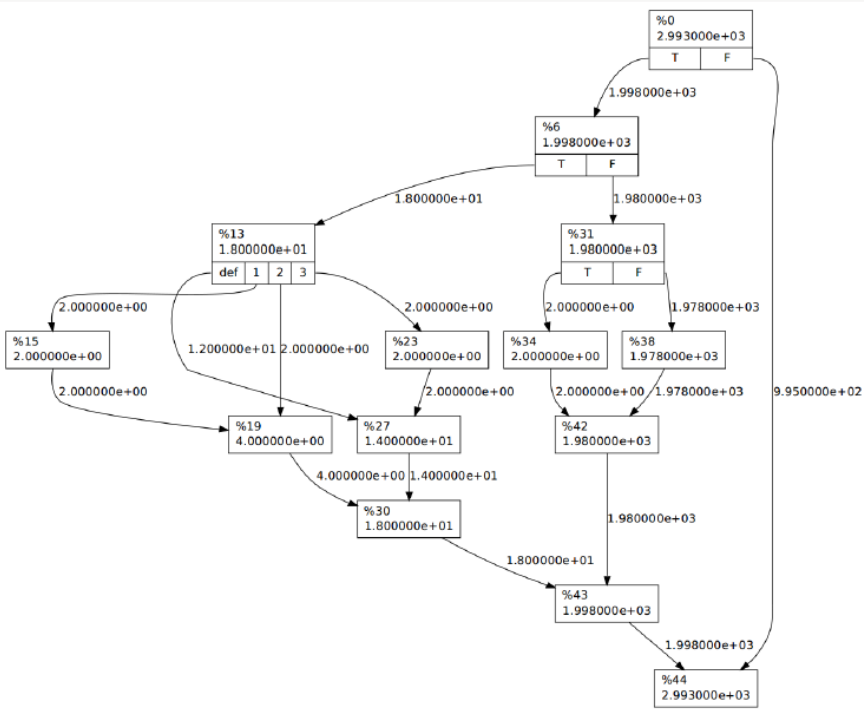


Рис. 1 Пример аннотированного графа функции

3.3 Вынос "холодных" участков кода в отдельные функции

Для оптимизации предлагается рассматривать функции, которые исполняются наибольшее число раз (”горячие”). Если функцию условно можно разделить на 2 части - большой редко исполняемый участок кода, относительно малый "горячий" участок. Из таких функций предлагается выносить ”холодную” часть функции в отдельную новую функцию, уменьшая при этом размер рассматриваемой функции и расстояния в памяти между часто исполняемыми участками кода[12].

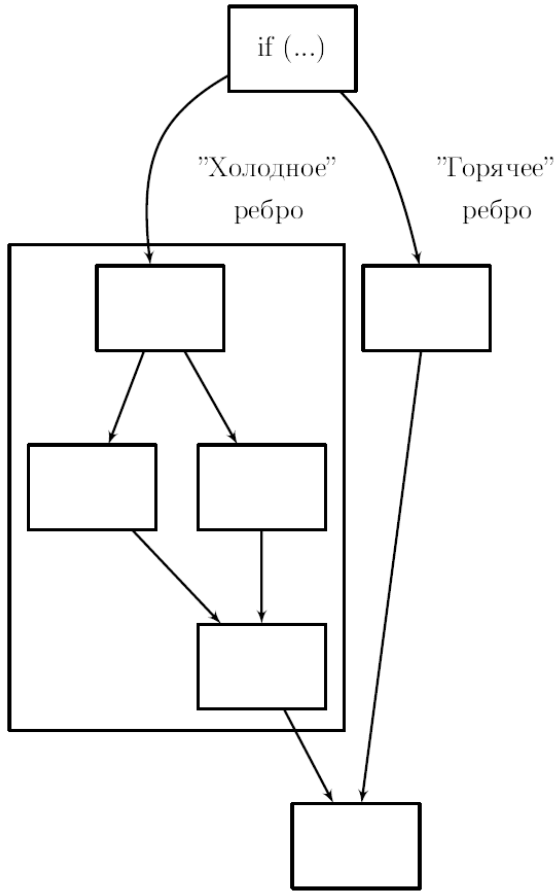
Для выделения ”горячих” функций используется одномерной кластеризация по весу из профиля с помощью алгоритма k-средних[13] при $k = 3$. Мы разделяем функции на 3 класса:

- ”горячие”
- ”средние”
- ”холодные”

Начальными центрами масс каждого класса выбраны максимальный, средний и минимальный веса функций соответственно.

Для выделения ”холодных” ребер внутри ”горячих” функций рассматриваются условные переходы - считаем ребро холодным, если оно имеет вес в 100 и более раз меньший другого ребра данного условного перехода. После выделения "холодного" ребра, мы должны выделить максимальный набор базовых блоков, в которые может попасть поток управления только при прохождении по выделенному на предыдущем шаге ребру. Выбираются все

Рис. 2 Рассматриваемый граф с выделенным для вынесения подграфом



блоки, над которыми доминирует блок, в который и входит выделенное ребро [12][14].

Рассмотрим некоторую часть графа потока управления ”горячей” функции (см. рисунок 2).

Стоит заметить, что требуется корректно обрабатывать следующие случаи:

- множественные выходы из выносимого подграфа
- внешние - функции, содержащие переменные из выносимых блоков

Для случая с множественными выходами, мы создаем специальный блок, который будет являться единственной точкой выхода из функции. В созданном блоке автоматически выбирается нужный код возврата. Проблема с -функциями решается их разделением на две части, одна остается во внешнем коде, другая вставляется в последний блок выносимого кода, обрабатывая соответствующие переходы.

На тестах SQLite, Expedite, Cray и Coremark дала средний прирост скорости в 0,8%. При использовании ее вместе с оптимизацией встраивания получен средний прирост в ~3%. Размер исполняемого файла увеличивается на 1- 7%, в зависимости от приложения.

3.4 Спекулятивная девиртуализация

Для объектно-ориентированных языков программирования решение о вставке виртуальных функций является проблемой, для решения которой недостаточно знать количество ее вызовов. В программах, написанных на языке Си++, могут быть два типа виртуальных вызовов функций: классические вызовы по указателю на функцию и вызовы виртуальных методов классов. Когда компилятор встречает такие вызовы, он не может определить, какая функция будет вызываться. Чтобы понять, какая функция будет вызвана, необходимо произвести дополнительный анализ [15]. Этот анализ включает в себя: сравнение сигнатур функций, анализ иерархии наследования классов и анализ типов существующих в точке вызова. Сравнение сигнатур отсекает «неподходящие» по возвращаемому значению и параметрам функции. Анализ иерархии наследования – выявляет классы, для которых существует реализация виртуального метода и отсекает классы, находящиеся выше по иерархии, чем тип указателя. Анализ существующих в точке вызова объектов рассматривает, объекты каких классов были созданы и еще не уничтожены в момент вызова функции.

Помимо этого была добавлена возможность инструментирования вызовов виртуальных функций сохранением информации о количестве вызовов конкретной виртуальной функции. Таким образом, используя профиль, мы можем определить наиболее вероятного кандидата на девиртуализацию.

Реализованный алгоритм сочетает в себе вышеописанные методики. После проведения девиртуализации и принятия решения о вставке функции, если оказывается, что кандидат на вставку всего один, – вставляется он. Если кандидатов несколько - производится спекулятивная девиртуализация: по данным профилирования мы можем сказать, какая реализация виртуальной функции исполнялась наиболее часто, и вставляем инструкцию “if”, в теле которой производится вставка «горячей» функции, а в ветке “else” произведется вызов альтернативной, «холодной» версии функции.

Поскольку биткод LLVM не содержит высокоуровневой информации, анализ иерархии должен производиться в компиляторе переднего плана, с сохранением результатов в метаинформации биткода. Таким образом, понадобилось дополнительно реализовать экспорт метаинформации в компиляторе переднего плана Clang[16]. Во время тестирования был отмечен прирост производительности в 3% при увеличении размера кода всего на 1%.

Помимо этого, алгоритм успешно проходит синтетические тесты для тестирования девиртуализации предложенные сообществом GCC[17]. В настоящее время ведутся работы направленные на увеличение точности статического анализа

3.5 Использование команд предвыборки при обработке массивов в цикле

Оптимизация вставляет команды предвыборки для оптимизации использования кэша процессора во время последовательной загрузки данных из массивов в циклах.

Процессоры архитектуры ARM серии Cortex-A9 имеют встроенный автоматический механизм предвыборки данных, который загружает данные в кэш, учитывая промахи кэша[18], массив загружается в кэш после нескольких итераций и промахов кэша. Такое поведение не оптимально 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.

Данная оптимизация анализирует циклы с индукционной переменной в канонической форме – индукционная переменная имеет тип целого числа, инициализируется нулем и увеличивается на единицу каждую итерацию. В цикле в такой форме выделяются все загрузки из массивов, использующие индукционную переменную.

Команды предвыборки должны быть выполнены заранее, чтобы данные находились в кэше в тот момент, когда они будут использоваться. Слишком позднее выполнение команды предвыборки приведет к тому, что данные не будут загружены в кэш к моменту их использования. Оптимизация производится над промежуточным представлением LLVM, поэтому, чтобы эффективно генерировать команды предвыборки, необходимо оценивать

количество машинных команд в цикле. Мы используем приближенную оценку – считаем, что вызов функции преобразуется в столько машинных команд, сколько у функции аргументов плюс 1, некоторые команды промежуточного представления не преобразуются в машинные команды (фи-функции, команды “GetElementPtr”, некоторые команды преобразования типов), остальные команды преобразуются в одну машинную инструкцию.

После оценки размера цикла, вычисляется, на сколько итераций вперед следует совершать предвыборку данных. Эта величина равна отношению задержки загрузки данных в кэш после выполнения команды предвыборки к количеству команд в цикле. На процессоре ARM Cortex-A9 требуется выполнение около 200 команд после команды предвыборки, чтобы данные были загружены в кэш.

Также производится попытка определить количество итераций цикла. Если количество итераций не является константой и вычисляется во время выполнения программы, то оно определяется из собранного профиля программы, если он доступен.

Если же и профиль программы недоступен, то количество итераций цикла оценивается на основе вероятностной оценки ветвления программы. Подход заключается в оценке количества переходов по каждому ребру на основе эвристических признаков и статистических данных [19]. Данный метод оценивает количество раз выполнения каждого базового блока. Для того, чтобы оценка была целочисленной, количество раз выполнения первого базового блока функции считается равным 1024. Далее в каждой точке ветвления программы производится попытка определить сколько раз произойдет переход по каждому ребру.

Используемые эвристические признаки:

– В цикле переход в заголовок происходит чаще, чем выход из цикла (вероятность перехода в заголовок цикла – 97%).

– Переход в базовый блок с командой возврата происходит редко (его вероятность 25%).

– Сравнение двух указателей или указателя с NULL чаще всего дает отрицательный результат (вероятность этого 62.5%).

– Сравнение переменной с нулем чаще всего дает отрицательный результат (вероятность этого 62.5%).

Эвристики применяются последовательно до первого подходящего. Если ни один из признаков не применим к точке ветвления, то все переходы считаются равновероятными.

После применения эвристик каждому ребру графа потока управления соответствует вероятность перехода по этому ребру. Используя эту

информацию можно получить ожидаемую частоту выполнения каждого базового блока. По всем управляющим конструкциям языка, кроме цикла, частота распространяется тривиальным образом – пропорционально вероятностям исходящих из базового блока ребер. Для циклов вычисляется вероятность, учитывающая переходы по обратным ребрам. Используя данную вероятность вычисляется частота заголовка цикла. Поделив частоту заголовка цикла на частоту первого базового блока функции можно получить статистическую оценку количества итераций цикла.

Предвыборка данных улучшает производительность генерируемого кода, когда команды обращения к памяти чередуются с вычислительными командами ЦПУ. Если в оптимизируемом цикле нет достаточного количества вычислительных команд, то предвыборка не даст значительного выигрыша в производительности. В разработанной оптимизации используется оценка отношения количества вычислительных команд к командам работы с памятью. Для выигрыша в производительности это отношение должно быть больше установленного значения. Для процессора ARM Cortex-A9 значение данной величины равно 3.

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 дала прирост производительности от 0,5 до 5 %, средний прирост составляет ~2.5%

4 Сервер приложений

Предлагаемый метод двухэтапной компиляции позволяет проводить оптимизацию программы с учетом собранного профиля, как при динамической компиляции, так и во время простоя системы. Но для мобильных устройств зачастую оптимизация программ на устройстве является затруднительной. Для снижения нагрузки на устройство предлагается использовать специальный сервер приложений [2], при таком подходе приложения, скомпилированные в промежуточное представление LLVM, будут храниться в специальном облачном хранилище, там же будет происходить генерация бинарного кода и оптимизация программы с учетом информации о ее профиле. Поскольку для каждого приложения будет поступать профиль от нескольких пользователей, то усреднив полученный набор профилей и проведя машинно-независимую оптимизацию, мы получим промежуточное представление, более отвечающее реальным вариантам

использования. Используя информацию усредненного профиля, мы можем сократить расходы при компиляции приложений для новых пользователей, а также повысить производительность динамической компиляции для пользователей, использующих ее на своих устройствах.

Помимо решения задач связанных с производительностью, применение сервера приложений позволит проводить запутывание приложений и осуществлять статическую проверку их кода инструментами типа SVACE[20], что поможет обеспечить защиту от обратного проектирования, и повысить устойчивость приложений к использованию их уязвимостей. Применяя различные варианты запутывающих преобразований [21],[22] мы можем получить множество различных вариантов кода одного приложения, что затруднит создание универсальной программы эксплуатирующей уязвимость.

4.1 Динамический выбор уровня оптимизаций

Механизм динамического переключения между уровнями оптимизации предназначен для решения вопроса экономии времени компиляции на функциях, которые исполняются редко при одновременном улучшении кода часто используемых функций. Как показано в работе [23], посредством только изменения опций можно достичь значительных результатов даже для статических компиляторов.

Идея оптимизации состоит в том, чтобы не применять оптимизации для «холодных» функций, поскольку их оптимизация не оказывает существенного влияния на производительность программы. Но поскольку таких функций достаточно много, время затрачиваемое на их анализ и оптимизацию может составлять большую часть времени расходуемого на компиляцию и оптимизацию программы[24][25][26]. Такая методика применима, как для динамической компиляции, где важна быстрая компиляция, а дополнительная оптимизация может производиться во время работы программы, так и для первичной компиляции больших программных комплексов для сбора профиля, и последующей быстрой генерации оптимизированной версии программы[27][28][29].

Было реализовано 3 варианта сочетаний наборов оптимизаций:

- минимальный O0 (не оптимизировать вовсе) для «холодных» на O2(стандартный набор) для «горячих»
- средний O1(минимальный набор) для «холодных» и O3(агрессивные оптимизации) для «горячих»
- максимальный - O2 для «холодных», O3 для «горячих»

При тестировании на SQLite было выявлено, что при минимальном уровне экономится до 90% времени компиляции, при аналогичной производительности. Для среднего экономия составляет 2-5% при

производительности, лучшей, чем при обычной компиляции с O3 на 1-3%, и максимальный дает экономию ~5% при ускорении на 3-4% в сравнении с O3.

5 Заключение

В данной статье были рассмотрены изменения, внесенные в систему двухэтапной компиляции позволяющие повысить производительность и совместимость системы. Описывается подход позволяющий снизить накладные расходы на сбор профиля программы с помощью инструментации. Для обеспечения учета профиля во время работы динамического компилятора был реализован механизм замещения на стеке. В качестве вспомогательного инструмента для отладки оптимизаций была реализована возможность аннотации графа потока управления весами из профиля программы, при выводе его графическом виде. Были предложены оптимизации учитывающие профиль программы. На основе приложенных методов и разработанных технологий возможно создать облачное хранилище позволяющее обеспечить как переносимость программ, в пределах одной архитектуры, так и учет специфики конкретной аппаратуры на которой производится развертывание программы. С другой стороны предлагаемый метод распространения программы в промежуточном представлении позволяет обеспечить высокую степень надежности и безопасности приложений, распространяемых через облачное хранилище, поскольку делает возможным применение специализированных инструментов статического анализа и запутывания кода.

Список литературы

[1]. Chris Lattner. LLVM: An Infrastructure for Multi-Stage Optimization.— Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL.

[2]. А. И. Аветисян. Двухэтапная компиляция для оптимизации и развертывания программ на языках общего назначения. – Труды ИСП РАН – 2012. - №12.

[3]. А.И. Аветисян, К. Ю. Долгорукова; Ш. Ф. Курмангалеев, Динамическое профилирование программы для системы LLVM, 201171-82

[4]. OProfile official website. http://oprofile.sourceforge.net.

[5]. Ian Lance Taylor (2008). "A New ELF Linker". GCC Developers' Summit 2008. pp. 129–136. Retrieved 2013-03-06.

[6]. GNU Binutils, http://www.gnu.org/software/binutils

[7]. Introduction to the Autotools, http://www.dwheeler.com/autotools/

[8]. CheckInstall, http://asic-linux.com.mx/~izto/checkinstall/

[9]. Pratschner, S. An overview of the .net compact framework garbage collector.— 2004. http://blogs.msdn.com/b/stevenpr/archive/2004/07/26/197254.aspx.

[10]. Soman, S. Efficient and General On-Stack Replacement for Aggressive Program Specialization / Sunil Soman, Chandra Krintz // International Conference on Programming Languages and Compilers. — 2006. — . — http://www.cs.ucsb.edu/~{}ckrintz/papers/osr.pdf.

[11]. Graphviz Documentation, http://www.graphviz.org/Documentation.php

[12]. Peng Zhao “Code and Data Outlining”, 2005.
[13]. Мандель И. Д. Кластерный анализ. — М.: Финансы и статистика, 1988.
[14]. Jun-Pyo Lee, Jae-Jin Kim, Soo-Mook Moon, Suhyun Kim ”Aggressive Function Splitting for Partial Inlining”, Seoul, South Korea, 2011.
[15]. David F. Bacon and Peter F. Sweeny, Fast Static Analysis of C++ Virtual Functuion Call, 1996
[16]. Clang: a C language family frontend for LLVM, http://clang.llvm.org/
[17]. GCC Free software foundation, http://gcc.gnu.org
[18]. Справочное руководство по процессорной архитектуре ARM., http://infocenter.arm.com
[19]. Youfeng Wu, James R. Larus. Static Branch Frequency and Program Profile Analysis. 27th International Symposium on Microarchitecture, 1994
[20]. Арутюн Аветисян, Андрей Белеванцев, Алексей Бородин, Владимир Несов. Использование статического анализа для поиска уязвимостей и критических ошибок в исходном коде программ. Труды ИСП РАН том 21, 2011, стр. 23-38.
[21]. Ш.Ф. Курмангалеев, В.П. Корчагин, Р.А. Матевосян. Описание подхода к разработке обфусцирующего компилятора. Труды Института системного программирования РАН, том 23, 2012 г. Стр. 67-76.
[22]. Ш.Ф. Курмангалеев, В.П. Корчагин, В.В. Савченко, С.С. Саргсян. Построение обфусцирующего компилятора на основе инфраструктуры LLVM. Труды Института системного программирования РАН, том 23, 2012 г. Стр. 77-92.
[23]. Р. Жуйков, Д. Плотников, М. Варданян. Автоматическая настройка оптимизационных преобразований компилятора GCC для платформы ARM. Труды Института системного программирования РАН, том 22, 2012 г. Стр. 49-66.
[24]. K. Pettis, R. C. Hansen. Profile guided code positioning . SIGPLAN Not. 1990 June.Vol. 25, no. 6, pp. 16-27. http://dx.doi.org/10.1145/93548.93550.
[25]. P. P. Chang, S. A. Mahlke, W.-m. W. Hwu. Using profile information to assist classic code optimizations. Center for Reliable and High-Performance Computing. Urbana-Champaign: University of Illinois, 1991.
[26]. Da Silva, A. F. Our Experiences with Optimizations in Sun’s Java Just-In-Time Compilers / Anderson F. Da Silva, Vitor S. Costa // Journal of Universal Computer Science. — 2006. — Vol. 12.
[27]. M. Arnold, Stephen J. Fink, D. Grove, M. Hind, Peter F. Sweeney. A Survey of Adaptive Optimization in Virtual Machines. IBM T. J. Watson Research Center, Hawthorne, USA, 2004.
[28]. Stephen Fink, David Grove, Michael Hind. Dynamic Compilation and Adaptive Optimization in Virtual Machines. IBM, June 2004.
[29]. Holzle, U. Adaptive Optimization for SELF: Reconciling High Performance with Exploratory Programming: Ph. D. thesis / Stanford University. —1994. — http://research.sun.com/self/papers/hoelzle-thesis.ps.gz.

Machine-specific optimization methods for C/C++ applications that are distributed in the LLVM intermediate representation format

Kurmangaleev Sh.F.
kursh@ispras.ru
ISP RAS, Moscow, Russia

Annotation. This paper analyzes approaches for optimizing C/C++ applications used in two-stage compilation system, allowing distributing such applications in the LLVM (low level virtual machine) intermediate representation. The on-stack replacement technique implemented in the LLVM just-in-time compiler is described. The paper presents a static instrumentation technique with incomplete control flow edge covering and its correction to the full control flow profile. The proposed approach allows the derived profile information to be comparable in quality to the classical approach while significantly reducing the profiling overhead.

Also, the technique for converting dynamically collected profile to LLVM profile format for statically compiled applications is presented. This paper evaluates both existing optimizations modified for using the collected profile and the newly developed ones. Implemented profile based inlining, outlining, speculative devirtualization. Also implemented annotation of control flow graph by profile weights to make profile based optimizations easy for debugging. Proposed the technique which allows using the prefetch instructions to improve the effectiveness of array processing codes. This paper describes the approach for building a specific cloud application storage that allows solving program optimization and protection problems. Finally, we present the approach for reducing compilation and optimization overhead in the cloud infrastructure.

Keywords: Two-stage compilation, optimization, LLVM, cloud storage

References

[1]. Chris Lattner. LLVM: An Infrastructure for Multi-Stage Optimization.— Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL.

[2]. Arutyun Avetisyan. Dvukhehtapnaya kompilyatsiya dlya optimizatsii i razvertyvaniya programm na yazykakh obshhego naznacheniya. [Two-stage compilation for optimizing and deploying programs in general purpose languages]. Trudy ISP RAN [The Proceedings of ISP RAS], 2012, vol. 22, pp. 11-18 (in Russian).

[3]. A.I. Avetisyan, K.U. Dolgorukova; Sh.F. Kurmangaleev. Dinamicheskoe profilirovanie programmy dlya sistemy LLVM [Dynamic profile collection for LLVM]. Trudy ISP RAN [The Proceedings of ISP RAS], 2011, vol. 21, pp. 71-82 (in Russian)

[4]. OProfile official website. http://oprofile.sourceforge.net.

[5]. Ian Lance Taylor (2008). "A New ELF Linker". GCC Developers' Summit 2008. pp. 129–136.

[6]. GNU Binutils, http://www.gnu.org/software/binutils
[7]. Introduction to the Autotools, http://www.dwheeler.com/autotools/
[8]. CheckInstall, http://asic-linux.com.mx/~izto/checkinstall/
[9]. Pratschner, S. An overview of the .net compact framework garbage collector.— 2004. http://blogs.msdn.com/b/stevenpr/archive/2004/07/26/197254.aspx.
[10]. Soman, S. Efficient and General On-Stack Replacement for Aggressive Program Specialization / Sunil Soman, Chandra Krintz // International Conference on Programming Languages and Compilers. — 2006. — . — http://www.cs.ucsb.edu/~{}ckrintz/papers/osr.pdf.
[11]. Graphviz Documentation, http://www.graphviz.org/Documentation.php
[12]. Peng Zhao “Code and Data Outlining”, 2005.
[13]. Mandel' I. D. Klasternyj analiz [Cluster Analysis]. — M.: Finansy i statistika, 1988. (in Russian)
[14]. Jun-Pyo Lee, Jae-Jin Kim, Soo-Mook Moon, Suhyun Kim ”Aggressive Function Splitting for Partial Inlining”, Seoul, South Korea, 2011.
[15]. David F. Bacon and Peter F. Sweeny, Fast Static Analysis of C++ Virtual Functuion Call, 1996
[16]. Clang: a C language family frontend for LLVM, http://clang.llvm.org/
[17]. GCC Free software foundation, http://gcc.gnu.org
[18]. ARM architecture., http://infocenter.arm.com
[19]. Youfeng Wu, James R. Larus. Static Branch Frequency and Program Profile Analysis. 27th International Symposium on Microarchitecture, 1994
[20]. Arutyun Avetisyan, Andrey Belevantsev, Alexey Borodin, Vladimir Nesov. Ispol'zovanie staticheskogo analiza dlya poiska uyazvimostej i kriticheskikh oshibok v iskhodnom kode programm [Using static analysis for finding security vulnerabilities and critical errors in source code]. Trudy ISP RAN [The Proceedings of ISP RAS], 2011, vol. 21, pp. 23-38. (in Russian)
[21]. Kurmangaleev S.F. Korchagin V.P. Matevosyan H.A. Opisanie podkhoda k razrabotke obfustsiruyushhego kompilyatora. [Description of the approach to development of the obfuscating compiler]. Trudy ISP RAN [The Proceedings of ISP RAS], 2012, vol.23, pp. 67-76 (in Russian)
[22]. Kurmangaleev S.F. Korchagin V.P. Savchenko V.V. Sargsyan S.S. Postroenie obfustsiruyushhego kompilyatora na osnove infrastruktury LLVM. [Building obfuscation compiler based on LLVM infrastructure]. Trudy ISP RAN [The Proceedings of ISP RAS], 2012, vol. 23, pp. 77-92. (in Russian)
[23]. Roman Zhuykov, Dmitry Plotnikov, Mamikon Vardanyan. Avtomaticheskaya nastrojka optimizatsionnykh preobrazovanij kompilyatora GCC dlya platformy ARM. [Automatic tuning of GCC optimizations for ARM platform]. Trudy ISP RAN [The Proceedings of ISP RAS], 2012, vol. 22, pp. 49-66, (in Russian)
[24]. K. Pettis, R. C. Hansen. Profile guided code positioning . SIGPLAN Not. 1990 June.Vol. 25, no. 6, pp. 16-27. http://dx.doi.org/10.1145/93548.93550.
[25]. P. P. Chang, S. A. Mahlke, W.-m. W. Hwu. Using profile information to assist classic code optimizations. Center for Reliable and High-Performance Computing. Urbana-Champaign: University of Illinois, 1991.
[26]. Da Silva, A. F. Our Experiences with Optimizations in Sun’s Java Just-In-Time Compilers / Anderson F. Da Silva, Vitor S. Costa // Journal of Universal Computer Science. — 2006. — Vol. 12.

[27]. M. Arnold, Stephen J. Fink, D. Grove, M. Hind, Peter F. Sweeney. A Survey of Adaptive Optimization in Virtual Machines. IBM T. J. Watson Research Center, Hawthorne, USA, 2004.
[28]. Stephen Fink, David Grove, Michael Hind. Dynamic Compilation and Adaptive Optimization in Virtual Machines. IBM, June 2004.
[29]. Holzle, U. Adaptive Optimization for SELF: Reconciling High Performance with Exploratory Programming: Ph. D. thesis / Stanford University. —1994. — http://research.sun.com/self/papers/hoelzle-thesis.ps.gz.