Двухэтапная компиляция для оптимизации и развертывания программ на языках общего назначения

Арутюн Аветисян <arut@ispras.ru>

Аннотация. В статье описывается метод двухэтапной компиляции программ на языках общего назначения (Си/Си++), основанный на компиляторной системе LLVM и позволяющий проводить оптимизации программ с учетом профиля пользователя и особенностей его целевой машины, а также организовывать развертывание программ в облачном хранилище с дополнительной прозрачной оптимизацией и поиском дефектов программ. Особенностью метода является применимость к языкам общего назначения и использование общей компиляторной инфраструктуры на всех этапах оптимизации и развертывания программы.

Ключевые слова: динамическая оптимизация, LLVM, уязвимость, облачное хранилище.

1. Введение

Широко применяемые статические оптимизации и учет профиля программы на машине разработчика являются основными видами оптимизации программ на языках Си и Си++. Очевидно, что при этом возникают сложности с точным учетом особенностей поведения пользователя и архитектуры его машины, поскольку при сборке на машине разработчика эти особенности либо не могут быть выяснены, либо требуется поддержка многих версий собранной программы для точной подгонки под необходимую архитектуру. Для решения этих задач, как впервые предложено в [1, 5], предлагается разделить этап изначальной сборки программы с применением машинно-независимых оптимизаций и этап окончательной специализации программы для конкретной машины пользователя и поведения пользователя. Такой подход мы называем методом двухэтапной компиляции.

Для осуществления двухэтапной компиляции необходимо распространение программы в объектных файлах, содержащих ее внутреннее представление, сохраняющее информацию высокого уровня и позволяющее проводить профилирование и машинно-зависимую оптимизацию программы. Наличие общей инфраструктуры двухэтапной компиляции позволяет решать следующие задачи:

- Учет точного профиля пользователя при динамической оптимизации на его машине, т.е. программа оптимизируется на конкретном наборе входных данных для данного конкретного запуска, а разные запуски программы могут приводить к различным оптимизациям. При этом значительно упрощается процесс разработки без потерь в производительности получаемой программы;
- Учет параметров машины пользователя при адаптивной оптимизации на его машине. Машинно-зависимые оптимизации, требующие информации об особенности архитектуры пользователя (распределение регистров, планирование команд, упреждающая загрузка, векторизация и т.п.) могут выполняться по-разному, существенно (десятки процентов) влияя на производительность;
- Оптимизация во время простоя, т.е. применение обоих описанных выше подходов во время простоя машины пользователя по тем же сохраненным данным профиля. Можно реализовывать разные оптимизационные стратегии на все время жизни программы (например, регулярная оптимизация по новым данным либо пересборка по поступлении сильно отличающегося профиля пользователя);
- Сокращение затрат на распространение и поддержку программы (достаточно поддерживать одну версию программы, при сборке которой применялись лишь машинно-независимые оптимизации). Конечно, для языка Си невозможно добиться портируемости, схожей с Java, из-за того, что особенности целевой машины попадают в программу на самых ранних этапах компиляции (а именно – на этапе препроцессирования). Разумнее говорить о портируемости единой версии программы между вариантами одного семейства архитектур (например, ARM) либо между архитектурами с совпадающими размерами базовых типов данных (например, ARM и x86). Тем не менее, созданием специальной “виртуальной” переносимой целевой архитектуры и компиляцией пользовательских программ под нее, а потом – их “подгонкой” под реальную целевую архитектуру возможно частично ослабить эти ограничения. Вопросом для исследований остается, во-первых, доля реальных программ, для которых можно добиться портируемости таким образом, а, во-вторых, необходимые дополнительные затраты на сглаживание разницы между реальной и виртуальной архитектурами и, как следствие, падение производительности. Примером подобного подхода является проект Google Portable Native Client [4].

Основой для реализации метода двухэтапной компиляции может служить LLVM [2, 3] – популярная компиляторная инфраструктура с открытыми исходными кодами на языке Си++. В рамках LLVM реализованы статический компилятор, компоновщик, виртуальная машина, JIT-компилятор и другие библиотеки. Функционирование системы обеспечивается единым внутренним

представлением, которое может быть представлено в текстовом виде, в виде структур данных в оперативной памяти, а также в двоичном виде как бит-код. Этот бит-код может быть сохранен в промежуточных объектных файлах для дальнейшей оптимизации, в том числе динамической. При этом возможно использовать все предоставляемые LLVM возможности по обработке внутреннего представления (включая различные анализы, трансформации и т.п.).

Важно упомянуть отличия предлагаемого метода от похожих подходов, использующихся в динамических языках, особенно на платформе .NET [6] с применением на машине пользователя компилятора NGEN [7] для генерации из байт-кода MSIL бинарной программы для нужной целевой архитектуры. Во-первых, платформа .NET ограничивает использование языков общего назначения, например, поддерживая язык C++/CLI [8], который по сути является отдельным динамическим языком, лишь происходящим от Си++. Более того, компилятор NGEN может не поддерживать преобразование некоторых методов байт-кода MSIL в бинарный код, полагаясь для таких случаев на обычный JIT-компилятор. Во-вторых, насколько можно судить, при компиляции NGEN не использует собранный профиль конкретного пользователя, ограничиваясь статическими оптимизациями. Наконец, преимущество нашего подхода в использовании единой инфраструктуры LLVM на всех этапах компиляции, от оптимизаций на машине разработчика до машины пользователя, что позволяет переиспользовать код при поддержке динамических, статических оптимизаций, оптимизаций во время простоя, а также переиспользовать компоненты поддержки профиля.

Далее в разделе 2 описывается реализация метода двухэтапной компиляции на базе LLVM, рассчитанная в том числе на применение на встраиваемых устройствах с процессором ARM. Раздел 3 посвящен концепции совместного использования облачного хранилища и двухэтапной компиляции для повышения производительности и безопасности программ. Раздел 4 заключает статью.

2. Метод двухэтапной компиляции

В предлагаемой реализации метода двухэтапной компиляции на первом этапе приложение компилируется на машинах разработчиков специальным набором компиляторных инструментов на базе LLVM, при этом выполняются лишь машинно-независимые оптимизации. Результат компиляции сохраняется в бит-кодовых файлах LLVM, дополнительно автоматически генерируется информация об устройстве программного пакета и о схеме его инсталляции. На втором этапе программа оптимизируется на машине пользователя, возможно, с учетом его поведения и особенностей его вычислительной системы. Поддерживается несколько режимов работы: а) автоматическая кодогенерация бинарной программы, оптимизированной под конкретную архитектуру, и инсталляция программы с помощью сохраненной на первом этапе информации; б) динамическая оптимизация программы во время её

работы с учетом собранного профиля пользователя с помощью реализованной инфраструктуры сбора профиля на основе пакета Oprofile и JIT-оптимизации на основе JIT-компилятора LLVM (подробнее см. [5]); в) оптимизация программы с учетом профиля пользователя во время простоя системы для экономии ресурсов (idle-time optimization).

Опишем разработанную схему двухэтапной компиляции для программ, система сборки которых основана на использовании утилит configure и make. Доработка инструментов LLVM производилась из-за того, что изначально в LLVM не предусмотрены средства прозрачного, автоматического получения бит-кода с учетом зависимостей между модулями, а также отсутствует поддержка динамического связывания модулей с бит-кодом. На первом этапе происходит генерация установочного пакета, содержащего в себе модули бит-кода и скрипты автоматического развертывания. Вместо использования оригинальной утилиты configure предлагается использовать специальную обертку configure-proxy, осуществляющую необходимые подстановки и вызывающую оригинальную утилиту, результатом работы которой является скрипт make.sh. Принцип работы сгенерированного скрипта для сборки аналогичен – вызывается утилита make с передачей всех необходимых параметров, а дополнительно конфигурируются вызовы компилятора LLVM-GCC. Для компиляторов реализованы обертки, основанные на том же подходе – для каждого вызова изначального компилятора с помощью конфигурации, заданной измененными скриптами сборки, запускается компилятор LLVM-GCC или Clang, позволяющий получить (параллельно с изначальным объектным файлом) и файл с бит-кодовым представлением. Для того, чтобы не влиять на ход сборки программы и не модифицировать скрипты сборки, необходимо полностью сохранять все исходные результаты сборки.

Помимо этого, необходимые изменения внесены в компиляторы переднего плана и компоновщик – эти изменения позволяют отследить зависимости между отдельными модулями программы. После окончания компиляции программы с помощью скриптов пост-обработки происходит, создание инсталляционного пакета на основе сгенерированных зависимостей. Инсталляционный пакет содержит файлы с бит-кодом, файлы, помеченные как зависимости на этапе постобработки, и скрипты компиляции и установки.

На втором этапе во время установки программы существует две альтернативы – статическая компиляция или использование динамической компиляции. В обоих случаях все происходит прозрачно для пользователя за счет создания скриптов компиляции и установки на первом этапе – в зависимости от заданного режима работы скрипты вызывают либо статический компилятор LLVM для получения объектного кода по файлам с бит-кодом, либо вызывает интерпретатор и JIT-компилятор LLVM, который обеспечивает сборку необходимых частей программы во время ее работы.

При экспериментальной проверке корректности работы созданной системы двухпроходной компиляции на системах x86 и ARM дополнительно была

проведена оптимизация компонент LLVM для их более быстрой работы и потребления меньшего объема памяти, что существенно для встраиваемых архитектур. При кодогенерации программ на платформе ARM было достигнуто сокращение использование памяти на 1.6-10.9% и времени компиляции на 10-20%.

3. Облачное хранилище и двухэтапная компиляция

Метод двухэтапной компиляции позволяет оптимизировать программы пользователя на его машине как динамически, через создание соответствующих JIT-компиляторов, так и во время простоя программы, с учетом собранного профиля, отражающего поведение пользователя, и характеристик его машины. Тем не менее, для мобильных устройств часто полная оптимизация программ на устройстве является затруднительной, требуя большого количества ресурсов. Возможно избежать этих затрат путем переноса второго этапа компиляции на сервер приложений, а на устройство возложить лишь задачу сбора и передачи профиля на сервер.

Для организации такого сервера требуется новый метод распространения приложений. Такое распространение предлагается организовать через «облачное хранилище» приложений нового поколения, обеспечивающее как переносимость программ в рамках одного семейства процессорных архитектур ARM, так и высокую степень надежности и безопасности хранимых приложений. Для построения облачного хранилища необходимо использование предложенной схемы двухэтапной компиляции для получения представления программы в виде бит-код файлов LLVM, а также метаинформации об устройстве приложения и о схеме его установки (см. рисунок 1).

На первом этапе двухпроходной компиляции создается установочный пакет приложения с бит-кодом LLVM и указанной выше метаинформацией. После помещения пакета приложения в хранилище для обеспечения безопасности приложение может быть автоматически проверено инструментами среды Svace [9] на наличие критических ошибок и уязвимостей (используя бит-код LLVM, а также информацию о связях программы), а также в случае необходимости может быть проведен более глубокий динамический анализ, например, сборкой приложения и запуском в песочнице, соответствующей целевому устройству. Таким образом, обеспечивается необходимая степень безопасности приложений.

Для обеспечения продуктивности приложения после проверки его пакета на безопасность по запросу загрузки приложения конкретным пользовательским устройством в хранилище осуществляется второй этап схемы двухэтапной компиляции и генерируется установочный пакет с объектным кодом для конкретного устройства. После этого во время жизни приложения на устройстве по желанию пользователя может быть организован сбор динамического профиля на устройстве и сохранение его на диск [5]. В

периоды слабой загруженности устройства собранный профиль может быть передан в облачное хранилище для оптимизации приложения с его использованием, а на устройство загружена новая оптимизированная версия приложения.

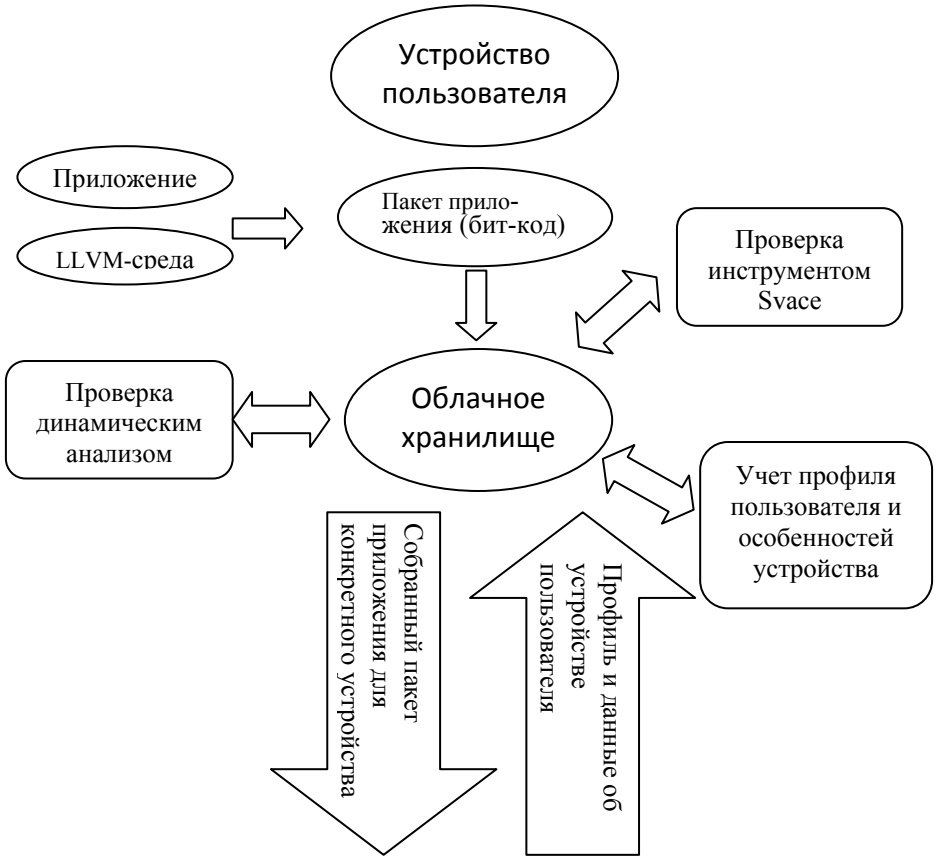

Рис. 1. Устройство облачного хранилища приложений.

Данный подход может быть использован для приложений как с открытым, так и с закрытым исходным кодом без каких-либо опасений со стороны разработчиков, так как восстановление исходного кода приложения по бит-код файлам LLVM является затруднительным. Отметим, что похожий подход к распространению программ широко применяется для динамических языков типа Java, но без дополнительной оптимизации и проверки на безопасность программ, помещенных в хранилище, что и обеспечивает перспетивность предлагаемой схемы распространения приложений.

4. Заключение

В данной статье был рассмотрен метод двухпроходной компиляции для программ, созданных на языках общего назначения, и позволяющий повысить как производительность программ учетом профиля конкретного пользователя и особенностей его машины, так и безопасность программ проверкой на дефекты и уязвимости в облачном хранилище. В настоящее время в ИСП РАН ведутся работы в двух направлениях – по улучшению методов динамической оптимизации на базе двухпроходной компиляции с применением собранного профиля и подменой кода на стеке и по разработке и реализации форматов развертывания и сборки приложений для облачного хранилища.

Список литературы

[1] А. Белеванцев, Д. Журихин, Д. Мельник. Компиляция программ для современных архитектур. Труды Института системного программирования РАН, том 17, 2009 г. стр. 31-50.

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

[3] Компиляторная инфраструктура LLVM. http://llvm.org/

[4] Portable Native Client Introduction. http://www.chromium.org/nativeclient/pnacl/building-and-testing-portable-native-client

[5] А.И. Аветисян, К.Ю. Долгорукова, Ш.Ф. Курмангалеев. Динамическое профилирование программы для системы LLVM. Труды ИСП РАН том 21, 2011, стр. 71-82.

[6] Инфраструктура .NET. http://msdn.microsoft.com/en-us/netframework/aa496123

[7] Компилятор NGEN. http://msdn.microsoft.com/en-us/library/6t9t5wcf.aspx

[8] Stanley B. Lippman. Pure C++: Hello, C++/CLI. MSDN Magazine, Visual Studio 2005 Guided Tour, 2006. http://msdn.microsoft.com/en-us/magazine/cc163681.aspx

[9] 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.

Two-stage compilation for optimizing and deploying programs in general purpose languages

Arutyun Avetisyan <arut@ispras.ru>
ISP RAS, Moscow, Russia

Abstract. We describe the approach for two-stage compilation of C/C++ programs using the LLVM compiler infrastructure that allows optimizing programs taking into account the user profile and his/her target machine features, as well as deploying programs in a cloud storage while transparently optimizing and checking for defects. The notable features of the approach are its applicability to programs written using general-purpose languages and utilizing the common compiler infrastructure on all optimization and deployment stages.

On the first stage (host machine) the build process is transparently captured and the LLVM bitcode files are generated with the proper support for archive/library files. LTO optimizations are also performed at this time, and the program dependencies and installation structure (e.g. libraries, installed resource or documentation files) are also captured. On the second stage (target machine) the final code generation and installation is performed with the transparent path translation. Either static code generation or dynamic JIT-based compilation is supported.

As mentioned, a cloud-based deployment strategy can also be used allowing for additional features like defect and vulnerability checking while the cloud store has access to the deployed programs in the LLVM bitcode form.

While optimizing LLVM toolchain for working on ARM embedded devices we have achieved the memory consumption reduction of up to 10% and 10-20% compile time decrease.

Keywords: dynamic optimization, LLVM, security vulnerability, cloud storage.

References

[1]. A. Belevantsev, D. Zhurikhin, D. Melnik. Kompilyatsiya programm dlya sovremennykh arkhitektur. [Program compilation for modern architectures] Trudy ISP RAN [The Proceedings of ISP RAS], volume 16, p. 31-50, 2009. (In Russian)

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

[3]. LLVM Compiler infrastructure, http://llvm.org/

[4]. Portable Native Client Introduction, http://www.chromium.org/nativeclient/pnacl/introduction-to-portable-native-client

[5]. 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)

[6]. The .NET infrastructure, http://msdn.microsoft.com/en-us/netframework/aa496123

[7]. The NGEN Compiler, http://msdn.microsoft.com/en-us/library/6t9t5wcf.aspx

[8]. Stanley B. Lippman. Pure C++: Hello, C++/CLI. MSDN Magazine, Visual Studio 2005 Guided Tour, 2006, http://msdn.microsoft.com/en-us/magazine/cc163681.aspx

[9]. A. Avetisyan, A. Belevantsev, A. Borodin, V. 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).