Применение метода двухфазной компиляции на основе LLVM для распространения приложений с использованием облачного хранилища[1]

С.С. Гайсарян, Ш.Ф. Курмангалеев, К.Ю. Долгорукова, В.В. Савченко, С.С. Саргсян
{ssg@ispras.ru, kursh@ispras.ru, unerkannt@ispras.ru, sinmipt@ispras.ru, sevaksargsyan}@ispras.ru

Аннотация. В статье описывается метод двухфазной компиляции программ на языках Си/Си++, позволяющий распространять приложения в промежуточном представлении LLVM. Описывается модификация компонентов LLVM с целью сокращения времени генерации кода. Описываются разработанные оптимизации с использованием профиля выполнения программы. Рассматривается организация специализированного облачного хранилища приложений.

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

1. Введение

Процесс распространения ПО через магазины приложений состоит в следующем: разработчик передаёт программный продукт владельцу магазина приложений, приложение размещается в интернет-магазине и становится доступным пользователю. Чтобы программный продукт оставался конкурентоспособным в течение длительного времени, разработчику необходимо обеспечивать функционирование приложения на большинстве программно-аппаратных платформ.

Как правило, эта проблема решается следующим образом: приложение или компилируется и оптимизируется разработчиком повторно для каждой новой платформы, с последующим добавлением в магазин новой версии бинарного кода, или реализуется с использованием динамических языков.

При применении JavaVM приложение распространяется в промежуточном представлении. Переносимость достигается ценой потери производительности

[1] Работа выполнена при поддержке РФФИ, грант 11-01-00954-а

из-за использования дополнительного слоя абстракции, скрывающего реальное оборудование от исполняемой программы. В этом случае для обеспечения приемлемого уровня производительности дополнительно применяются динамические оптимизации на целевой архитектуре во время исполнения приложения, в том числе оптимизации, учитывающие профиль исполнения программы.

Разработка на традиционных языках (например, Си/Си++) позволяет учитывать особенности платформ, используя машинно-зависимые оптимизации (распределение регистров, планирование и конвейеризация кода, векторизация), выигрыш от применения которых может достигать нескольких десятков процентов. По этой причине производитель предоставляет Native SDK, позволяющий использовать такие языки. Вместе с тем, разработка приложений на традиционных языках и их распространение через магазины приложений требуют существенных дополнительных накладных расходов разработчиков на перенос приложений в связи с необходимостью поддержки большого количества различных платформ.

В настоящее время активно ведутся работы по автоматизации переноса приложений на языках Си/Си++ на различные платформы. Эти работы базируются на идее компилирования Си/Си++-приложений в промежуточное представление и их распространение по аналогии с приложениями на динамических языках. Наиболее интересные результаты получены компанией Google, которая разрабатывает проект Portable Native Client, имеющий целью обеспечить запуск единой версии программы на архитектурах ARM и x86. Эта разработка рассчитана на небольшие приложения, работающие в браузере Chrome, и имеет ряд ограничений, в том числе ухудшение показателей производительности.

В настоящей статье в разделе 2 описывается система двухэтапной компиляции программ на основе LLVM и изменения, внесенные в динамический компилятор LLVM. Раздел 3 описывает реализованные оптимизации. В разделе 4 описывается сервер приложений. Раздел 5 завершает статью.

2. Двухфазная компиляция

В предлагаемой реализации метода двухфазной компиляции на первом этапе приложение компилируется на машинах разработчиков специальным набором компиляторных инструментов на базе LLVM [1], при этом выполняются лишь машинно-независимые оптимизации. Результат компиляции сохраняется в файлах с биткодом 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 генерация кода бинарной программы, оптимизированной под конкретную

архитектуру, и ее развертывание с помощью сохраненной на первом этапе информации; б) динамическая оптимизация программы во время её работы с учетом собранного профиля пользователя; в) оптимизация программы с учетом профиля пользователя во время простоя системы (idle-time optimization) для экономии ресурсов.

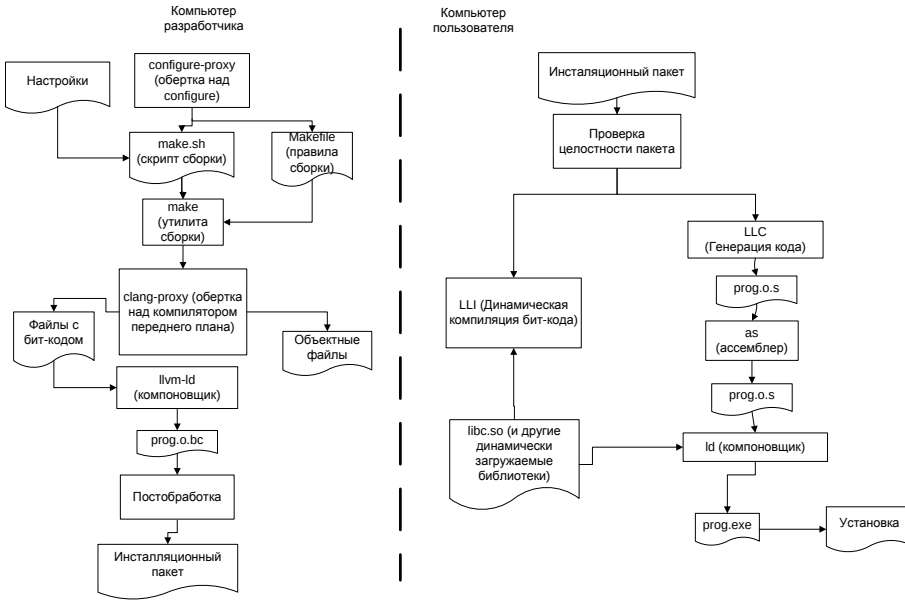

Рис. 1. Схема двухфазной компиляции

Схема работы системы двухфазной компиляции для программ, сборка которых основана на использовании утилит configure и make, представлена на рисунке 1. Указанные на рисунке дополнительные инструменты были разработаны и реализованы из-за того, что в LLVM не предусмотрены средства прозрачного, автоматического получения биткода с учетом зависимостей между модулями, а также отсутствует поддержка динамического связывания модулей с биткодом.

Предлагаемый метод распространения программ, написанных на языках Си/Си++, в промежуточном представлении позволяет решить проблему 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 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].

2.1. Описание модификаций в стандартных инструментах

Изменения в компиляторе Clang

Компилятор переднего плана CLANG [3] лишь частично поддерживает так называемую "кросс-компиляцию", т.е. запуск на одной архитектуре с генерацией кода для другой. Для обеспечения полной поддержки такого режима работы потребовалось обеспечить возможность указания путей к заголовочным файлам и перекомпилированным библиотекам целевой системы, а также изменить механизм поиска соответствующих файлов. Все необходимые параметры задаются на этапе сборки проекта. Необходимо отметить, что CLANG не имеет стабильной собственной реализации стандартной библиотеки языка Си++ и некоторых других необходимых во время компоновки библиотек, поэтому при сборке программ используются соответствующие библиотеки компилятора GCC[4]. Кроме того, поскольку для обоих вариантов компиляции, описанных в предыдущем разделе, требуется запуск модифицированного компоновщика, были выполнены необходимые изменения в драйвере.

Изменения в компоновщике

Поскольку на этапе компоновки мы можем определить порядок сборки и точные зависимости между модулями программы, были внесены необходимые изменения для сохранения данной информации. Такая информация требуется для формирования корректной очереди компиляции на второй фазе сборки и включения в инстанционный пакет только необходимых файлов. Кроме того, наличие промежуточного представления на этапе компоновки позволяет выполнить дополнительные оптимизации времени связывания (LTO-link time optimizations), область видимости которых будет расширена до нескольких единиц трансляции.

После завершения компоновки модулей, содержащих биткод LLVM, и их оптимизации происходит сохранение объединенного модуля, содержащего биткод, запись всех необходимых ему зависимостей для корректной компиляции на второй фазе сборки и модификация путей к используемым

системным библиотекам. В модули, которые содержат биткод LLVM и соответствуют финальным исполняемым файлам приложения, внедряется дополнительная информация об именах, требуемых для динамической компиляции других модулей, содержащих промежуточное представления LLVM.

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. Идея 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. Такая методика применима как для динамической компиляции, где важна быстрая компиляция, а дополнительная оптимизация может производиться во время работы программы, так и для компиляции больших программных комплексов.

Было реализовано 3 варианта сочетаний наборов оптимизаций:

- минимальный "O0" (не оптимизировать вовсе) для «холодных» и "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 уровня экономия составляет 2-5% при производительности, лучшей, чем при обычной компиляции с "O3" на 1-3%, и максимальный уровень дает экономию ~5% при ускорении на 3-4% в сравнении с "O3".

Оптимизация работы ключевых частей инфраструктуры

Дополнительно была проведена оптимизация компонент LLVM для их ускорения их работы и уменьшения потребления памяти, что существенно для встраиваемых архитектур. Для этого был снят и проанализирован профиль выполнения; выяснилось, что в коде присутствовали множественные выделения/освобождения памяти малого размера – порядка нескольких байт, а также неэффективная работа с локальными переменными – массивами. Так, например, во время генерации кода для программы SQLite LLVM выполняет 6098391 операций выделения памяти. Примерно 20% составляют запросы на выделение памяти размером 16 байт, 14% размером 8 байт, 11% размером 48

байт. Для устранения описанных проблем, была применена библиотека DLMalloc и переписан проблемный код. Во время генерации кода для программ на платформе ARM было достигнуто сокращение использования памяти на 1.6-10.9% и времени компиляции на 10-20%.

3. Используемые оптимизации

В настоящем разделе описываются основные оптимизации, выполненные в рамках разработанной инфраструктуры за последние два года. Другие реализованные оптимизации, в частности, на основе профиля программы и исправления кодогенерации LLVM для ARM, можно найти в статье [9].

3.1. Открытая вставка функций

Открытая вставка функций – оптимизирующее преобразование компилятора, вставляющее код функции на место его вызова в тело вызывающей функции. На тестах SQLite, Expedite, Cray и Coremark учет информации о профиле выполнения программы при проведении оптимизации привел к приросту скорости работы ~2%.

3.2. Клонирование блоков

Клонирование блоков - удвоение часто исполняемых базовых блоков графа потока управления, имеющих более одного исходящего ребра и более одного входящего. Суть алгоритма показана на рисунке 2.

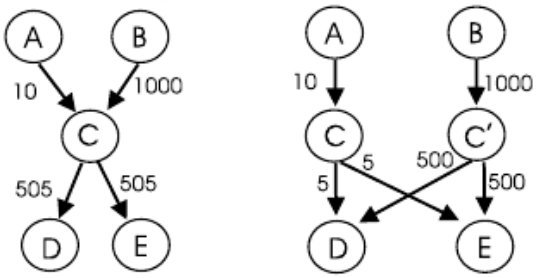

Рис. 2 Клонирование соединенных базовых блоков: слева – до разбиения, справа – после

Данное преобразование не является оптимизирующим само по себе, но позволяет оптимизациям, основанным на анализе потока данных, таким, как удаление загрузок, удаление границ массивов, замещение на стеке и пр., работать более эффективно.

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 две 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.

Вынос редко исполняемого кода в отдельную функцию [5] повышает эффективность программы за счет более эффективного использования кэша процессора.

На тестах SQLite, Expedite, Cray и Coremark оптимизация дала средний прирост скорости в 0,8%. При использовании ее вместе с оптимизацией открытой вставки функций 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 [6]. 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. 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 методы. 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.

Во время тестирования был отмечен прирост производительности в ~5% при использовании только статического анализа и до ~7% с использованием спекулятивного алгоритма. Тестирование производилось на программе Clucene совместно с оптимизацией вставки функций. Алгоритм успешно проходит синтетические тесты девиртуализации, предложенные сообществом GCC.

3.5. Формирование суперблоков

Классические оптимизации используют статические методы анализа, такие как анализ времени жизни переменных или анализ достигающих определений для обеспечения корректности преобразований кода. Эти методы не различают часто и редко исполняемые пути. Однако часто бывают случаи, когда значение портится на редко исполняемом пути, который существует, например, для обработки событий. В результате невозможно применить оптимизации к часто исполняемым путям, пока редко исполняемые пути не будут исключены из анализа. Это требует точной оценки поведения программы во время исполнения.

Рассмотрим взвешенный граф потока управления, который получился после сбора профиля от отработавшей программы.

Весом ребра является вероятность перехода от одного блока к другому. Для формирования суперблоков мы последовательно находим новую трассу, т.е. последовательность базовых блоков, исполненную чаще других, и копируем хвост трассы для каждого блока, через который можно покинуть трассу. Поиск длится до тех пор, пока он возможен (суперблоки из одного базового блока не имеют смысла).

3.6. Ускорение оптимизаций времени связывания

Распараллеливание этапа LTO (оптимизаций времени компоновки) позволяет существенно сократить общее время компиляции программы, а также уменьшить требования к объему оперативной памяти, что существенно в случае сборки больших программных проектов. На данный момент использование простой эвристики – разбиение на задания с приблизительно равными по размеру наборами модулей биткода – позволило добиться сокращения времени компиляции до 60% (тест Expedite) при использовании 4 потоков, с сохранением производительности выходного приложения. Для малых тестов, таких как Coremark и Cray, сокращение времени сборки находится в пределах погрешности, изменение производительности составляет «-11%» и 12% процентов соответственно.

4. Сервер приложений

Предлагаемый метод двухфазной 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. 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 полно 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.

4.1. Описание механизма применения на сервере приложений профиля, собранного на клиенте

Были реализованы специальные системные сервисы, позволяющие пересылать клиентские данные о профиле выполнения программы на сервер приложений. Клиентская программа осуществляет мониторинг указанной директории и, как только в ней появляются файлы с профилем, производит их пересылку на сервер приложений. Сервер получает данные файлы и перекомпилирует приложение, применяя оптимизации с учетом профиля выполнения программы. После этого оптимизированное приложение пересылается клиенту (рисунок 3).

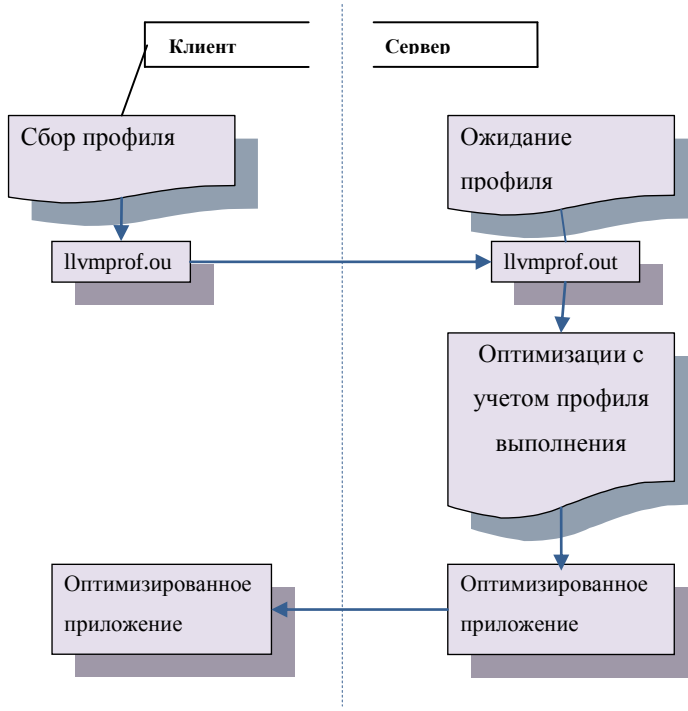

Рис. 3. Схема обмена профилем между клиентом и сервером.

4.2. Описание протокола пересылки файлов

Для реализации протокола используется технология Protocol Buffers [8]. Основное сообщение протокола называется "ProfileData" и содержит информацию об уникальном идентификаторе приложения и о каждом модуле приложения (рисунок 3) . Информация о модуле приложения включает в себя имя файла и данные о профиле выполнения. Поскольку приложения могут состоять из набора модулей, запускаемых в разное время, количество соответствующих полей в сообщении может быть различным.

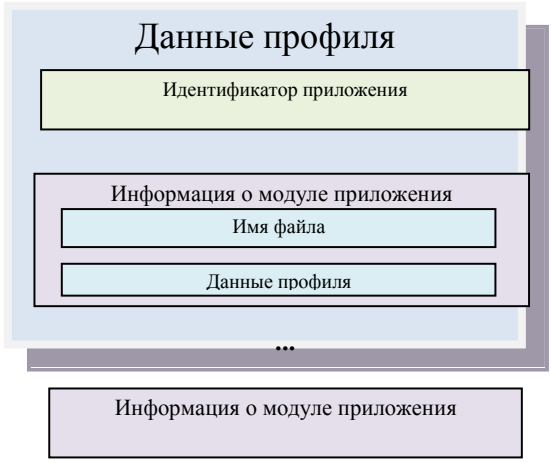

Рис. 4. Описание формата сообщения "Данные профиля"

5. Заключение

В данной статье рассмотрена система двухэтапной компиляции, реализованная на базе LLVM. Описаны изменения, внесенные в компоненты LLVM. Были предложены оптимизации, учитывающие профиль программы. На основе приложенных методов и разработанных технологий можно создать облачное хранилище позволяющее обеспечить как переносимость программ в пределах одной архитектуры, так и учет специфики конкретной аппаратуры, на которой производится развертывание программы.

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. - №22. DOI: 10.15514/ISPRAS-2012-22-1.
[3] Clang: a C language family frontend for LLVM, http://clang.llvm.org/
[4] GCC Free software foundation, http://gcc.gnu.org
[5] Peng Zhao "Code and Data Outlining", 2005.
[6] David F. Bacon and Peter F. Sweeny, Fast Static Analysis of C++ Virtual Functuion Call, 1996
[7] Справочное руководство по процессорной архитектуре ARM., http://infocenter.arm.com
[8] Protocol Buffers. URL: https://developers.google.com/protocol-buffers/

[9] 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.

Applying two-stage LLVM-based compilation approach to application deployment via cloud storage

Sergey Gaissaryan <ssg@ispras.ru> ISP RAS, Moscow, Russia
Shamil Kurmangaleev <kursh@ispras.ru> ISP RAS, Moscow, Russia
Kseniya Dolgorukova <unerkannt@ispras.ru> ISP RAS, Moscow, Russia
Valery Savchenko <sinmipt@ispras.ru> ISP RAS, Moscow, Russia
Sevak Sargsyan <sevaksargsyan@ispras.ru> ISP RAS, Moscow, Russia

Abstract. The process of software distribution via the app stores is as follows: software developer sends application to the online store and it becomes available to the user. To remain competitive software product for a long time, the developer needs to ensure the functioning of the application on most hardware and software platforms. Typically, this problem is solved as follows: an application either compiled and optimized developer repeatedly for every new platform, followed by the addition of a new shop in binary versions, or implemented using a dynamic language.

The paper describes two-stage compilation approach for C/C++ languages that allows deploying application in the LLVM (low level virtual machine) intermediate representation. The LLVM modifications for optimizing code generation time as well as the profile-based optimizations are presented. Also approach is presented and evaluated to dividing the linking process on parallel threads to reduce compile time for large applications.

The specialized application cloud storage architecture is also suggested. We describe implementation of the mechanism for applying profile information, collected on client side on the application server. Also the profile exchange protocol and the specification of message data format are described.

Keywords: Two-stage compilation, program 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. DOI: 10.15514/ISPRAS-2012-22-1. (in Russian).
[3]. Clang: a C language family frontend for LLVM, http://clang.llvm.org/
[4]. GCC Free software foundation, http://gcc.gnu.org
[5]. Peng Zhao “Code and Data Outlining”, 2005.
[6]. David F. Bacon and Peter F. Sweeny, Fast Static Analysis of C++ Virtual Functuion Call, 1996
[7]. ARM architecture., http://infocenter.arm.com
[8]. Protocol Buffers. URL: https://developers.google.com/protocol-buffers/
[9]. Sh.F. Kurmangaleev. 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], ISP RAN [The Proceedings of ISP RAS], 2013, vol. 24, pp. 127-144, 2013. DOI: 10.15514/ISPRAS-2013-24-7. (in Russian).