Разработка и реализация метода масштабирования по памяти для систем межмодульных оптимизаций и статического анализа на основе LLVM

Долгорукова К.Ю. <unerkannt@ispras.ru>
Институт системного программирования РАН,
109004, Россия, г. Москва, ул. А. Солженицына, дом 25

Аннотация. Проблема масштабируемости систем оптимизации времени связывания и систем статического анализа не потеряла своей актуальности в настоящее время: несмотря на рост производительности и увеличение объема памяти компьютеров, программы растут в размерах и сложности пропорэ иона льно, особенно когда дело касается таких сложных многомодульных программ, как, например, операционные системы, браузеры и другие. Эффективная оптимизация таких программ с использованием таких мощных инструментов, как межпроцедурные оптимизирующие преобразования, проводимые во время связывания, и преобразования с использованием профиля исполнения программы, требует существенных вычислительных ресурсов. В статье рассматривается подход к масштабированию по памяти системы оптимизаций времени связывания в целях ограничения потреб ляемЙ й памяти заданным пороговым значением. Предложенный метод включает в себя следующие этапы: аннотирование промежуточного представления на этапе генерации промежуточного кода; во время компоновки чтение объявлений и аннотаций из файлов с промежуточным представлением, предварительный анализ, в котором происходит построение и анализ графа вызовов, отложенную загрузку участков кода во время оптимизаций и выгрузку участков кода по требованию. Также предложен подход к применению масштабирования по памяти к системе статического анали за. ОЙ исанный метод масштабирования был реализован на основе инструмента GOLD-plugin системы LLVM[1]. Представленные предварительные результаты тестирования реализации данного подхода на тестах SPEC CPU2000[2] показывают увеличение размера кода на 6%, увеличение накладных расходов по времени на 0.2% и по памяти на 36%.Ключевые слова: межмодульные анализ и оптимизации; системы межмодульных оптимизаций; масштабирование.

DOI: 10.15514/ISPRAS-2015-27(6)-7

Для цитирования: Долгорукова К.Ю. Разработка и реализация метода масштабирования по памяти для систем межмодульных оптимизаций и статического анализа на основе LLVM. Труды ИСП РАН, том 27, вып. 6, 2015 г., стр. 97-110. DOI: 10.15514/ISPRAS-2015-27(6)-7.

1. Введение

Традиционный метод сборки программ из исходного кода состоит из двух этапов: компиляции и связывания. Обычно все модули программы компилируются отдельно и независимо в объектные файлы, которые потом связываются компоновщиком. В случае раздельной независимой оптимизации у компилятора нет программного кода других модулей, поэтому эффективность межпроцедурных оптимизаций ограничена одним файлом с исходным кодом. Если же есть возможность оптимизировать все входящие в программу модули вместе, эффективность этих оптимизаций значительно возрастает. Оптимизации, проводимые на всей программе целиком, называются межмодульными оптимизациями.

На втором этапе сборки компоновщик получает все скомпилированные файлы программы и файлы библиотек, разрешает коллизии и зависимости между ними и строит исполняемый файл. Практика показала, что целесообразно проводить межмодульные оптимизации на этапе связывания, так как именно на этом этапе системе сборки доступна вся программа целиком: для этого в объектные файлы программ, полученные на этапе компиляции, добавляется некоторая информация о программе, достаточная, чтобы построить промежуточное представление программы, используемое компилятором. Такие системы называются системами оптимизаций времени связывания.

Межмодульные оптимизирующие преобразования, проводимые над всеми модулями программы, всегда связаны с построением промежуточного представления для всей программы. В случае больших приложений этот процесс может потребовать огромных ресурсов. Для сборки с оптимизациями времени связывания таких программ, как операционные системы или интернет-браузеры, состоящих из нескольких тысяч файлов исходного кода, может требоваться память, которой не обладают не только обычные настольные компьютеры, − но и далеко не все серверные архитектуры способны собрать такие программы без задействования отгрузки на диск.

Методы регулирования потребления ресурсов системой для различных архитектур называются масштабированием системы.

Масштабирование может проводиться как по времени, так и по памяти, ускоряя процесс сборки – или уменьшая количество потребляемой памяти. Ускорение сборки может проводиться как посредством ручной оптимизации существующих решений, так и посредством распараллеливания для запуска сборки на нескольких ядрах. Уменьшить количество потребляемой памяти можно также с помощью ручной оптимизации, то есть посредством поиска ошибок в использовании памяти или усовершенствования алгоритмов, либо с помощью механизмов управления потреблением ресурсов.

Существует несколько примеров компиляторных инструментов, которые дают возможность так или иначе управлять потреблением ресурсов во время проведения оптимизаций времени связывания, в том числе GCC[3], компиляторы HP[4,5], Google’s LIPO[6] и другие; подробно принципы их

работы описаны в статье [7]. Для большинства из них применяется одна схема работы, включающая в себя 3 этапа: генерацию промежуточного кода (ГПК), стадию межпроцедурного анализа и стадию генерацию машинного кода (ГМК).

Во время генерации промежуточного кода компилятор получает файлы с кодом на исходном языке программирования, проводит небольшое количество локальных оптимизаций в каждом файле отдельно, а также генерирует дополнительную информацию о зависимостях между функциями в так называемые аннотации, представляющие собой некую информацию о программе, необходимую для проведения межпроцедурного анализа. В результате работы первого этапа генерируется расширенный файл в промежуточном представлении (либо в двоичном формате), содержащий также дополнительную информацию. Обычно этап ГПК легко параллелизуем.

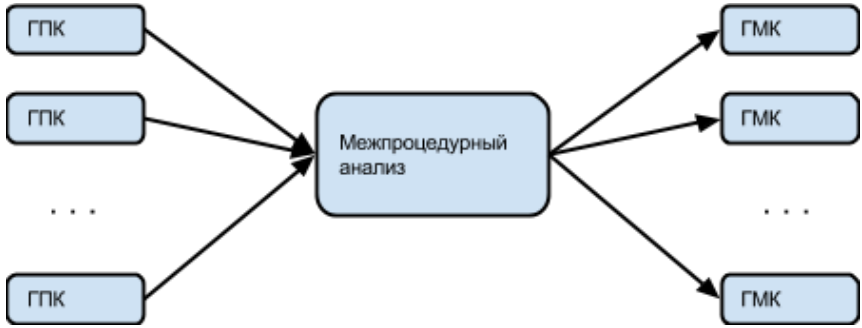

Рис. 1. Схема работы систем оптимизаций времени связывания

Второй этап, как правило, проходит во время связывания и читает файлы, полученные на первом этапе, анализирует межпроцедурные зависимости, производит некоторые оптимизирующие преобразования, и снова генерирует файлы в промежуточном представлении и некоторое количество дополнительной информации. Этот процесс трудно параллелизуем, и все компиляторные системы с открытым исходным кодом выполняют межмодульный анализ в один поток, поэтому именно анализ – узкое место этих систем.

Последняя фаза принимает файлы, сгенерированные на втором этапе и генерирует из них объектный код, который затем подается ассемблеру. Иногда на этой стадии проводятся дополнительные оптимизации. Эта фаза у разных компиляторов может быть параллелизуемой или непараллелизуемой, и в результате может получаться либо один объектный файл, либо несколько, которые впоследствии передаются стандартному системному компоновщику.

2. Особенности сборки программ в системе LLVM

Система LLVM – это набор библиотек и компиляторных утилит для анализа и оптимизации кода. В основе LLVM лежит его характерное промежуточное представление – похожий на ассемблер язык с трёхадресными инструкциями, находящимися в SSA-форме. Файлы, содержащие код на языке LLVM, называются биткодом.

Когда пользователь собирает программу на языке Си или Си++ утилитами LLVM с оптимизациями времени связывания, он должен вызвать компилятор CLANG с опцией -flto, например, так:

clang -flto file1.c -file2.c … -fileN.c -o program.exe

В этом вызове прозрачно для пользователя происходит следующее: сначала каждый файл программы транслируются компиляторами CLANG в промежуточное представление LLVM и сохраняется в файлы с биткодом, затем вызывается компоновщик LD-GOLD[8], который считывает сгенерированные посредством CLANG файлы с биткодом и передает их на обработку плагину llvm-gold-plugin.

Каждый считанный плагином файл преобразуется в единицу абстракции промежуточного представления, называемую модулем. Модуль (Module) состоит из набора объявлений и определений глобальных переменных, тел функций, констант, набора синонимов и дополнительной информации, необходимой для некоторых оптимизаций, или мета-данных (MetaData). Функции, в свою очередь, состоят из базовых блоков, а последние, в свою очередь, – из инструкций.

Библиотека оптимизаций времени связывания (LTO – Link-Time Optimization) позволяет создавать так называемый LTO-модуль – структуру, в которую происходит чтение символов файла. Для запуска оптимизаций времени связывания также необходим объект генератора кода (LTO Code Generator), который производит анализ и оптимизации скомпонованного модуля, а затем генерирует объектный код.

Во время сборки программы компоновщик передает их по одному на обработку плагину llvm-gold. Плагин создает для текущего файла объект LTO-модуль, читает и переводит во внутреннее представление содержимое файлов, затем передает LTO-модуль компоновщику модулей, а тот, в свою очередь, компонует его в хранящийся в памяти композитный модуль, разрешая коллизии типов и составляя связи с новым участком кода. После того как все файлы оказываются обработаны, компоновщик посылает плагину запрос на генерацию кода. Тогда генератор кода проверяет композитный модуль на правильность, запускает на нем последовательно анализ и оптимизации, и, наконец, генерирует объектный код.

Анализ и оптимизации в LLVM реализованы в виде проходов компилятора. Проходы разделены соответственно абстракциям промежуточного представления, на которых они могут производиться независимо. Существуют

проходы, исполняемые на модуле, на функции, на базовых блоках. Помимо этого, есть проходы, выполняемые на сгенерированных структурах: на графе вызовов функций, на циклах. Также есть постоянные проходы, которые исполняются в самом начале работы компилятора на всём модуле. Для управления порядком запуска компиляторных проходов в LLVM применяется стек менеджеров проходов, каждый из которых выстраивает порядок пассов согласно зависимостям между ними. В стеке менеджеры разделяются по типу проходов, которыми они управляют, и для каждого уровня абстракции есть свой менеджер. Сами менеджеры также являются проходами. Когда компилятор запускает оптимизации, он начинает оптимизировать с самой верхней абстракции – модуля. Таким образом, сначала запускаются модульные проходы, среди которых есть менеджер следующих абстракций – функций и сильносвязных компонент графа. Менеджеры компонент графа, в свою очередь, запускают проходы, работающие на сильносвязных компонентах, а менеджеры функций – проходы, работающие на телах функций, а также менеджеры циклов, – и так далее вплоть до инструкций.

При работе оптимизаций всегда неявно подразумевается, что модуль полностью сформирован в памяти, в нем присутствуют все необходимые объявления и тела функций, разрешены все коллизии, сформированы все зависимости. Таким образом, пиком потребления памяти при сборке программы с помощью утилит LLVM будет именно стадия оптимизаций.

В задачу масштабирования, таким образом, входят:

- разделение фазы анализа и фазы оптимизаций;
- разработка метода распараллеливания межпроцедурных оптимизаций;
- разработка метода ограничения потребляемой памяти с учетом желания пользователя или согласно ограничениям оборудования, на котором производится сборка программы.

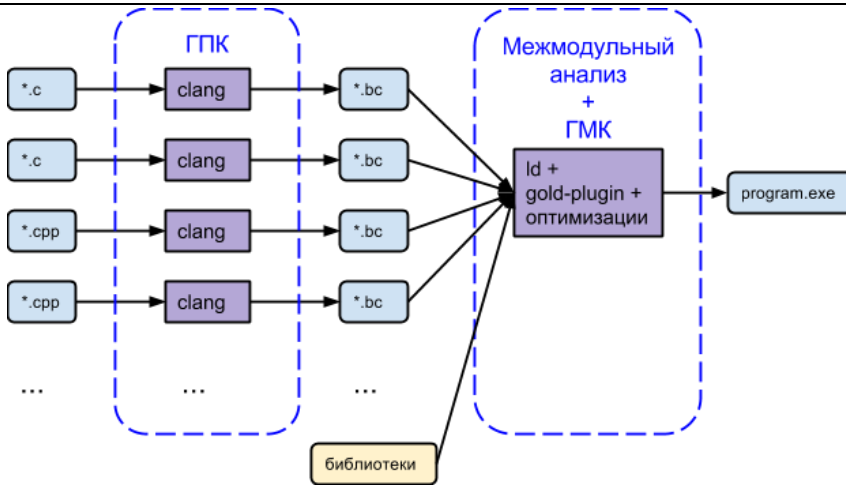

Рис. 2. Схема сборки программы с помощью утилит LLVM

3. Управление потреблением памяти

Существует несколько основных подходов к регулированию потребления памяти, реализованных в разных компиляторах [3]. Один из них, предложенный разработчиками HP в системе HLO (High-Level Optimization) для компилятора HP-UX, использует диспетчеризацию ресурсов, в которой отслеживается доступ к участкам кода и потребление памяти. При достижении указанного пользователем порога занимаемой памяти компилятор начинает отгружать давно неиспользуемые участки промежуточного представления кода на диск. Когда компилятор пытается получить доступ к отгруженным участкам кода, диспетчер подгружает их в память. Данный подход позволяет выполнять оптимизации с произвольным ограничением ресурсов, но платой за такую гибкость будет время: выгрузка кода на жесткий диск – очень тяжеловесный процесс, требующий доступа к более медленной памяти, а также операций архивирования и разархивирования кода.

Другой подход предлагают разработчики GCC в рамках проекта WHOPR (WHOle-PRogram optimization – оптимизация программы целиком). Межмодульные анализ и оптимизации строго разделены на разные стадии, причем, стадия оптимизаций может быть выполнена в несколько потоков. Для того, чтобы иметь возможность выполнять оптимизации параллельно, на стадии анализа обеспечивается независимость файлов друг от друга при возможности использовать информацию из других модулей.

3.1 Предлагаемый подход к управлению памятью для систем межмодульных оптимизаций

Авторами статьи был учтен опыт существующих решений, и предложен комбинированный подход к управлению использованием ресурсов. Наш метод условно разделяет процесс сборки на две стадии:

1. Загрузка информации о глобальных структурах и данных для всех компонуемых модулей: данные о типах, описание функций, описание типов, глобальные переменные. На основе этих данных можно проводить некоторый легковесный анализ, например, строить граф вызовов. При этом доступ к коду функций на данной стадии ограничен только глобальной метаинформацией, поставляемой с кодом. К самим телам функций доступа нет.
2. Загрузка кода тел функций и оптимизирующие преобразования над ним. Данный этап допускает любые оптимизирующие преобразования. Загрузка функций осуществляется последовательно, по мере запроса алгоритмов оптимизации. Диспетчер памяти следит за потреблением ресурсов, и, когда размер занятой памяти переходит заданный пользователем порог, отгружает неиспользуемый код на диск.

Таким образом, с одной стороны, достигается гибкость в управлении памятью, а с другой, представляется возможность производить часть анализа программы вовсе без загрузки кода в память.

3.2 Предлагаемый подход к управлению памятью для систем статического анализа

Для статического анализа кода был предложен несколько иной подход, основанный на обходе графа. Для этого предполагается использовать граф вызовов в качестве структуры для задания порядка обхода кода программ. Для того, чтобы иметь граф вызовов еще до момента загрузки кода, необходимо на этапе генерации промежуточного кода встраивать метаинформацию о вызовах функций. Получить доступ к графу вызовов можно с помощью интерфейса прикладного программирования (API – application programming interface). По мере завершения работы над анализом функции, код функции можно выгрузить, вызвав соответствующую функцию. Таким образом, интерфейс прикладного программирования для статического анализа представлен методами:

- получения доступа к графу вызовов;
- загрузки тела функции;
- выгрузки тела функции.

4. Реализация метода ленивой загрузки в системе LLVM

Чтобы иметь возможность ограничить потребление ресурсов компоновщиком, необходимы ключевые изменения в стратегии оптимизации, чтения и компоновки файлов с биткодом в модуль промежуточного представления. В общем случае необходимо отделение стадии анализа от стадии оптимизаций, а также реализация анализа, способного работать лишь на описаниях кода, не имея доступа к самому коду компилируемой программы. Также необходим механизм разделения файлов с биткодом на группы, которые можно оптимизировать независимо, без доступа к телам функций.

Для реализации ограничения потребления ресурсов был внедрен механизм загрузки тел функций в память по запросу. Для этого в механизме чтения биткода из файла были реализованы пропуски тел функций, а компоновщик модулей допускал подобные пропуски, помечая пропущенные функции, как «встреченные». Тел у встреченных функций нет, но они отличаются от объявлений тем, что могут быть загружены в любой момент по требованию какой-либо из оптимизаций. Также смещения тел встреченных функций сохраняются в специальном журнале для более быстрого доступа к ним при повторном чтении файла.

Таким образом, загрузка файла с биткодом состоит из двух этапов: загрузка объявлений, глобальных переменных и констант с предварительной их компоновкой в композитный модуль, и загрузка тел функций с окончательной компоновкой.

Специально для возможности загрузки тела функции из очереди оптимизирующих проходов был реализован дополнительный проход ленивой загрузки. Каждый оптимизирующий или анализирующий компиляторный проход, требующий доступа к телам функций, должен запросить при инициализации проход-загрузчик, чтобы менеджер проходов инициировал ленивую загрузку раньше, чем запуск прохода, требующего тело функции.

Когда проход-загрузчик инициирует загрузку тела функции, в работу вступает менеджер ленивой загрузки, который достает из журнала информацию о функции, такую как: файл, где лежит тело функции, смещение, размер кода и таблицы глобальных значений и типов исходного модуля, которые также были сохранены во время предварительной загрузки. Затем менеджер запускает загрузчик кода, который считывает участок файла с кодом в буфер и преобразует его в промежуточное представление. После этого менеджер ленивой загрузки вновь запускает компоновщик, который разрешает и уточняет типы загруженных переменных и копирует их в композитный модуль.

После всех этих действий код функции готов к оптимизациям.

5. Реализация легковесного построения графа вызовов

В библиотеке LLVM есть реализация графа вызовов, широко используемая в оптимизациях, но использовать её вместе с ленивой загрузкой оказалось невозможно: эта реализация тесно привязана к инструкциям вызова, что, во-первых, не нужно при анализе зависимостей между файлами, а во-вторых, невозможно обеспечить в отсутствие кода тел функций. Чтобы всё-таки получить граф вызовов без доступа к телам функций, в механизме записи биткода в файлы, работающего на этапе генерации промежуточного представления, были реализованы запись информации о вызовах перед телами функций. Для чтения информации были произведены соответствующие изменения в механизме чтения биткода. Построенный таким образом граф вызовов содержит информацию о факте вызова функции, но не предоставляет информации, например, о параметрах вызова. Тем не менее, имея такой граф, возможно проанализировать связи между модулями, а также строить ленивый диспетчер оптимизаций, итерирующийся не последовательно по функциям, а методом обхода вглубь дерева вызовов. Этот диспетчер можно использовать как при оптимизациях в условиях нехватки ресурсов, так и для ресурсоемкого анализа, например, статического анализа с поиском ошибок или анализа сущностей и связей программ для понимания кода [8].

Рисунки 3 и 4 показывает сравнение двух графов одной программы: первый получен в результате обычного построения графа через обход инструкций вызова, а второй построен посредством аннотаций без доступа к инструкциям. Как можно видеть из рисунка, графы различаются наличием источника и стока, рёбра от и к которым добавляются автоматически всем узлам при построении графа в LLVM.

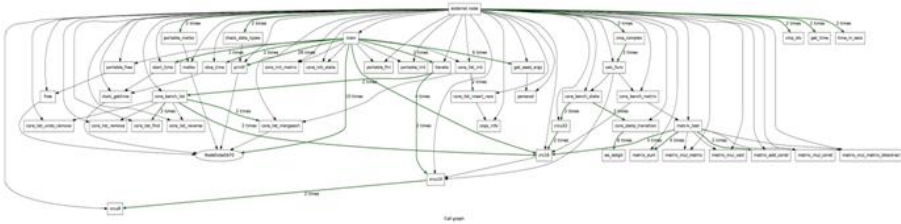

Рис. 3. Граф вызовов, построенный стандартным проходом построения графа вызовов в LLVM

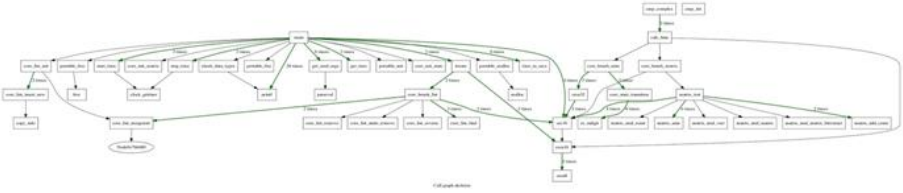

Рис. 4. Граф вызовов, построенный только с помощью аннотаций

6. Результаты работы

В ходе работы были разработаны и реализованы:

- метод ленивой загрузки кода в два этапа: предварительное чтение и чтение тел функций;
- метод выгрузки функций;
- метод легковесного получения графа вызовов без доступа к инструкциям;
- интерфейс прикладного программирования для систем статического анализа.

Существующее решение было протестировано на некоторых тестах SPEC CPU2000 на архитектуре x86_64 Intel Core i5-2500 с 32ГБ оперативной памяти. Таблица 1 показывает, насколько увеличивается объем, занимаемый файлами с промежуточным представлением, когда в него записываются аннотации.

Табл. 1. Увеличение объема биткода от аннотаций

Название теста	**Размер файлов с биткодом в байтах при обычной сборке**	**Размер файлов с биткодом в байтах при сборке с аннотациями**	**Увеличение размера кода в процентах**
gcc	2797940	2970212	6.16
gzip	89156	96808	8.58
vpr	303208	318008	4.88
mcf	43863	45656	4.09

crafty	571744	583700	2.09
parser	312748	377888	20.83
bzip2	86516	92292	6.68
Суммарное значение	**4205175**	**4484564**	**6.64**

Исходя из результатов таблицы 1, размер биткода увеличился в среднем на 6,6%. При этом размер аннотаций зависит от количества функций и вызовов в программе. Для программ с большим количеством маленьких функций относительный объем аннотаций может достигать 20%, как это видно на тесте parser, тогда как для программы с несколькими крупными функциями этот показатель составляет всего 2%.

Табл. 2. Временные издержки на ленивую загрузку функций без выгрузки.

Название теста	**Время сборки в секундах**	**Время ленивой сборки в секундах**	**Увеличение времени в процентах**
gzip	0.2993	0.3001	0.27
vpr	0.9942	0.996	0.18
gcc	13.0642	13.0932	0.22
mcf	0.1164	0.1174	0.86
crafty	1.4256	1.4185	-0.50
parser	1.0047	1.0176	1.28
bzip2	0.3827	0.385	0.60
Суммарное значение	**17.29**	**17.33**	**0.24**

Время сборки программ, как видно из таблицы 2, от ленивой загрузки, отложенной до момента оптимизаций, практически не страдает, и увеличивается всего на 0,2%. Тестирование сборки проводилось по 10 раз для каждого теста и было усреднено.

Табл. 3. Накладные расходы на ленивую загрузку

Название теста	**Пик потребления при обычной сборке, Кб**	**Пик потребления при ленивой сборке, Кб**	**Накладные расходы, %**	**Среднее потребление при обычной сборке, Кб**	**Среднее потребление при ленивой сборке, Кб**	**Накладные расходы, %**
gzip	324	404	24.69	177	269	51.98
vpr	18720	21652	15.66	4484	4765	6.27
gcc	168992	212468	25.73	94811	132811	40.08
mcf	n/a	n/a	n/a	n/a	n/a	n/a
crafty	23328	28700	23.03	6287	7319	16.41
parser	20296	25264	24.48	4924	5742	16.61
bzip2	n/a	n/a	n/a	n/a	n/a	n/a
Сум-марное значение	**231660**	**288488**	**24.53**	**110683**	**150906**	**36.34**

Таблица 3 показывает накладные расходы по памяти на поддержание журнала для ленивой загрузки. Тест проводился при выключенной отгрузке кода функций, таким образом, чтобы замерить объемы журнала. Тестирование проводилось с помощью утилиты vmstat с частотой 1 с. При учете, что большинство программ компоновались всего несколько секунд, тестирование не исключает погрешность. Мало того, для двух тестов не удалось получить данных ввиду очень малого времени сборки.

7. Заключение

В результате работы была получен работоспособный прототип системы, на основе которого возможно разрабатывать межмодульные анализирующие и оптимизирующие проходы, работающие только на аннотациях, а также строить системы статического анализа. Тем не менее, экспериментальные данные показали, что необходим менеджер ресурсов, который смог бы отслеживать необходимости выгрузки функций по достижению

определенного пользователем порога потребления памяти. Также планируется оптимизировать способ хранения структур в журнале с целью уменьшения накладных расходов.

Список литературы

[1]. The LLVM Compiler Infrastructure. http://LLVM.org/
[2]. SPEC CPU benchmark. https://www.spec.org/cpu2000/
[3]. Preston Briggs, Doug Evans, Brian Grant, Robert Hundt, William Maddox, Diego Novillo, Seongbae Park, David Sehr, Ian Taylor, Ollie Wild. WHOPR - Fast and Scalable Whole Program Optimizations in GCC. Initial Draft, 12 Dec 2007.
[4]. Andrew Ayers, Stuart de Jong, John Peyton, and Richard Schooler. Scalable crossmodule optimization. SIGPLAN Not., 33(5):301–312, 1998. ISSN 0362-1340. doi:http://doi.acm.org/10.1145/277652.277745.
[5]. Sungdo Moon, Xinliang D. Li, Robert Hundt, Dhruva R. Chakrabarti, Luis A. Lozano, Uma Srinivasan, and Shin-Ming Liu. SYZYGY - a framework for scalable cross-module IPO. In CGO '04: Proceedings of the international symposium on Code generation and optimization, page 65, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2102-9
[6]. Xinliang David Li, Raksit Ashok, Robert Hundt. Lightweight Feedback-Directed CrossModule Optimization. CGO'10, April 24–28, 2010, Toronto, Ontario, Canada. ACM 978-1-60558-635-9/10/04.
[7]. К.Ю. Долгорукова. Обзор масштабируемых систем межмодульных оптимизаций. Труды Института системного программирования РАН Том 26, выпуск 3, 2014, стр. 69-90. ISSN 2220-6426 (Online), ISSN 2079-8156 (Print). DOI: 10.15514/ISPRAS-2014-26(3)-3
[8]. А.А. Белеванцев, Е.А. Велесевич. Анализ сущностей программ на языках Си/Си++ и связей между ними для понимания программ. Труды Института системного программирования РАН. Том 27, выпуск 2, 2015, стр. 53-64. ISSN 2220-6426 (Online), ISSN 2079-8156 (Print). DOI: 10.15514/ISPRAS-2015-27(2)-4
[9]. Пакет GNU Binutils. http://www.gnu.org/software/binutils/

Implementation of Memory Scalability Approach for LLVM-Based Link-Time Optimization and Static Analyzing Systems

Ksenia Dolgorukova <unerkannt@ispras.ru>
Institute for System Programming of the RAS,
25 Alexander Solzhenitsyn Str., Moscow, 109004, Russian Federation

Abstract. Link-time optimization and static analyzing systems scalability problem is of current importance: in spite of growth of performance and memory volume of modern computers programs grow in size and complexity as much. In particular, this is actual for such complex and large programs as browsers, operating systems, etc. To improve performance of these programs as much as possible, there are several aggressive optimising

techniques like interprocedural optim ization a nd profile-guided optimization. These techniques applied to large programs claim for large memory and need a lot of time to be performed. This paper introduces memory scalability approach for link-time optimization system and proposes technique for applying this approach to static analyzing systems. The approach involve several steps: adding a summary information to intermediate representation at compile time, reading declarations and summaries from IR files, analysing summary and computing call graph at special pre-analysis phase, lazy code loading during optimization phase and code unloading on demand. Proposed approach was implemented as the linking tool based on the LLVM GOLD-plugin. The tool was tested on SPEC CPU2000 benchmark suite. Preliminary results show increasing of average intermediate code on 6%, increasing of average time on 0.2% and increasing of total memory usage to 36%.

Keywords: link-time optimization; cross-module optimization systems; scalability.

DOI: 10.15514/ISPRAS-2015-27(6)-7

For citation: Dolgorukova Ksenia. Implementation of Memory Scalability Approach for LLVM-Based Link-Time Optimization and Static Analyzing Systems. *Trudy ISP RAN/Proc. ISP RAS*, vol. 27, issue 6, 2015, pp. 173-192 (in Russian). DOI: 10.15514/ISPRAS-2015-27(6)-7.

References

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

[2]. SPEC CPU benchmark. https://www.spec.org/cpu2000/

[3]. Preston Briggs, Doug Evans, Brian Grant, Robert Hundt, William Maddox, Diego Novillo, Seongbae Park, David Sehr, Ian Taylor, Ollie Wild. WHOPR - Fast and Scalable Whole Program Optimizations in GCC. Initial Draft, 12 Dec 2007.

[4]. Andrew Ayers, Stuart de Jong, John Peyton, and Richard Schooler. Scalable crossmodule optimization. SIGPLAN Not., 33(5):301–312, 1998. ISSN 0362-1340. doi:http://doi.acm.org/10.1145/277652.277745.

[5]. Sungdo Moon, Xinliang D. Li, Robert Hundt, Dhruva R. Chakrabarti, Luis A. Lozano, Uma Srinivasan, and Shin-Ming Liu. SYZYGY - a framework for scalable cross-module IPO. In CGO '04: Proceedings of the international symposium on Code generation and optimization, page 65, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2102-9

[6]. Xinliang David Li, Raksit Ashok, Robert Hundt. Lightweight Feedback-Directed CrossModule Optimization. CGO'10, April 24–28, 2010, Toronto, Ontario, Canada. ACM 978-1-60558-635-9/10/04.

[7]. Ksenia Dolgorukova. Obzor masshtabiruemyh sistem mezhmodul'nyh optimizacij [Overview of Scalable Frameworks of Cross-Module Optimization]. Trudy ISP RAN [The Proceedings of ISP RAS], vol. 26, issue 3, 2014, pp. 69-90. ISSN 2220-6426 (Online), ISSN 2079-8156 (Print). DOI: 10.15514/ISPRAS-2014-26(3)-3. (in Russian)

[8]. A. Belevantsev, E. Velesevich. Analiz sushhnostej programm na jazykah C/C++ i svjazej mezhdu nimi dlja ponimanija programm [Analyzing C/C++ code entities and relations for program understanding]. Trudy ISP RAN [The Proceedings of ISP RAS], vol. 27, issue 2, 2015, pp. 53-64. ISSN 2220-6426 (Online), ISSN 2079-8156 (Print). DOI: 10.15514/ISPRAS-2015-27(2)-4. (in Russian)

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