Министерство образования и науки Российской Федерации
МОСКОВСКИЙ ФИЗИКО-ТЕХНИЧЕСКИЙ ИНСТИТУТ
(государственный университет)
КАФЕДРА ИНФОРМАТИКИ

Оптимизация линковки проектов, написанных на C/C++

Выпускная квалификационная работа
(магистерская работа)

Направление подготовки: 03.04.01 Прикладная математика и физика

Выполнил:
студент 6 курса 073 группы
Татунов Кирилл Юрьевич

Научный руководитель:
к.ф.-м.н.
Чуканова Ольга Владимировна

Москва, 2016

Содержание

1 Введение

Начнем с того, что определим, на что направлено данное исследование. Данное исследование направлено в основном на проекты, которые собираются в автоматическом режиме на регулярной основе. Такого рода проекты, над которыми трудится большой коллектив программистов, имеют тенденцию [1] пересобирать после каждого коммита в систему контроля версий для раннего нахождения возможных дефектов и багов. Такая пересборка является занимает время, которое в-первых ждёт программист для получения результата от автоматической системы сборки, во-вторых это процессорное время, будь оно на собственном кластере компании, ведущей разработки програмного продукта, или в облаке, стоит определённых денег.

Программисты, когда составляют правила для используемой системы сборки, будь то Make, CMake, qmake или что-то другое, не всегда способны проследить, что зависимости, которые они указывают, действительно являются зависимостями после каждого коммита, который они делают. Возможно зависимость требовалась несколько коммитов назад, до того как программист убрал соответствующую зависимость из исходного кода, но не потрудился убрать её из системы сборки.

Подобного рода проблемы требуют координации со стороны системы сборки (требуется ответ на вопрос в духе «считает ли CMake, что библиотека A зависит от библиотеки B?»), и в общем случае рашение будет своё для каждой системы сборки, поэтому в этом исследовании мы сосредоточим своё внимание на решении, не зависящем от конкретной системы сборки.

В данной работе в основном пойдёт речь о удалении ненужных за-

висимостей, возникающих при линковке одной библиотеки к другой. В последующих главах мы определим несколько необходимых понятий, определим интересующий нас аспект линковки, рассмотрим, что такое «неоптимальная зависимость», предложим модель, на основе которой находить и решать неоптимальные зависимости и рассмотрим результаты на некоторых проектах с открытым исходным кодом.

2 Постановка задачи

Целью работы является сокращение времени линковки отдельных целей в проекте за счет удаления ненужных зависимостей. Нас интересует решение, не зависящее от конкретной системы сборки, потому что во-первых, такое решение должно быть своим для каждой системы сборки, а во-вторых понятие «зависимость» может быть довольно расплытым. Например, в то время как в системе сборки CMake существует возможность указания явных зависимостей при помощи `target_link_libraries`, там также существует возможность добавления собственных правил и комманд при помощи `add_custom_target` и `add_custom_command`.[1] Подобные возможности, которые позволяют идти в обход того, что система сборки могла бы выдать в качестве зависимости для изучения, существенно усложняют построение графа зависимостей.

Подобное требование вынуждает изучать зависимости в проекте на уровне линковщика. Современные линковщики, например ld, обладают обширным набором встроенных возможностей.(todo link) Среди них нас будет интересовать только основная задача линковщика: сборка объектных файлов, генерируемых компилятором, и библиотек в один файл, представляющий собой другую библиотеку, исполняемый файл или ещё один объектный файл.

Таким образом, если во время линковки обнаруживается, что исполняемый файл E собирается из объектных файлов o_1, o_2 и библиотеки L, то E зависит от o_1, o_2 и от L (здесь и далее объектные файлы обозначаются маленькими буквами, а библиотеки и исполняемые файлы,

[1] На момент написания на сайте github.com ~680000 результатов при поиске `target_link_libraries` и суммарно ~300000 при поиске `add_custom_target` и `add_custom_command`, то есть примерно половина от `target_link_libraries`.

уже полученные из линковщика — большими). Обозначаться подобная зависимость будет как

$$E \to \{o_1, o_2, L\}.$$

Некоторые библиотеки собираются только из объектных файлов. В этом случае, конечно, верно, что $L \to \{o_3, o_4, o_5\}$, но помимо этого мы будем обозначать подобную конструкцию как $L = [o_3, o_4, o_5]$. Квадратные скобки обозначают, что библиотека L была собрана из объектных файлов o_3, o_4 и o_5 или, иными словами, разбивается на соответствующие объектные файлы.

В проектах программисты не всегда следят за оптимальной расстановкой зависимостей. Нашей основной целью в этой работе будет поиск неоптимальных зависимостей следующего вида. Пусть E — исполняемый файл или библиотека, а A и B — библиотеки. Рассмотрим такую ситуацию:

$$E \to A \to B.$$

Если существует способ разбить библиотеку A на две библиотеки A_1 и A_2 такие, что $A = [A_1, A_2]$, и A_2 не зависит от A_1, то описанный пример можно будет переписать как

$$E \to [A_1, A_2] \to B.$$

Используя независимость A_2 от A_1 можно исключить лишнюю зависимость:

$$E \to A_1 \qquad A_2 \to B.$$

Исключение подобных зависимостей позволит сократить время сборки отдельных целей в проекте, но скорее всего не сократит время сборки всего проекта, потому что несмотря на то, что для сборки E теперь не

требуется собирать и линковать B, собрать B потребуется для сборки другой цели проекта, которая действительно зависит от B.

3 Модель

3.1 Описание модели

Вначале мы опишем, какой аспект линковки для нас важен. Линковщик, когда собирает файлы в другой файл, занимается разрешением символов. Нас интересует два типа символов: определённый и неопределённый. Определённый символ — функция или переменная, определённая в данном модуле и предоставляемая для использования другим модулям, в то время как неопределённый символ — функция или переменная, на которые ссылается модуль, но не определяет их внутри себя.

Определённые символы, как правило[2], определены только в одном файле, в то время как неопределённые символы могут использоваться в нескольких модулях. При этом модуль с каким-либо неопределённым символом будет зависеть от модуля, в котором он определён, потому что прежде чем собрать испольняемый файл, линкеру необходимо найти определение символа и поставить соответствующую “ссылку”, где искать этот символ.

На основе этих зависимостей мы будем строить направленный граф, который их описывает следующим образом. Допустим, есть библиотека A, зависящая от библиотеки B через какой-то символ: $A \to B$. Вместо того, чтобы следить за тем, как уже собранные библиотеки зависят друг от друга, мы будем разбивать библиотеки на объектные файлы, из которых они собраны. Каждый объектный файл будет представлен вершиной в нашем графе. Рёбра графа описывают зависимости между объектными

[2]Символы могут быть определены в нескольких файлах. Они либо не линкуются вместе, либо используются специальные опции линовщика, позволяющие перезаписывать уже определённые символы.

файлами: если объектный файл o_1 зависит от объектного файла o_2 через какой-либо символ, то в графе будет ребро, напраленное из вершины o_1 в вершину o_2.

$$o_1 \to o_2.$$

В проектах объектные файлы объединяются в библиотеки, что мы будем отражать объединением вершин в группы. Так, предыдущий пример может быть расширен следующим образом:

$$o_3 \to o_2, \quad o_4 \to o_5,$$
$$L_1 = [o_1, o_3, o_4], \quad L_2 = [o_2, o_5].$$

После введения групп на вершинах они будут выглядеть так:

$$[o_1, o_3, o_4] \to [o_2, o_5].$$

Таким образом символьные зависимости в проектах мы представляем в виде ориентированного графа с дополнительным объединением вершин в группы.[3]

Далее мы опишем как мы будем выделять неоптимальные зависимости в построенном графе.

[3]Эти группы могут быть вполне произвольными, например они могут пересекаться.

3.2 Выделение неоптимальных зависимостей.

После того, как граф построен, мы можем исследовать его на существование и выделение неоптимальных зависимостей. Алгоритм будет заключаться в следующем.

Из списка всех групп выберем какую-нибудь группу G. Во-первых, среди групп, которые зависят от G выберем группу F, и во-вторых, среди всех групп, от которых зависит G, выберем произвольную группу H. Так мы получим тройку $F \to G \to H$.

Группы F и H (и соответствующие им библиотеки) транзитивно зависят друг от друга. Вспоминая, что F, G и H являются группами вершин, мы хотим понять, зависят ли вершины в группе F от каких-нибудь вершин группы H, или иными словами, существует ли путь от какой-либо вершины из F до какой-либо вершины из H. Несуществование подобного пути даёт нам понять, что зависимость $F \to G \to H$ неоптимальна и группу G (и соответствующую ей библиотеку) можно разделить на две G_1, G_2 так, чтобы $F \to G_1, G_1 \not\to G_2, G_2 \to H$.

Псевдокод подобного алгоритма представлен ниже.

Algorithm 1 Поиск неоптимальной зависимости.

```
1: procedure FindUnoptimalDependencies(Graph, Groups)
2:     r ← {}                          ▷ Result that is returned from the function
3:     for G ∈ Groups do
4:         for F ∈ {X ∈ Groups | X → G} do
5:             for H ∈ {Y ∈ Groups | G → Y} do
6:                 if path from F to H doesn't exist then
7:                     r ← {F, G, H}
8:                 end if
9:             end for
10:        end for
11:    end for
12:    return r
13: end procedure
```

4 Реализация

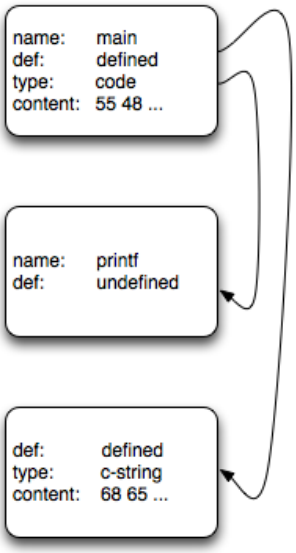
4.1 Выбор линкера

В качестве линкера, был использован линкер lld, разрабатываемый в рамках проекта LLVM [2]. К его преимуществам относительно устоявшегося и широко используемого линкера ld можно отнести: скорость — lld от 1.2 до 2 раз быстрее линкера gold, который в свою очередь быстрее ld; относительно маленький размер исходного кода — ELF часть lld на июнь 2016 составляет 13k строчек кода и COFF часть составляет еще 7k[4], в то время как gold составляет 146k строчек кода; модульность и расширяемость — в lld относительно просто встроить дополнительную логику, за счет чего, например, в lld была добавлена такае функция как дедупликация идентичного кода. Также как и линкер gold, lld можно использовать как прямую замену ld.

Более того, дизайн линкера, а точнее внутреннее представление файлов, также помогает в реализации целей данного исследования. lld внутри себя использует модель «атомов» — неделимых кусков кода или данных. Обычно каждая написанная пользователем функция или глобальная переменная — это атом. Кроме этого компилятор может добавлять атомы, например, строковые литералы или некоторые константы.

Простая “hello world” программа выглядит так:

[4]здесь не учитывается код из самого проекта LLVM, на базе которого написан lld



Атомы, также как и символы, помимо прочего, могут быть определёнными и неопределёнными, неопределённые атомы зависят от определённых и естественным образом образуют граф зависимостей. Этот факт помогает при постреении графа, используемого в данном исследовании.

4.2 Дополнительные проверки

Построенная модель является довольно общим инструментом изучения зависимостей в проектах. Она позволяет применять и другие алгоритмы на ориентированных графах [3].

Следующие алгоритмы были реализованы на основе данной модели:

- поиск циклов,
- поиск мостов.

Эти алгоритмы являются классическими алгоритмами в теории графов, и их теоретическое описание можно найти например в [4].

5 Основные результаты

Для исследования эффективности предложенного алгоритма, были взяты проекты с открытым исходным кодом, а именно: Qt, SFML, tungsten.

В Qt [5], в модуле Qt Tools, была найдена 1 неоптимальная зависимость, улучшение которой ускоряет сборку Qt Tools на 12%.

В проекте SFML [6] была также найдена 1 неоптимальная зависимость, улучшение которой ускоряет сборку SFML Window на 18%.

В проекте Magnum [7] были найдены неоптимальные зависимости в модулях MagnumDebugTools и MagnumWavAudioImporter, улучшение которых ускоряет их сборку на 11% и 24% соответственно.

Qt Base	12%
SFML	18%
Magnum	11%, 24%

6 Заключение

В первой части работы было введено понятие неоптимальной зависимости, затем была рассмотрена графовая модель, позволяющая находить и устранять подобные зависимости. На основе предложенной модели были реализованы такие алгоритмы, как поиск циклов и поиск мостов. Исследование было проверено на проектах с открытым исходным кодом: Qt, SFML, Magnum, на которых были получены положительные результаты.

Список литературы

[1] https://blog.travis-ci.com/2012-12-17-numbers/.

[2] http://lld.llvm.org/.

[3] Manfred Nagl. *Graph-theoretic concepts in computer science*. Springer, 1990.

[4] Х Томас. Кормен, Чарльз И. Лейзерсон, Рональд Л. Ривест, Клиффорд Штайн Алгоритмы: построение и анализ—2-е изд. *М.:«Вильямс*, page 1296, 2006.

[5] Qt — cross-platform multipurpose application framework. https://www.qt.io/.

[6] SFML — cross-platform library providing various multimedia API. http://www.sfml-dev.org/.

[7] Magnum — C++11/C++14 OpenGL graphics engine. http://mosra.cz/blog/magnum.php.

[8] Andrea Corradini, Luciana Foss, and Leila Ribeiro. Graph transformation with dependencies for the specification of interactive systems. In *International Workshop on Algebraic Development Techniques*, pages 102–118. Springer, 2008.

[9] Tom Mens, Gabriele Taentzer, and Olga Runge. Analysing refactoring dependencies using graph transformation. *Software & Systems Modeling*, 6(3):269–285, 2007.

[10] Andrea GM Cilio and Henk Corporaal. A linker for effective whole-program optimizations. In *International Conference on High-Performance Computing and Networking*, pages 643–652. Springer, 1999.