Ускорение оптимизации программ во время связывания

К.Ю. Долгорукова <unerkannt@ispras.ru>
С.В. Аришин <arishin@phystech.edu>
Институт системного программирования РАН,
109004, Россия, г. Москва, ул. А. Солженицына, д. 25

Аннотация. В данной статье речь идет о двух методах ускорения процесса сборки программы: распараллеливании межпроцедурной оптимизации и о легковесном методе проведения оптимизаций. Ускорение в первом случае достигается за счет горизонтального масштабирования системы оптимизации времени связывания. Во втором случае метод представляет собой работу исключительно на аннотациях, что позволяет работать с минимально необходимой информацией вместо кода всей программы целиком. Проблема горизонтальной масштабируемости системы всегда связана с необходимостью разделения большой задачи на несколько подзадач, которые могут быть выполнены независимо и параллельно. Масштабирование оптимизаций времени связывания – непростая задача, так как оптимизирующие преобразования работают последовательно на всем промежуточном представлении программы, и результат их работы зависит от предыдущих преобразований. Для оптимизации на этапе связывания необходимо разделить промежуточное представление компилируемой программы на участки, минимизируя зависимости между ними, и оптимизировать эти участки отдельно. Для анализа программы используется граф вызовов. Таким образом, задача сводится к тому, чтобы разделить граф вызовов на несколько слабо связанных друг между другом компонент. Данная задача относится к одной из сложных комбинаторных проблем, и нахождение оптимального решения – NP-полная задача. Тем не менее, качество работы алгоритма зависит от свойств графа, поэтому целесообразно исследовать свойства графа вызовов в контексте оптимизаций времени связывания и подобрать к нему приемлемый алгоритм, проверив работу на реальных программах. Основная задача данного исследования – найти легковесный и эффективный метод разбиения структуры программы таким образом, чтобы как можно меньше ухудшить производительность собираемых программ при независимой оптимизации участков кода. В статье представлен новый метод разбиения графа вызовов программ, проведено его сравнение с некоторыми другими существующими методами для графов вызовов тестовых программ. Также описана реализация предложенного метода в системе LLVM, представлены результаты сравнения производительности программ, собранных в один поток и в несколько потоков. Запуск на 4-х потоках показал ускорение процесса сборки в среднем на 31%, тогда как производительность по сравнению с собранными в один поток программами упала в среднем на 3%. Для легковесного метода оптимизаций описана реализация преобразования удаления мертвого кода. Также приведены результаты тестирования в совокупности с ленивой загрузкой кода.

K. Dolgorukova, S. Arishin. Link-time optimization speedup. *Trudy ISP RAN/Proc. ISP RAS*, vol. 28, issue 5, 2016, pp. 175-198.

Ключевые слова: компиляторы; оптимизация времени связывания; масштабирование; разбиение графов.

DOI: 10.15514/ISPRAS-2016-28(5)-11

Для цитирования: К.Ю. Долгорукова, С.В. Аришин. Ускорение оптимизации программ во время связывания. Труды ИСП РАН, том 28, вып. 5, 2016, стр. 175-198. DOI: 10.15514/ISPRAS-2016-28(5)-11

1. Введение

Оптимизации времени связывания зарекомендовали себя как эффективный инструмент оптимизации программ [1]. Проводимые на этапе компоновки объектного кода с внедренным промежуточным представлением программ, они способны работать над всем кодом целиком, эффективно оптимизируя межпроцедурные зависимости. Также полное представление о коде дает возможность проводить более агрессивные локальные оптимизации и оптимизации с использованием профиля.

Сложность подхода оптимизаций времени связывания заключается в том же, что и его сила: весь код программы находится в памяти. Этот факт накладывает дополнительные требования к ресурсам, в особенности, для крупных программ. К примеру, сборка состоящего из 36,5 тысяч исходных C/C++ файлов браузера Firefox на компиляторах GCC и LLVM с оптимизацией времени связывания требует от 6 до 34 Гб ОЗУ в зависимости от настроек, и работает от 11 до 26 минут на x86-64 [2]. Для офисного текстового редактора LibreOffice, состоящего почти из 20 тысяч C/C++ файлов, эти числа будут иметь значения 8-14 Гб и 61-68 минут соответственно [3].

Данная статья является частью исследования, посвященной разработке масштабируемой системы оптимизации времени связывания. Полученные ранее результаты, а также подробное описание системы можно найти в [4] и [5]. Масштабируемость системы подразумевает как возможность работать на устройствах с ограниченными ресурсами, так и на высокопроизводительных машинах. И, если в ранних работах мы вели речь о сборке программ на устройствах с низкой производительностью, то в данной работе упор будет сделан на масштабировании под устройства с несколькими ядрами. Как было показано в [4], традиционный путь масштабирования подразумевает наличие трёх стадий при сборке программы из исходного кода: генерацию промежуточного представления, межпроцедурный анализ и генерацию машинного кода. Первая и последняя стадии теоретически могут проводиться параллельно. На практике же возможность распараллеленной оптимизации во время генерации машинного кода спорна, и применяется далеко не во всех оптимизирующих компоновщиках. В частности, в системе LLVM ранее такой возможности не было.

Статья построена следующим образом: в разделе 2 речь пойдет о разбиении графа вызовов для проведения межпроцедурной оптимизации в несколько

потоков. В подразделе 2.1 приведен краткий обзор существующих методов разбиения графа. В 2.2 приводятся оценки разбиения, а также обсуждается применимость алгоритмов к графам вызовов программ. В 2.3 описывается предлагаемый для разбиения алгоритм, а в 2.4 приводится его сравнение с некоторыми другими алгоритмами по предложенным в 2.2 метрикам. В разделе 3 описываются легковесные межпроцедурные оптимизирующие преобразования, поясняется их место в разрабатываемой системе оптимизации времени связывания. В 3.1 описывается общий метод работы таких преобразований. В 3.2 проводится краткий обзор существующих оптимизаций времени связывания и обсуждается применимость к ним предложенного метода. Раздел 4 посвящен реализации предлагаемых методов в системе LLVM. Раздел 5 содержит результаты тестирования. Заключение завершает статью выводами и планами дальнейших работ.

2. Горизонтальное масштабирование системы оптимизации времени связывания

Как было упомянуто в [4], этап анализа не может быть распараллелен, но, как правило, распараллеливается этап генерации машинного кода. Для этого на этапе анализа производится такое разбиение промежуточного представления программы на части, чтобы можно было проводить независимую оптимизацию этих частей без отрицательного влияния на результирующую производительность программы.

Большинство межпроцедурных оптимизаций работает с инструкциями вызовов: например, встраивание вставляет тело функции в место его вызова, если оценщик встраивания признает эту процедуру целесообразной, межпроцедурное распространение констант вставляет константы в параметры вызова и распространяет в тело вызываемой функции, если это возможно; распространение аргументов производит похожие манипуляции с переменными, и так далее.

Несложно догадаться, что ключевой зависимостью для межпроцедурных оптимизаций является зависимость по вызовам функций программ. При этом, локальные данные функций и инструкции, работающие с ними и не выводящие поток управления программы за пределы тела функции, в целом не повлияют на работу межпроцедурных оптимизаций, если они будут проводиться независимо на функциях. Таким образом, функцию можно взять за единицу анализа, а вызовы между функциями – за отношения между функциями.

По причине того, что зависимости между функциями играют ключевую роль в работе оптимизаций времени связывания, произвольное разбиение кода на части для независимой оптимизации может сильно испортить результаты межпроцедурных оптимизаций, снизив производительность собираемой программы. Таким образом, необходимо распределить функции на группы так, чтобы связей между группами было как можно меньше. Это могут быть связи

вызовов, частота вызовов, а также дополнительная информация о контексте вызовов, необходимая, например, для оптимизации встраивания.

С этой целью в данной работе производится обзор алгоритмов разбиения графов, а также исследуются кластерные свойства графов вызовов для подбора подходящего алгоритма разбиения.

2.1 Определения

Для графа G = (V, E), где V - множество вершин, E – множество ребер, положим число вершин n=|V| и число ребер m=|E|. Граф называется ориентированным, если для ребра {u, v} важен порядок следования вершин, – и неориентированным, если не важен, тогда {u, v} ≡ {v, u}.

Все нижеописанные определения даны для неориентированных графов, но тривиальным образом переопределяются для ориентированных.

Определим плотность графа:

$$\delta(G) = \frac{m}{\binom{n}{2}}$$

Для *n={0,1}* положим (G)=0. Граф, плотность которого равна 1, называется полным. Если *{u, v}∈E*, то u и v называются соседями.

Матрица смежности A_G графа *G = (V, E)* порядка *n* – это матрица n×n $A_G=(a^G_{v,u})$, где

$$a\frac{G}{v,u} = \begin{cases} 1, & \{v,\ u\} \in E, \\ 0, & \{v,u\} \notin E \end{cases}$$

Разбиением графа назовем совокупность подмножеств $C_1,C_2,\ldots,C_k$ графа *G=(V,E)*, CiV, i=1,k, что $\bigcup_{i=1}^{k} C_i = V$. Разбиения графа могут быть пересекающимися и непересекающимися. В нашей статье речь пойдет, в основном, о непересекающихся разбиениях, таких, что $C_i \cap C_j = 0, \forall i \neq j$. Далее будем называть такие подмножества *кластерами*.

Пусть S – подмножество вершин графа G=(V,E), S⊆*V*. Индуцированным S графом называется подграф E(S): *{{u,v}|u∈V, v∈V, {u,v}∈E}*. Назовём *локальной плотностью* подграфа E(S) величину *δ*(G(S))=|E(S)||S|.

Степенью вершины назовём число инцидентных ей ребер, то есть deg(v) = |{{u,v}| u∈ V, u≠v}|. Внутренней степенью кластера назовём величину $\deg_{int}$ (C) = |{{v, u} ∈ E | v, u ∈ C}|, а внешней – $\deg_{ext}$ (C) = |{{v, u} ∈ E | v ∈ C, u ∈ V\C}|. Также нам понадобится определение компоненты сильной связности ориентированного графа. Сильно связным графом G=(V,E) называется такой граф, что для любых двух его вершин u∈V, v∈V существует ориентированный путь из u в v. Компонентами сильной связности, или сильно связными компонентами, называются максимальные по включению его сильно связные подграфы.

2.2 Краткий обзор методов разбиения графов

В связи с бурным развитием области анализа больших данных, – будь то социальные сети, биологические сети, сети коммутации, – соответственно развивалась область анализа графов.

В данной работе мы не будем углубляться в обзор методов разбиения графов, а лишь приведем классификацию и коротко упомянем некоторые из них. Более подробный обзор можно прочитать в статье Shaeffer [6].

По доступной области данных методы разбиения графов делятся на глобальные и локальные. Глобальные имеют в каждый момент времени доступ ко всему графу, тогда как локальные работают с несколькими так называемыми *зерновыми* вершинами и информацией об их соседях второго уровня. Зерновые вершины могут выбираться какими-то эвристическими методами, либо случайно. В случае больших данных глобальные методы оказываются требовательны к ресурсам, требования же локальных методов не зависят от размера исходного графа. Результаты глобальных методов в целом превосходят результаты локальных, но продвинутые модификации локальных методов также хорошо работают на некоторых графах [7].

Глобальные методы, в свою очередь, делятся на итеративные и иерархические. Итеративные методы – это такие широко известные методы, как k-средних и его модификации; иерархические же методы основаны на построении множества вложенных кластеров. Проблемы итеративных методов, прежде всего, состоят в том, что необходимо точно определить метрику схожести элементов. Для графов за нее обычно берут расстояние. Также итеративные методы чувствительны к порядку обхода и начальному разбиению. Иерархические методы не чувствительны к порядку обхода и начальным данным, но основная их проблема заключается в спорном моменте, как определить конечное разбиение при полученном в результате работы алгоритма дереве кластеров.

Иерархические методы, в свою очередь, делятся на дивизивные и аггломеративные по порядку обхода дерева иерархии: сверху вниз или снизу вверх. При дивизивном разбиении в начале алгоритма задан один кластер C_0=G, который итеративно разбивается на подкластеры. При аггломеративном разбиении наблюдается обратный процесс: начальное разбиение представляет собой n кластеров, содержащих по одной вершине, и на каждой итерации происходит объединение существующих кластеров в более крупные.

Дивизивные методы классифицируются соответственно метрикам качества разбиения. Методы, основанные на разрезах, ищут минимальный разрез; методы, основанные на мере промежуточности, стараются максимизировать ее; спектральные методы ищут собственные вектора на матрице Лапласа. Более экзотические методы типа методов на основе цепей Маркова, электрических цепей, сводятся к описанным выше.

Самые точные результаты дают спектральные методы, но они требуют огромных математических вычислений, а значит, ресурсов. Остальные глобальные методы обладают сложностью начиная от O(n log n) и заканчивая $O(n^4)$. Некоторые вероятностностые методы имеют сложность O(n), но дают хороший результат с некоторой вероятностью. К тому же, у методов разная чувствительность к шумам в виде добавления или удаления случайных ребер.

В целом же выбор того или иного метода прежде всего зависит от свойств анализируемого графа.

2.3 Методы оценки разбиения

Оценка качества разбиения – тема, на которую по сей день ведутся споры, так как не все метрики одинаково отражают свойства полученных разбиений [9]. Поэтому для оценки качества разбиения применялись две метрики: модулярность и средняя относительная внутрикластерная плотность.

Для заданной группы кластеров $(C_1, \dots C_k)$ вычисляется матрица относительных степеней кластеров E = $\{\mathbf{e}_{ij}\}$, где для кластеров C_i и C_j $\boldsymbol{e}_{ij} = |(u,v)|/m$, $u \in C_i$, $v \in C_j$, а $\boldsymbol{e}_{ii} = deg(C_i)/m$.

Мера модулярности кластера $Q = \boldsymbol{\Sigma}_i(\boldsymbol{e}_{ii} - \boldsymbol{a}_i^2)$, где $\boldsymbol{a}_i = \boldsymbol{\Sigma}_j \boldsymbol{e}_{ij}$ была введена Ньюманом[7] для оценки разбиения крупных биологических и социальных сетей. Отметим, что Q≤1, и только для “идеальных” кластеров, представляющих собой несвязные клики, Q=1. Между тем, “хорошими” можно считать кластеры, модулярность которых порядка 10^{-1}, и разбиение можно назвать “плохим”, если Q<0.

Вторая мера – относительная внутрикластерная плотность, – это отношение плотности кластера к плотности графа целиком. Относительной плотностью подграфа назовем отношение $\frac{\delta(G(S))}{\delta(G)}$. Это более очевидная метрика, показывающая, насколько полученный кластер “плотнее” всего графа целиком.

2.4 Исследования свойств графов вызовов программ

О графе вызовов априори известно то, что это разреженные ориентированные графы. Так как иногда вызовов одной и той же функции в теле другой функции может быть несколько, граф также является взвешенным: каждому ребру мы сопоставим количество вызовов одной функции из тела другой функции. Также может оказаться полезным присваивать веса и самим узлам, указывая, например, размер функции в инструкциях: это может помочь равномерно загружать вычислительные модули при оптимизации.

Если собираемый код представляет собой исполняемую программу, то у нее будет одна точка входа – функция main. В этом случае программа имеет древоподобный вид с выраженным корнем. Если же компонуется библиотека, то точек входа может быть несколько. В случае применения callback-функций в совокупности с системными вызовами, точек входа может быть несколько

даже в исполняемой программе; тем не менее, как правило, в графе все равно прослеживается древовидность. Более всего нарушают древоподобную структуру сильно связные компоненты. В случае с графами вызовов компоненты сильной связности можно интерпретировать как рекурсивные вызовы. При разбиении графа такие подмножества целесообразно помещать в один кластер в виду высокой связности. Чтобы проверить вышеописанные предположения, мы провели статистическое исследование графов вызовов реальных программ, посчитав в них количество компонент сильной связности, а также разбив эти графы достаточно точным итеративным алгоритмом Girvan & Newman [7][8], максимизирующим меру промежуточности [10].

Диаграммы 1 и 2 показывают влияние различных форм для одних и тех же графов на качество разбиения.

Первый (синий) столбец показывает, насколько хорошо локализованы связи в исходных файлах: то есть, мы принимаем файлы за кластер и измеряем для них модулярность и относительную плотность.

Второй (красный) показывает характеристики кластеров, если в качестве исходного графа использовать граф файлов G_f, где каждому узлу v соответствует файл с исходным кодом F, а каждому ребру (u,v) – вызовы функции g внутри функций f, где g – функция из файла F'≠F. При этом, из-за небольшого количества файлов в некоторых тестах, граф для них разбить на 4 компоненты не удалось.

Третий (желтый) показывает, насколько удачно получается разбить графы вызовов посредством алгоритма Girvan & Newman.

Четвертый (зелёный) соответствует кластеризации графа компонент сильной связности.

Следующие три – это работа алгоритма с учетом весов ребер.

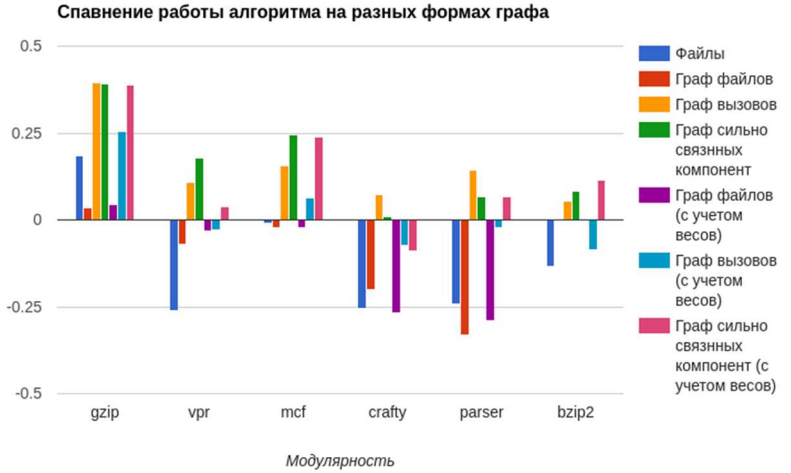

Диаграмма 1. Сравнение модулярности графов

Diagram 1. Graph modularity comparison

Результаты исследования мы трактовали следующим образом. Графы вызовов – разреженные графы, в основном связные, с небольшой плотностью и плохо выраженными естественными кластерами. При этом функции далеко не всегда грамотно разбиты на файлы, что и обуславливает необходимость оптимизации времени связывания.

Проведение разбиения на графах компонент сильной связности почти так же эффективно, как на самих графах вызовов, при этом учет весов незначительно ухудшает результат.

Проведение разбиения графа файлов возможно, но эффективность полученного разбиения зависит от аккуратности разработчиков программы.

Заметим, что для алгоритма разбиения графа на компоненты совсем не обязательно уметь выявлять естественные кластеры, ведь их размеры сильно варьируются, а поиск – очень дорогостоящая задача, на тему которой в данный ведутся многочисленные исследования [11] и даже проводятся конкурсы [12]. Для использования в задаче разбиения графа вызовов подойдет алгоритм, который будет работать за линейное или псевдолинейное время, и на выходе которого будут получены одинаковые по размеру разбиения с достаточно высокой плотностью.

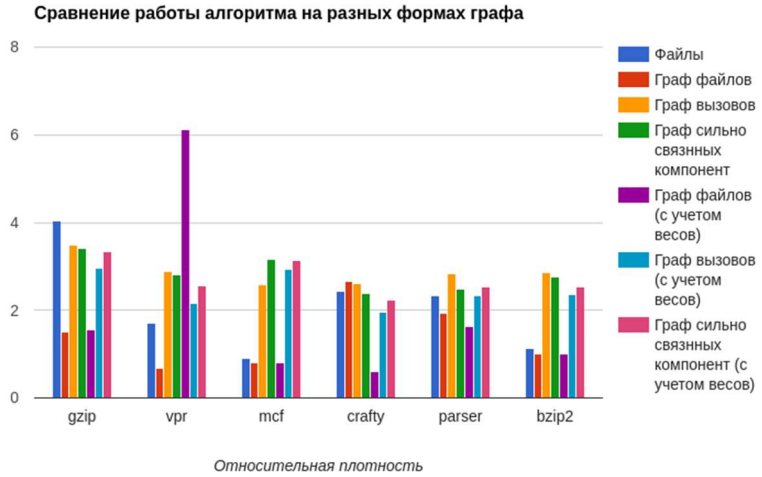

Диаграмма 2. Сравнение относительной плотности графов

Diagram 2. Graph relative density comparison

2.5 Описание предлагаемого алгоритма

Алгоритм работает в два этапа: на первом этапе формируется граф $G_c=(V_c, E_c)$ компонент сильной связности и ориентированный граф приводится к неориентированному виду, на втором этапе полученный граф разбивается на k частей.

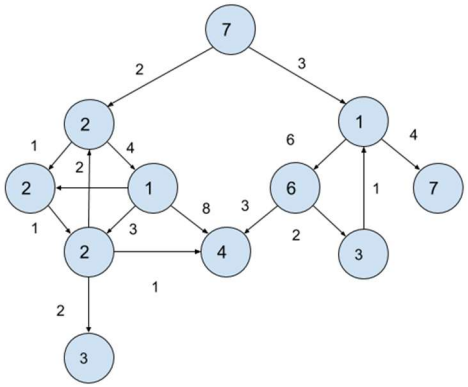

Рис. 1. Пример взвешенного ориентированного графа.

Fig. 1. Weighted oriented graph example

На начальном этапе в нашем распоряжении взвешенный ориентированный граф G = (V,E). Компоненты сильной связности могут быть найдены посредством алгоритма Косарайю или Тарьяна.

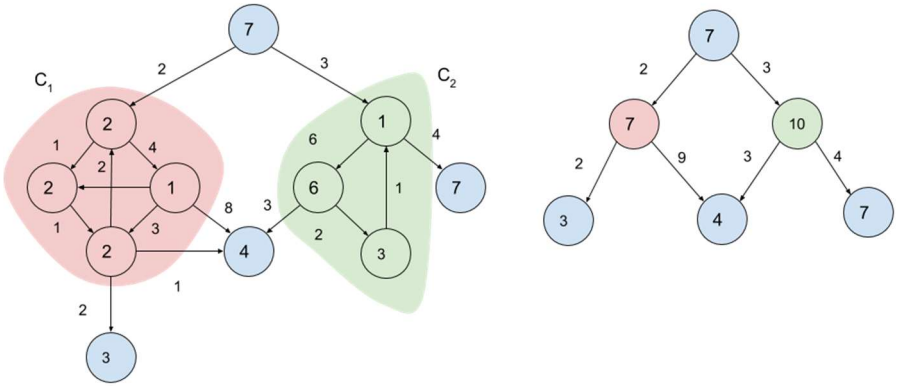

Рис. 2. Преобразование графа в граф сильно связных компонент.

Fig. 2. Graph transformation to strong-connected component graph

Затем для найденных компонент связности формируется множество V_c вершин G_c ***sc***: $v \longrightarrow v_c$. При этом, если вершина $v \in V$ не входит ни в одну найденную компоненту из V_c, то сама вершина v помещается в V_c. Ребра получаются тривиальным образом: $\forall v \in V$: $v \longrightarrow v_c$, $u \in V$: $u \longrightarrow u_c$, $u_c \neq v_c$, если $\{v,u\} \in E$, то $\{v_c,u_c\} \in E_c$. Веса ребер также: $w(vc,uc)=\sum_{v \to v_c,\ u \to u_c} w(v,u)$. Если имеют место взвешенные вершины, то вес тоже получается за счет суммирования весов вершин, входящих в компоненту.

Затем граф приводится к неориентированному виду тривиальным образом. Так, матрица смежности графа становится симметричной относительно главной диагонали. В результате этих манипуляций получаем граф *G'=(V', E')*.

Перейдем к самому разбиению. Положим $n=|V'|$, $m=|E'|$, а k – число кластеров, на которые нужно разбить граф, $k < n$. Первая наша задача – выбрать потенциальные центры кластеров, обладающие большой “гравитацией ”. Для этого выбираем $2k$ вершин с максимальным удельным весом. Удельный вес каждой вершины $v \in V'$ считается по формуле $\omega(v)=w(v) \cdot \sum_{(v,u) \in E'} w(v,u)$, где $w(v)$ – вес вершины или 1, если вес не задан, а $w(v,u)$ – вес ребра (v,u). Далее для каждого потенциального центра алгоритмом Дейкстры ищем расстояния от данной вершины до других центров. В определении расстояния могут возникнуть разночтения из-за того, что мы имеем дело со взвешенным графом. То, как мы трактуем веса ребер, зависит исключительно от контекста. Так как алгоритм рассчитан на работу с графами вызовов программ, и веса ребер в

нашем случае – результат профилирования или количество статических вызовов, то для расстояния между соседними вершинами логично использовать величину, обратную весу ребер. Подробно о том, как вычислялись расстояния в нашем графе, описано в 4.1.

Сортируем эти вершины по удаленности от остальных в порядке убывания. В качестве меры удаленности от других вершин можно брать среднее или среднее квадратичное расстояние. Для простоты в наших примерах возьмём среднее. Получаем вектор $\boldsymbol{c}=\{c_1,c_2,\ldots,c_k\}$, $c_i \in V'$ $\forall\ i=1..k$, и выполняется неравенство $\sum_{i\neq j, i\leq k, j\leq k} d(c_i,c_j) > \sum_{j\neq p, p\leq k, p\leq k} d(c_j,c_p)$, $\forall\ i\in[1..k],\ j\in[1..k],$ i<j. Далее для каждого центра C_i жадным алгоритмом с помощью обхода в ширину набираем в кластер C_i [N/k] вершин. Если у какого-то центра закончились соседи, то назначаем центром следующий по списку центр, пока не наберется требуемое количество вершин или не кончатся центры.

Если случилось так, что центры все равно находились слишком близко, и быстро исчерпали всех соседей, то добавляем в кластеры оставшиеся вершины случайным образом.

Центров кластеров *2 k* лишь потому, что может возникнуть ситуация взаимного поглощения кластеров, когда 2 центра оказываются слишком близко, и один из них поглощает всех соседей второго.

Сложность полученного алгоритма зависит от свойств графа, от выбранного *k*, а также от используемых вспомогательных алгоритмов для промежуточных целей. В случае разреженных графов алгоритм показывает асимптотику роста *O(k n log n + k m log n)*.

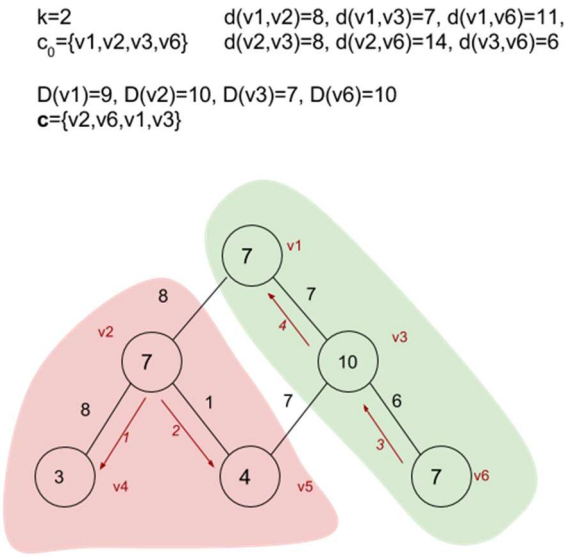

Рис. 3. Пример работы алгоритма на графе. Нумерованными стрелками показан порядок обхода вершин.

Fig. 3. Applying the algorithm to SCC-graph example. Order of vertex handling is denoted by numbered arrows

3. Легковесные оптимизации

Необходимость облегчать оптимизации обусловлена не только необходимостью сокращения времени сборки приложений; в большинстве масштабируемых систем оптимизации времени связывания это неотъемлемая часть фазы анализа программы, когда в памяти находится не весь код программы, а лишь информация, необходимая для анализа связей в программе [13][14].

Суть метода заключается в разбиении преобразования на 2 стадии: анализирующую и оптимизирующую. Первая стадия проводится на этапе компиляции и генерации промежуточного представления для файлов исходного кода. Когда промежуточное представление оказывается сформировано, анализатор обходит его и формирует дополнительную минимально необходимую информацию, достаточную для проведения оптимизации конкретного типа, – *аннотации*. На второй стадии аннотации считываются и обрабатываются оптимизирующим пребразованием, непосредственно меняющим код. Вышеописанный метод приводит к увеличению размера промежуточного представления, но ускоряет процесс

анализа кода программы, а самое главное – позволяет компоновщику не загружать часть кода программы в память. Таким образом, метод позволяет проводить оптимизации при использовании метода ленивой загрузки кода еще до загрузки тел функций.

К сожалению, не каждое существующее межпроцедурное оптимизирующее преобразование можно с легкостью привести к аналогичному, работающему только на аннотациях. В рамках исследования уже было реализовано построение графа вызовов с помощью аннотаций [5], который может быть использован межпроцедурными преобразованиями, не требующими сигнатуру вызова. Для преобразований, требующих доступ к локальным переменным и инструкциям не всегда оптимально использовать аннотации, поскольку загрузка тел функций оказывается дешевле, чем накладные расходы на поддержание аннотаций.

Таким образом, необходимо определить, какие преобразования возможно и необходимо адаптировать в первую очередь. Так как работа ведется на основе компиляторной системы LLVM, были исследованы уже реализованные в ней оптимизирующие проходы.

Оптимизации времени связывания в LLVM представлены спектром зарекомендовавших себя методов, в их числе: встраивание кода, межпроцедурное удаление мёртвого кода, межпроцедурное распространение констант и т.д. В сочетании с локальными методами оптимизации и использованием профиля разработчик может получить увеличение производительности программы до 30%.

Мы проанализировали существующие на данный момент оптимизации и сравнили влияние на производительность существующих методов без использования профиля. Оценить влияние оптимизирующих преобразований оптимизации времени связывания LLVM на улучшение производительности довольно сложно из-за их большого количества и влияния друг на друга.

Поэтому для оценки влияния была предложена следующая схема:

1. Для каждого преобразования запускалась серия тестов, в которой оно исключалось из списка оптимизирующих преобразований.
2. Запускалась серия тестов, в которой участвовали все преобразования.
3. Для каждого теста из серии строился массив влияния преобразования на производительность. Элемент массива представляет собой разность результатов тестов с пунктов 1 и 2 для каждого преобразования. Массив сортировался, лучшие 4 преобразования получали баллы от 4 до 1, худшие 4 от -4 до -1, остальные – 0.
4. Баллы для каждого преобразования суммировались по всей серии тестов. На основе данной схемы оценки была построена следующая диаграмма для оптимизирующих преобразований LTO LLVM.

Мы также проанализировали, какую информацию используют оптимизирующие преобразования, чтобы выяснить, какие из них могут проводиться в облегченном варианте, а какие – нет. Большинству оптимизирующих преобразований необходим доступ к инструкциям, базовым блокам и инструкциям вызовов. Это значит, что им необходим доступ к телам функций для выполнения анализа, что нежелательно в случае ленивой загрузки кода. В числе таких трудноадаптируемых межпроцедурных методов оказались: продвижение аргументов, распространение констант и даже встраивание. Тем не менее, самое эффективное преобразование с точки зрения тестирования – межпроцедурное удаление мертвого кода, – практически не затрагивает код, а лишь просматривает использование глобальных переменных в коде.

Таким образом, для реализации метода легковесной оптимизации был выбран метод межпроцедурного удаления мёртвого кода (GlobalDCE).

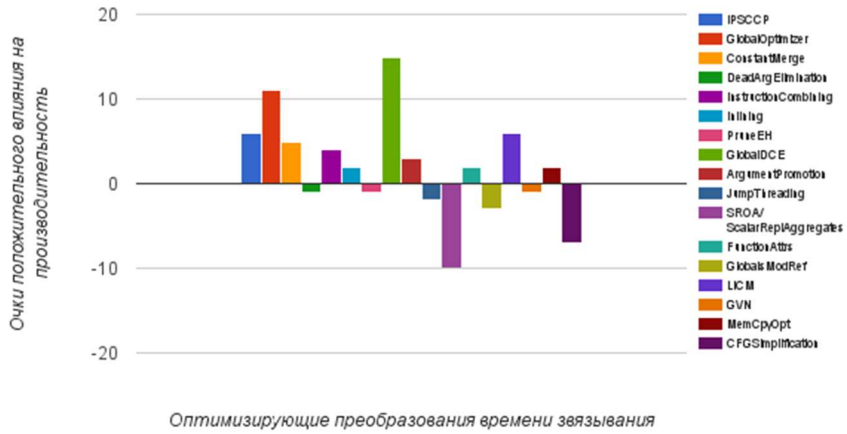

Диаграмма 3. Влияние оптимизирующих проходов на производительность тестов

Diagram 3. Performance impact of optimization passes

4. Реализация методов

Оба метода были реализованы в разрабатываемой в Институте системного программирования масштабируемой системе оптимизаций времени связывания на основе LLVM [4][5].

4.1 Реализация метода разбиения

За счет наличия в LLVM эффективной алгоритмической базы, некоторые вспомогательные алгоритмы были реализованы с их помощью. Например, в системе LLVM уже присутствует алгоритм поиска компонент сильной связности в графе, поэтому он был использован для построения графа

компонент сильной связности. Для сортировки массивов использовались функции стандартной библиотеки STL. Тем не менее, дополнительно была реализована двоичная куча для подсчета расстояний между вершинами графа.

Для поиска расстояний в графе использовалась следующая формула преобразования весов ребер: $w'(v)=max_{u \in E'}(w(u)) + 1 - w(v)$, которая использовалась для нахождения наиболее удаленных вершин в массиве.

Для исследования были взяты некоторые тесты из тестового набора SPEC CPU2000, для которых удалось построить все структуры и на которых удалось запустить оба алгоритма поиска разбиения.

Табл. 1. Сравнение модулярности при разбиении предложенным алгоритмом.

Table 1. Modularity measurements of graph division performed by proposed algorithm

Модулярность	Граф файлов	Граф вызовов	Граф сильно связных компонент	Граф файлов (с учетом весов)	Граф вызовов (с учетом весов)	Граф сильно связных компонент (с учетом весов)
gzip	0.02	0.02	-0.15	-0.04	-0.06	-0.03
vpr	-0.28	-0.28	-0.34	-0.33	-0.05	-0.06
mcf	-0.01	-0.06	-0.25	-0.01	0.19	0.16
crafty	-0.28	-0.43	-0.42	-0.36	-0.43	-0.44
parser	-0.36	-0.28	-0.42	-0.21	-0.32	-0.41
bzip2	0	-0.54	-0.55	0	-0.45	-0.45

Табл. 2. Сравнение относительной плотности при разбиении предложенным алгоритмом.

Table 2. Relative density measurements of graph division performed by proposed algorithm

Относительная плотность	Граф файлов	Граф вызовов	Граф сильно связнных компонент	Граф файлов (с учетом весов)	Граф вызовов (с учетом весов)	Граф сильно связнных компонент (с учетом весов)
gzip	2.37	2.75	2.14	3.03	2.81	2.78
vpr	1.17	1.7	1.48	1.05	2.57	2.55
mcf	2.44	3.17	1.62	1.7	3.89	4.21
crafty	1.68	1.62	1.33	1	1.58	2.2
parser	1.81	1.76	1.6	1.09	2.61	2.38
bzip2	0.25	1.11	1.81	0.25	2.42	2.47

В сравнении с алгоритмом Джирвана и Ньюмана, а также с простым жадным алгоритм показывает результаты, представленные на диаграммах 4 и 5.

Алгоритм не показал высокой эффективности в поиске “естественных” кластеров, и его результаты немногим лучше случайного жадного алгоритма, тогда как при учете весов он проявил себя в поиске наиболее плотных кластеров, что с учетом дополнительного ограничения на размер кластеров и небольшой сложности алгоритма – весомый аргумент в его пользу.

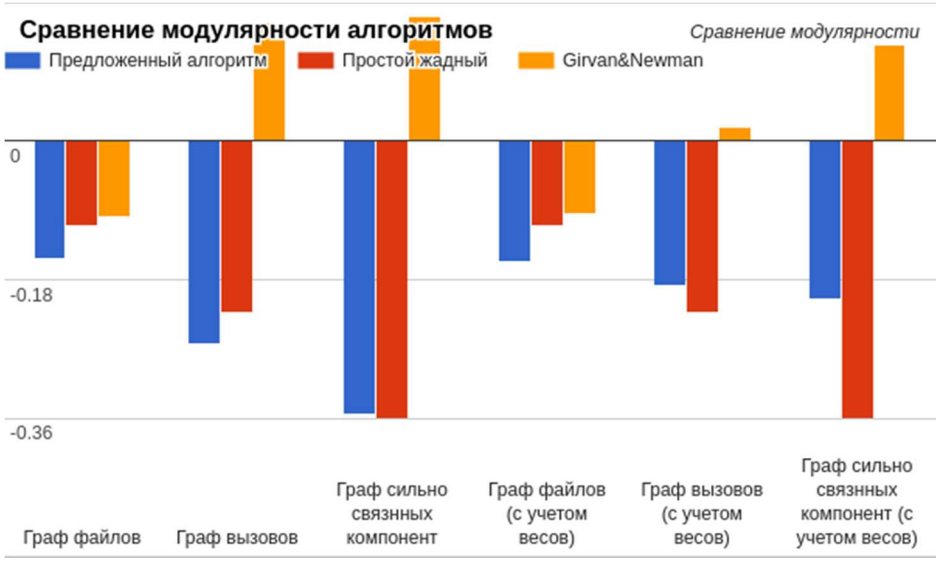

Диаграмма 4. Сравнение работы различных алгоритмов относительно модулярности

Diagram 4. Comparison of modularity of graph division performed by different algorithms

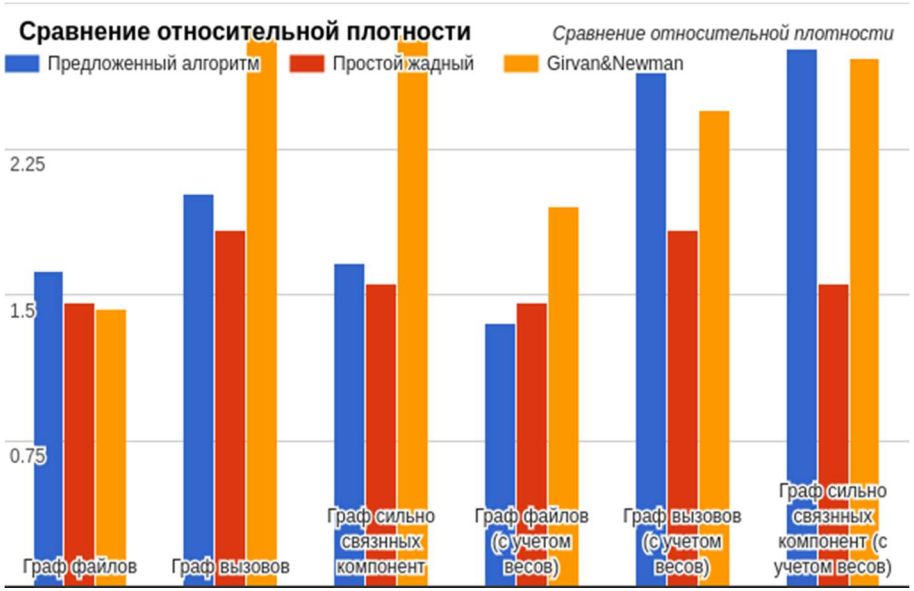

Диаграмма 5.Сравнение алгоритмов относительно относительной плотности

Diagram 5. Comparison of relative density of graph division performed by different algorithms

4.2 Реализация оптимизизации удаления мертвых глобальных переменных

GlobalDCE (Global Dead Code Elimination) – оптимизация, выполняющая поиск и удаление из программы мертвого кода. Мертвым кодом называют код, результат выполнения которого не влияет на результат работы программы. Эта оптимизация полезна тем, что позволяет уменьшить количество выполняемых операций и размер промежуточного представления.

В LTO LLVM GlobalDCE реализован как агрессивный алгоритм, который ищет глобальные переменные и функции, которые «живы», хранит их в списке. В дальнейшем все, что не попало в этот список, удаляется. В частности, GlobalDCE также позволяет удалить части недостижимого кода. В процессе работы плагина Gold GlobalDCE запускается несколько раз.

Предлагается заменить это оптимизирующее преобразование двумя. Одно, как уже говорилось выше, будет запускаться на этапе работы Clang и, аналогично принципам поиска «живых» переменных в GlobalDCE, искать глобальные переменные, которые используются в модуле. Второе, основываясь на результате первого преобразования, в процессе работы плагина Gold проанализирует полученную информацию и удалит неиспользуемые переменные.

Одним из способов прикрепления дополнительной информации к инструкциям является использование метаданных. Метаданные позволяют передать дополнительную информацию о коде оптимизирующим преобразованиям или генераторам кода. Примером может служить отладочная информация. В LLVM есть два примитива метаданных – строки и узлы. Синтаксически метаданные помечаются специальным символом '!'. Метаданные можно прикреплять к модулям, функциям, инструкциям. Также метаданные не имеют типа и это не значения. Рассмотрим каждый примитив подробнее.

Строка метаданных – строка, заключенная в двойные кавычки. Причем в строку может быть включен любой символ, в том числе даже непечатаемые символы вида '\xx', где xx – двузначный шестнадцатеричный код символа.

Узлы метаданных похожи по устройству на структуры – список разнообразных элементов, разделенных запятой и заключенных в кавычки. Элементом узла могут быть не только строки, но также операнды, константы, функции и т.д.

Отдельно стоит отметить, что узлам метаданных можно давать имена, по которым легко найти узел в любой момент компиляции либо просто определить, есть ли такие данные.

Для оптимизирующего преобразования оптимальным является использование именованного узла метаданных, прикрепленного к каждой глобальной переменной.

5. Результаты

Тестирование проводилось на четырехъядерном Intel(R) Core(TM) i5-2500 на тестовом наборе SPEC CPU 2000 (int) [15].

Диаграмма 6 показывает сравнение времени работы жадного алгоритма, работающего за время O(m), с предложенным алгоритмом разбиения. Как видно из диаграммы, отличия присутствуют лишь на тесте gcc ввиду большого размера графа вызовов программы и составляют 2%.

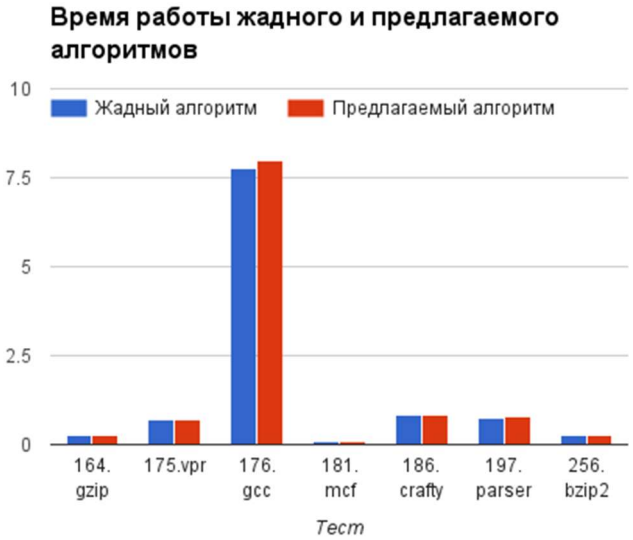

Диаграмма 6. Сравнение времени работы разработанного алгоритма с жадным

Diagram 6. Run time comparison of proposed algorithm vs greedy

Табл. 3. Измерение времени работы сборки с разбиением посредством предлагаемого алгоритма разбиения

Table 3. Run time of building with proposed algorithm graph division and parallel optimization

Тест	**Сборка без распараллеливания, с**	**Сборка с распараллеливанием в 4 потока, с**	**Ускорение, %**
164.gzip	0.28	0.26	7.28
175.vpr	1.06	0.71	32.9
176.gcc	13.88	7.97	42.60
181.mcf	0.13	0.09	31.44
186.crafty	1.5	0.85	43.14
197.parser	1.07	0.78	27.66
256.bzip2	0.4	0.24	38.67
Суммарное время	**18.31**	**10.89**	**31.9**

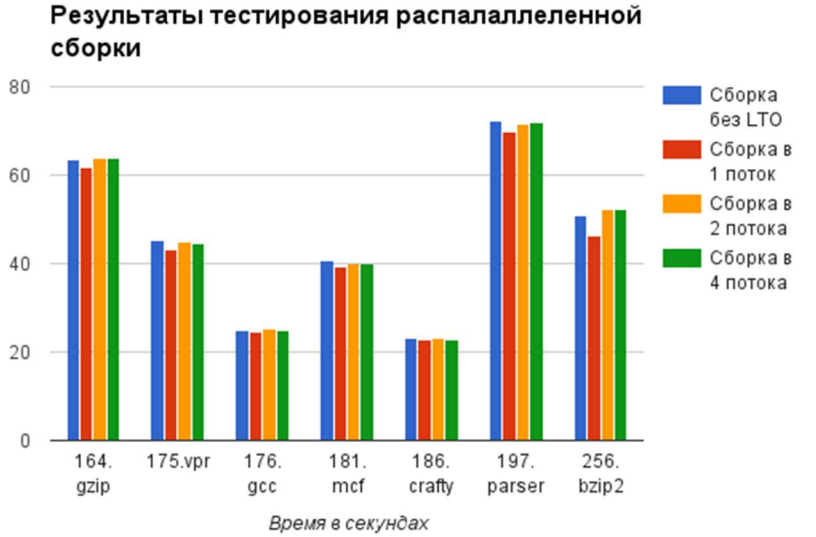

Диаграмма 7. Сравнение производительности программ, собранных с разными уровнями оптимизации времени связывания

Diagram 7. Performance measurements of benchmark building with different LTO parallelization levels

На *таблице 3* видно ускорение сборки программ в среднем на 31%, при этом алгоритм разбиения замедляет работу программы чуть более, чем работающий за O(m) жадный алгоритм.

Так как распараллеливание затрагивает лишь оптимизационную часть сборки и не может быть распространено на этап чтения и компоновки файлов с промежуточным представлением, а также требует дополнительных накладных расходов по времени, теоретический максимум ускорения сборки на 4х потоках составляет всего 50-60% в зависимости от программы.

На *диаграмме 7* приведены сравнения времен работы программ, собранных без оптимизации времени связывания, с оптимизацией в 1 поток, 2 и 4 соответственно.

При тестировании оптимизация проводилась исключительно на разных потоках, и ни предварительного анализа, ни оптимизации перед разбиением проведено не было. Также отсутствовал сильно влияющий на производительность фактор – данные профилирования. Это влечет за собой необходимость более тщательного тестирования, а также работы над предварительным анализом программ. Поэтому довольно сложно сказать, что точно в некоторых случаях отрицательно сказалось на производительности: алгоритм разбиения, отсутствие анализа всей программы или специфика программы. Также ввиду того, что выбранный тестовый набор представляет собой относительно небольшие тесты с небольшим временем сборки, сами

результаты межпроцедурной оптимизации весьма невелики, а «разрыв» некоторых межпроцедурных связей ведет к деградации производительности. Для полной картины необходимо тестирование на целевых для нашей задачи крупных программах: Mozilla Firefox, LibreOffice, Android OS.

Результаты тестирования, указанные в *таблице 4*, являются усредненными значениями серии из 3х тестов. В среднем прирост производительности составляет 0,51%. Стоит отметить, что на отдельных тестах прирост производительности выше — gzip, gcc. Эти тесты состоят из большого числа модулей, на такие программы рассчитана оптимизация. Этим же объясняется ухудшение производительности на тесте bzip2.

Тест состоит всего из одного модуля, поэтому оптимизация ведет к небольшому ухудшению. Из результатов видно, что в большинстве случаев использование аннотаций приводит к улучшению производительности. Стоит также отметить, что тестирование производилось при ленивой загрузке модулей программы на этапе оптимизации времени связывания.

Табл. 4. Измерение влияния легковесного оптимизирующего прохода на производительность.

Table 4. Performance measurements of lightweight GlobalDCE pass

Название теста	Время работы без оптимизации, с	Время работы с оптимизацией, с	Прирост производительности, %
164.gzip	78.8	77.9	1.14
175.vpr	58.4	58.3	0.17
176.gcc	33.4	32.8	1.79
181.mcf	86.9	86.6	0.34
197.parser	95.4	94.2	0.21
256.bzip2	69.3	69.7	-0.57
	421.2	**419.5**	**0.51**

Заключение

Данная статья является продолжением исследования на тему “Масштабирование системы оптимизации времени связывания” и описывает работы по проведению горизонтальной масштабируемости, а также разработке легковесных оптимизирующих преобразований. В статье проведено исследование разбиения графов вызовов программ на компоненты для обеспечения параллельной оптимизации, а также предложен алгоритм разбиения, обеспечивающий высокую внутреннюю плотность компонент при равном числе узлов. Также приведено описание реализации оптимизирующего прохода межпроцедурного удаления мёртвого кода для работы только с

аннотациями. В статье приведены результаты тестирования разработанных методов на тестовом наборе SPEC CPU2000 (INT).

В дальнейшем планируется расширить возможности системы для работы с профилем программы, а также провести тестирование на больших современных программах с открытым исходным кодом, таких как LibreOffice и браузер Firefox с использованием более мощных многоядерных архитектур с возможностью выполнять значительно большее число потоков одновременно.

Литература

[1]. Andrew Ayers, Richard Schooler, and Robert Gottlieb. Aggressive inlining. SIGPLAN Not., 32(5):134–145, 1997. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/258916.258928.

[2]. Honza Hubicka. Linktime optimization in GCC, part 2 – Firefox. http://hubicka.blogspot.ru/2014/04/linktime-optimization-in-gcc-2-firefox.html.

[3]. Honza Hubicka. Linktime optimization in GCC, part 3 – LibreOffice. http://hubicka.blogspot.ru/2014/09/linktime-optimization-in-gcc-part-3.html.

[4]. К. Ю. Долгорукова. Обзор масштабируемых систем межмодульных оптимизаций. Труды ИСП РАН, том 26, вып. 3, 2014 г., стр. 69-90. DOI: 10.15514/ISPRAS-2014-26(3)-3.

[5]. К. Ю. Долгорукова. 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.

[6]. Satu Elisa Schaeffer. Graph clustering. Survey. Computer Science Review. Volume 1, Issue 1, August 2007, Pages 27–64. DOI:10.1016/j.cosrev.2007.05.001.

[7]. M. Girvan, M. E. J. Newman. Community structure in social biological networks. PNAS, June 11, 2002, vol.99, no.12. DOI: 10.1073/pnas.122653799.

[8]. M. E. J. Newman. Scientific collaboration networks. II. Shortest paths, weighted networks, and centrality. Physical review E, vol. 64, 016132. DOI: 10.1103/PhysRevE.64.016132.

[9]. L. da F. Costa, F. A. Rodrigues, G. Travieso & P. R. Villas Boas (2007). Characterization of complex networks: A survey of measurements, Advances in Physics, 56:1, 167-242. DOI:10.1080/00018730601170527

[10]. M. E. J. Newman, M. Girvan. Finding and evaluating community structure in networks. Phys. Rev. E 69, 026113 (2004). arXiv:cond-mat/0308217 DOI:10.1103/PhysRevE.69.026113.

[11]. L. Danon, A. Díaz Guilera, J. Duch, A. Arenas, Comparing community structure identification, Journal of Statistical Mechanics Theory and Experiment (2005) P09008.

[12]. D. A. Bader, H. Meyerhenke, P. Sanders, D. Wagner. Contemporary mathematics: graph partitioning and graph clustering. 10th DIMACS Implementation Challenge Workshop, February 13–14, 2012, Georgia Institute of Technology, Atlanta, GA.

[13]. 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.

[14]. Gautam Chakrabarti, Fred Chow. Structure Layout Optimizations in the Open64 Compiler: Design, Implementation and Measurements. Open64 Workshop at CGO 2008, April 6, 2008. Boston, Massachusetts.

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

Link-time optimization speedup

K. Dolgorukova <unerkannt@ispras.ru>
S. Arishin <arishin@phystech.edu>
Institute for System Programming of the Russian Academy of Sciences,
109004, Moscow, Alexander Solzhenitsyn st., 25.

Abstract. This paper proposes the two different approaches to speed-up program build: making link-time optimization work in parallel and lightweight optimization approach. The former technique is achieved by scaling LTO system. The latter makes link to work faster because of using summaries to manage some of interprocedural optimization passes instead of full IR code in memory.

The problem of horizontal LTO system scaling leads to the problem of partition of the large task to several subtasks that can be performed concurrently. The problem is complicated by the compiler pipeline model: interprocedural optimization passes works consequentially and depends on previous performed ones. That means we can divide just data on which passes works, not passes themselves. We need to divide IR code to sub-independent parts and run LTO on them in parallel. We use program call graph analysis to divide a program to parts. Therefore, our goal is to divide call graph that is one of NP-compete problems. Nevertheless, the choice of the dividing algorithm strongly depends on properties of divided graph.

The main goal of our investigation is to find lightweight graph partition algorithm that works efficiently on call graphs of real programs and that does not spoil LTO performance achievements after optimizing code pieces separately.

This paper proposes new graph partition algorithm for program call graphs, results of comparing this one with two other methods on SPEC CPU2000 benchmark and implementation of the algorithm in scalable LLVM-based LTO system. The implementation of this approach in LTO system shows 31% link speedup and 3% of performance degradation for 4 threads. The lightweight optimization shows 0,5% speedup for single run in lazy code loading mode.

Keywords: compilers; link-time optimization; scalability; graph division; graph clustering.

DOI: 10.15514/ISPRAS-2016-28(5)-11

For citation: K. Dolgorukova, S. Arishin. Link-time optimization speedup. Trudy ISP RAN/Proc. ISP RAS, vol. 28, issue 5, 2016, pp. 175-198 (in Russian). DOI: 10.15514/ISPRAS-2016-28(5)-11

References

[1]. Andrew Ayers, Richard Schooler, and Robert Gottlieb. Aggressive inlining. SIGPLAN Not., 32(5):134–145, 1997. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/258916.258928.
[2]. Honza Hubicka. Linktime optimization in GCC, part 2 – Firefox. http://hubicka.blogspot.ru/2014/04/linktime-optimization-in-gcc-2-firefox.html.
[3]. Honza Hubicka. Linktime optimization in GCC, part 3 – LibreOffice. http://hubicka.blogspot.ru/2014/09/linktime-optimization-in-gcc-part-3.html.
[4]. K. Dolgorukova. [Overview of Scalable Frameworks of Cross-Module Optimization]. Trudy ISP RAN/Proc. ISP RAS, vol. 26, issue 3, 2014, pp. 69-90 (in Russian). DOI: 10.15514/ISPRAS-2014-26(3)-3.
[5]. K. Dolgorukova. [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.
[6]. Satu Elisa Schaeffer. Graph clustering. Survey. Computer Science Review. Volume 1, Issue 1, August 2007, Pages 27–64. DOI:10.1016/j.cosrev.2007.05.001.
[7]. M. Girvan, M. E. J. Newman. Community structure in social biological networks. PNAS, June 11, 2002, vol.99, no.12. DOI: 10.1073/pnas.122653799.
[8]. M. E. J. Newman. Scientific collaboration networks. II. Shortest paths, weighted networks, and centrality. Physical review E, vol. 64, 016132. DOI: 10.1103/PhysRevE.64.016132.
[9]. L. da F. Costa, F. A. Rodrigues, G. Travieso & P. R. Villas Boas (2007). Characterization of complex networks: A survey of measurements, Advances in Physics, 56:1, 167-242. DOI:10.1080/00018730601170527
[10]. M. E. J. Newman, M. Girvan. Finding and evaluating community structure in networks. Phys. Rev. E 69, 026113 (2004). arXiv:cond-mat/0308217 DOI:10.1103/PhysRevE.69.026113.
[11]. L. Danon, A. Díaz Guilera, J. Duch, A. Arenas, Comparing community structure identification, Journal of Statistical Mechanics Theory and Experiment (2005) P09008.
[12]. D. A. Bader, H. Meyerhenke, P. Sanders, D. Wagner. Contemporary mathematics: graph partitioning and graph clustering. 10th DIMACS Implementation Challenge Workshop, February 13–14, 2012, Georgia Institute of Technology, Atlanta, GA.
[13]. 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.
[14]. Gautam Chakrabarti, Fred Chow. Structure Layout Optimizations in the Open64 Compiler: Design, Implementation and Measurements. Open64 Workshop at CGO 2008, April 6, 2008. Boston, Massachusetts.
[15]. SPEC CPU benchmark. https://www.spec.org/cpu2000.