[10] *Nielson R., Nielson H., Hankin C.* Principles of Program Analysis. Springer-Verlag, 1999. 476 p.

[11] Prechelt L., Kramer G. Functionality versus practicality: Employing existing tools for recovering structural design patterns // Journal of Universal Computer Science, 4(12). 1998. P. 866–882.

[12] *Steensgaard B.* Points-to analysis in almost linear time // ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. 1996. P. 32–41.

[13] *Tip F.* A Survey of Program Slicing Techniques // Journal of Programming Languages. Vol. 3. N. 3. 1995. P. 121–189.

[14] *Ullman J. D.* Principles of Database and Knowledge-Base Systems. Vol. I and II. W.H. Freeman and Company, 1989.

[15] *Vivien F., Rinard M.* Incrementalized pointer and escape analysis // Conference on Programming Language Design and Implementation. 2001. P. 35–46.

[16] *De Volder K.* JQuery: A Generic Code Browser with a Declarative Configuration Language. University of British Columbia, Vancouver, Canada. http://www.cs.ubc.ca/labs/spl/projects/jquery. 15 p.

[17] *De Volder K.* Type-Oriented Logic Meta Programming. Ph. D. Thesis, Vrije Universiteit Brussel. http://tyruba.sourceforge.net. 226 p.

[18] *Wagner T., Graham S.* Incremental Analysis of Real Programming Languages // Proc. SIGPLAN Conference on Programming Language Design and Implementation. 1997. P. 31–43.

[19] *Whaley J., Lam M.* Cloning-based context-sensitive pointer alias analyses using binary decision diagrams // Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation. 2004. P. 131–144.

[20] *Yur J., Ryder B. G., Landi W.* An incremental flow- and context-sensitive pointer aliasing analysis // Proc. Twenty-First International Conference on Software Engineering. 1999. P. 442–451.

[21] *Zhang X., Gupta R., Zhang Y.* Efficient Forward Computation of Dynamic Slices Using Reduced Ordered Binary Decision Diagrams // IEEE/ACM International Conference on Software Engineering. 2004.

[22] *Zhu J., Calman S.* Symbolic pointer analysis revisited // Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation. 2004. P. 145–157.

Линеаризация графа потока управления с учётом результатов профилирования

О. В. Медведев
dours@tercom.ru

В данной работе рассматривается построение оптимальной линеаризации (линейного упорядочения вершин) графа потока управления с учётом результатов профилирования. Под профилированием в данном случае понимается подсчёт количества инструкций перехода, выполненных процессором на репрезентативном наборе тестов. Критерий оптимальности линеаризации — количество тактов, потраченное процессором на команды перехода при выполнении оптимизируемой программы на том же наборе тестов. Предложены два алгоритма — полиномиальный и перебор с отсечениями. Произведено их сравнение с известными алгоритмами по результатам линеризации нескольких стандартных программ.

Введение

Одним из важных средств ускорения выполнения инструкций в современных процессорах является конвейер (pipeline). Идея его использования заключается в разбиении процедуры выполнения команды на несколько стадий с целью их параллельного исполнения. В некоторых процессорах число стадий очень велико (например, 31 в Intel Pentium на ядре Prescott[1]). Этот метод используется не только в процессорах общего назначения, но и в DSP-процессорах.

Слабым местом такой схемы, как известно, являются условные переходы. В момент выборки инструкции условного перехода процессор не знает, произойдёт ли этот переход. Тем не менее на сле-

[1] http://en.wikipedia.org/wiki/Pentium_4.

© О. В. Медведев, 2006.

дующих тактах ему необходимо продолжить заполнение конвейера, поэтому он вынужден предсказать, произойдёт ли данный переход и на основании этого предсказания заполнять конвейер следующими инструкциями из соответствующей ветви. Если при выполнении инструкции перехода выясняется, что предсказание было неверным, процессор удаляет из конвейера все выбранные и дешифрованные инструкции, загруженные из ошибочно предсказанной ветви перехода, и начинает загрузку из другой ветви. При этом такты, потраченные на обработку удалённых инструкций, оказываются потерянными.

Чтобы избежать этих потерь, было изобретено множество аппаратных и программных методов предсказания адреса, на который произойдёт переход — как использующих, так и не использующих результаты профиляции.

Один из методов заключается в подборе оптимальной *линеаризации* графа потока управления — такого порядка вершин, при котором наиболее часто случающиеся переходы будут занимать меньше времени. Он применяется на этапе компиляции.

В данной работе рассмотрены алгоритмы линеаризации, учитывающие статистическую информацию, построенные на базе точного полиномиального алгоритма для дэгов, алгоритм, реализующий метод ветвей и границ и произведено их сравнение друг с другом и с чисто эвристическим «жадным» алгоритмом, а также произведена попытка их скомбинировать. Произведено также сравнение ответов данных алгоритмов с оптимальными на нескольких небольших входах, на которых удалось вычислить правильный ответ. Эти алгоритмы предназначены в основном для использования при компиляции для DSP-процессоров, так как во многих из них (в силу ограничений на стоимость и энергопотребление) блок предсказания правильного адреса перехода очень прост — условные переходы всегда предсказываются как непроисходящие. С другой стороны, как бы хорошо процессор ни предсказывал условные переходы, часть его времени отнимают безусловные переходы. Качественно выполнив линеаризацию, можно избавить процессор от выполнения части таких переходов, что выгодно в любом случае.

1. Постановка задачи

В данной работе задачи и утверждения формулируются для графов с исходящей степенью вершин, не превосходящей двух. Вершина с исходящей степенью больше двух соответствует структуре управления «switch», которая может быть реализована двумя способами.

- Путём преобразования её в несколько вершин с операторами вида «if-then-else»; такой граф будет удовлетворять ограничению на степень.

- Путём генерации таблицы переходов в паре с инструкцией перехода с косвенным операндом. В данной работе предполагается что выполнение такой инструкции всегда повлечёт сброс конвейера, поэтому делать исходящие из соответствующей вершины рёбра «провалами» смысла не имеет; таким образом, алгоритм линеаризации можно запускать на граф без всех этих рёбер.

Данное ограничение на граф взято из [3].

Тем не менее ни один из приведённых в работе алгоритмов не пользуется этим ограничением, а все данные ниже определения и утверждения можно обобщить на случай произвольного графа.

1.1. Некоторые обозначения и определения

Рассмотрим граф потока управления $G = (V, E)$ с весами на рёбрах, вершины которого — линейные участки программы, рёбра — возможные способы передачи управления между ними.

Введём несколько обозначений:

- $out(v)$ — множество рёбер, выходящих из вершины v;

- $w : E \to \mathbb{Z}$ — функция, задающая вес ребра;

- «условной вершиной» будем называть вершину, из которой выходит два ребра.

При генерации машинного кода по графу G мы можем расположить код, соответствующий вершинам графа, в любом порядке $(v_{k_1}, v_{k_2}, \ldots, v_{k_n})$, причём если ребро e соединяет вершины v_{k_i} и $v_{k_{i+1}}$, то оно в машинном коде никак не отражается — процессор и так перейдёт от последней инструкции участка v_{k_i} к первой $v_{k_{i+1}}$, поскольку выполняет код последовательно (если данное ребро соответствует условному переходу, то, возможно, придётся заменить

условие на отрицание, чтобы добиться такого эффекта). Такие рёбра (или ситуацию передачи управления по такому ребру) мы будет называть «провал» (перевод англ. «fallthrough»). Любое другое ребро отразится в коде инструкцией перехода, указывающей процессору на нетривиальную передачу управления в этом месте. Причём если появление инструкции условного перехода неизбежно, то количество и расположение инструкций безусловного перехода зависит *только* от выбранного расположения вершин. В случае, когда ребро, идущее между v_{k_i} и $v_{k_{i+1}}$, соответствует условному переходу, может оказаться выгодным сделать его *не* «провалом» (подробнее данная ситуация рассмотрена в разделе 3).

Определение 1. *Линеаризацией графа потока управления G назовём порядок, в котором код вершин будет вставлен в код генерируемой функции, считая, что условия всех условных переходов фиксированы.*

Для данной работы сам по себе порядок вершин не важен. Важно лишь множество рёбер, которые будут соответствовать «провалам» в коде. Пусть даны линеаризация L и ребро e. Предикат «ребро e является "провалом" в линеаризации L» мы будем обозначать как $e \in L$.

Определение 2. *Простым путём в графе G назовём последовательность вершин $p = (v_1, v_2, \ldots, v_k)$, такую что $\forall i : 1 \leq i \leq (k-1)$ $(v_i, v_{i+1}) \in E$ и ни одна вершина в последовательности не повторяется дважды. v_1 называется началом пути, v_k — концом.*

Определение 3. *Покрытием путями графа G будем называть множество попарно вершинно-непересекающихся простых путей, проходящих через все вершины графа.*

Покрытие графа путями также задаёт набор «провалов». Пусть

$$P = \{p_1, p_2, \ldots, p_l\}$$

некоторое покрытие путями графа G, где

$$p_i = (v_{i,1}, v_{i,2}, \ldots, v_{i,k_i}).$$

Покрытие задаёт набор «провалов» — множество всех рёбер, покрытых путями. Очевидно, оно задаёт частичный порядок на вер-

шинах, в соответствии с которым можно построить линеаризацию, содержащую все «провалы», заданные покрытием, например, $(v_{1,1}, v_{2,1}, \ldots, v_{k_1,1}, v_{2,1}, v_{2,2}, \ldots)$. Кроме того, оно фиксирует условия для тех условных переходов, одно из рёбер которых оказывается покрытым. Интересно отметить, что множество «провалов», заданное линеаризацией, полученной по покрытию, может оказаться строго больше множества, заданного покрытием. Например, так наверняка окажется, если $P = \{(v_1), (v_2), \ldots, (v_n)\}$, где $\{v_1, v_2, \ldots, v_n\}$ — множество вершин графа. Однако по любой линеаризации можно построить покрытие путями, задающее ровно то же множество «провалов», поэтому определять оптимальное множество «провалов» можно как в терминах линеаризаций, так и в терминах покрытий. Большинство статей на тему оптимизации переходов посвящены именно поиску оптимального покрытия путями. Данная работа не является исключением.

1.2. Постановка задачи

Дан граф потока управления G. Дана w — функция весов рёбер. Вес ребра e равен суммарному количеству передач управления по данному ребру при выполнении программы на репрезентативном наборе тестов.

Фиксированы следующие константы:

- a — количество тактов, которое процессор тратит на выполнение условного перехода в случае, если условие оказывается истинным (включая возможный сброс конвейера);
- b — то же самое в случае ложности условия;
- c — количество тактов, которое тратится на выполнение безусловного перехода.

Определим целевую функцию, задающую количество тактов, которое процессор потратил бы на выполнение инструкций перехода, если бы выполнял программу, сгенерированную по линеаризации L графа $G = (V, E)$, на том же наборе тестов, по которому была подсчитана весовая функция w:

$$Penalty(G, w, L) = \sum_{v \in V} Penalty(v), \tag{1}$$

где значение $Penalty(v)$ вычисляется следующим образом:

- если $out(v) = \emptyset$, то $Penalty(v) = 0$;
- если $out(v) = \{e\}$ и $e \in L$, то $Penalty(v) = 0$;
- если $out(v) = \{e\}$ и $e \notin L$, то $Penalty(v) = c \cdot w(e)$;
- если $out(v) = \{e_1, e_2\}$ и $e_2 \in L$, то

$$Penalty(v) = a \cdot w(e_1) + b \cdot w(e_2);$$

- если $out(v) = \{e_1, e_2\}$ и $e_1, e_2 \notin L$, то

$$Penalty(v) = \min(a \cdot w(e_1) + (b+c) \cdot w(e_2), (b+c) \cdot w(e_1) + a \cdot w(e_2)).$$

Последний случай соответствует условной вершине, ни одно из исходящих рёбер которой не попало в линеаризацию. В машинном коде это будет реализовано двумя инструкциями — по одному ребру управление передаст инструкция условного перехода, а по второму — следующая за ней инструкция безусловного перехода — отсюда и множитель $(b + c)$. Штраф в этом случае — это минимум из штрафов в двух возможных случаях:

- условие перехода заменили на его отрицание, адреса переходов поменяли местами;
- использовали исходное условие.

Данная работа посвящена поиску линеаризации с *минимальным* значением $Penalty$ или близким к нему.

Следует отметить два частных случая этой задачи.

1. $a = c = 1, b = 0$ — минимизация общего количества переходов, неважно каких. Основной плюс такой постановки — сильное упрощение вида целевой функции:

$$Penalty(G, w, L) = \sum_{e \notin L} w(e).$$

2. $a = c = 1$, $b = 0$, $\forall e \in E\, w(e) = 1$ — ещё бóльшее упрощение. Случай, когда статистика выполнения отсутствует и все рёбра считаются одинаково важными. Целевая функция преобразуется к такому виду:

$$Penalty(G, w, L) = |\{e \notin L\}| = |E| - |\{e \in L\}|.$$

То есть ищется покрытие, содержащее как можно больше рёбер.

Заметим также, что фиксация констант a, b, c означает, что данная задача нацелена в основном на применении для процессоров без блока предсказания условного перехода.

2. Обзор существующих решений

Историю рассматриваемой задачи можно разделить на два периода — до появления процессоров с длинными конвейерами и после. Они отражены в первых двух подразделах.

2.1. Невзвешенный случай

В начале 80-х годов рассматривался в основном второй частный случай приведённой выше формулировки. Его решение позволяет получить покрытие, содержащее максимальное количество рёбер, и, соответственно, код с минимальным числом инструкций безусловного перехода.

Про эту задачу известно следующее:

- покрытие, содержащее максимальное суммарное количество рёбер, содержит минимальное количество путей;
- задача поиска покрытия с минимальным количеством путей NP-трудна, поскольку к ней сводится задача поиска гамильтонова пути;
- та же задача для дэгов сводится к поиску максимального паросочетания в двудольном графе (сведение приведено в разделе 2.3.).

Вышеперечисленные факты приведены и обоснованы в [2].

Кроме того, существует алгоритм, решающий задачу для графа, полученного из структурной программы при помощи динамического программирования за $O(E)$. К сожалению, алгоритм предназначен для слишком узкого класса графов. Он приведён в [6].

2.2. Общий случай

С появлением процессоров с длинными конвейерами встала проблема условных переходов. Были разработаны разнообразные эври-

стические алгоритмы, предназначенные для предсказания процессором целевого адреса условного перехода [8, 7]. Кроме того, стали появляться методы, призванные помочь компилятору *подсказать* процессору наиболее частое направление перехода, опираясь на собранную статистическую информацию. Это привело к появлению методов линеаризации *взвешенных* графов с учётом разницы в стоимости выполнения разных инструкций перехода в разных случаях.

В начале рассматривался первый частный случай общей задачи. Оптимальная линеаризация в этом случае задаётся покрытием графа путями с максимальным суммарным весом (оно, очевидно, доставляет минимум целевой функции, равной сумме весов рёбер, не являющихся «провалами»).

В [5] приведён жадный алгоритм нахождения такого покрытия. Он начинает работу с покрытия, в котором каждая вершина является отдельным путём. Рёбра перебираются в порядке убывания веса. Очередное ребро (u, v) добавляется в покрытие, если на текущем шаге u является концом какого-либо пути, а v — началом другого пути. Два соответствующих пути объединяются в один.

В статье [3] приведены два эвристических алгоритма, решающие основную задачу, поставленную в данной работе в столь же общей формулировке. Интересно, что, несмотря на напрашивающуюся аналогию с задачей поиска покрытия путями максимального суммарного веса, авторы [3], судя по всему, не пытались свести общую задачу к ней. Они модифицировали жадный алгоритм так, чтобы на очередном шаге он проверял целесообразность добавления очередного ребра согласно целевой функции. При этом рёбра всё равно перебирались в порядке убывания исходных весов. Они также предложили алгоритм, основанный на переборе перестановок вершин. Предлагалось взять 15 самых тяжёлых рёбер, перебрать все перестановки вершин, им инцидентных, после чего к полученной части линеаризации добавить лучшую перестановку вершин для 15 следующих рёбер, и так далее.

Кроме того, в [3] была предложена очень хорошая эвристика для ускорения работы переборных алгоритмов. Перед началом перебора предлагалось удалить из графа набор самых лёгких рёбер, сумма весов которых не превосходит 0,001% от суммарного веса всех рёбер.

В существующих работах не было обнаружено попытки воспользоваться полиномиальным алгоритмом для дэгов с целью приближённо решить задачу.

2.3. Сведение задачи покрытия дэга к поиску паросочетания

В [2] было приведено сведение задачи поиска покрытия путями с максимальным суммарным количеством рёбер в дэге к поиску максимального по мощности паросочетания в двудольном графе. Далее будет приведено сведение поиска покрытия путями с максимальным суммарным весом в дэге к поиску паросочетания максимального веса в двудольном взвешенном графе. При переходе ко взвешенному случаю доказательство корректности сведения практически не меняется.

Дан граф $G = (V, E)$ и весовая функция $w : E \to \mathbb{Z}$. Построим двудольный граф G' с долями L и R, множеством рёбер E', весовой функцией w'. Для каждой вершины $v \in V$ заведём две вершины $v^l \in L$ и $v^r \in R$. Каждому ребру $(u, v) \in E$ сопоставим (u^l, v^r), $w'((u^l, v^r)) = w((u, v))$. В полученном графе найдём паросочетание максимального веса. В покрытие путями исходного графа добавим те рёбра, образы которых принадлежат паросочетанию.

3. Сведение общей задачи к задаче покрытия путями

В данном разделе будет показана несостоятельность двух самых очевидных сведений, приходящих на ум первыми, а также будет предъявлено и доказано корректное сведение.

3.1. Обоснование необходимости сведения

Заметим, что для решения первого частного случая общей задачи достаточно найти покрытие графа путями с максимальным суммарным весом (это очевидно из вида соответствующей целевой функции).

Покажем, что такой способ не даёт оптимального ответа относительно целевой функции, заданной равенствами (1). Контрпример приведён на рис. 1. Допустим, $a = 4$, $b = 1$, $c = 1$. Алгоритм поиска покрытия максимального веса покроет оба ребра $(S, 3)$ и $(1, 2)$, при этом значение целевой функции будет равно

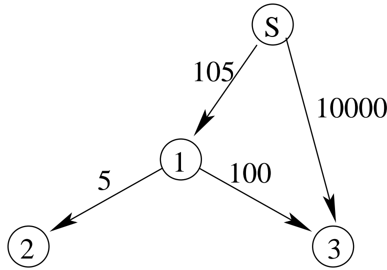

Рис. 1. Пример, для которого покрытие путями
с максимальным суммарным весом не даёт оптимального ответа

$10000 + 4 \cdot 105 + 5 + 4 \cdot 100 = 10825$. Если же покрыть *только* ребро $(S, 3)$, штраф будет $10000 + 4 \cdot 105 + \min(4 \cdot 100 + 2 \cdot 5, 4 \cdot 5 + 2 \cdot 100) = 10640$, что меньше.

На этом же примере видно, что недостаточно просто домножить веса всех рёбер, выходящих из «условных» вершин, на a, а остальных — на c и применить поиск покрытия максимального веса (хотя на первый взгляд кажется, что такое сведение отражает всю разницу между условными и безусловными переходами).

В связи с этим мы рассмотрим несложное преобразование, сводящее общую задачу к задаче поиска покрытия путями максимального веса.

3.2. Описание сведения

Пусть дан граф G с весовой функцией w. Построим по нему граф G' по следующему правилу (далее все штрихованные объекты будут относиться к штрихованному графу):

- вершине $v : out(v) = \emptyset$ сопоставим $v' : out(v') = \emptyset$;
- $v : out(v) = \{e\}$, $w(e) = x$ сопоставим

$$v' : out(v') = \{e'\}, w'(e') = c \cdot x;$$

- в случае $v : out(v) = \{e_1, e_2\}$, $w(e_1) = w_1$, $w(e_2) = w_2$ введём следующие обозначения:

$$\begin{aligned} \alpha_1 &= a \cdot w_1 + b \cdot w_2; \\ \alpha_2 &= b \cdot w_1 + a \cdot w_2; \\ \beta &= \min(a \cdot w_1 + (b + c) \cdot w_2,\ (b + c) \cdot w_1 + a \cdot w_2). \end{aligned} \tag{2}$$

Сопоставим v вершины v' и v'', как показано на рис. 2. В дальнейшем будем называть v'' «фиктивной» вершиной, а (v', v'') — «фиктивным» ребром.

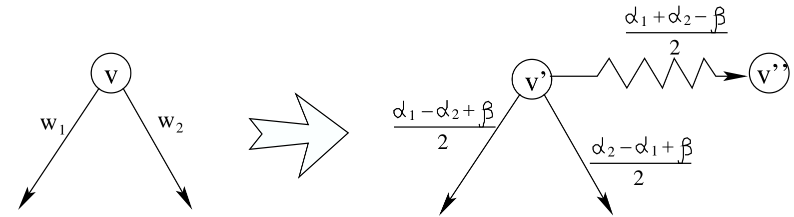

Рис. 2. Сведение задачи минимизации штрафа
к задачи поиска максимального путевого покрытия

Заметим, что от такого преобразования количество вершин возрастает не больше чем в два раза, количество рёбер — не больше чем наполовину.

Заметим, что «фиктивным» ребром путь в G' может только заканчиваться, поэтому удаление таких рёбер практически не повлияет на структуру путевого покрытия. В частности, не изменится количество путей.

В G' ищется покрытие путями максимального суммарного веса, а покрытие исходного графа получается из него путём удаления всех «фиктивных» вершин и рёбер и перехода к прообразам.

3.3. Корректность сведения

Теорема 1. *Если $a \geq c$, то оптимальное покрытие путями G получается из покрытия путями максимального суммарного веса G' в результате удаления из его путей «фиктивных» вершин и рёбер.*

Замечание 1. *Ограничение $a \geq c$ весьма естественно — вряд ли на каком-то процессоре безусловный переход будет выполняться дольше, чем неверно предсказанный условный.*

Доказательство.

1. Заметим что

$$\begin{aligned}
&a \geq c \Rightarrow \\
&a \cdot w_1 + b \cdot w_2 \geq c \cdot w_1 \Rightarrow \\
&(a+b) \cdot (w_1 + w_2) \geq a \cdot w_2 + (b+c) \cdot w_1 \Rightarrow \\
&(a+b) \cdot (w_1 + w_2) \geq \min(a \cdot w_2 + (b+c) \cdot w_1, a \cdot w_1 + \\
&+ (b+c) \cdot w_2) \Rightarrow \frac{\alpha_1 + \alpha_2 - \beta}{2} \geq 0,
\end{aligned}$$

где последняя дробь — вес «фиктивного» ребра.

2. Введём целевую функцию

$$Penalty'(G', w', L') = \sum_{e' \notin L'} w'(e').$$

Её минимизация соответствует поиску покрытия G' путями с максимальным суммарным весом.

Докажем что тогда для любого L' — покрытия путями G' такого, что

$$\forall v' \in V' : |outs(v')| = 3 \;\; \exists e \in outs(v') : e \in L', \tag{3}$$

выполнено равенство

$$Penalty'(G', w', L') = Penalty(G, w, L),$$

где L — покрытие, полученное из L' удалением «фиктивных» рёбер и вершин.

Очевидно, функцию $Penalty'$ можно записать как

$$Penalty'(G', w', L') = \sum_{v' \in V'} \sum_{e' \in outs(v') : e' \notin L'} w'(e'). \tag{4}$$

В таком случае обе штрафные функции являются суммами по всем вершинам некоторых величин.

Мы докажем, что эти суммы равны покомпонентно. Чтобы доказать последнее утверждение, рассмотрим произвольную вершину $v \in V$. Если из неё не выходит ни одного ребра, то она вносит нулевой вклад в обе суммы. В случае одного ребра e вклад равен $c \cdot w(e)$, если e содержится в линеаризации, либо 0 в противном случае, что очевидно.

Пусть теперь $out(v) = \{e_1, e_2\}$, $out(v') = \{e'_1, e'_2, e'_3\}$.

Рассмотрим три случая:

(a) $e_1 \in L$. Значит $e'_1 \in L'$; соответственно вклад v' в $Penalty'(\dots)$ равен

$$\frac{\alpha_1 + \alpha_2 - \beta}{2} + \frac{\alpha_2 - \alpha_1 + \beta}{2} = \alpha_2 = a \cdot w_2 + b \cdot w_1;$$

(b) $e_2 \in L$ — полностью аналогично предыдущему.

(c) $e_1 \notin L$ и $e_2 \notin L$. Тогда, согласно (3), $e'_3 \in L'$. В таком случае вклад v' в $Penalty'(\dots)$ равен

$$\frac{\alpha_1 - \alpha_2 + \beta}{2} + \frac{\alpha_2 - \alpha_1 + \beta}{2} = \beta,$$
$$\beta = \min(a \cdot w_2 + (b+c) \cdot w_1,\; a \cdot w_1 + (b+c) \cdot w_2),$$

что, в свою очередь, равно вкладу вершины v в $Penalty(\dots)$.

3. Рассмотрим покрытие L' с минимальным значением $Penalty'$. Каждый путь, который заканчивается в вершине v' : $|outs(v')| = 3$ продлим «фиктивным» ребром, выходящим из этой вершины (очевидно, это всегда можно сделать). Назовём это покрытие L'_1. Поскольку, как было доказано в начале теоремы, веса «фиктивных» рёбер неотрицательны, $Penalty'(G', w', L') \geq Penalty'(G', w', L'_1)$. Поскольку L' минимально, имеет место и неравенство в другую сторону, а значит, и равенство. Покрытия графа G, получающиеся из L' и L'_1, будут одинаковы. Обозначим их оба через L. Поскольку L'_1 удовлетворяет (3), $Penalty(G, w, L) = Penalty'(G', w', L'_1)$, а $Penalty'(G', w', L'_1) = Penalty'(G', w', L')$. Значит, покрытие, полученное из покрытия G' с максимальным суммарным весом рёбер, будет оптимальным.

□

4. Сравниваемые алгоритмы

В разделе 3 мы свели наш случай задачи поиска оптимальной линеаризации обратно к поиску максимального по весу покрытия путями во взвешенном графе. Далее будут перечислены приближённые алгоритмы решения этой задачи, реализованные в данной работе.

4.1. «Жадный» алгоритм

Самый простой из всех, взят из [3]. Начнём с покрытия, в котором каждая вершина является отдельным путём. Будем пытаться добавить в это покрытие все рёбра графа в порядке убывания весов. Очередное ребро добавляется, если оно соединяет конец одного пути с началом другого. При этом пути объединяются.

4.2. Взвешенное паросочетание

Покрытие путями графа без циклов сводится к поиску паросочетания в двудольном графе. Простейший способ сведения задачи на произвольном графе к задаче на графе без циклов — построить дерево поиска в глубину из стартовой вершины и выкинуть все обратные рёбра. В качестве ответа выдать покрытие путями полученного графа.

4.3. Взвешенное паросочетание, дополненное «жадным» алгоритмом

Это усовершенствование алгоритма из раздела 4.2, которое предлагает «жадному» алгоритму начинать не с пустого покрытия (как это делает алгоритм из раздела 4.1.), а с построенного алгоритмом из раздела 4.2.

Мотивировка использования данного алгоритма такова: часто встречаются циклы, подобные циклу на рис. 3 — с очень разветвлённым телом с рёбрами среднего веса и одним-двумя обратными рёбрами с огромным весом. Алгоритм, использующий только паросочетания, выкинет эти рёбра, чем сильно ухудшит ответ. Естественно попытаться добавлять такие рёбра к уже имеющемуся оптимальному покрытию графа без циклов.

Алгоритм использовался в двух вариантах:

- добавляем очередное ребро, если это приносит локальное

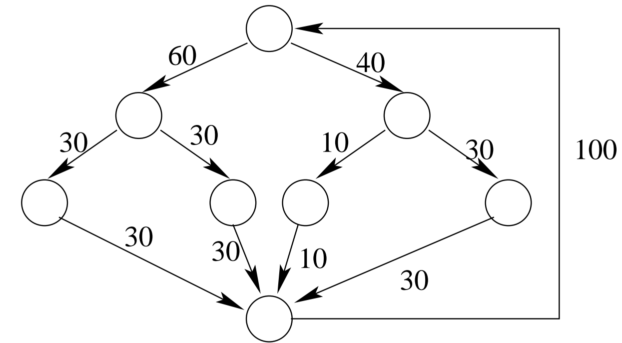

Рис. 3. Цикл с «тяжёлым» обратным ребром

улучшение и добавить его в текущее покрытие путями возможно;
- добавляем, если необходимо поменять часть путей (возможно удалить/добавить новые), но после изменений общий вес покрытия возрастёт.

Последний вариант будем условно называть агрессивным.

4.4. Невзвешенное паросочетание с дополнением

Этот алгоритм работает на исходном графе (без «фиктивных» рёбер). Сначала строится линеаризация без учёта весов тем же сведением к паросочетаниям, что и во взвешенном паросочетании (см. раздел 4.2). Очевидно, что такая линеаризация окажется неконкурентоспособной по сравнению со взвешенными алгоритмами, поскольку она не учитывает частот выполнения условных инструкций, а штраф за «лишнее» выполнение условного перехода мы предполагаем достаточно большим. Поэтому происходит её «доводка» — инструкция

```
          jumpIfCondition label;
          <else code>
          .....
label:
          <then code>
```

заменяется на

```
            jumpIfNotCondition label2;
            jmp label;
    label2:
            <else code>
            .....
    label:
            <then code>
```

если это выгодно. То есть сама линеаризация не меняется, а процессору таким образом подсказывают, на какую ветвь чаще происходит переход. Фактически, это способ подсказать процессору наиболее вероятную ветвь перехода.

Сравнение с данным алгоритмом призвано показать, что использование статистической информации полезно не только для того, чтобы подсказать процессору более частое направления перехода, но и именно при линеаризации.

4.5. Ветви и границы с отсечением по времени

Этот алгоритм осуществляет перебор по подмножествам множества всех рёбер при помощи широко известной рекурсивной процедуры перебора всех подмножеств некоторого множества. Отсекаются ветви перебора, в которых ни одно подмножество заведомо не окажется покрытием путями или верхняя оценка на ответ в которых не больше текущего максимума.

Верхняя оценка определяется следующим образом: предположим, что мы уже покрыли какими-то путями часть вершин, а U — множество ещё не покрытых. Тогда в качестве верхней оценки на ответ можно использовать выражение

$$\sum_{v \in U} \max_{e_i \in outs(v)} w_i. \tag{5}$$

Перебор ведётся на массиве рёбер, отсортированных по убыванию веса (внешний уровень рекурсии соответствует самому тяжёлому ребру). Таким образом он начинает с ответа, который дал бы жадный алгоритм.

Также оказалось полезным не учитывать при переборе часть рёбер, общий вес которых в исходном графе не превосходил 0,0001% от суммарного веса всех рёбер. Данная эвристика (взятая из [3]) нацелена на удаление из рассмотрения рёбер с нулевым или другим очень маленьким весом, которых может оказаться достаточно много.

К сожалению, несмотря на все отсечения, дождаться завершения перебора на больших функциях ни разу не удавалось, поэтому его работа останавливается через некоторое время и в качестве ответа берётся текущий максимум.

Таким образом, это скорее следует назвать не перебором, а локальным поиском в районе ответа «жадного» алгоритма.

5. Реализация

Все вышеперечисленные алгоритмы были реализованы в качестве модулей библиотеки PRANLIB[2] на языке OCaml[3]. Также эта библиотека была интегрирована с проектом CIL (C Intermediate Language)[4] — фронт-эндом языка C, также написанном на OCaml, что позволило запускать алгоритмы на нетривиальных программах.

Было написано несколько скриптов на языках bash и perl для автоматизированного запуска разных алгоритмов линеаризации на разных программах с разными параметрами и генерации по полученным данным таблиц и гистограмм.

6. Результаты экспериментов

Тестирование было произведено на нескольких обычных программах. Дождаться завершения метода ветвей и границ на них не удалось. Кроме того, специально с целью убедиться в качестве ответов, выдаваемых методом ветвей и границ с отсечением по времени, было произведено тестирование на трёх программа, реализующих небольшие по размеру кода алгоритмы. Для них были найдены оптимальные покрытия.

6.1. Программы, точный ответ для которых найти не удалось

В данном разделе приведены таблички с результатами экспериментов для следующих программ:

[2] http://oops.tepkom.ru/projects/pranlib.
[3] http://caml.inria.fr/.
[4] http://manju.cs.berkeley.edu/cil/.

- `gzip-1.2.4` — известная программа для сжатия файлов без потерь;
- `oggdec` — программа преобразования сжатого музыкального файла в формате «ogg-vorbis» в несжатый .wav файл из пакета `vorbis-tools-1.1.1`. Оптимизировалась вместе с соответствующим библиотечным кодом (библиотеки `libvorbis-1.1.2` и `libogg-1.1.3`);
- `UnitWalk` — SAT–solver.

Сравниваются следующие алгоритмы:

- ag — простой «жадный» алгоритм (см. раздел 4.1);
- wpm — взвешенные паросочетания (см. раздел 4.2);
- wpmalg — взвешенные паросочетания, дополненные агрессивным «жадным» алгоритмом (см. раздел 4.3, второй вариант);
- obf — алгоритм ветвей и границ (см. раздел 4.5), в котором обработку каждого графа отводится не более $0.1 \times n$ секунд, где n — это число рёбер в графе;
- obfalg — obf, дополненный агрессивным «жадным» алгоритмом;
- uwpmf — невзвешенные паросочетания (см. раздел 4.4);
- best — результат лучшего алгоритма на данной функции.

Строкам таблицы соответствуют функции, обладающие 99% суммарного веса рёбер после профиляции. Последняя строка — суммы результатов алгоритмов по всем этим функциям. Имена функций приведены не полностью, поскольку они зачастую оказываются слишком длинными. Кроме того, они не столь важны.

Эксперименты проводились на процессоре Athlon64 3000+ — константа 0.1 для алгоритмов obf и obfalg была подобрана именно для него, поскольку все изменения в оптимальном ответе перебор обычно делал именно за это время. Дальше он мог работать часами, не трогая текущий максимум.

В таблицах для каждого алгоритма и функции указано значение штрафа, определённого равенством 1 для линеаризации, построенной данным алгоритмом для данной функции. Для алгоритма «ag»

указано абсолютное значение. Для остальных — разница в процентах с «ag». Таким образом, можно примерно представить, насколько «умные» алгоритмы оказались лучше самого прямолинейного.

Алгоритмы оценивались в двух весовых моделях, параметры которых приведены в таблице 1. Модель «Fallthrough» соответствует процессору, не имеющему блока предсказания адреса условного перехода. Переход всегда предсказывается как непроисходящий. Модель «Dynamic», наоборот, соответствует процессору с совершенным блоком предсказания — условный переход приводит к такому же штрафу, что и безусловный.

Таблица 1. **Использованные весовые модели**

Штраф в тактах за переход	Fallthrough	Dynamic
условный произошедший (a)	8	2
условный непроизошедший (b)	1	1
безусловный (c)	2	2

Сводная гистограмма по таблицам результатов приведена на рис. 4. Миниатюрную разницу между «obf», «obfalg» и «best» она уловить не позволяет.

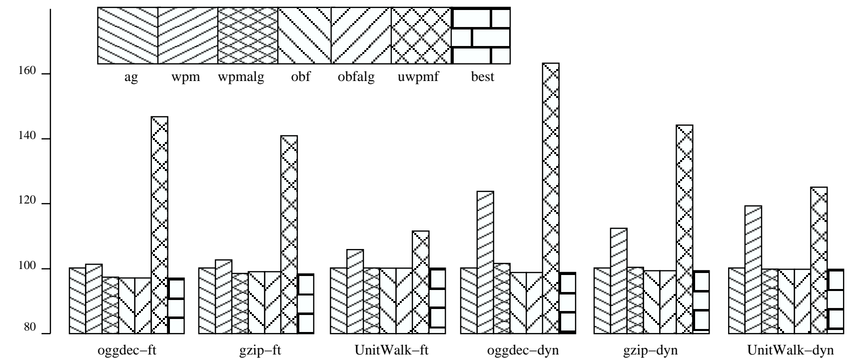

Рис. 4. Результаты работы алгоритмов на больших программах. Для каждого алгоритма показано отношение значения целевой функции на ответе данного алгоритма к значению, полученному жадным алгоритмом, в процентах. Суффикс «ft» в названии программы соответствует результатам для модели «Fallthrough», «dyn» — для «Dynamic»

6.2. Программы с вычисленными точными ответами

Были подобраны несколько программ специально для того чтобы посчитать для них оптимальную линеаризацию перебором. Их список таков:

- `fib` — библиотека, реализующая фибоначчиеву кучу — довольно сложный вариант очереди с приоритетами + программа, осуществляющая добавление в кучу случайных чисел и удаление минимума (самый крупный граф — 49 рёбер);
- `KthNumber` — программа, решающая задачу K с соревнования NEERC2005 (73 ребра максимум);
- `linprog` — решение задач линейного программирования в пространствах малой размерности (39 рёбер максимум).

Гистограмма приведена на рис. 5. Точному ответу, посчитанному на графе без рёбер с суммарным весом 0,0001% от общего веса графа, соответствует столбец «precise».

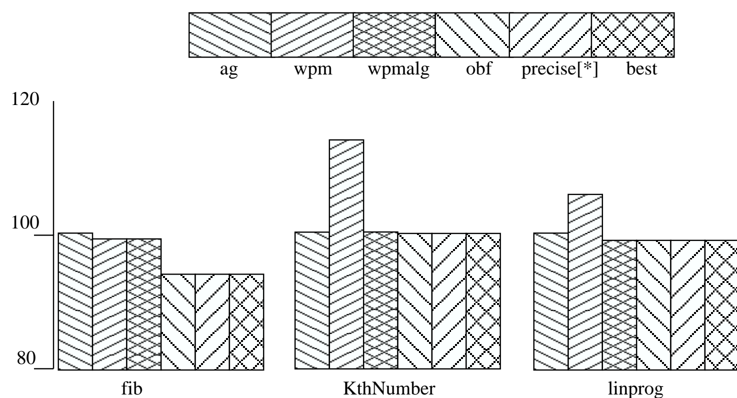

Рис. 5. Результаты работы алгоритмов на программах с небольшими функциями. Для каждого алгоритма показано отношение значения целевой функции на ответе данного алгоритма к значению, полученному жадным алгоритмом, в процентах

7. Анализ результатов

Первое, самое бесспорное замечание: сравнивая работу «uwpmf» с другими алгоритмами, можно заметить, что учёт статистической информации именно при линеаризации, а не только в «подсказывании» процессору правильного направления условного перехода, даёт существенное улучшение. Это видно из результатов «uwpmf» как на модели «Fallthrough» (порядка 40% ухудшение в двух программах), так и на модели «Dynamic» (16%).

По обратным рёбрам действительно очень часто передаётся управление, и это приводит к плохим результатам алгоритма «wpm», основанного на разрыве циклов за счёт обратных рёбер и точном решении задачи для полученного графа без циклов. Он практически не оказывался лучше жадного. Дополнение его «жадным» алгоритмом, пытающимся добавить в том числе и обратные рёбра к оптимальному для дэга ответу существенно улучшает алгоритм — он оказался хуже «жадного» только на 3 функциях из обработанных 69.

Судя по всему, большинство обратных рёбер соответствуют безусловным переходам, поскольку «wpm» очень плохо работает именно на модели «Dynamic», которая предназначена именно для их оптимизации. Для штрафа за выполненный условный переход в 3 вместо 2 тактов «wpm» на «oggdec» работал только на 5% хуже «ag».

Типичная разница между «obf» и «obfalg» — порядка 0.01%. Результаты работы «obf» совпадали с точными ответами в случаях, когда их удавалось получить. Все изменения в оптимальном ответе происходят в первые секунды работы «obf». Из 69 функций «wpmalg» обогнал «obf» 10 раз, в среднем на 0,3228%, максимум — на 1,903%. «obf» обогнал «wpmalg» 27 раз в среднем на 3,35%, максимум на 11,51%. Он оказался лучше жадного в среднем на 3,12%. Всё это — аргументы в пользу того, что «obf» в среднем даёт практически точный ответ и не делает резких «скачков» от него на «неудачных» входных данных.

Средняя разница между «жадным» алгоритмом и точным ответом на функциях, где он известен, оказалась 5,527%, максимальная — 10,392%.

Заключение

В данной работе исследовалась задача минимизации затрат процессора на выполнение инструкций перехода за счёт оптимальной линеаризации графа потока управления с учётом информации о частоте выполнения переходов, собранной в результате выполнения программы на наборе тестов.

Было приведено и обосновано сведение данной задачи к задаче покрытия графа набором вершинно-непересекающихся простых путей максимального суммарного веса, предложено два приближённых алгоритма, реализованных в рамках библиотеки анализа потока управления. Было произведено сравнение предложенных алгоритмов с известным «жадным» алгоритмом на нескольких программах, а также сравнение с точными оптимальными ответами на программах с небольшими функциями.

Окончательный вывод таков: «жадный» алгоритм даёт неплохие по оптимальности ответы, при этом очень прост в реализации. При необходимости получить ещё более точный ответ следует пользоваться методом ветвей и границ с отсечением по времени, возможно, дополняя его более интеллектуальными эвристиками, чем в данной работе. Небольшое дополнительное улучшение может дать запуск как перебора, так и полиномиального алгоритма и использование лучшего из двух ответов.

Список литературы

[1] *Касьянов В. Н., Евстигнеев В. А.* Графы в программировании: обработка, визуализация и применение. СПб.: БХВ-Петербург, 2003. 1104 с.

[2] *Boesch F. T., Gimpel J. F.* Covering the points of a digraph with point-disjoint pahs and its application to code optimization // Journal of the Association for Computing Machinery. Vol. 24. No 2. April 1977. P. 192–198.

[3] *Calder B., Grunwald D.* Reducing branch costs via branch alignment // Proc. 6th Intl. Conference on Architectural Support for Programming Languages and Operating Systems. October 1994. P. 242–251.

[4] *Hirata T., Maruoka A., Kimura M.* A polynomial time algorithm to find a path cover of a reducible flow graph // Syst. Comput. Control. Vol. 10. No. 3. 1979. P. 71–78.

[5] *Pettis K., Hansen R. C.* Profile guided code positioning // Proc. ACM SIGPLAN '90, Conference on Programming Language Design and Implementation. 1990. P. 16–27.

[6] *Ramanath M. V. S, Solomon M.* Jump minimization in linear time // ACM TOPLAS. Vol. 6. No. 4. 1984. P. 527–545.

[7] *Ramirez A., Larriba-Pey J. L., Valero M.* Branch prediction using profile data // Proc. Euro-Par. 2001. P. 386–393.

[8] *Hwu Wen-mei W., Conte T. M., Chang P. P.* Comparing Software and hardware schemes for reducing the cost of branches // Proc. 16th Annual International Symposium on Computer Architecture. 1989. P. 224–233.