1 Введение

На этапе компиляции любой программы неизбежно возникает проблема распределения регистров. Многие работы рассматривают эту проблему, мы же сфокусируемся на решениях предложенных в работах Чайтина (Chatin), Бриггса (Briggs), Полетто (Poletto).

Существует различные подходы к решению этой проблемы, в рассматриваемых работах есть два подхода. Первый, через построение графа, этот подход был предложен еще до работ Чайтина в работах Кокка (Cocke) в 1980, Ершова в 1971, Шварца (Schwartz) в 1973, однако именно в работе Чайтина в 1980 была предложена реализация этой идеи. Второй же подход это линейная аллокация, то есть для распределения регистров используется не граф, а линейное представление кода при помощи времени жизни переменных. Это упрощает работу за счет не всегда верного определения хроматического числа.

2 Постановка задачи

В процессе компиляции программа, представленная в виде промежуточного представления (IR), предполагает наличие неограниченного числа регистров. Однако реальная архитектура процессора предоставляет ограниченное их количество. Это ставит перед компилятором задачу корректного распределения переменных по имеющимся регистрах. При этом важно не только обеспечить правильность выполнения программы, но и минимизировать количество обращений к памяти для повышения её производительности.

Для понимания задачи рассмотрим пример: программа использует три регистра, в то время как архитектура процессора предусматривает лишь 2 регистра. В этом случае однозначно распределить переменные по регистрам не получится. Возникает необходимо проводить выгрузку (в англоязычной литературе *spill*) какой-то из переменных в память. Таким образом, при обращении к этой переменной потребуется загрузить её из памяти, а при изменении — снова сохранить обратно.

Таким образом, перед компилятором возникает необходимость выбора, какие из переменных следует выгрузить в память. Этот выбор напрямую влияет на производительность программы: частые обращения к памяти могут существенно замедлить её исполнение. Поэтому задача распределения регистров включает не только обеспечение корректной работы, но и поиск оптимального решения, минимизирующего влияние выгрузок.

3 Определение сложности задачи

Для того чтобы показать, что задача является NP полной необходимо показать что задача сводится к некоторой NP полной задачи, а так же показать что некоторая NP полная задача сводится к этой задаче. Нетрудно показать что проблема распределение регистров сводится к раскраске графа.

> **Определение.** Будем говорить что переменная *жива* в момент τ, если существуют моменты в программе τ_{def} и τ_{use} такие, что $\tau_{\text{def}} \leq \tau \leq \tau_{\text{use}}$ и в момент τ_{def} происходит объявление переменной, а в момент τ_{use} происходит использование переменной. И из точки программы τ исполнение может дойти до точки τ_{use}.

> **Определение.** Будем говорить что переменные a и b *интерферируют*, если одна из переменных жива в момент объявления другой переменной и их значения не совпадают.

Граф помех будет строиться следующим образом.

1. Все переменные исходного кода являются вершинами в этом графе.
2. В графе существуют ребро (a, b), если переменные a и b интерферируют в исходном коде.

Для простоты будем считать что никакие значения переменных в наших примерах не совпадают

Рассмотрим простой пример:

1. def a
2. def b
3. use(a)
4. def c
5. use(c)
6. use(d)

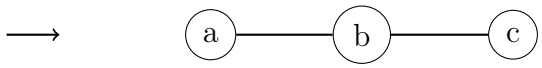


Рис. 1: Пример преобразования кода в граф

В графе на рисунке 1 есть ребро (a, b) потому что в момент 2 переменная a жива, но в графе нет ребра (a, c) потому что в момент 4 у переменной a не найдется такого момента τ_{def}.

В своей статье Чайтин подчеркнул что задача является NP полной. То есть, что любой граф можно свести некоторому к исходному коду программы. Для этого построим исходный код по следующему правилу:

1. Если в графе существует вершина NODE_i, то в исходном коде будет объявление переменной с таким названием.

2. Если в графе есть ребро $(\mathrm{NODE}_i, \mathrm{NODE}_j)$, то в исходном коде добавим использование переменных например суммирование. Так, чтобы эти переменные не могли занимать один регистр.

Видно что этот алгоритм использует немного другое определение интерференции, однако, несложно убедиться что эти определения совпадают

Этот метод действительно работает, однако не всегда все так просто, иногда требуются дополнительные конструкции например операторы ветвления, для того чтобы проблема раскраски графа действительно свилась к проблеме распределения регистров. Рассмотрим пример когда такие преобразования потребуются.

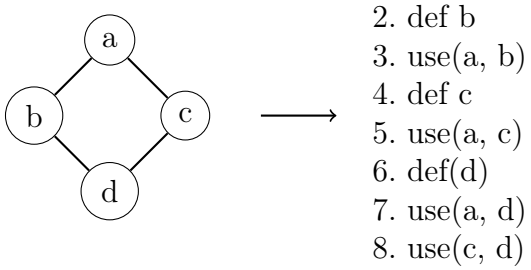


Рис. 2: Пример преобразования графа в код

На этом примере видно, что код построенный по алгоритму который был предложен Чайтина не всегда соответствует изначальному графу. В этом случае в исходном коде переменные a и d интерферируют, хотя в изначальном графе ребра (a, d) не было. Код построенный по графу из рисунка 2 должен выглядеть как на рисунке 3.

```
01. def a
02. if statement:
03.     def b
04.     use(a)
05.     def d
06. else:
07.     def c
08.     use(a)
09.     def d
10. use(d)
```

Рис. 3: Правильный вид исходного кода

4 Идея Чайтина

Как уже было сказано выше, идея распределения регистров при помощи построения графа помех уже была предложена, однако ее реализация впервые появилась в работе Чайтина.

В чем заключается идея изложенная в этой статье:

1. Построить граф помех.
2. Удалить из графа все вершины, степень которых меньше n. Если граф стал пустым, это означает, что исходный граф можно было раскрасить в n цветов. В противном случае выбрать вершину, степень которой больше n, и **удалить** её (**выгрузить**). При этом необходимо реализовать механизм выгрузки вершины из памяти перед её использованием и последующей загрузки обратно. После этого перейти к пункту 1.

Теперь разберёмся в деталях. Как строить граф помех описано в разделе 3. Далее удалим все вершины, у которых меньше n соседей. Важно понять, что это действие никак не повлияет на хроматическое число графа. Действительно, если у вершины меньше n соседей, всегда найдётся цвет, в который её можно покрасить. Таким образом, удаление таких вершин не изменит хроматического числа графа. Это упрощает задачу, поскольку мы можем просто убрать все такие вершины. После этого остаются только вершины, у которых n или более соседей. Теперь необходимо выбрать одну из вершин для удаления. Принцип выбора следующий: для каждой вершины вычисляется значение, называемое **стоимостью выгрузки**. Сначала посчитаем количество объявлений и

использований вершины. При этом нужно учитывать вес каждого использования и объявления, равный частоте их появления. А затем для того, чтобы получить **стоимость выгрузки** конкретной вершины, возьмем отношение $\frac{\textit{количество использований}}{\textit{степень вершины}}$. Теперь когда необходимо выбрать вершину для выгрузки, выберем вершину с наименьшей *стоимостью выгрузки*.

Затем необходимо перестроить граф помех, так как после добавления кода для выгрузки, граф может измениться. После этого нужно еще раз попытаться раскрасить граф.

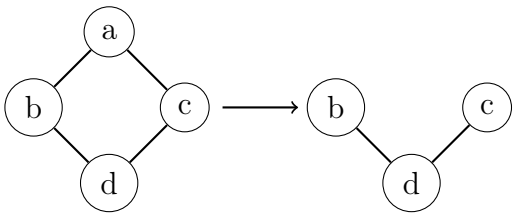
Рассмотрим, как раскрасить граф, если известно, что его можно раскрасить в n цветов. Для этого, каждый раз, когда вершина удаляется из графа, будем помещать её в стек. Когда граф станет пустым, начнём извлекать вершины из стека по одной и выбирать для каждой цвет так, чтобы никакая соседняя вершина не имела того же цвета.

Это всегда возможно, так как на момент помещения вершины в стек её степень была меньше n, что гарантирует наличие свободного цвета. Таким образом, мы получим корректную раскраску графа, а вместе с ней — распределение регистров.

5 Проблемы в алгоритме Чайтина

Однако в этом алгоритме есть некоторые проблемы которые обнаружил Бриггс и исправил в своей работе. Например, алгоритм Чайтина по умолчанию считает, что если у вершины больше или n соседей, то выбрать для нее цвет не получится. Однако это не так. Рассмотрим несколько примеров чтобы проблемы алгоритма стали понятны.

Первый пример это граф с рисунка 2. Нетрудно заметить что такой граф можно покрасить в два цвета. Однако алгоритм Чайтина не найдет такую раскраску и выгрузит одну вершину.



Еще один пример неэффективной работы алгоритма Чайтина — это алгоритм SVD-разложения. В данном алгоритме используются несколько глобальных переменных и вложенных циклов. Проблема распределения регистров возникает именно из-за глобальных переменных. Однако, поскольку стоимость выгрузки глобальных переменных

слишком велика, в первую очередь выгружаются переменные циклов. Это не решает проблему, так как основная сложность связана не с переменными циклов. В результате некоторые регистры могут оставаться свободными, несмотря на то, что переменные циклов выгружаются в память, что ещё больше снижает эффективность работы алгоритма.

Структура кода выглядит приблизительно так:

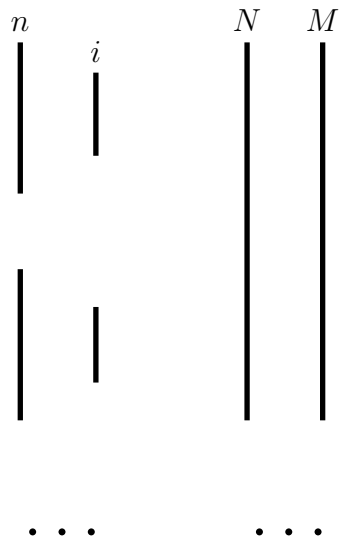
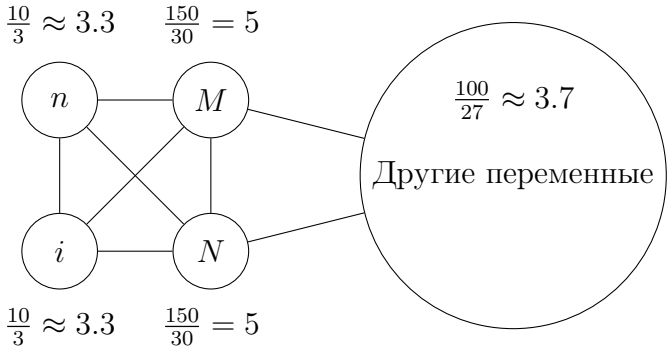
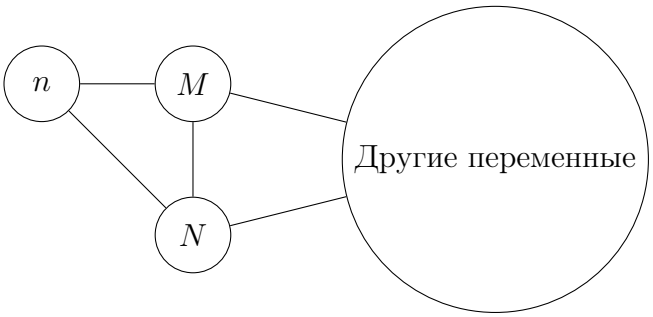


Рис. 4: Структура кода

Есть несколько дешевых для выгрузки переменных i и n, и несколько дорогих переменных N и M. Граф помех выглядит следующим образом.



Для простоты представим, что других переменных 27 и каждый из них имеет 100 исполнений. При этом i и n имеют 3 соседа, и используются 10 раз. У глобальных переменных N и M таким образом получается 30 соседей, и пусть их каждая из них используется 150 раз. Будем считать что граф необходимо покрасить в 2 цвета. Рассчитаем для каждой вершины стоимость выгрузки $\text{cost}(i) = \text{cost}(n) = \frac{10}{3} \approx 3.3$, $\text{cost}(N) = \text{cost}(M) = \frac{150}{30} = 5$, $\text{cost}(\text{other}) = \frac{100}{27} \approx 3.7$. Как видно в первую очередь будет производиться выгрузка переменных i или n.



TO BE CONTINUED...