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 оптимального решения* задачи.

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*. Поскольку в настоящее время скорость работы с регистрами может отличаться от скорости работы с *внешней памятью* может отличаться на порядки.

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.

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 трудной**, необходимо построить *полиномиальную редукцию* для некоторой *NP полной* задачи, а так же показать что решение задачи позволяет получить решение некоторой *NP полной* задачи.

Сначала введем некоторые определения.

> **Определение 1.** 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 2.** Будем говорить что переменные a и b *интерферируют*, если одна из переменных жива в момент объявления другой переменной и их значения не совпадают.

Покажем что можно построить **граф помех**. И таким образом покажем что решение нашей задачи, поможет решить *NP полную* задачу, в нашем случае задачу о *раскраске графа*.

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.

Пусть Var множество переменных, Reg множество регистров, V множество вершин, E множество ребер такого графа, C множество цветов. Тогда существует отображение $\varphi : \text{Var} \to V$ биективное. При этом выполняются следующее свойство:

$$\forall a, b \in \text{Var a и b интерферируют} \Leftrightarrow (\varphi(a), \varphi(b)) \in E$$

Предположим что у нас есть распределение регистров. То есть имеется отображение $\psi : \text{Var} \to \text{Reg}$, такое, что $\forall a, b \in \text{Var}$; если a и b интерферируют $\Rightarrow \psi(a) \neq \psi(b)$. Для того чтобы раскрасить граф, необходимо предъявить отображение $\text{color} : V \to C$ обладающее следующим свойством:

$$\forall A, B \in V; (A, B) \in E \Rightarrow c(A) \neq c(B) \qquad (*)$$

Введем дополнительное отображение $\mu : \mathrm{Reg} \to C$. Оно биективно сопоставляет регистры и цвета.

Построим требуемое отображение c следующим образом:

$$\forall v \in V \; \mathrm{color}(v) = \mu(\psi(\varphi^{-1}(v)))$$

Покажем что такое отображение действительно обладает свойством *. Рассмотрим $A, B \in V$, если $(A, B) \in V$, то переменные $\varphi^{-1}(A)$ и $\varphi^{-1}(B)$ интерферируют. А так как они интерферируют, то $\psi(\varphi^{-1}(A)) \neq \psi(\varphi^{-1}(B))$. Значит под действием μ эти вершины попадут в разные цвета. ■

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

Рассмотрим простой пример:

Пример 1.

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

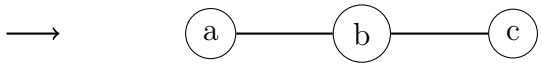


Рис. 1: Пример преобразования кода в граф

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_{use}.

В своей статье Чайтин предложил следующий алгоритм построения графа из исходного кода:

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.

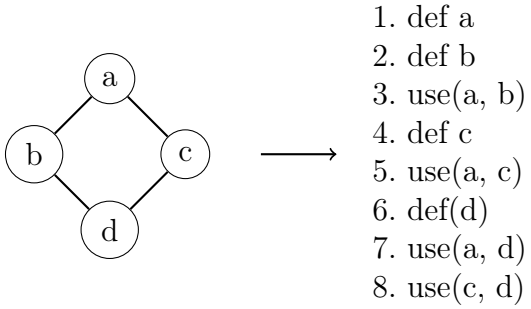


Рис. 2: Пример преобразования графа в код

```
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

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

Этот метод работает не всегда. 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.

Пример 2.

На этом примере видно, что код построенный по алгоритму который был предложен Чайтином 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.

4 Статья Чайтина

4.1 Идея

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

Чтобы распределить переменные по регистрам в статье Чайтина предлагается:

1. Построить граф помех.
2. - Если граф стал пустым, это означает, что исходный граф можно было раскрасить в n цветов.
   - Если есть вершина со степенью меньше n, убираем ее из графа, и кладем ее в стек, переходим к 2
   - 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.
3. Теперь можно доставать из стека вершину и присваивать ей некоторый цвет.

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, предложенный Чайтином, 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.

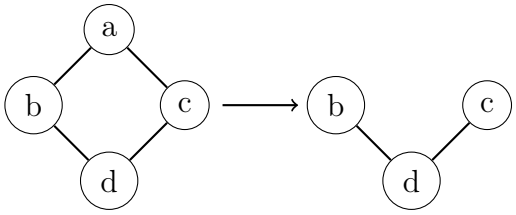
Рассмотрим, как раскрасить граф, если известно, что его можно раскрасить в n цветов. Для этого, каждый раз, когда вершина удаляется из графа, будем помещать её в стек. Когда граф станет пустым, начнём извлекать вершины из стека по одной и выбирать для каждой цвет так, чтобы никакая соседняя вершина не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

4.2 Проблемы в алгоритме Чайтина

В этом алгоритме есть некоторые проблемы которые обнаружил и исправил Бриггс 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:

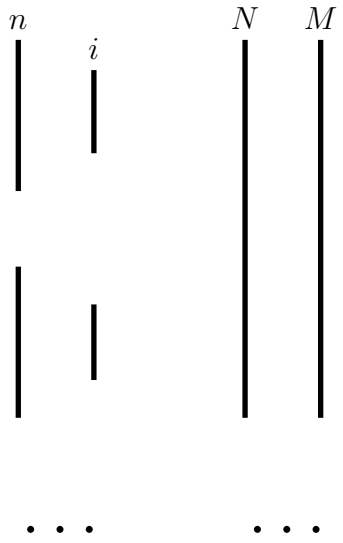
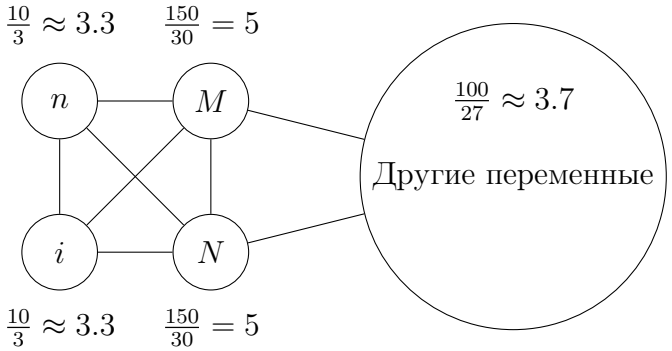


Рис. 4: Структура кода

В этом коде есть переменные i и n, это переменные циклов. А так же есть переменные N и M это некие глобальные переменные. Для простоты можно считать что они определяют границы циклов, при этом в самих циклах не участвуют.



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.

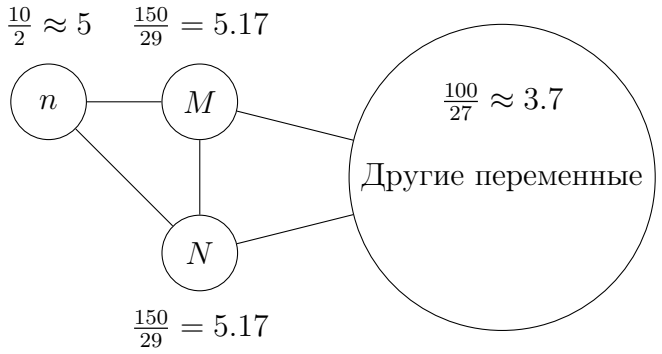


Рис. 5: Пересчитанные стоимости

Пересчитаем значение *стоимости выгрузки* (см. Рисунок 4.2). И снова алгоритм предлагает выгрузить переменную n. Таким образом алгоритм вновь выбирает переменную n, переменную цикла, к выгрузке, однако это снова не решит проблему. В конце переменные M и N будут выгружены, как и переменные i и n. А так как переменные i и n используются в цикле, то их придется очень часто выгружать и загружать, что *значительно* повлияет на производительность.

5 Статья Бриггса

5.1 Анализ проблем

В своей статье Бриггс описывает модифицированный алгоритм раскраски графа, он называет его **optimistic coloring**. Разберем какая именно проблема скрывается в выше описанных примерах, и как можно их решить.

В примере с рисунка 2, как уже было сказано, алгоритм Чайтина *выгрузит* какую-то переменную, хотя очевидно что обойтись можно и без этого. Эта проблема связана с тем что по умолчанию алгоритм считает, что если есть вершина графа со степенью большей n, то такой граф нельзя раскрасить в n цветов.

В примере SVD разложения проблема возникает не когда алгоритм выбирает *сгрузить* переменные i и n, а когда после выгрузки N и M он не обнаруживает что переменные n и i вполне *могут поместиться* на регистры. Несмотря на то, что сначала казалось иначе.

5.2 Идея

Для решения вышеперечисленных проблем Бриггс предлагает внести следующие корректировки в алгоритм Чайтина:

1. В пункте 2 алгоритма, в случае обнаружения вершины у которой больше n соседей, не будем сразу выгружать вершину, а положим ее на стек.

2. Соответственно в пункте 3 возникает проблема, ведь теперь не всегда вершину можно будет покрасить. В этом случае оставим их не покрашенными, это те переменные которые нужно выгрузить.

Эти изменения помогут решить проблемы алгоритма Чайтина. Первая проблема решается в пункте 1. Теперь в примере с рисунка 2 хотя некоторая вершина и будет выбрана для выгрузки, во время этапа 3 всем вершинам удастся получить цвет.

Вторая проблема также решена. Теперь несмотря на то, что переменные i, n, M, N будут выгружены (можно считать что в таком порядке), при попытке подобрать цвета для M, N станет понятно что подобрать для них цвета не получится. Поэтому переменные i, n не придется выгружать.

Эвристика предыдущего алгоритма скорее отвечала на вопрос "имеет ли вершина $< n$ соседей?", чем на вопрос "можно ли подобрать цвет для этой вершины?". Новый алгоритм дает ответ на последний вопрос.

6 Статья Полетто

В этой секции рассмотрим идею, описанную в статье Полетто. В отличие от первых двух алгоритмов этот алгоритм базируется не на раскраске графа. Этот подход основан на изучении *интервалов жизни* переменных. Поскольку определение *интервалов* можно произвести в один проход по коду, и сам алгоритм работает за линейное время, то этот метод работает значительно быстрее по сравнению с предыдущими. При этом исходя из результатов предоставленных в этой статье следует что и производительность страдает не значительно.

6.1 Идея

Как уже было сказано идея базируется на *интервалах жизни*.

```
01. def a
02. if statement:
03.     use(b)
04. else:
05.     use(a)
```

Рис. 6: Псевдокод

Определение 3. Будем говорить что $[i, j]$ является **интервалом жизни** переменной v если не существует таких моментов i' и j' что $i' < i$ и $j < j'$ и при этом переменная v *жива* в эти моменты. Интервалы жизни для переменной a будем обозначать interval_a

Определение 4. Будем говорить что переменная a жива в интервальном смысле в момент τ, если $\tau \in \text{interval}_a$.

Стоит заметить что в *интервале жизни* переменной могут встречаться интервалы в которых она будет не жива. В дальнейшем это поможет понять почему этот алгоритм не может решить проблему раскраски графа. То есть он на сама деле не решает NP полную задачу, а лишь ее некоторое приближение.

Пример 3.

Рассмотрим пример 6. В данном случае интервалы жизни переменной $a \in [1, 5]$, однако в интервале $[2, 3]$ переменная не жива, так как если исполнение пошло по этому потоку то переменная больше не будет использована.

Теперь перейдем к алгоритму. Предположим что нужно распределить переменные по R регистрам. В каждый момент времени τ есть живые в интервальном смысле переменные, предположим их n. В таком случае в момент τ необходимо выгрузить $n - R$ регистров. Количество живых переменных меняться только если какая-нибудь переменная станет живой в интервальном смысле, или наоборот какая нибудь перестанет жить.

Для того чтобы удобнее отслеживать живые переменные алгоритм хранит их в массиве отсортированном по возрастанию правой границе интервалов жизни. Аналогично список еще не живых переменных удобно хранить в массиве, который отсортирован по возрастанию левых границ интервалов жизни.

1. Удалить все не живые на момент τ переменные из списка живых.

2. Добавить живые на момент τ переменные из еще не живых в список живых. На этом этапе возникает необходимость выгрузить некоторые переменные.

На этапе 1 необходимо просмотреть список живых переменных. При этом стоить отметить что если необходимо убрать k переменных, то нужно просмотреть не более $k+1$ переменной. Это происходит за счет того что переменные отсортированы в порядке возрастания правой границы. То есть если переменная с индексом i остаётся живой в момент времени τ, то проверять $i+1$ переменную нет необходимости.

На этапе 2 необходимо просмотреть список еще не живых переменных. Аналогично из-за того как отсортирован список, необходимо просмотреть только те переменные, которые станут живыми в текущий момент времени.

6.2 Оценка эффективности