1. Введение

В процессе компиляции программы в машинный код конкретного процессора возникает проблема **распределения регистров**. Множество исследований посвящено данной проблеме, мы же рассмотрим решения предложенные в работах Чайтина (Chatin) [2], Бриггса (Briggs) [1], Полетто (Poletto) [5]. Отметим что задача является NP полной (сложность задачи будет рассмотрена в секции 3), и поэтому все алгоритмы, описанные ниже, являются эвристическими.

В рассматриваемых работах предложено два подхода к решению этой проблемы. Первый, через построение **графа**, этот подход был предложен еще до работ Чайтина в работах Кокка (Cocke) в 1970 [3], Ершова в 1962 [4], Шварца (Schwartz) в 1973 [6]. Второй подход это **линейная аллокация**, который для распределения регистров использует не граф, а линейное представление кода. Этот метод был предложен в работе Полетто.

2. Постановка задачи

В процессе компиляции программы обычно используется **промежуточное представление программы (IR)**, оно допускает наличие неограниченного числа переменных. Однако реальная архитектура процессора предоставляет ограниченное количество регистров. Это ставит перед компилятором задачу корректного распределения переменных по имеющимся регистрам, при нехватке регистров необходимо некоторые переменные разместить в памяти.

Для понимания задачи рассмотрим пример: программа содержит выражение, в котором используются 3 переменные, в то время как архитектура процессора предусматривает лишь 2 регистра. В этом случае однозначно распределить переменные по регистрам не получится. Возникает необходимость проводить **выгрузку** (в англоязычной литературе *spill*) какой-то из переменных в память. При обращении к этой переменной потребуется загрузить её из памяти, а при изменении — снова сохранить обратно.

Таким образом, важно не только обеспечить корректность выполнения программы, но и минимизировать количество обращений к памяти, поскольку в настоящее время скорость работы с регистрами может отличаться от скорости работы с внешней памятью на порядки.

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.

Представим исходный код в виде CFG (control flow graph). Будем говорить что переменная x *жива* в точке p, если существует путь в CFG из точки определения, проходящий через p и использование переменной, на котором не встречается другого определения переменной x.

Определение 2.

Будем говорить что переменные a и b *зацеплены*, если одна из переменных жива в точке объявления другой переменной.

Определение 3.

$IG(V)$ – *граф зацепленности* для множества переменных некоторой программы. $IG(V) = (V, E)$, где V – множество переменных, а

$$E = \{(a, b) \mid a, b \in V \wedge \ a \text{ и } b - \text{зацеплены}\}.$$

Если удастся получить раскраску этого графа в n цветов, то и проблема распределения переменных по n регистрам для этой программы будет решена.

Рассмотрим простой пример:

Пример 1.

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

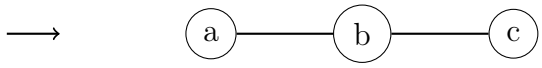


Рис. 1: Пример преобразования кода в граф
def: Объявление переменной
use: Использование переменной

В графе на рисунке 1 есть ребро (a, b), потому что в точке 2 переменная a *жива*, но в графе нет ребра (a, c), потому что в момент 4 у переменной a не найдется пути который прошел бы через использование.

Теперь покажем, что по заданному графу можно построить программу, распределение регистров которой даст раскраску.

Пусть Var – множество переменных, R – множество регистров. Тогда пусть существует отображение $\varphi : \text{Var} \to R$. При этом выполняются следующее свойство:

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

Если построить биективное отображение $\psi : R \to C$, где множество C – множество цветов. То раскраска графа $IG(V)$ строится как $\zeta = \psi \circ \varphi$.

В своей статье Чайтин [2] предложил следующий алгоритм построения кода из графа:

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

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

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.

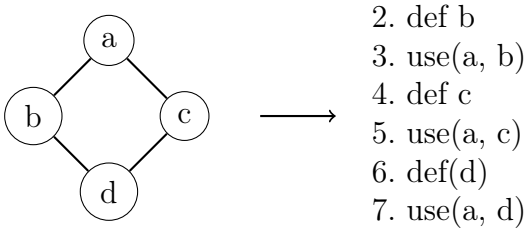


Рис. 2: Пример преобразования графа в код

На рисунке 2 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.

```
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

Алгоритм Чайтина легко исправляется следующим подходом: (Алгоритм под вопросом)

1. Для каждой вершины исходного графа создать переменную.
2. Если между вершинами существует ребро, то создать `if statement`, в теле которого объявить эти переменные и использовать таким образом, чтобы они стали зацеплены.

4. Подход Чайтина

4.1. Идея

Чтобы распределить переменные по N регистрам в статье Чайтина предлагается:

1. Построить граф зацепленности.
2. Опустошить граф, переложив все вершины в стек согласно следующим правилам:

   (a) Если есть вершина со степенью меньше N, убираем ее из графа, и кладем ее в стек, переходим к 2;

   (b) Если граф стал пустым перейти на пункт 3;

   (c) В противном случае выбрать вершину, степень которой больше N, и **удалить** её (**выгрузить** соответствующую переменную). 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 по одной вершине из стека, и присваивать ей цвет.

Теперь разберёмся в деталях. Как строить граф зацепленности 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. 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. Можно считать, что если использование переменной или ее объявление происходит в цикле, то его вес будет равен 10. 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 зацепленности, так как после добавления кода для выгрузки, граф может измениться. После этого нужно еще раз попытаться раскрасить граф.

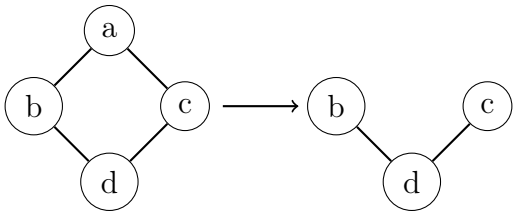
Рассмотрим как раскрасить граф, если известно, что его можно раскрасить в N цветов. Для этого, каждый раз, когда вершина удаляется из графа, будем помещать её в стек. Когда граф станет пустым, начнём извлекать вершины из стека по одной и выбирать для каждой цвет так, чтобы никакая соседняя вершина не 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 проблемы которые обнаружил и исправил Бриггс в своей работе [1]. Рассмотрим несколько примеров.

Рассмотрим 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 она не связана 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 программы.

На рисунке 4 представлена приблизительная структура кода.

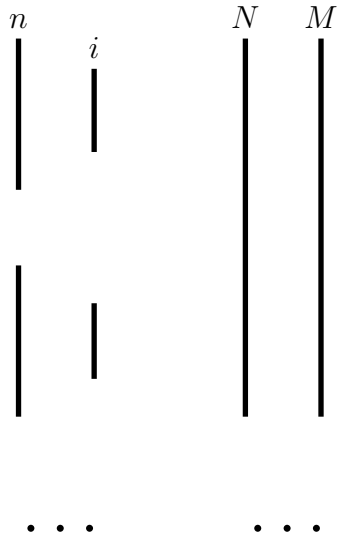


Рис. 4: Структура кода

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

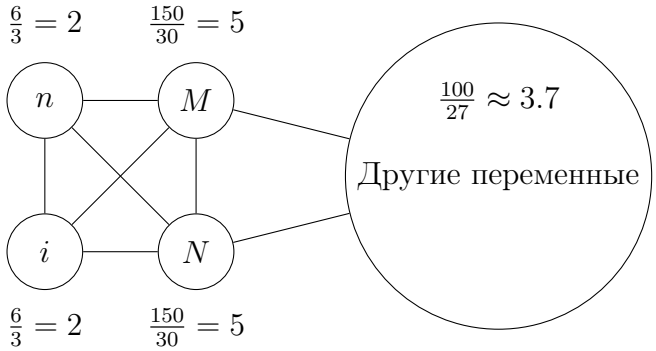


Рис. 5: Граф зацепленности и стоимости выгрузки

Пусть также есть другие переменные, их 27 и каждая из них имеет 100 использований. При этом i и n имеют 3 соседа, и используются 6 раз. У глобальных переменных N и M 30 соседей, и пусть каждая из них используется 150 раз. Будем считать что граф необходимо покрасить в 2 цвета. Рассчитаем для каждой вершины стоимость выгрузки $\text{cost}(i) = \text{cost}(n) = \frac{6}{3} = 2$, $\text{cost}(N) = \text{cost}(M) = \frac{150}{30} = 5$, $\text{cost}(\text{other}) = \frac{100}{27} \approx 3.7$. Как видно в первую очередь будет производиться выгрузка переменных i или n.

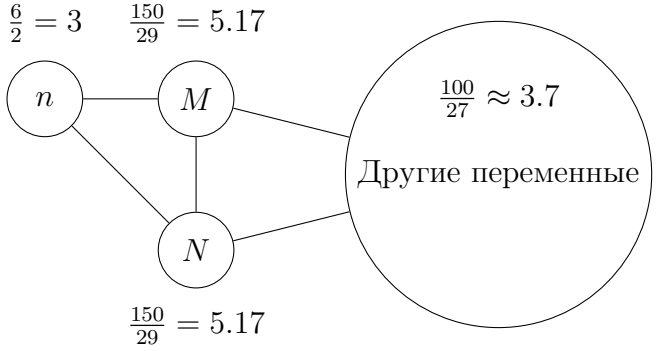


Рис. 6: Пересчитанные стоимости

Пересчитаем значение стоимости выгрузки (см. Рисунок 6). И снова алгоритм предлагает выгрузить переменную n, переменную цикла, однако это 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. Подход Бриггса

5.1. Анализ проблем

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

В примере, представленном на рисунке 2, как было отмечено ранее, алгоритм Чайтина производит выгрузку хотя бы одной переменной. Однако выгрузку можно избежать. Эта проблема связана с тем, что по умолчанию алгоритм считает, что если в графе есть вершина со степенью большей N, то такой граф нельзя раскрасить в N цветов.

В примере с SVD разложением 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 могут поместиться на регистры.

5.2. Идея

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

1. В пункте 2c алгоритма при обнаружении вершины с количеством соседей большим или равным N, вместо немедленной выгрузки, разместим 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 найденные проблемы алгоритма Чайтина. Первая проблема решается в пункте 1. Теперь в примере, представленном на рисунке 2, хотя некоторая вершина и будет выбрана для выгрузки, во время этапа 3 всем вершинам удастся получить цвет.

Вторая проблема также решена. Теперь хотя 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. Подход 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 кода, порожденного таким алгоритмом, страдает незначительно.

6.1. Идея

Представим программу в некотором IR и пронумеруем его произвольным образом. Например, если имеется дерево, то в качестве нумерации такого IR можно взять порядок обхода DFS. Хотя алгоритм не опирается на какую-то конкретную нумерацию, выбор нумерации может значительно повлиять на результат аллокации.

Определение 4. Будем говорить что $[i, j]$ является **интервалом жизни** переменной v если не существует таких точек i' и j' что $i' < i$ и $j < j'$ и при этом переменная v *жива* в этих точках. Интервалы жизни для переменной a будем обозначать interval_a

Определение 5. Будем говорить что переменная a жива в интервальном смысле в точке p, если $p \in \text{interval}_a$.

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.

Пример 3.

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

Рис. 7: Псевдокод

Рассмотрим пример с рисунка 7. 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 пути то переменная больше не будет использована.

(Опять пошли моменты, переписать на точки)

Теперь перейдем к алгоритму. Пусть необходимо распределить переменные по R регистрам. В каждой точке p есть живые в интервальном смысле переменные, предположим их n. В таком случае в точке p 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 выбора переменных для выгрузки используется эвристика, при которой отдается предпочтение переменной с более поздним окончанием интервала жизни.

Алгоритм, предложенный Полетто:

1. Удалить все не живые в точке p переменные из списка живых.
2. Добавить живые в точке p переменные из еще не живых в список живых. На этом этапе возникает необходимость выгрузить некоторые переменные.
3. Перейти к следующей точке.

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.

На этапе 1 необходимо просмотреть список живых переменных. При этом, стоит 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.

Пример 4.

Рассмотрим пример работы алгоритма линейной аллокации. На рисунке 8 представлены интервалы жизни некоторых переменных. Пусть в данном случае необходимо разместить переменные на 2 регистрах. В момент *2* живы переменные $\{\mathbf{A}, \mathbf{B}\}$, и они обе помещаются в регистры. В момент *3* живы переменные $\{\mathbf{A}, \mathbf{B}, \mathbf{C}\}$, значит необходимо принять решение о выгрузке. Так

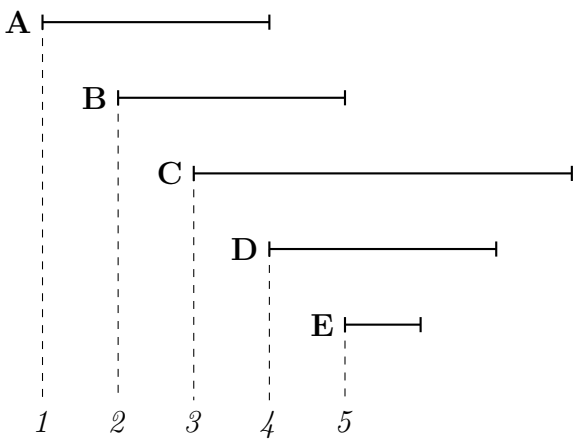


Рис. 8: Пример интервалов жизни

как у переменной **C** интервал жизни заканчивается позже чем интервалы **A** и **B**, то она и будет выгружена. В момент *4* переменная **A** умирает и оживает переменная **D**, то есть живы $\{\mathbf{B}, \mathbf{C}, \mathbf{D}\}$. Вновь необходимо сделать выгрузку, и снова переменная **C** выбирается для выгрузки. В момент *5* переменная **B** умирает и на регистрах освобождается место для переменной **E**. В результате работы алгоритма выгруженной оказалась лишь переменная **C**.

6.2. Оценка сложности

Пусть V количество переменных, и R количество регистров. Тогда сложность алгоритма $O(V \cdot R)$ если для составления массивов интервалов жизни будет использоваться линейный поиск. Напомним что список интервалов отсортирован. Если использовать бинарное дерево, то сложность составит $O(V \cdot \log R)$.

7. Сравнение подходов

В этой секции рассмотрим сравнение описанных выше подходов. [5] Будем сравнивать как различия во времени компиляции программы, так и влияние на производительность во время исполнения.

7.1. Время компиляции

На рисунке 9 представлен график времени компиляции. На вертикальной оси отмечены затраты на компиляцию в циклах на сгенерированную инструкцию. Чем больше

значение тем больше затраты. По горизонтальной оси отложены различные тесты. Для каждого теста есть 3 столбца. Первый столбец **U** — алгоритм распределяющий регистры при помощи подсчета количества использований. Второй столбец **L** — линейной аллокации. Третий столбец **C** — раскраски графа. Так как столбец **U** нас не интересует, то комментировать его не будем. Горизонтальные столбцы разделены на 3 части:

1. **Анализ живности**.
2. **Подготовка к распределению регистров**: Для **L** это расчет интервалов жизни, для **C** построение графа зацепленности.
3. **Распределение регистров**: Для **L** это проход по интервалам, для **C** это раскраска графа.

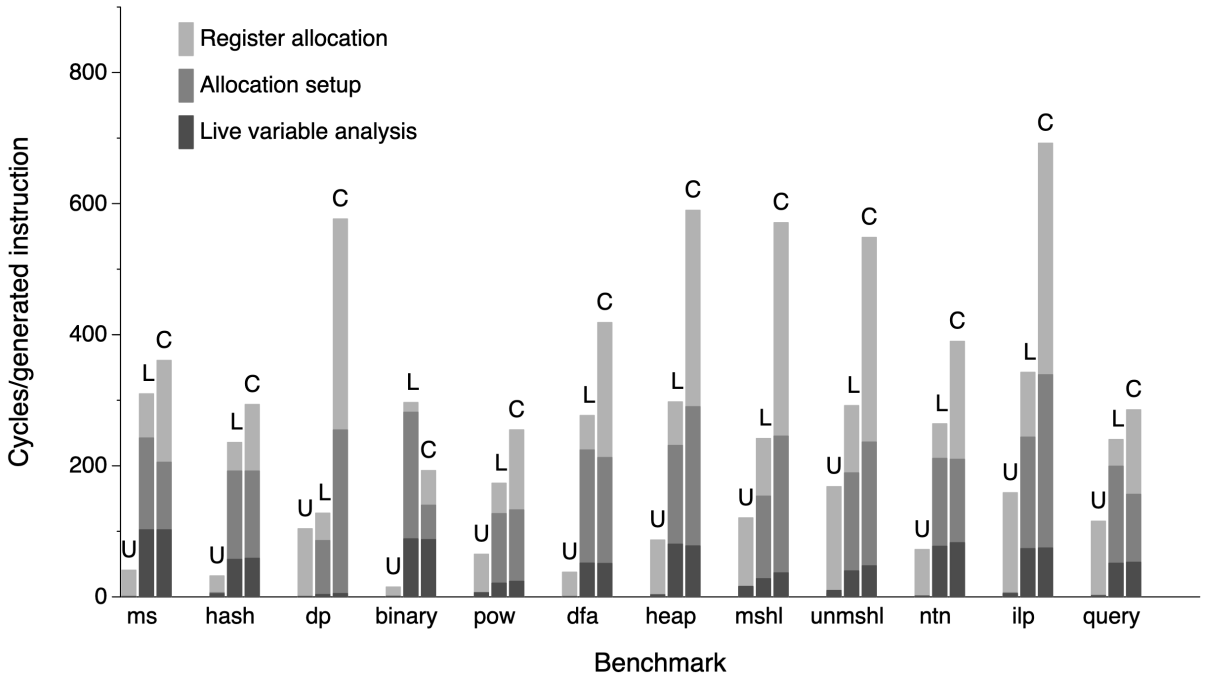


Рис. 9: Время компиляции

Видно, что практически для всех тестов алгоритм линейной аллокации быстрее чем алгоритм раскраски. Интересные значения можно наблюдать в тесте binary. Такие результаты объясняются тем, что в этом тесте небольшое количество переменных и очень много ветвлений. В этом случае построить небольшой граф зацепленности быстрее, чем анализировать длинные интервалы жизни. Однако стоит отметить, что время распределения регистров в этом тесте все же быстрее для линейной аллокации.

7.2. Время исполнения

На рисунке 10 представлены результаты тестирования программ, скомпилированных при помощи различных алгоритмов аллокации регистров. В таблице представлены четыре алгоритма, нас интересуют только Linear scan и Graph coloring. Значения в скобках представляют отношение времени исполнения метода, скомпилированного с данным алгоритмом, к времени исполнения метода, скомпилированного с алгоритмом раскраски графа.

Видно, что результаты линейной аллокации отличаются в пределах 10% от алгоритма раскраски графа.

Benchmark	**Time in seconds (ratio to graph coloring)**			
	Usage counts	Linear scan	Graph coloring	Binpacking
espresso	21.3 (6.26)	4.0 (1.18)	3.4 (1.00)	4.0 (1.18)
compress	131.7 (3.42)	43.1 (1.12)	38.5 (1.00)	42.9 (1.11)
li	13.7 (2.80)	5.4 (1.10)	4.9 (1.00)	5.1 (1.04)
alvinn	26.8 (1.15)	24.8 (1.06)	23.3 (1.00)	24.8 (1.06)
tomcatv	263.9 (4.62)	60.5 (1.06)	57.1 (1.00)	59.7 (1.05)
swim	273.6 (6.66)	44.6 (1.09)	41.1 (1.00)	44.5 (1.08)
fpppp	1039.7 (11.64)	90.8 (1.02)	89.3 (1.00)	87.8 (0.98)
wc	18.7 (4.67)	5.7 (1.43)	4.0 (1.00)	4.3 (1.07)
sort	9.8 (2.97)	3.5 (1.06)	3.3 (1.00)	3.3 (1.00)

Рис. 10: Время исполнения

Список литературы

[1] Briggs P., Cooper K. D., Torczon L. Improvements to graph coloring register allocation // ACM Transactions on Programming Languages and Systems (TOPLAS). — 1994. — Vol. 16, no. 3. — P. 428–455.

[2] Chaitin G. J. Register allocation & spilling via graph coloring // ACM Sigplan Notices. — 1982. — Vol. 17, no. 6. — P. 98–101.

[3] Cocke J., Schwartz J. Programming languages and their compilers: Preliminary notes, 2nd rev. version, new york, ny: Courant inst. of math // Sci., New York University. — 1970.

[4] Ershov A. P. Reduction of the Problem of Memory Allocation in a Block-Structured Language to a Problem of Coloring the Vertices of a Graph // Dokl. Akad. Nauk SSSR. — 1962. — Vol. 142. — P. 785–787.

[5] Poletto M., Sarkar V. Linear scan register allocation // ACM Transactions on Programming Languages and Systems (TOPLAS). — 1999. — Vol. 21, no. 5. — P. 895–913.

[6] Schwartz J. T. On Programming: An Interim Report on the SETL Project. Installment I: Generalities // Lecture Notes, Courant Institute, New York University, New York. — 1973. — P. 1–20.